UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Data Science

Master's Thesis

Submitted to the Data Science Research Group
in Partial Fullfilment of the Requirements for the Degree of

Master of Science (M.Sc.)

in the Field of Computer Science

Grammar-based Compression of RDF Graphs

by
Philip Frerk

Thesis Advisor:

Michael Röder, M.Sc.

Thesis Supervisors:

Prof. Dr. Axel-Cyrille Ngonga Ngomo
Prof. Dr. Stefan Böttcher

Paderborn, June 22, 2019

Declaration

(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

City, Date

Signature

Contents

1 Introduction

The majority of data on the Internet is unstructured, because it is mainly text. That makes it difficult for machines to extract knowledge from the data and answer specific queries. In the context of the Semantic Web, an attempt is made to build a knowledge base using structured data. A possible framework for structured data is the Resource Description Framework (RDF)[1], in which triples of the form $(subject, predicate, object)$ are used in order to express certain facts. A set of triples naturally forms a graph, whereas the different *subjects* and *objects* are nodes and the *predicates* are edges of that graph. As those graphs express facts, they are called knowledge graphs.

In reality, such knowledge graphs can become very large with millions or even billions of nodes and edges. The following three use cases can then become very hard or even infeasible: storage, transmission and processing. One way to circumvent this problem is using compression. There are many existing compressors which work for all kinds of data, but the problem is that the compressed data is not query-able. Also, their compression might not be so strong, since they do not take advantage of the special features of RDF. For these reasons RDF-specific compression techniques are of interest. An example of such an RDF compressor is Header Dictionary Triples (HDT) from [FMPG+13]. Here, the data is made smaller by a compact representation of the triples.

There is a completely different technique which is called grammar-based compression. This method has so far been tested very little for RDF graphs. As the name suggests, it is based on the principle of a formal grammar, with productions that can be nested among each other. There are not yet many grammar-based compression algorithms for graphs. One example is GraphRePair (GRP) [MP18]. This is a compressor that works for any type of graph and is therefore also applicable for RDF.

In this thesis, it will be investigated to what extent grammar-based compression is suitable for RDF and whether it delivers even better results than HDT.

[1] https://www.w3.org/RDF/

1.1 General Idea

As already mentioned, there are essentially three use cases for RDF data:

1. Storage
2. Transmission
3. Processing/Consumption

The idea is to make all these use cases easier or possible by compression, especially if very large amounts of data have to be handled. In the first two use cases, compression can be helpful by reducing the size of the data. Thus, the data can be stored more compactly and also can be transferred faster, which occurs frequently in reality and can become a problem due to possibly slow transfer speeds.
The third use case (Processing/Consumption) is about reading or writing access to data. The more common case of read access is typically the execution of queries on RDF data. These are most often neighborhood queries. A compression algorithm can help in this respect by making it possible to answer such queries directly on the compressed graphs. This may even be faster than on the original data due to the reduced size.
As stated in [MP18], a graph compressed by GRP is not well suited for those just mentioned neighborhood queries. Such queries will take much longer than on the original graph. Therefore, the thesis will focus on GRP's potential of compressing RDF data strongly and the query times will not be evaluated.

1.2 Thesis Structure

In Ch. 2, the necessary fundamentals are presented. It will mainly be about the two compressors HDT and GRP.
The following parts of the thesis consist of two main tasks. The first one is comparing GRP with HDT. First results will show that GRP outperforms HDT in many cases. Therefore, the second task is to suggest improvements of GRP, which can partly also be used for HDT.
In Ch. 3, a formal model for compressors is introduced, which will be used in all following chapters. Afterwards, approaches to the main tasks will be presented. Implementation details will be discussed in Ch. 4. In Ch. 5, the approaches to the different tasks will be evaluated by using real world RDF data. Finally, the results of the thesis are summarized in Ch. 6 and future work on this topic is presented.

2 Related Work

This chapter gives an introduction to RDF and presents the different compression algorithms discussed in the thesis.

2.1 RDF

RDF is a framework for structured data on the web. According to [HKRS08], it can be formally described as follows. Let the following sets be mutually disjoint:

- U (URI references, typically represent unique entities or properties)
- B (Blank nodes, represent an arbitrary value, necessary for displaying more complex logical statements)
- L (Literals, represent fixed values, e.g. numbers or strings)

An RDF triple $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ displays the statement that the subject s is related to the object o via the predicate p. So, a subject can be anything but a literal. A predicate can only be a URI and an object can be anything.
Similar to entities or properties, each blank node has a unique ID in order reference it in the graph. But in contrast to URIs, these IDs do not have a semantic meaning. That fact will be used in Ch. 3.4.2.

2.1.1 Ontology

In an RDF graph, relationships between entities or properties are displayed. In addition, entities or properties can also be assigned to types. In an ontology, relationships between these types can be defined. More importantly for this thesis, features of types or properties are also described there. An ontology is itself also an RDF graph, which is described in one of the languages RDF Schema (RDFS) [1] or Web Ontology Language (OWL) [2] of which OWL is the more powerful one. [HKRS08]
In the course of this thesis, ontologies will be used to achieve a better compression. More precisely, specific features of properties will be used to do that.

[1] https://www.w3.org/TR/rdf-schema/
[2] https://www.w3.org/TR/owl-ref/

2.2 HDT

HDT ([FMPG+13]) is an approach for compressing RDF data which is directly tailored to RDF and the data compressed by it is still query-able. The idea behind it is to make RDF's extensive representation more compact, thus reducing memory size. To achieve that, HDT creates three different components during the compression: A header, a dictionary and the triples.
The header contains statistical information about the RDF graph (e.g. number of subjects) and is not needed to decompress the file.
In the dictionary, all URIs and literals (which are usually quite long) are mapped to unique IDs (integers) and that mapping is stored persistently. HDT also compresses the dictionary to store it more efficiently. Since URIs typically have long common prefixes, a prefix-based text compression ([CW84]) is used here. It identifies longest common prefixes and takes advantage of the redundancy by storing the prefixes only once.
The triples component only uses the IDs produced by the dictionary in order to denote the triples. Usually, in many plain text formats for RDF (e.g. N-Triples [3]) each triple is written down after another. That simple notation is called 'Plain Triples' in HDT and can be seen in Fig. 2.1. This does not only require much space, but is also not good for query performance, since each single triple has to be traversed within a query execution.
It is easy to see that a more compact representation can be achieved by cleverly grouping the triples. That is exactly what HDT does. It is assumed that the following triples all have the same subject s. These would then be written in N-Triples as follows:

$$(s, p_1, o_{11}), ..., (s, p_1, o_{1n_1}), (s, p_2, o_{21}), ..., \\ (s, p_2, o_{2n_2}), ..., (s, p_k, o_{kn_k})$$

In HDT the triples are then grouped by s:

$$s \rightarrow [(p_1, (o_{11}, ..., o_{1n_1})), (p_2, (o_{21}, ..., o_{2n_2})), ..., \\ (p_k, (o_{kn_k}))]$$

Since all triples have the same subject s, it does not have to be written down at the beginning of each triple anymore. On the second level, the triples are then grouped according to the predicates. Thus, all triples with the predicate p_1 come first, then all with the predicate p_2 and so on. Finally, only the different objects have to be enumerated since the subject and predicate are already implicitly given. It will now be explained how HDT implements this mechanism.
The above mentioned grouped representation is called 'Compact Triples' and can be seen in Fig. 2.1 in the middle. In the array 'Predicates' are the IDs of the predicates. A '0' denotes a change of the subject. For example, the first two entries in the 'Predicates' are linked to the first subject. At position 3 in 'Predicates', there is a '0'. Therefore, the following predicates are linked to the next subject (subject 2 in this case).
This works analogously for the objects, in the array 'Objects' the IDs of the objects are listed, and a '0' here denotes a change of the predicate. For instance, the first entry in 'Objects' is a '6', so the first derived triple would be $(1, 2, 6)$. The second entry of 'Objects' is a '0', since there is a change from predicate '2' to '3' in 'Predicates'.
The last representation of triples is called 'Bitmap Triples'. This is a slight adaption of 'Compact Triples'. The array S_p has the same content as 'Predicates', but without the zeros. That job is

[3] https://www.w3.org/TR/n-triples/

done by the bit-array B_p in which a '1' at position i means that at position i in S_p there comes a change of the subject (this change was previously denoted by a '0' in 'Predicates'). That works analogously for the objects where S_o has the same content as 'Objects', but without zeros.
The advantage of 'Bitmap Triples' is that queries can be executed faster on this representation. Also, the compressed size is slightly smaller than with 'Compact Triples'. Therefore, 'Bitmap Triples' are always used in the current version of HDT.
HDT has also procedures for answering queries, but those will not be mentioned here, because the thesis will focus on HDT's compression ability.

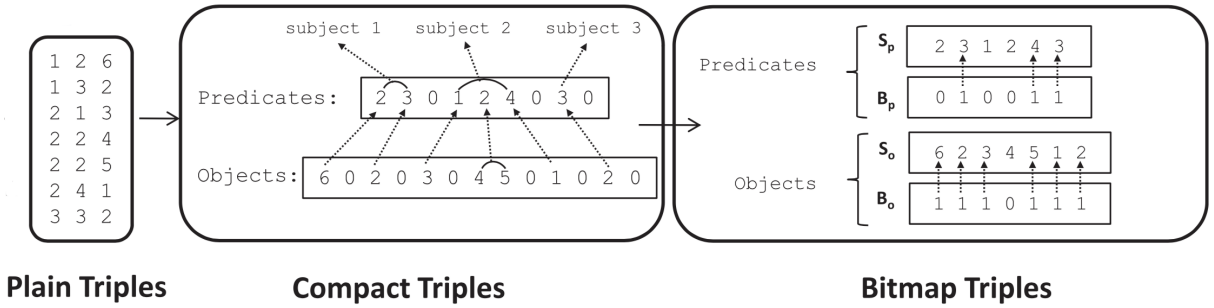


Figure 2.1: Three different representations of triples in HDT, figure from [FMPG+13].

2.3 Grammar-based Graph Compression

In this chapter, two different approaches to grammar-based graph compression are discussed. First, [Dü16] is introduced and it is briefly explained why the approach is less suitable for RDF. Then GRP ([MP18]), which the thesis will focus on, is introduced.
In general, grammar-based compressors try to find patterns that occur multiple times in the graph. Such patterns are subgraphs that can be of different kinds. Usually, small patterns are searched for, since it is possible to find more occurrences of these. In addition, it is possible to combine several small patterns into one large one, so that even large parts can be compressed. These small patterns are often called digrams, because they consist of two elements. The exact shape of the digrams depends on the respective algorithm.

2.3.1 Dürksen's Algorithm

An approach to grammar-based compression of graphs was developed in [Dü16]. Here, the authors assume a hyper graph with node and edge labels. Such a graph can be seen in Fig. 2.2 on the left. To simplify the compression, a transformation is now executed, in which a new node is inserted for each edge e, which then has the same label as e. The original structure of the graph is obtained by connecting two nodes, which were previously connected by an edge, indirectly by the new node. The result of the transformation is a graph, which only has node labels, but no edge labels. Moreover, hyper edges (edges with more than two incident nodes) are no longer present due to the transformation.

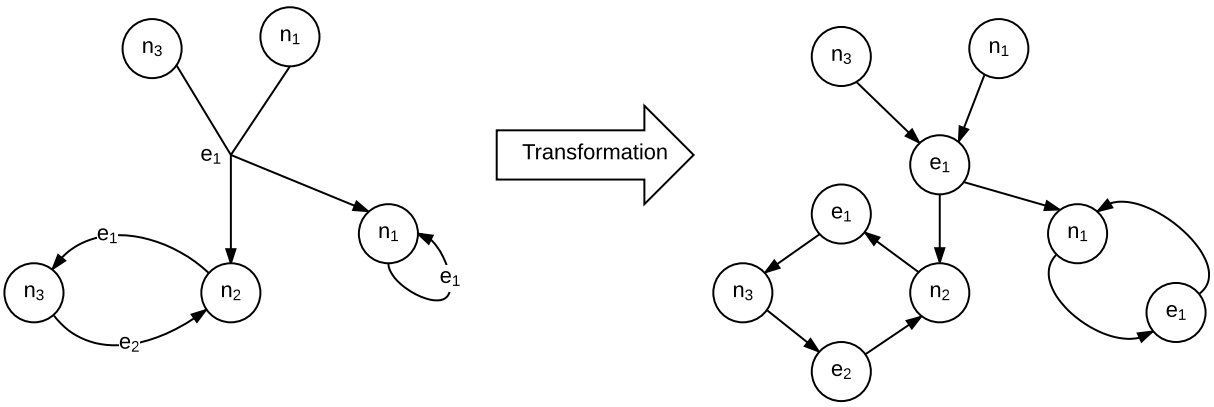


Figure 2.2: Transformation of a labeled graph (each edge is replaced by a new node having the label of the replaced edge).

Thus, digrams like in Fig. 2.3 can be replaced. Here it is twice the case that a node with the label X is connected with a node with the label Y. This digram is then stored in a central location and can be referenced via the label X'. Hence, only two nodes remain in the compressed graph (with X' as label) and the graph was reduced to a smaller size. There are some details which make this digram replacement possible, but they are neglected at this point. The interested reader is referred to [Dü16].
Dürksen's algorithm does not seem to be suitable for RDF, because it is based on the fact that there are several nodes with the same label. However, since in RDF a node represents an entity that does not occur twice, Dürksen's basic assumption is not fulfilled in RDF.
Additionally, Dürksen's algorithm is not yet in a mature state, i.e., there is only a rudimentary implementation that does not work reliably for large graphs. Also, work is currently underway to find a compact representation of such a compressed graph in order to save the data with little memory. [Dü16]
For these reasons, the thesis will focus on the compressor GRP, which is presented in Ch. 2.3.2.

2.3.2 GraphRePair

This chapter deals with GRP, the compressor from [MP18]. It firstly presents some foundations and then explains what digrams and digram occurrences are in GRP. Finally, the main routine of the compressor and the encoding of the compressed data are discussed.

Foundations

For a set M, $M^+ = \{x_1 \cdot x_2 \cdot ... \cdot x_n | x_1, x_2, .., x_n \in M\}$ is defined as the set of all non-empty strings of M where $\cdot$ stands for the concatenation of two symbols. $M^* = M^+ \cup \{\epsilon\}$ also includes

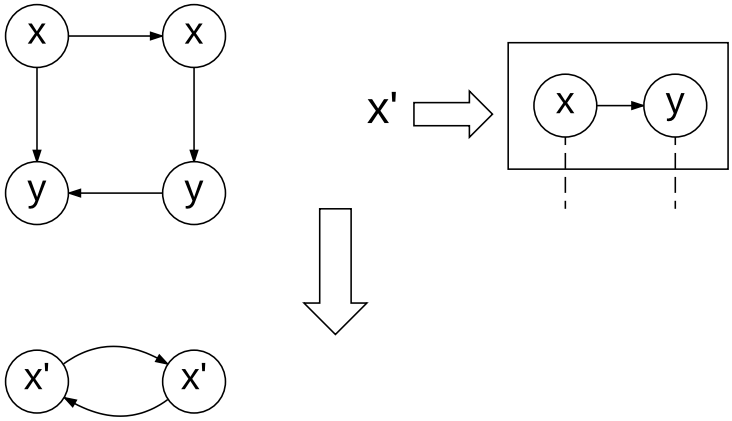


Figure 2.3: Replacement of two occurrences of the digram $X \to Y$ by nodes with the label X'.

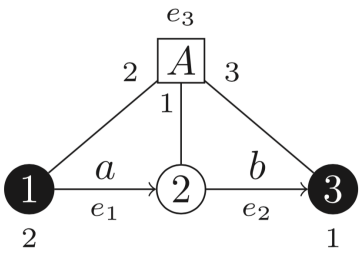


Figure 2.4: A hyper graph as defined in [MP18]. The black nodes 1 and 3 (consider the numbers within the nodes) are external nodes. The numbers below them denote their order within ext. The white node 2 is internal. e_3 (with the label A) is a hyper edge while e_1 and e_2 (labeled with a and b, respectively) are normal edges.

The numbers 2, 1 and 3 around e_3 denote the order of the nodes attached to e_3.

the empty string ϵ.
Let Σ be an alphabet. A hypergraph over Σ is a tuple $g = (V, E, att, lab, ext)$ where $V = \{1, ..., n\}$ is the set of nodes. $E \subseteq \{(i, j)|i, j \in V\}$ is the set of edges. $att : E \to V^+$ is the attachment mapping assigning start and end nodes to all edges. $lab : E \to \Sigma$ is the edge label mapping, and $ext \in V^*$ is a series of external nodes. A hypergraph does not contain multi-edges, which means for two edges $e_1 \neq e_2$ it holds $att(e_1) \neq att(e_2) \vee lab(e_1) \neq lab(e_2)$. For a hypergraph $g = (V, E, att, lab, ext)$ $V_g, E_g, att_g, lab_g, ext_g$ are used to refer to its components.
An example for a hypergraph is illustrated in Fig. 2.4. Formally, the graph can be described as $V = \{1, 2, 3\}$, $E = \{e_1, e_2, e_3\}$, $att = \{e_1 \mapsto 1 \cdot 2, e_2 \mapsto 2 \cdot 3, e_3 \mapsto 2 \cdot 1 \cdot 3\}$, $lab = \{e_1 \mapsto a, e_2 \mapsto b, e_3 \mapsto A\}$ and $ext = 3 \cdot 1$.
External nodes are black and the numbers below them indicate indexes for their position in ext. Analogously, hyper edges (e_3 in this case) have indexes indicating the order of the attached nodes. The utility of external nodes is explained below. [MP18]

Digrams in GRP

A digram d is a hyper graph with $E_d = \{e_1, e_2\}$ so that the following conditions hold:

1. $\forall v \in V_d : v \in att_d(e_1) \vee v \in att_d(e_2)$
2. $\exists v \in V_d : v \in att_d(e_1) \wedge v \in att_d(e_2)$
3. $ext_d \neq \epsilon$

Condition 1 ensures that all nodes in d are incident to one of the two edges of d. Condition 2 is used to make sure that there is one 'middle node' incident to both edges of d. Finally, condition 3 ensures that there are external nodes. External nodes are not a real part of the digram, they represent other nodes of the overall graph which are connected to elements of the digram. [MP18]

Digram Occurrences in GRP

Digram occurrences are concrete instances of a digram. Let g be a hyper graph and d be a digram with the two edges e_1^d, e_2^d. Let $o = \{e_1, e_2\} \subseteq E_g$ and let V_o be the set of nodes incident with edges in o. Then o is an occurrence of d in g if there exits a bijection $b : V_o \to V_d$ so that for $i \in \{1, 2\}$ and $v \in V_o$ all following conditions hold:

1. $b(v) \in att_d(e_i^d)$ iff $v \in att_g(e_i)$
2. $lab_d(e_i^d) = lab_g(e_i)$
3. $b(v) \in ext_d$ iff $v \in att_g(e)$ for some $e \in E_g \setminus o$

Condition 1 and 2 ensure that the two edges of o form a graph isomorphic to d. Condition 3 makes sure that every external node of d is mapped to a node in g that is incident to at least one edge in g that is not contained in o. [MP18]

Example Digram Replacement

In order to better comprehend how GRP is working, an example is shown in Fig. 2.5. On the left, there is an uncompressed graph with two occurrences of the same digram. After the compression, these two occurrences are replaced by edges labeled with A. The compressed graph and the digram can be seen on the right. In the digram, the black nodes are external, i.e., they are still present in the compressed graph. Only the white node and its attached edges (labeled a and b) do not exist anymore. It is important to notice that the node ordering in A is not related with the node ordering in the graph. Every digram is itself a (hyper-)graph and has its own order which is necessary for decompression. Due to direction of the A-edges, it is stored which of the attached nodes corresponds to which external node in A. As an example, the edge from node 1 to 4 is considered. Since 4 is the end node, it is clear that it corresponds to the external node 3 in A, because the external node 3 has the position 2 in *ext* of A.
Also, it has to be noted that the two digram occurrences could not be replaced if node 2 or 6 (in the graph on the left side) were connected to other nodes of the graph.
In general, there are many more digram types which can be seen in [MP18]. The digram shown here is only one specific kind.

Algorithm

Algorithm 1 is the main routine of GRP. The algorithm take a graph as the input and returns a grammar whereas N is the set of non terminals, P is the set of productions and $S \in P$ is the start production. It maintains a list of digram occurrences in g. As long as this list contains multiple occurrences for one digram the loop will continue. In the loop, the most frequent digram is found, then replaced and the grammar is extended. After a digram replacement, the occurrence list has to be updated, because the graph has changed and former digram occurrences may not exist anymore. Moreover, new digram occurrences can be present. This occurrence update is a complex process and will not be discussed here, since it is not relevant for the thesis. [MP18]

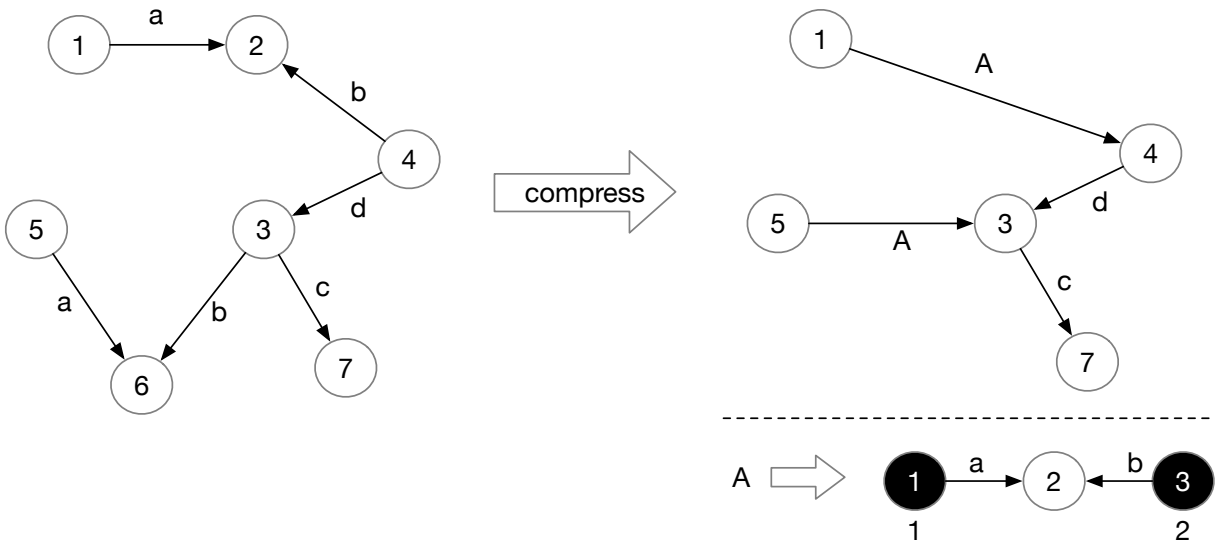


Figure 2.5: Replacement of two occurrences of the digram A. Original graph is on the left. Compressed graph and digram are on the right.

Algorithm 1 GraphRePair (Graph $g = (V, E, att, lab, ext)$)

1: $N, P \leftarrow \emptyset$
2: $S \leftarrow g$
3: $L(d) \leftarrow$ list of non-overlapping occurrences of every digram d appearing in g
4: **while** $|L(d)| > 1$ for at least one digram d **do**
5: $\quad mfd \leftarrow$ most frequent digram
6: $\quad A \leftarrow$ new non terminal for mfd
7: $\quad$ Replace every occurrence of mfd in g
8: $\quad N \leftarrow N \cup \{A\}$
9: $\quad P \leftarrow P \cup \{A \rightarrow mfd\}$
10: $\quad$ Update the occurrence list L
11: **return** Grammar (N, P, S)

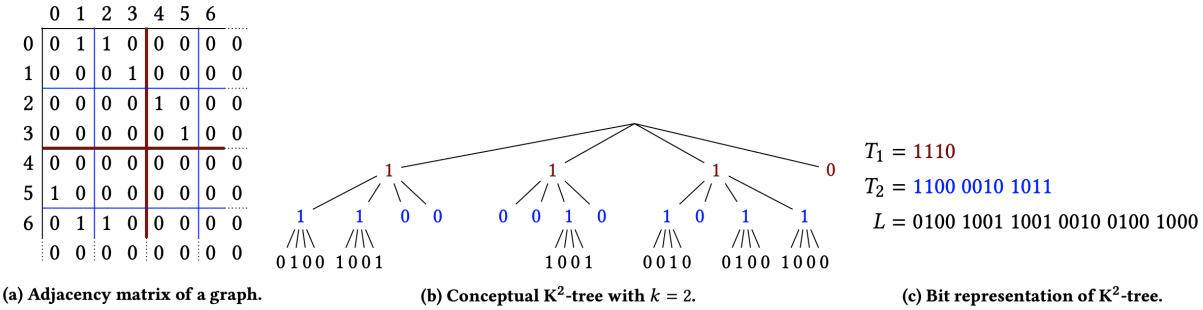


Figure 2.6: An adjacency matrix, its k^2 tree representation and the tree's bit representation.

Grammar Encoding

After GRP has constructed a grammar for some graph, this grammar has to be stored in an efficient way. The start production (the graph) is most often much bigger than the other productions and is therefore encoded differently.

The start production is encoded using k^2 trees which is illustrated in Fig. 2.6. The adjacency matrix of the compressed graph (start rule of the grammar) is considered here. First, that matrix is extended with zeros to the next power of two. Next, it is partitioned into k^2 equally large partitions ($k = 2$ here).

The tree's root represents the whole matrix and its child order corresponds to the order of the just created partitions. If a partition only contains zeros, a zero leaf is added to the tree (e.g. in Fig. 2.6 (a), partition 4, right bottom). Otherwise a one-node is added and the corresponding condition will be partitioned itself. The recursive procedure continues until there is zero-leaf for each path of the tree. Afterwards, the tree is represented by bit-strings which is shown in Fig. 2.6 (c). T_1 and T_2 encode the first and second level and L encodes the level of the leaves.

Since the graph has also edge labels, an adjacency matrix and its corresponding tree will be created for each of those labels. Furthermore, the compressed graph grammar contains hyper edges. An adjacency matrix only displays the nodes connected to the hyper edge, but not the order they are connected to it. Hence, for each hyper edge a permutation is stored, in addition.

That encoding method does not always work well together with the grammar, i.e., a smaller grammar sometimes results in a bigger encoded size of the start rule. If GRP can compress well that results in a high number of non terminals and hyper edges, in most cases. That will increase the storage amount for the start rule. Moreover, the compression potential of the k^2-tree highly depends on the content of the adjacency matrix. That is, a small change in the matrix can deliver a much higher storage amount of the tree.

The remaining productions of the grammar are encoded differently, because they are usually quite small compared to the start production. Here, δ-codes with variable length are used (like in [Eli75]). Essentially, this is a way of displaying objects as a bit-string. Every production starts with the number of edges. For each edge, one bit is used to mark terminal/non-terminal edges, followed by the number of attached nodes. Next, the δ-codes of the list of IDs is mentioned. Finally, a δ-code is used for the edge label. [MP18]

Implementation

GRP has been implemented by the authors of [MP18] in the Scala language [4]. It has to be seen as a proof of concept implementation, i.e., it has quite a high run time for bigger graphs. The

[4] https://www.scala-lang.org/

software can take an RDF file (in N-triples format) as the input and produces three different output files:

Prod: Contains all productions expect the start rule.

Perms: Contains the hyper edge permutations.

Start : Contains the start rule.

All these files are necessary to decompress the graph.[MP18]

3 Approach

This chapter contains the solution approaches to the parts of the thesis. Firstly, a formal model of a compressor is defined. Based on that, the key performance indicators which will be used to measure the performance of the single compressors are introduced. Secondly, the two existing compressors HDT and GRP are compared. Finally, improvements of the compression will be suggested.

3.1 Compressor Models

Here, two compressor models will be introduced, one for general compressors and another one for RDF compressors.

3.1.1 General Compressor Model

A general purpose compressor C can be described as follows: C takes an input *in* and produces an output *out*. That step is called compression. Decompression can be described as taking *out* as an input and producing *in*. An illustration can be seen in Fig. 3.1. The terms CT_C and DCT_C denote the time for compression and decompression, respectively. They will be explained in more details in Ch. 3.2.2.

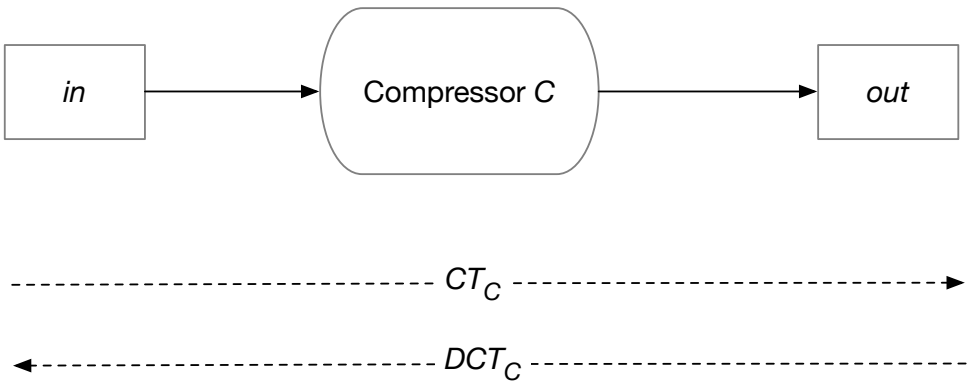


Figure 3.1: Visualization of the General Compressor Model.

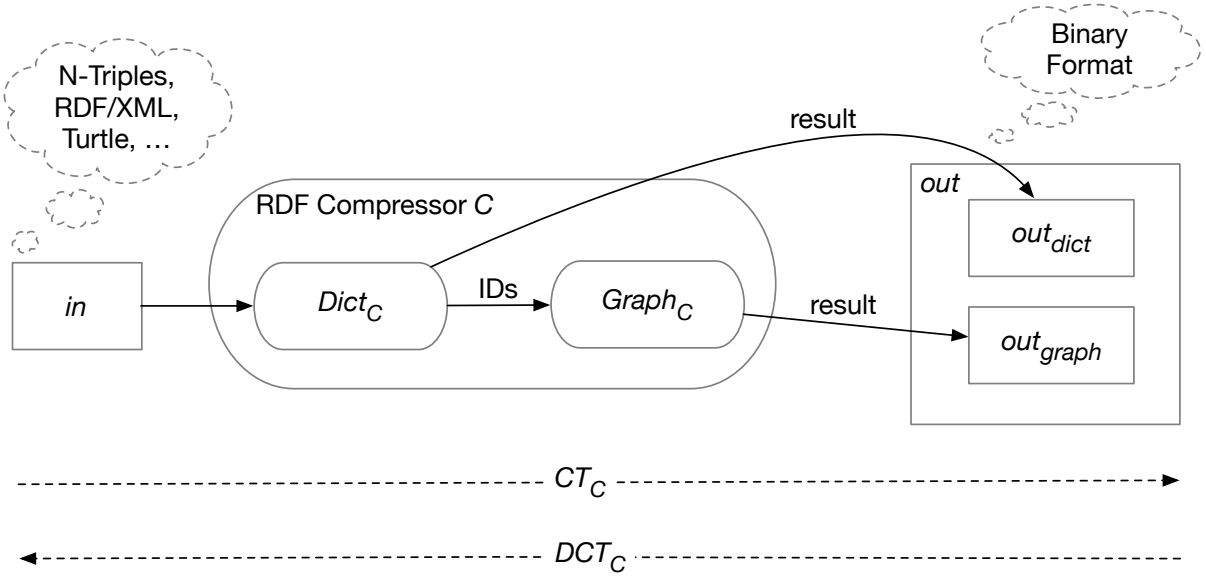


Figure 3.2: Visualization of the RDF Compressor Model.

An example for a general compressor is GZip [1] which will be used in Ch. 5.2.3 as a baseline for GRP and HDT.

3.1.2 RDF Compressor Model

This chapter introduces a formal model for an RDF compressor which can be seen as a sub class or extension of the general model in Ch. 3.1.1. It is illustrated in Fig. 3.2. Here, the compressor C is more complex, as it consists of two components. The first one is called $Dict_C$ and it handles the dictionary compression. That mechanism has already been introduced in the context of HDT (see Ch. 2.2). It assigns an ID to each URI, literal or blank node ID of the graph and can then also further compress the dictionary (like in HDT with prefix trees). The result of $Dict_C$ is out_{dict} which is displayed in the right of Fig. 3.2.
The IDs created by $Dict_C$ are then delivered to the second component, $Graph_C$, which is responsible for compressing the RDF graph. It only uses the IDs from $Dict_C$ and does not know about the URIs, literals and blank node IDs. Its result is called out_{graph} which, together with out_{dict}, forms the complete output out ($out = \{out_{dict}, out_{graph}\}$). out_{graph} and out_{dict} can each be a single file or a set of files.

3.1.3 Existing RDF Compressors

Tab. 3.1 shows an overview of the RDF compressors discussed in this thesis and in which way they fulfill the model from Ch. 3.1.2. As already explained in Ch. 2.2, $Dict_{HDT}$ assigns IDs and compresses the dictionary. In contrast, $Dict_{GRP}$ only assigns IDs and omits the dictionary afterwards. $Dict_{GRP}$ is therefore not a real dictionary compressor. We solve this problem by replacing $Dict_{GRP}$ with $Dict_{HDT}$. This way, comparing HDT and GRP in a fair way is possible. It must also be mentioned that $Graph_{HDT}$ includes the HDT header (see Ch. 2.2). Even if the header is not needed for decompression, its size will be part of out_{graph}. But this is not a problem, because the header's size is so small that it does not add a significant amount to out_{graph}.

[1] https://www.gzip.org/

	$Dict_C$	$Graph_C$
HDT	✓	✓
GRP	only ID assignment (no storage of out_{dict})	✓

Table 3.1: Overview of RDF compressors presented in this thesis.

3.2 Key Performance Indicators

In this chapter, key performance indicators for general and RDF compressors are introduced. Those indicators will be measured in order to determine the overall performance of the different compressors.

3.2.1 Compression Ratio

One of the key performance indicators for a compressor C is its compression ratio. Let m be a single file or a set of files. Then $|m|$ is defined as the size which is measured in bytes. The compression ratio is defined by the following equations:

$$CR_C = \frac{|out|}{|in|} \tag{3.1}$$

$$CR_{Dict_C} = \frac{|out_{dict}|}{|in|} \tag{3.2}$$

$$CR_{Graph_C} = \frac{|out_{graph}|}{|in|} \tag{3.3}$$

Eq. 3.1 defines the compression ratio for the whole output out. Therefore, it is applicable to general and RDF compressors.
In contrast, Eq. 3.2 and 3.3 define the compression ratio only with regard to out_{dict} or out_{graph}, respectively. They are only applicable for RDF compressors, since a general compressor has no distinction between out_{dict} and out_{graph}. In some cases, it is of interest to only consider either the dictionary or graph compression.
Sometimes, CR is used instead of CR_C if it is clear from the context, which compressor C is considered.
As shown in Fig. 3.2, in can have different formats. That has to be taken into account with regard to CR as those formats implicate different input sizes. When CR of two compressors is compared, their input has to have the same format.

3.2.2 (De-)Compression Time

Another key performance indicator of a compressor C is its compression time (CT_C) and decompression time (DCT_C). These metrics also depend on the input data and indicate the run time needed for compression and decompression of the data, respectively. The run time is typically measured in milliseconds. CT_C and DCT_C are also shown in Fig. 3.1 and 3.2. They are defined the same for general compressors and RDF compressors. Furthermore, CT_C and DCT_C are only measured for the whole compressor C, not for $Dict_C$ or $Graph_C$.
Analogously to CR, if is clear from the context which compressor is considered, CT and DCT are used instead of CT_C and DCT_C, respectively.

3.3 GRP vs HDT

In this chapter, the two existing compressors - HDT and GRP - will be compared. Therefore, the features of the compressors and their applicability to certain features of RDF graphs are discussed.
The question is whether there are certain features that an RDF graph can have, and which have a positive or negative impact on the compression ratio of one or both algorithms.
As HDT and GRP use the same method for compressing the dictionary ($Dict_{HDT}$), we only compare $Graph_{HDT}$ and $Graph_{GRP}$ in this chapter.

3.3.1 Star Pattern

According to [SDB16], a star is a graph in which one node is connected to all other nodes. The other nodes are not connected to each other. While a star is an extreme case, there exist graphs with a few nodes having very high degrees. We denote them to be similar to the star pattern.
In order to measure that we define the Star Pattern Similarity (SPS). Let $deg(n)$ be the degree of a node n. Let N be a list of all nodes sorted by their deg-values in descending order. Let N_{top} be the first x nodes of N. Hence, N_{top} contains x nodes with the highest degrees. x should be chosen related to the size of N, we choose $x = 0.001 \times |N|$. Then, SPS is defined as follows:

$$SPS = \frac{\sum_{n \in N_{top}} deg(n)}{\sum_{n \in N} deg(n)} \in [0, 1]$$

The higher the value of SPS is, the more similar the graph is to the star pattern, because then there a few nodes having a high proportion of the sum of all degrees.
The star pattern can be further divided into the hub and authority patterns. As the name suggests, the hub pattern corresponds to a graph in which a few nodes have very high outgoing degrees, the authority pattern is the opposing case (high ingoing degrees). That distinction is needed in the following chapter.

3.3.2 HDT

Next, HDT's behavior with respect to the input is discussed. As HDT's way of compressing the graph structure is fairly simple compared to GRP, there do not seem to be many potential features of the input data that have a high impact on HDT's compression ratio. But one feature is clearly visible: When Fig. 2.1 is considered, it is noticeable that the number of 1's in B_p becomes less if there are a only a few different subjects. This enables HDT to store B_p more efficiently and, thus, $CR_{Graph_{HDT}}$ becomes smaller. Consequently, HDT should perform better if the graph is similar to the hub pattern.

3.3.3 GRP

For GRP, that input feature analysis is more complex. Since GRP constructs grammar rules by using the graph's structure, it can make use of sub graphs that are much more complex than the star pattern HDT is using. There can be many features that can lead to different constructible grammar rules. Those will not be discussed here, as these pattern can become arbitrarily complex, because they can be nested among each other. But one insight is that GRP's compression ratio tends to be bigger when there are more different predicates in the graph. This is true, because GRP's grammar rules are based on repeating patterns and, hence, repeating edge labels as part of these patterns. Let

$$ELR = \frac{\text{number of different edge labels}}{\text{number of edges}}$$

(Edge Label Ratio) be the ratio of the edge labels or properties to the total number of edges of the graph.
A lower value of ELR should increase the likelihood of a lower compression ratio for GRP. However, if the graph's structure becomes unfavorable for GRP, the compression ratio may still be worse at a lower value for ELR.
In [MP18], the authors mention that a graph similar to the star pattern is beneficial for GRP, because GRP can create many digrams around those nodes with high degrees whereby, in contrast to HDT, GRP does not rely on the hub pattern.

3.4 Compression Improvements

First comparisons of $Graph_{HDT}$ and $Graph_{GRP}$ (see Ch. 5.1) showed that $Graph_{GRP}$ achieves a better compression ratio in many cases. Since this thesis' topic is grammar-based compression, Ch. 3.4.1 will focus on improving $Graph_{GRP}$. But Ch. 3.4.2 is about the improvement of $Dict_{HDT}$ which is used by both HDT and GRP.

3.4.1 Ontology Knowledge

As already discussed in Ch. 2.1.1, an ontology contains meta data about an RDF graph.
This chapter will investigate whether it is possible to change the structure of an RDF graph by applying knowledge from its ontology so that it is better compressible for GRP, but at the same time remains semantically equivalent to the original graph. In this way, no data would be lost after the compression.
In Ch. 3.3.3, it has already been mentioned that GRP makes use of much more complex sub structures than HDT. It will therefore be interesting to see how applying ontology knowledge influences GRP's compression ratio.
In the following chapters, some concepts of OWL are introduced and it is analyzed how they can be used in order to produce a graph structure which is favorable for GRP.

Symmetric Properties

There is a class in [owl] called `owl:SymmetricProperty` [2] which expresses that a certain property p is symmetric. This means, if there is a triple (s, p, o) in the graph, then there can also be a triple (o, p, s) at the same time. In reality, however, it can happen that only one of the two triples is explicitly mentioned and the second triple is only implicitly present. The idea is to always add the other triple to the graph in such a case. This makes the graph larger at first, but more grammar rules can be found. This is because ELR can be reduced by the adding, which can lead to a better compression ratio. Furthermore, digram occurrences like in Fig. 3.3 can be constructed after the edges are added. In this example, p is the only symmetric property. Hence, the green edges were added to the graph on the left. This leads to a graph in which two occurrences of the digram A can be found. The compressed graph is shown on the right side. The A-edges are indirected, because the digram A contains only one external node. Therefore, it is clear that the nodes 3 and 6 each correspond to that external node, respectively. It can be seen that adding symmetric edges can produce digram occurrences which contain the two

[2] The prefix `owl:` is used for every entity or property in the context of OWL.

corresponding symmetric edges. It is likely that GRP finds many of those digrams after a huge amount of symmetric edges was added to a graph.

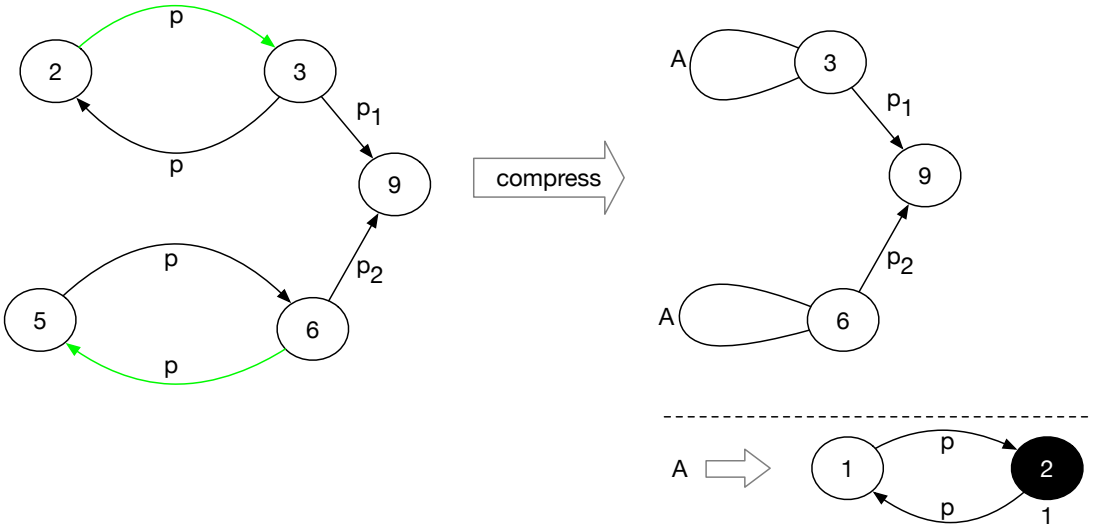
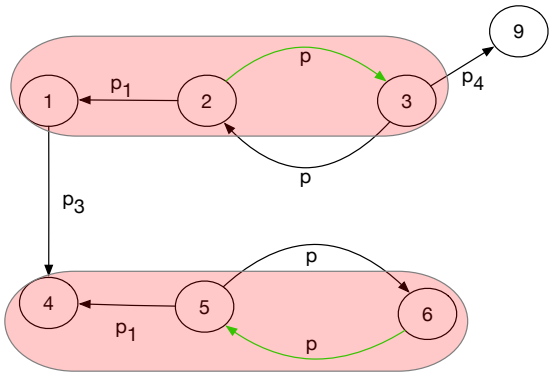
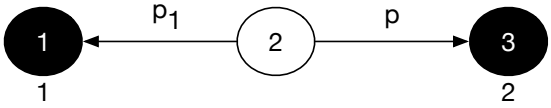


Figure 3.3: A sub graph with the symmetric predicate p (on the left side). The green edges have been added. Afterwards, two occurrences of the digram A can be found. The compressed graph is shown on the right side.



(a) A sub graph to which the green edges were added. Two occurrences of the digram in Fig. 3.4b are marked by red ovals.



(b) The digram that can be found twice in the graph of Fig. 3.4a after the green edges were added.

Figure 3.4: Visualization of the benefits of adding symmetric edges to the graph. p is symmetric, all other properties are not.

In Ch. 2.3.2, it was shown that more edges can also lead to a worse compression ratio, since internal nodes of a digram are not allowed to be connected to other nodes in the graph. So, it could be argued that adding symmetric edges makes GRP find less digram occurrences. But this is not the case, since the nodes, which the symmetric edges are added to, were already connected before. An example supporting this argument can be seen in Fig. 3.4. In Fig. 3.4a, a graph is shown whereby p is the only symmetric property. Hence, the green edges were added to the graph. Due to the addition, the digram of Fig. 3.4b can be found twice, whereas it was previously found only once. So, the addition delivers a better compression potential here. At the same time, the degrees of the nodes 2, 3, 5 and 6 are increased by one. But this should not decrease the probability of finding other digrams in the graph, since those nodes were already

connected before.
The replacement of the digram in Fig. 3.4b is not illustrated here, as that digram type was already explained in Ch. 2.3.2.

Inverse Properties

One property of [owl] is `owl:inverseOf`, which is defined for two properties p_1, p_2. If (s, p_1, o) exists then (o, p_2, s) should also exist and vice versa. Analogously to `owl:SymmetricProperty`, it can be the case that only one of the two triples is explicitly mentioned. It is reasonable to argue in a similar way for adding those edges to the graph here. It can decrease ELR if there are many occurrences of a few inverse properties. Also, the added edges can directly produce a high number of digram occurrences which the two edges are part of (similar to Fig. 3.3). Furthermore, it will not create an unfavorable graph structure, but enable GRP to find other digram occurrences (similar to Fig. 3.4).

Transitive Properties

In [owl], a predicate can be denoted as transitive (`owl:TransitiveProperty`). Let p be transitive. If the triples $(1, p, 2), (2, p, 3)$ exist then $(1, p, 3)$ should also exist. Consequently, this holds also for an arbitrarily long path from 1 to n, as illustrated in Fig. 3.5. Such a path, with the length of at least two, is called *transitive path* from now on. If there exists a *transitive path* between two nodes i and j, then the edge (i, j) is called *direct transitive path.* So, in Fig. 3.5, the edges $(1, 3)$ and $(1, n)$ are *direct transitive paths.*

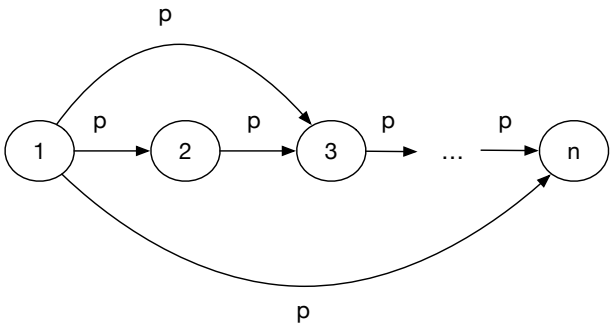
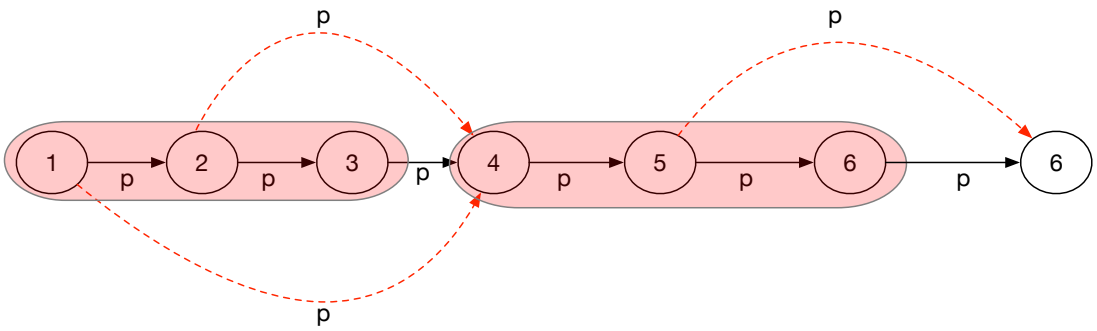


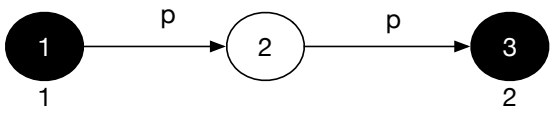
Figure 3.5: A sub graph with the transitive predicate p.

The shown graph shall again be seen as some sub graph. The approach is to remove all *direct transitive paths.* This is reasonable, as it gives the nodes i and j a lower degree, and therefore GRP can have a higher chance to find other digrams in which those nodes are involved. An example is shown in Fig. 3.6a where the red dashed edges are removed. After that, two occurrences of the digram in Fig. 3.6b can be found. Before the removing those occurrences were not present, since the nodes 2 and 5 were connected to other nodes which is not allowed.

The opposing approach is to add all *direct transitive paths.* But a strong argument against this approach is that it would dramatically increase the number of edges, because for each pair (i, j) (with $i, j \in \{1, ..., n\}$ and distance between i and j greater than 1), an edge (i, j) would be added. After increasing the graph's size so much it is unlikely that out_{graph} becomes smaller even if more digrams could be found for some reason.



(a) A sub graph from which the red dashed edges were removed. Two occurrences of the digram in Fig. 3.6b are marked by red ovals.



(b) The digram that can be found twice in the graph of Fig. 3.6a after the red dashed edges were removed.

Figure 3.6: Visualization of the benefits of removing *direct transitive paths* from the graph. p is transitive.

Equal Properties

Further properties of [owl] are `owl:equivalentProperty` and `owl:sameAs`. The first one denotes that two properties are equivalent, but that does not mean that they are equal. In contrast, the latter one is expressing equality. If there is a property p which is equal to other properties $p_1, .., p_n$, then one approach could be to replace each occurrence of $p_1, .., p_n$ with p. This would reduce ELR and, at the same time, not change the structure of the graph. However, this approach was not implemented, since the thesis focuses on compressing single RDF graphs and `owl:sameAs` typically connects multiple graphs with each other. Compressing multiple graphs would lead to more complexity, because not only properties can be the same, but single nodes can be marked as the same as well.

3.4.2 Dictionary Improvements

According to [FMPG+13], the dictionary (out_{dict}) makes up most of the memory of the complete output *out*. First results of Ch. 5.1 also show that. It is therefore worth investigating whether the dictionary can be compressed better. It can be taken advantage of certain features of the dictionary to achieve that. In order to do that we use $Dict_{HDT}$ as the basis and further improve it.

Literals

Objects in RDF can be literals. Literals typically contain constant values and usually have no common prefixes. Therefore, the compression of $Dict_{HDT}$ is not suitable for these. It would be possible to use different compression techniques for different types of data values (integer, double, string, etc.). The thesis will focus on compressing strings.
Since those strings can contain whole flow texts, a text compression would probably be well applicable. An example of such a text compression is a Huffman Code [Sha10]. Here, every

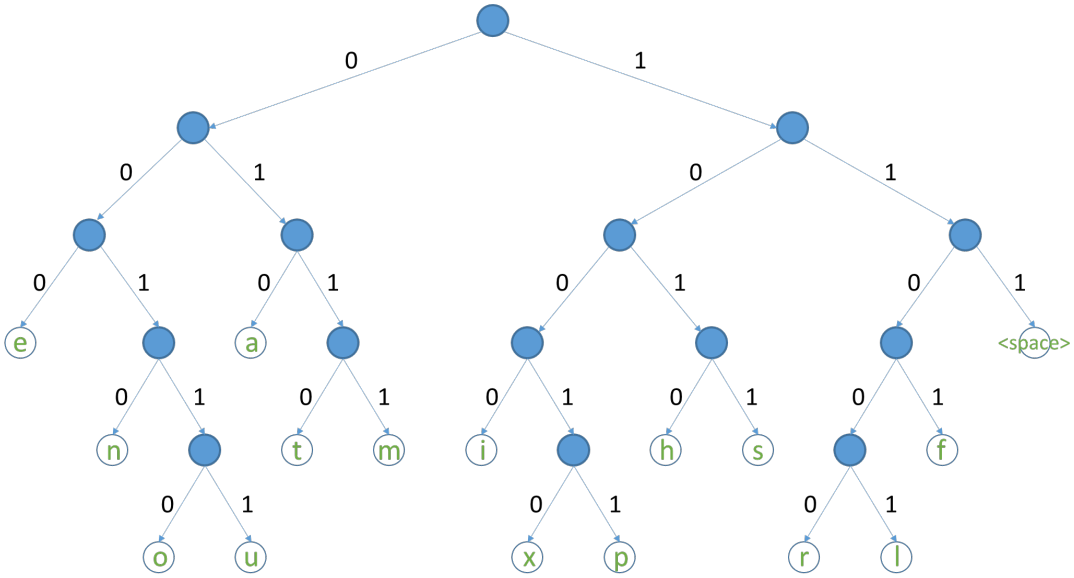


Figure 3.7: An example of a Huffman Tree.

single character of a text is binary coded, whereby frequently occurring characters get short and rare characters get longer codes. These codes are expressed by a binary Huffman tree. An example can be seen in Fig. 3.7. Each leaf contains a symbol whereas the ones and zeros on the path to the symbol define its code. The tree is constructed in such a way that paths to frequent characters are shorter than those to rare characters. The whole procedure can be seen in [Sha10].

Blank Nodes

As already mentioned in Ch. 2.1, every blank node gets an ID. These IDs are usually chosen arbitrarily and have no meaning beyond that. When reading an RDF graph with the Jena-API [3] (which is used by HDT according to [FMPG+13]) random strings are assigned to the blank nodes, which are quite long. They also have no common prefixes, which makes $Dict_{HDT}$ ineffective here.
To improve the compression, the IDs of the blank nodes could be reassigned. For example, numbers from 1 to n (n = number of blank nodes) can be used to get short IDs.
Another possibility is not to save the IDs of the blank nodes at all. In HDT all strings in the dictionary (including the blank node IDs) are mapped to short IDs. Thus, the blank node IDs are in principle already stored. They can therefore be removed from the dictionary.

[3] https://jena.apache.org/index.html

4 Implementation

This chapter will explain how the different approaches of Ch. 3 have been implemented. First, the comparison of HDT and GRP is discussed. Second, improvements for both graph and dictionary compression are presented.

4.1 GRP vs HDT

In this chapter, it will be explained how the comparison between HDT and GRP is implemented. As has just explained, $Graph_{HDT}$ can compress a graph that is similar to a hub pattern (few subjects, many objects) very well. So, $CR_{Graph_{HDT}}$ gets higher the further away the graph is from this pattern. The most unfavorable graph structure for $Graph_{HDT}$ is therefore the authority pattern (see Ch. 3.3.1).
The task is to create a series of RDF graphs $(G_1, ..., G_m)$ that first correspond to the hub pattern and then continue to change in the direction of the authority pattern. That can be done using Alg. 2. The first input parameter *steps* influences how many graphs will be created, as it defines by which amount the number of subjects (*subj*) is increased in every iteration of the while loop. It has to chosen low enough to ensure a smooth transition from the hub to authority pattern. The second parameter *nodesFactor* influences the amount of nodes of the graphs. Concrete values for the parameter are not important to mention.

Algorithm 2 BuildGraphs (*steps*, *nodesFactor*)

```
1: n ← nodesFactor * steps // n is the number of nodes
2: subj ← 1
3: i ← 1
4: while subj < n do
5:     obj ← n − subj
6:     G_i ← build graph with subj subjects and obj objects
7:     subj ← subj + steps
8:     i ← i + 1
9: return (G_1, ..., G_m)
```

An example output of Alg. 2 is shown in Fig. 4.1. All graphs (G_1 to G_m) have the same number of nodes and edges. That is ensured by Line 6 in Alg. 2. G_1 has only one subject connected to all objects. G_2 then has two subjects more and correspondingly 2 objects less. This continues

until there is only one object that is connected to all subjects (G_m). The edges are randomly distributed among the nodes, so that all nodes have a similar degree and each node has at least a degree of one.

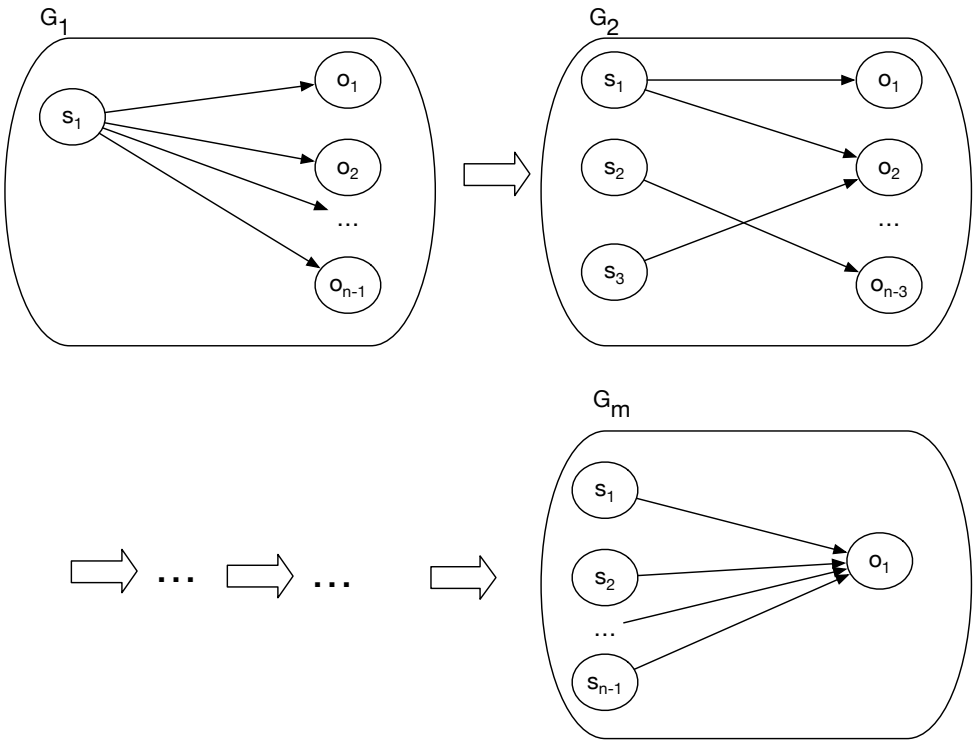


Figure 4.1: Example output of Alg. 2: Step-by-step transition from hub pattern to authority pattern. The number of nodes n is the same for each graph. The number of edges is also the same for each graph.

It is also ensured that each of the generated files has exactly the same size. This is made possible by ensuring that each URI has the same length and that every RDF graph also has the same number of triples. This same size is important for the evaluation, since only the graph's structure is of interest here. An unequal size of the RDF files would lead to undesired effects in terms of CR_{Graph_C}.

A section of such a file (for G_1) is shown in Fig. 4.2. In that example, there is only one distinct predicate for all triples. This is because, we first want to focus on the effect of the hub and authority patterns. The impact of an increasing number of predicates will be evaluated afterwards.

```
<http://subject/0000000000> <http://predicate/0000000000> <http://object/0000001036> .
<http://subject/0000000000> <http://predicate/0000000000> <http://object/0000000854> .
<http://subject/0000000000> <http://predicate/0000000000> <http://object/0000000991> .
<http://subject/0000000000> <http://predicate/0000000000> <http://object/0000000450> .
<http://subject/0000000000> <http://predicate/0000000000> <http://object/0000000456> .
<http://subject/0000000000> <http://predicate/0000000000> <http://object/0000000689> .
<http://subject/0000000000> <http://predicate/0000000000> <http://object/0000001166> .
<http://subject/0000000000> <http://predicate/0000000000> <http://object/0000000960> .
```

Figure 4.2: Excerpt from the generated RDF file for G_1 (see Fig. 4.1). Each triple has the same length.

4.2 Compression Improvements

This chapter introduces the implementation details of Ch. 3.4. First, applying ontology knowledge in order to achieve a better compression ratio is discussed. Afterwards, implementation details of the dictionary compression improvements are presented.

4.2.1 Datasets

Here, an overview of the different datasets, that have been used for the evaluation, is given. Although the datasets are used in Ch. 5, they are presented here, since some implementation details are dependent on the data.

Semantic Web Dog Food

Semantic Web Dog Food [1] is a collection of RDF files from the Semantic Web community. It contains data about papers, people, organizations, and events of the community. Open source tools are provided for users to contribute data to the project. This way, the dataset can continue to grow. [NGPG16]

DBPedia

DBPedia [2] is an RDF version of the knowledge from Wikipedia. According to [LIJ+15], it extracts data in 111 different languages. The Wikipedia info boxes are mapped to one single ontology. That is done by crowd sourcing effort. DBPedia is one of the most used datasets in the Semantic Web. It contains many different data files of a quite big size, with some of them having hundreds of millions of triples. [LIJ+15]

Wordnet

Wordnet [3] contains knowledge about the English language. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms which are called synsets, each expressing a distinct concept. Those sets are the nodes in an RDF graph and relations between them are edges/properties. It also has an ontology which is useful for Ch. 4.2.2. [MBF+90]

Nuts-RDF

The NUTS (Nomenclature of Territorial Units for Statistics) [4] contains the NUTS regions along with geographic information. It contains blank nodes and is therefore useful for the evaluation of the improvements regarding blank node IDs.

Government Open Data

This is RDF data [5] published by the Ministry of Housing, Communities Local Government of the United Kingdom.

[1] http://www.scholarlydata.org/dumps/
[2] https://wiki.dbpedia.org/Downloads2015-04
[3] http://wordnet-rdf.princeton.edu/about
[4] http://nuts.geovocab.org/
[5] http://opendatacommunities.org/home

4.2.2 Ontology Knowledge

This chapter is about how to manipulate the RDF graphs to match the features of Ch. 3.4.1. For this, the query language SPARQL [spa] is used.

Gathering Relevant Properties

DBPedia and Wordnet are used here, since both of them have ontologies. First, all relevant (symmetric, inverse, transitive) properties must be determined. This information should normally be directly contained in the ontology of the data, by triples of the form:

```
(myProperty, rdf:type, owl:SymmetricProperty)
```

This triple states that `myProperty` is symmetric. The concept is analogous for inverse or transitive properties.
However, this is not the case with DBPedia. In their ontology, such triples do not occur for the symmetric or inverse cases (only for transitive properties). Symmetric and inverse properties have to be determined in a different way. In DBPedia's ontology the equivalent properties of Wikidata [6] are given. It is possible to check in the Wikidata ontology whether the properties are symmetric or inverse, because there the information is given. Thus, it can be back-traced which DBPedia properties are relevant.
In Wordnet, transitive properties are explicitly given, but symmetric and inverse are not. It is necessary to determine them by understanding the meaning of Wordnet's relations. Properties connect different "synsets". One of those properties is `antonym`, which is like an opposite-relation between words. Therefore, the property can be seen as symmetric. The properties `hypernym` and `hyponym` are inverse. However, this only holds for nouns, not for verbs. Hence, a sub graph of Wordnet, which only contains nouns, has to be created in order to use those inverse properties.
After all relevant properties have been gathered, the manipulations (as suggested in Ch. 3.4.1) can be executed. The following chapters show how that can be achieved.

Symmetric Properties

In order to remove or add symmetric properties SPARQL can be used. The code of listing 4.1 will be used to do add symmetric triples.

```
INSERT {?o ?p ?s}
WHERE{
        {?s ?p ?o}
        MINUS {?o ?p ?s}
}
```

Listing 4.1: SPARQL update for adding triples with the symmetric property p.

That update has to be executed for each symmetric property p. If it is desired to remove the second triple, a delete update would have to be executed. That delete is shown in Listing 4.2. Here, it has to be taken care not to delete both directions, this can be done by using a filter.

[6] https://www.wikidata.org/wiki/Wikidata:Main_Page

```
DELETE {?o ?p ?s}
WHERE{
?s ?p ?o .
FILTER (EXISTS {?o ?p ?s } && (str(?s) > str(?o) )
}
```

Listing 4.2: SPARQL update for removing triples with the symmetric property p.

Inverse Properties

Assume that $p1$ and $p2$ are inverse properties. Listing 4.3 is used to add the triple $(o, p1, s)$ if $(s, p2, o)$ already exists. That update has to be made in both directions

$$(p1, p2) \text{ and } (p2, p1)$$

in case the other direction exists. If the triples should be removed instead of adding them, a delete has to be performed which is shown in Listing 4.4

```
INSERT {?o ?p1 ?s}
WHERE{
        {?s ?p2 ?o}
        MINUS {?o ?p1 ?s}
}
```

Listing 4.3: SPARQL update for adding triples with the inverse properties p1 and p2.

```
DELETE {?o ?p1 ?s}
WHERE{
?s ?p2 ?o .
FILTER (EXISTS { ?o ?p ?s })
}
```

Listing 4.4: SPARQL update for removing triples with the inverse properties p1 and p2.

Transitive Properties

Here, the triple (s, p, o) will be removed if there exists a path from s to o via the transitive predicate p of a length of at least two edges. That can be achieved by Listing 4.5 in which a property path is used in the first line of the where clause.
Of course, it is also possible to add the triple (s, p, o) if it does not exist. That can be done similarly with an insert and is shown in Listing 4.6.

Sub Graphs

Real datasets are quite big and it can happen that there are only a few relevant properties (symmetric/inverse/transitive). Even if there are many of those properties it can be that they do not occur often in the data. In that case, the effect of manipulating the data cannot have a big impact on the compression ratio. However, it is still desired to investigate if the effect

```
DELETE { ?s ?p ?o }
WHERE {
        ?s ?p/?p+ ?o.
        ?s ?p ?o
}
```

Listing 4.5: SPARQL update for removing triples with the transitive property p.

```
INSERT { ?s ?p ?o }
WHERE {
?s ?p/?p+ ?o.
FILTER (NOT EXISTS {?s ?p ?o })
}
```

Listing 4.6: SPARQL update for adding triples with the transitive property p.

is possibly there. Therefore, it is necessary to form a smaller graph in which those relevant properties occur often. So, a procedure to build a sub graph is needed.

First, all triples $t_1, ..., t_n$ are collected which contain one of the relevant properties. It would be possible to randomly add a number of remaining triples in order to get a more diversified graph. But that would result in a graph far away from the original and would probably not contain structural patterns from the original one.

Therefore, we choose to add triples that are directly connect to the subjects or objects of $t_1, ..., t_n$. By doing that, a real sub graph is extracted out of the original graph.

The procedure is illustrated in Fig. 4.3. There, p is the only relevant property. Consequently, the green triples are collected in the first step. The red triples are collected in the second step, because they are connected to triples from the first step.

The results of Ch. 5.1 have also shown that GRP has a quite high run time even for smaller graphs. It was also mentioned in [MP18] that the rudimentary implementation (see Ch. 2.3.2) cannot handle very large graphs. However, we still want to evaluate GRP for real datasets which are often very large. To solve this problem, a realistic sub graph of the big graphs has to be extracted. As mentioned earlier, choosing random triples does not fulfill that. Hence, we choose an approach similar to the one shown in Fig. 4.3. But here we start with some entity e (not

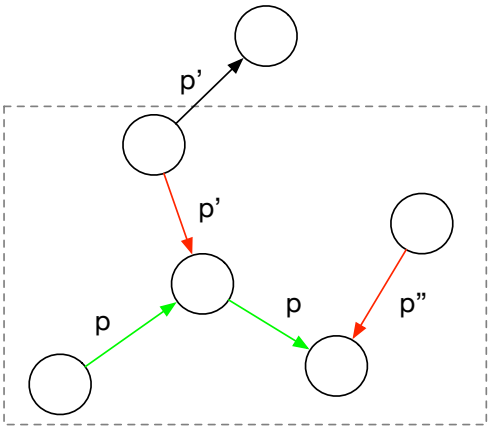


Figure 4.3: Extraction of a sub graph (marked by dashed line).

with a set of desired properties). Analogously to the other approach, we gather all triples e is part of. This gives us new entities $e_1, ..., e_n$. Then, triples are added which $e_1, ..., e_n$ are part of. That routine is repeated a few times until some defined number of triples has been reached.

4.2.3 Dictionary Improvements

For the dictionary improvements, HDT's dictionary compression ($Dict_{HDT}$) is used as a basis. Therefore, the HDT-Java code [7] has been extended.

Literals

As already mentioned in Ch. 3.4.2, literals will be compressed using a Huffman code. To achieve that, HDT is changed such that it does not compress literals by prefix trees, but rather gives the literals to the newly created `HuffmanHandler`, which compresses all literals and finally stores them in a binary format. In order to make that possible, the `HuffmanHandler` has to traverse all literals in the beginning of the compression to establish a Huffman Tree.
In addition, the Huffman code/tree itself has to be stored in order to be able to decompress the data. There is no standard way for storing the binary tree. One approach can be seen in Alg. 3 which has to be started at the root node. That method creates an unambiguous bit representation of the tree. It traverses the tree in a depth-first-search-manner and writes a one if the current node is a leaf. Otherwise, it writes a zero and the procedure is then called recursively for the current node's children. This encoding is unambiguous, because each node is either a leaf or has exactly two children.

Algorithm 3 EncodeNode (TreeNode node)

```
1: if node is leaf then
2:     writeBit(1)
3:     writeCharacter(node.character)
4: else
5:     writeBit(0)
6:     EncodeNode(node.leftChild)
7:     EncodeNode(node.rightChild)
```

Alternatively, there are pre-computed Huffman trees for natural languages such as English. There, it has already been investigated which letter occurs how often in English texts and in this way a generally valid Huffman code has been established. The advantage is that the Huffman tree does not have to be stored and it does not have to be calculated, which saves runtime. The disadvantage, however, is that the tree is not optimal for the text to be compressed, as it is more general. Another problem in our case is that the literals contain a lot of special characters that are not taken into account in prefabricated Huffman codes. Therefore, prefabricated codes will not be used.

Blank Nodes

HDT normally uses the arbitrary and long strings generated by the Jena API and tries to compress them using prefix trees.
Now, two approaches, which have both been implemented, for improving the compression of blank nodes are presented.

[7] https://github.com/rdfhdt/hdt-java

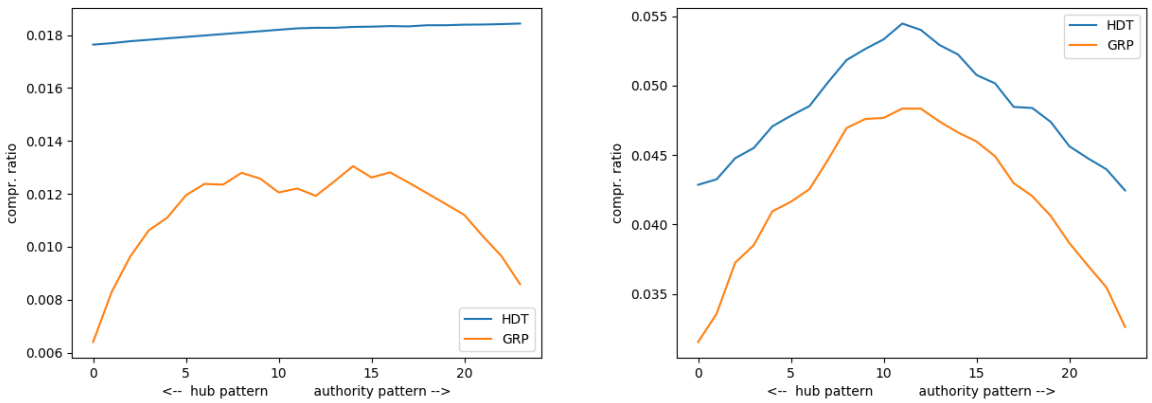
The first approach is to use shorter IDs (e.g. numbers from 1 to n). Then, during the run time a mapping from old ID to new ID is maintained by the new class `BlankNodeHandler` in order to make sure that the same blank node will get the same new ID if it occurs multiple times. That mapping does not have to be stored persistently and will therefore be omitted once the compression is finished.
The second approach is to omit blank node IDs completely. The HDT code is changed in such a way that skips blank nodes in the process of storing the dictionary. HDT must then be changed so that it can handle the case in which it does not find a corresponding string in the dictionary for a certain short ID. At this point, it would know that the considered node is a blank node and the longer blank node ID is unimportant. Such a situation will occur when a decompression is performed. That mechanism has not been implemented, because this thesis is only interested in potential compression ratio improvements.

5 Evaluation

This chapter presents the evaluation of the several approaches presented in Ch. 3 and 4. First, the performances of GRP and HDT are compared. Second, the proposed compression improvements are evaluated.
For the following experiments, HDT-Java 2.0 [1] (the currently newest version) has been used. For GRP the implementation presented in Ch. 2.3.2 has been used. The evaluated datasets are the ones introduced in Ch. 4.2.1.

5.1 GRP vs HDT

This chapter shows the evaluation results of the comparison between HDT and GRP. First, results for the synthetically generated graphs from Ch. 4.1 will be presented. Afterwards, the two compressors will also be evaluated for real world data.



(a) $CR_{Graph_{HDT}}$ and $CR_{Graph_{GRP}}$.

(b) CR_{HDT} and CR_{GRP}.

Figure 5.1: The compression ratios for GRP and HDT without and with dictionary sizes.

[1] https://github.com/rdfhdt/hdt-java/releases/tag/v2.0

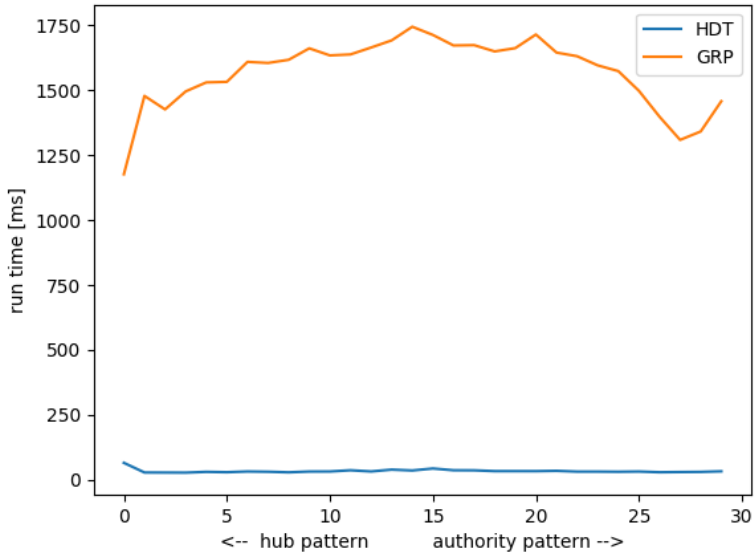


Figure 5.2: CT_{HDT} and CT_{GRP} (Average run time of 100 consecutive executions).

5.1.1 Results for Synthetic Data

Fig. 5.1 shows the results for the synthetically generated data (see Ch. 4.1). Fig. 5.1a shows CR_{Graph_C} (without the size of the dictionary). As expected, $CR_{Graph_{HDT}}$ becomes higher, as the graphs are further away from the hub pattern. Moreover, $Graph_{GRP}$ compresses better if the graph is similar to the star pattern (greater value of SPS), as stated earlier. It is noticeable that $CR_{Graph_{GRP}}$ fluctuates much more than $CR_{Graph_{HDT}}$ and the impact of the star pattern similarity is much higher for $Graph_{GRP}$.
Fig. 5.1b shows the values of CR_{HDT} and CR_{GRP}. It can be seen that the curves behave differently than in Fig. 5.1a, since the dictionary size $|out_{dict}|$ is much bigger than $|out_{graph}|$. As both GRP and HDT use the same dictionary compressor ($Dict_{HDT}$), they have been increased by the same amount and therefore CR_{GRP} is smaller than CR_{HDT}. Furthermore, it is noticeable that $|out_{dict}|$ is bigger when the graph is further away from the star pattern. This is due to the fact that the number of different URIs becomes higher at a lower value of SPS, because of the way these graphs are constructed (see Fig. 4.2).
Apart from the compression ratio, the run time is also important for the overall performance. Fig. 5.2 shows the average value of CT for both compressors, respectively. Here, the same scenario with the star pattern (and only one distinct predicate) was used. It has been executed 100 times to get a sophisticated run time measurement, because CT depends on the current CPU workload of the computer. Decompression is not supported by the currently implemented version of GRP and will therefore be omitted. It can be seen that CT_{GRP} is on average 48 times higher than CT_{HDT}. However, it should also be noted that the implementation of GRP is rather rudimentary (according to the authors of [MP18]), while that of HDT has been under development for some time. So, they are not on the same level in terms of quality. In theory, GRP should be able to compete with HDT, since it has linear run time with respect to the size of the graph [MP18]. In [FMPG+13], the authors do not mention HDT's run time in $\mathcal{O}$-notation.
In Ch. 4.1, it was mentioned that only one distinct predicate will be used in the first comparison. However, in the following it is shown how the compressors behave as the number of predicates increases. To realize that, the whole procedure from Ch. 4.1 is executed four times with an

increasing number of predicates each time. The plots for those executions are shown in Fig. 5.3. For both algorithms, the CR_{Graph_C}-values get bigger as the number of predicates increases. The curves with same line styles have to be compared, since those pairs have the same number of predicates. The fourth execution is marked by the solid red lines and this is the first case in which $CR_{Graph_{HDT}} < CR_{Graph_{GRP}}$ is true. So, there are cases in which $Graph_{HDT}$ compresses better. That continues, as the number of predicates is further increased. Hence, higher amounts of predicates have a higher impact on $Graph_{GRP}$ than on $Graph_{HDT}$. However, the next chapter will show that there is no fixed value for ELR at which $Graph_{HDT}$ is always better.

Iteration	Line Style
1	··
2	– –
3	▽▽
4	—

Table 5.1: Legend for the line styles in Fig. 5.3. A higher iteration number indicates a higher number of predicates.

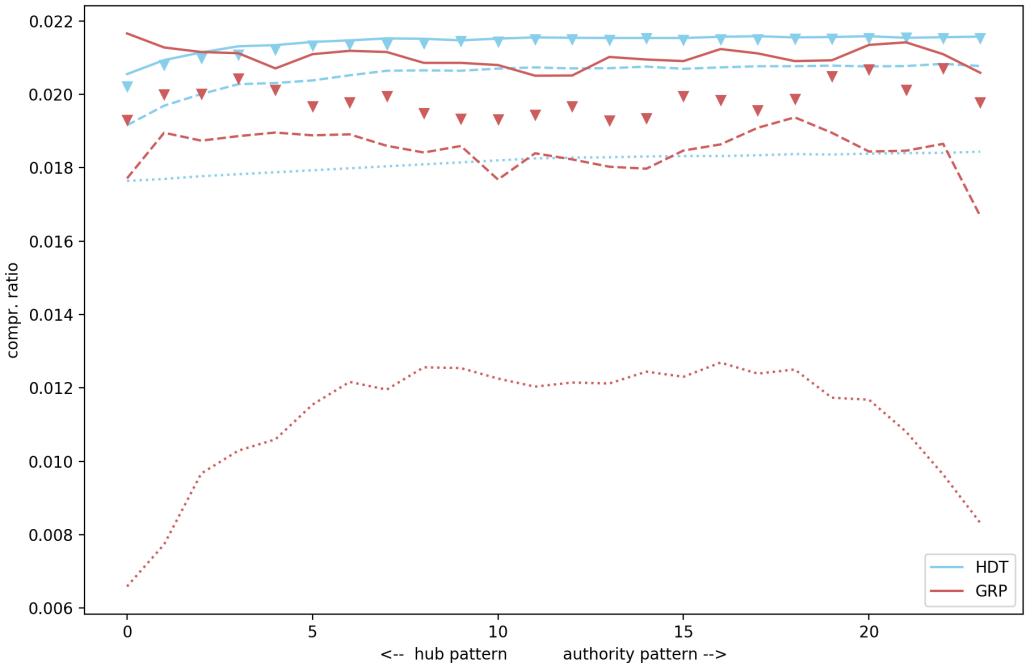


Figure 5.3: CR_{Graph_C}-values for HDT and GRP. Same line styles indicate that the number of distinct predicates is the same (legend is shown in Tab. 5.1). The two solid lines (iteration 4) show the first case in which $CR_{Graph_{HDT}} < CR_{Graph_{GRP}}$ holds.

5.1.2 Results for Real World Data

The previous chapter used synthetic data to detect the behavior of the compressors to certain changes in the structure of the input data. Next, real data will be used to compare the compression ratios of $Graph_{HDT}$ and $Graph_{GRP}$.

Dataset Overview

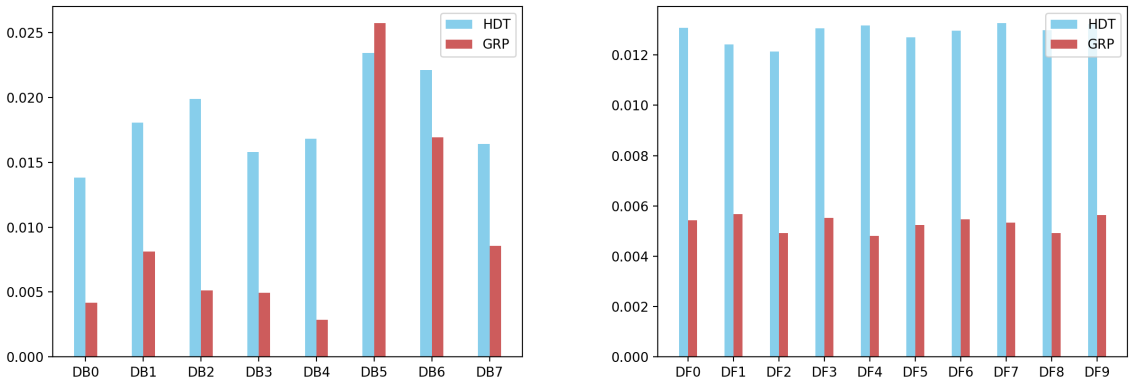
Here, an overview of the used RDF files is provided. They are contained in the following datasets: DBPedia, Semantic Web Dog Food, WikiData and Open Government Data. Tab. 5.2 shows the files that are used, as well as their number of triples and resources. Also, their values for ELR and SPS are shown. It has to be noted that the entity-based sub graphs are limited to 50000 triples.

Short Name	Long Name	#Triples	#Resources	ELR	SPS
DF0	dc-2010-complete-alignments	5919	821	0.008	0.000043
DF1	ekaw-2012-compl-alignments	13114	1604	0.004	0.052
DF2	eswc-2006-compl-alignments	6654	1259	0.004	0.071
DF3	eswc-2009-compl-alignments	9462	1247	0.004	0.054
DF4	eswc-2010-compl-alignments	18122	2226	0.002	0.091
DF5	eswc-2011-compl-alignments	25865	3071	0.002	0.114
DF6	iswc-2002-compl-alignments	13450	1953	0.003	0.057
DF7	iswc-2003-compl-alignments	18039	2565	0.002	0.102
DF8	iswc-2005-compl-alignments	28149	3877	0.001	0.132
DF9	iswc-2010-compl-alignments	32022	3842	0.001	0.116
DB0	external-links_en	50000	54931	2.0E-5	0.038
DB1	geo-coordinates_en	50000	12505	8.0E-5	0.2
DB2	homepages_en	50000	98584	2.0E-5	0.006
DB3	instance-types-transitive_en	50000	7722	2.0E-5	0.229
DB4	instance-types_en	50000	35677	2.0E-5	0.387
DB5	mappingbased-properties_en	50000	21590	0.01502	0.028
DB6	persondata_en	50000	10631	1.8E-4	0.112
DB7	transitive-redirects_en	50000	82318	2.0E-5	0.022
OD0	00f1dbeb-4ba1-4dac-92ec	50000	6094	4.39E-4	0.294
OD1	4eb92c99-76fe-4987-a475	50000	6613	1.99E-4	0.23
OD2	95414599-bfc3-4472-931f	50000	5349	3.59E-4	0.376
OD3	ac7e4a31-ab86-496b-98a9	50000	5349	3.59E-4	0.376
OD4	e4342f74-dd87-4667-8c31	50000	6616	1.99E-4	0.247
WD0	wikidata-20190426-lexemes	50000	9443	0.004	0.201
WD1	wikidata-20190510-lexemes	50000	9569	0.004	0.2
WD2	wikidata-20190517-lexemes	50000	9729	0.004	0.2
WD3	wikidata-20190524-lexemes	50000	9740	0.004	0.2
WD4	wikidata-20190607-lexemes	50000	9742	0.004	0.2

Table 5.2: RDF files for the comparison of GRP and HDT. URLs to the datasets are in Ch. 4.2.1.

On most of the used RDF graphs, $Graph_{GRP}$ achieves a better compression ratio than $Graph_{HDT}$. On average, $CR_{Graph_{HDT}}$ is 1.8 times higher than $CR_{Graph_{GRP}}$.
In order to investigate the correlation between the pairs (CR_{Graph_C}, ELR) and (CR_{Graph_C}, SPS) the Spearman correlation is used. That correlation coefficient produces values between -1 and 1. Negative values imply a negative correlation, positive values imply a positive correlation. The value 0 means that there is no correlation. [Zar05]
The results are shown in Tab. 5.3. Due to the value 0.646 it becomes clear that higher numbers of different predicates lead to a higher compression ratio for $Graph_{GRP}$. In contrast, a higher value of SPS implies a lower compression ratio for $Graph_{GRP}$, as expected. Although it has to be noted that for the latter case, the correlation is not as strong as with ELR.

$CR_{Graph_{HDT}}$ has a positive correlation with ELR, but that correlation is quite small. The correlation between $CR_{Graph_{HDT}}$ and SPS is positive (but very close to 0) which is reasonable, since $Graph_{HDT}$ only benefits from hub pattern, but SPS is defined for the general star pattern.



(a) DBPedia

(b) Semantic Web Dog Food

Figure 5.4: $CR_{GraphHDT}$ and $CR_{GraphGRP}$.

e mit vorsicht
ßen

	$CR_{Graph_{GRP}}$	$CR_{Graph_{HDT}}$
ELR	0.646	0.181
SPS	−0.381	0.009

Table 5.3: Spearman correlation values.

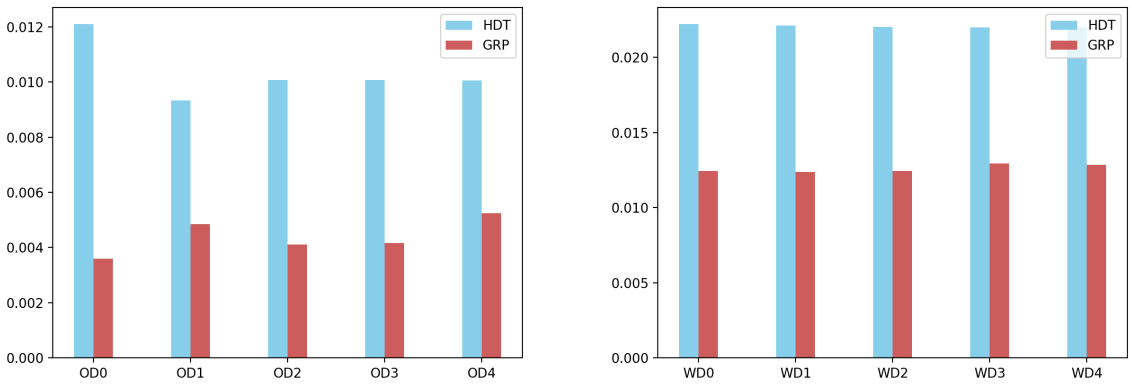
Fig. 5.4 and 5.5 show the compression ratios for the datasets, respectively. It has to be noticed that values of CR_{Graph_C} are not comparable between different RDF files. This is due to the fact that CR_{Graph_C} does not take the dictionary into account. Hence, if two files f_1, f_2 have the same graph structure, but f_1 is using longer URIs than $f2$, then CR_{Graph_C} will be smaller for f_1. This is because $|f_1| > |f_2|$ is true and $CR_{Graph_C} = \frac{out_{graph}}{in}$ (with $in \in \{f_1, f_2\}$ in this case). Therefore, the compression ratios of the two approaches can only be compared if they were achieved on the same input file.

5.2 Compression Improvements

This chapter shows the evaluation results for the compression improvements. First, results for the graph compression improvements ($Graph_{GRP}$) are discussed. Then, results for the dictionary compression improvements ($Dict_{HDT}$) are presented. In the end, both approaches will be combined in a final evaluation.

5.2.1 Ontology Knowledge

In this chapter, it will be evaluated whether using meta data from the ontology can result in a better compression ratio for $Graph_{GRP}$. Here, not only the file size of the output ($|out_{graph}|$) will be measured. Since these approaches have the intention to improve the ability for $Graph_{GRP}$ to produce a smaller grammar, the features of that grammar will be presented in more detail.



(a) Government Open Data

(b) WikiData

Figure 5.5: $CR_{GraphHDT}$ and $CR_{GraphGRP}$.

That is because the GRP's grammar encoding does not always produce a smaller result on a smaller graph (as mentioned in Ch. 2.3.2).

Dataset Overview

Tab. 5.4 shows an overview of the data used for the ontology-based manipulations. *MBP* stands for *mappingbased-properties_en*. DB_{full} is the full version of *MBP* and WN_{full} is the full version of Wordnet. The other datasets are adjusted to show the effects of the manipulations better. They contain a higher proportion of symmetric, inverse or transitive properties, respectively.

Short Name	Long Name	#Triples	#Resources	ELR	SPS
DB_{full}	MBP	32018293	4803076	0.04	0.158
DB_{inv}	MBP_manyinverses	1097	935	0.037	0.022
DB_{sym}	MBP_manysymmetrics	19954	18785	0.008	0.037
DB_{tra}	MBP_manytransitives	1994	1147	0.101	0.0004
WN_{full}	wordnet	2637168	601641	0.0	0.223
WN_{inv}	wordnet_manyinverses	1999	1351	0.008	0.113
WN_{sym}	wordnet_manysymmetrics	1999	2283	0.003	0.058
WN_{tra}	wordnet_manytransitives	3988	2033	0.004	0.08

Table 5.4: RDF files for the ontology-based improvements. URLs to the datasets are in Ch. 4.2.1.

Occurrence of Properties

In the following, symmetric, inverse or transitive properties are called relevant properties.
First, it is considered which relevant properties occur in real world datasets. Tab. 5.5 shows the relevant properties for DBPedia and Wordnet. It can be concluded that these amounts are relatively small, especially for symmetric properties, considering the fact that these datasets are relatively large.
Next, it is considered how often the relevant properties occur in the datasets. Thus, DB_{full} and WN_{full} will be analyzed. Fig. 5.6 shows how often the relevant properties occur. (Relative amount $= \frac{\#\text{occurrences}}{\#\text{triples}}$). It can be noticed that the amounts are quite low. This supports the hypothesis from Ch. 4.2.2 that building sub graphs with higher amounts of relevant properties

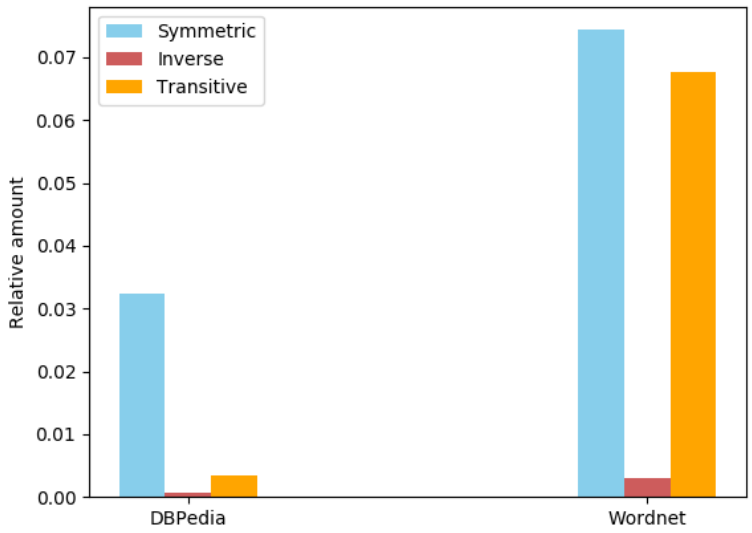


Figure 5.6: Relative amount of transitive/symmetric/inverse properties in real datasets. (Relative amount = $\frac{\#\text{occurrences}}{\#\text{triples}}$)

is necessary to observe the effect of the data manipulations. Consequently, those sub graphs (see Ch. 4.2.2) are used in the following evaluations.

	DBPedia	Wordnet
URL Prefix	http://dbpedia.org/ ontology/	http://wordnet-rdf.princeton.edu/ ontology#
Symmetric	spouse	antonym
Inverse	(doctoralStudent, doctoralAdvisor) (mother, child) (father, child) (follows, followedBy)	(hypernym, hyponym)
Transitive	isPartOf province locatedInArea city district county settlement	cause substance_holonym part_holonym entail member_holonym member_meronym instance_hypernym instance_hyponym hyponym par_meronym substance_meronym hypernym

Table 5.5: Overview of the relevant properties of DBPedia and Wordnet.

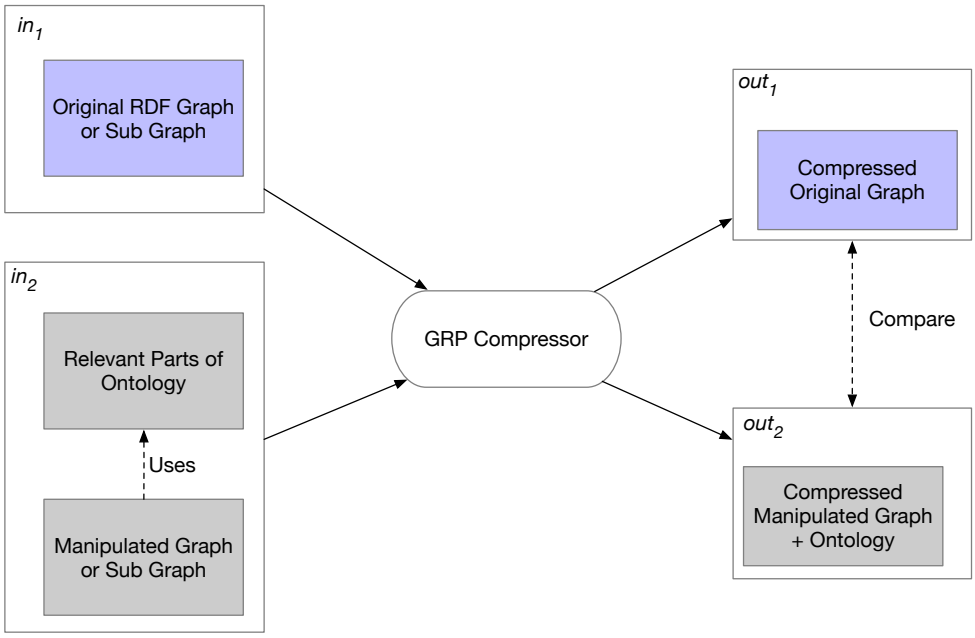


Figure 5.7: Overall process of applying ontology knowledge and comparing the compression results.

Evaluation Process

The overall process, which is used to evaluate the ontology-based manipulations, is illustrated in Fig. 5.7. First, the original graph or sub graph is given to GRP. That part is called in_1 in Fig. 5.7 and it will deliver the first result out_1.

In the next step, relevant properties have to be determined and the original graph will be manipulated. The manipulated graph is shown in the lower part of in_2. That manipulation step can be seen as a pre-processing and is not explicitly shown in Fig. 5.7. In addition, the relevant ontology triples (upper part of in_2) have to be compressed and stored as well. Otherwise, the original graph could not be restored. The manipulated graph and all relevant ontology triples are merged into one graph (in_2), which is then compressed by GRP and the result is called out_2.

Finally, out_1 and out_2 have to be compared. The next chapter will explain how that is done.

The evaluation process must be executed for the different manipulation aspects (symmetric, inverse etc.) independently. So, there is one manipulated graph where all symmetric edges are added or removed, and analogously for the other manipulations. This way, it is noticeable which effect the manipulations have, individually. A combined evaluation will be done in Ch. 5.2.3.

Metrics

This chapter explains how the two outputs out_1 and out_2 are compared. Since the ontology-based manipulations are intended to achieve a better compression ratio by enabling $Graph_{GRP}$ to produce a smaller grammar, out_1 and out_2 are not only compared at the file size level, but also in terms of their graph sizes. A reasonable metric for the size of a graph is its number of edges. In order to compare out_1 and out_2 with respect to their edge amounts, the metrics Input Edge Ratio (IER) and Output Edge Ratio (OER) are defined:

$$IER = \frac{\#\text{edges in } in_2}{\#\text{edges in } in_1}$$

$$OER = \frac{\#\text{edges in } out_2}{\#\text{edges in } out_1}$$

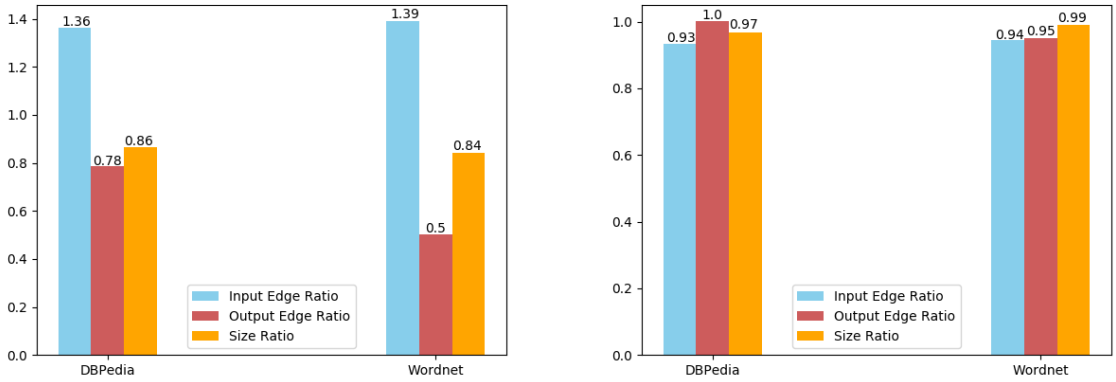
IER shows how many edges have been added or removed by the manipulation which indicates how big the impact of the manipulation is. OER shows whether the manipulation results in a better ($OER < 1$) or worse ($OER > 1$) compression ratio.
In order to measure the manipulation it is necessary to define a new metric instead of reusing $CR_{Graph_{GRP}}$ as defined in Ch. 3.1.2. For out_1, $CR_{Graph_{GRP}}$ would be in relation to in_1 (analogously for out_2 and in_2). But here it is necessary to compare out_1 and out_2. Therefore, the new metric Size Ratio (SR) is defined as follows:

$$SR = \frac{|out_{graph_2}|}{|out_{graph_1}|}$$

Here, out_{graph} is used instead of out. Normally, the dictionary size (out_{dict}) should also be taken into account, because the added ontology triples have increased $|out_{dict_2}|$. However, due to the very small amount of relevant properties that increase is not even noticeable and can be omitted. If $SR < 1$ holds then the manipulation has resulted in an improvement at the file size level, otherwise not. Of course, the file size level is, in the end, more relevant than the grammar level, because it is the size needed to store the compressed data.

Symmetric Properties

First, the approach suggested in Ch. 3.4.1 to add all possible triples with symmetric edges will be evaluated. Fig. 5.8a illustrates the results whereby DB_{sym} and WN_{sym} are used for DBPedia and Wordnet, respectively. As expected, IER is bigger than 1, as edges have been added to the graph. For both DBPedia and Wordnet, IER is about 1.4. Also, it can be noticed that $OER < 1$ holds, which means that adding symmetric triples is indeed beneficial for $Graph_{GRP}$ and even brings a significant improvement on the grammar level as OER is about 0.8 for DBPedia and about 0.5 for Wordnet.
Furthermore, $SR < 1$ is true. Hence, also on the file size level the manipulation has delivered a better result. However, the improvement is not as good as on the grammar level. So, the statement from Ch. 2.3.2, that GRP's encoding method does not always work well, is supported by these results.



(a) Results for adding symmetric properties.

(b) Results for deleting symmetric properties.

Figure 5.8: Results for adding or deleting symmetric properties, on the grammar level (IER, OER) and on the file size level (SR).

In order to show that adding symmetric properties is more beneficial than removing them, the results of the removal case are also shown (see Fig. 5.8b). There, it can be seen that $IER < 1$ holds, since edges have been removed. But IER is still quite close to 1, because there are not many cases in which an edge could be removed. Hence, in most cases only one of the two directions for symmetric properties is present in the graph. Both OER and SR are close or equal to 1, which indicates that adding symmetric edges is more beneficial for $Graph_{GRP}$. Nevertheless, it can still be argued that there are only a few cases in which edges were removed and that the potentially positive effect is therefore not recognizable. To further investigate the removal case, the decompressed version of the graph out_2 from Fig. 5.8b is taken as the input in_1 for a new evaluation. So, in_1 will be an artificial graph in which for each symmetric property only one direction of the triples exist. This extreme case can show whether adding or removing symmetric edges will be better.
The evaluation results are illustrated in Fig. 5.9. Of course, $IER > 1$ is true and IER is bigger than in Fig. 5.8a, since a higher amount of edges has been added. The fact that both OER and SR are lower than 1, supports our hypothesis that adding symmetric edges is more beneficial than removing them. So, we conclude that adding symmetric edges enables $Graph_{GRP}$ to find more digrams, and thus producing a smaller graph.

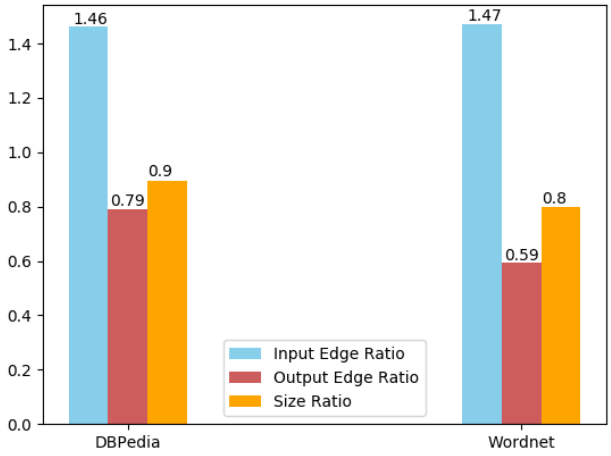
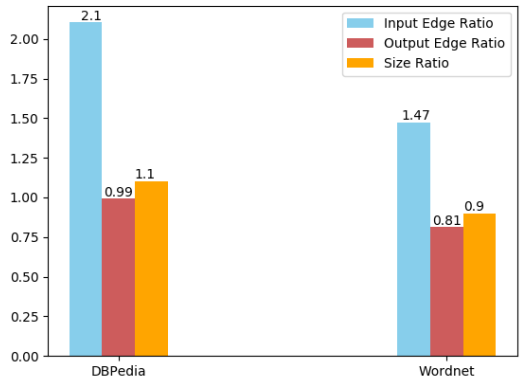


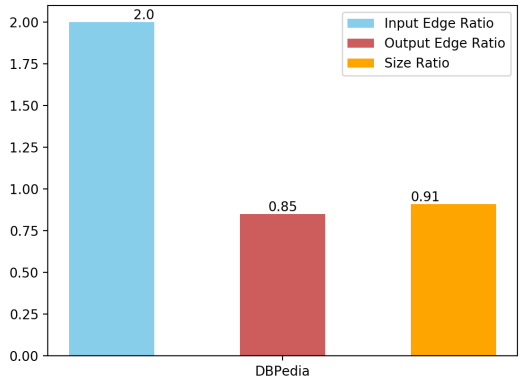
Figure 5.9: Results for adding symmetric properties, on the grammar level (IER, OER) and on the file size level (SR). In contrast to Fig. 5.8a, the input graph in_1 contains only one direction for each triple with a symmetric property.

Inverse Properties

Next, the usage of inverse properties is evaluated. First, the approach presented in Ch. 3.4.1 (adding triples with inverse properties) is considered. Here, DB_{inv} and WN_{inv} have been used for DBPedia and Wordnet, respectively. The results are shown in Fig. 5.10a. For DBPedia, the size of the input graph has been doubled ($IER = 2.1$). But $OER = 0.99$ is almost 1, thus the manipulation has not resulted in a better compression for DBPedia. SR is even bigger than 1, so the result is even worse on the file size level.
For Wordnet, IER is smaller than for DBPedia and more importantly, OER is significantly smaller than 1. Hence, in this case, the manipulation has delivered a positive result. On the file size level, the result is also better ($SR = 0.91$).



(a) Results for adding inverse properties.



(b) Results for adding inverse properties in DBPedia (bigger graph than in Fig. 5.10a).

Figure 5.10: Results for adding inverse properties, on the grammar level (IER, OER) and on the file size level (SR).

In the case of DBPedia, a reason for the negative result can be that DBPedia has a relatively high number of different inverse properties. Therefore, adding all the edges with many different URIs to the relatively small graph in Fig. 5.10a (1097 triples) does not have an effect that is positive enough to compensate the high amount of edges added to it. In order to further investigate that, a graph that is ten times bigger, is created. The results for it are shown in Fig. 5.10b which are much better. While IER is about 2 again, $OER = 0.85$ is much smaller and also $SR = 0.91$ is smaller. So, it can be stated that adding inverse properties has a positive effect on the graph compression potential, as mentioned in Ch. 3.4.1.
The opposing approach (removing triples with inverse properties) is evaluated as well. Thus, the same technique, which was already used for symmetric properties, is used again. For DBPedia and Wordnet, the graphs are manipulated such that only one direction exists for each occurrence of inverse properties. Those graphs are then used as the input in_1 for a new evaluation. The results are illustrated in Fig. 5.11. There, it can be seen that the values for IER are higher than before, as expected. Correspondingly, the OER-values are lower, which supports the hypothesis that removing inverse triples leads to an unfavorable graph structure for GRP.

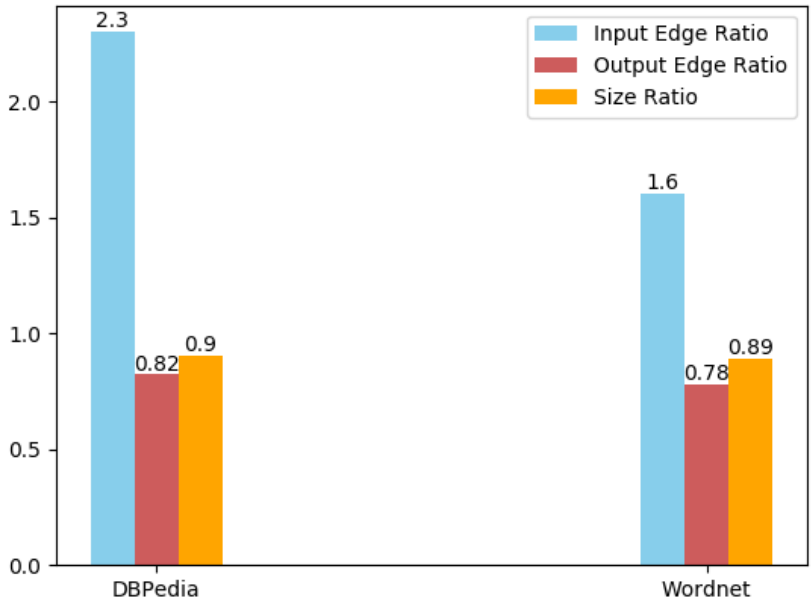
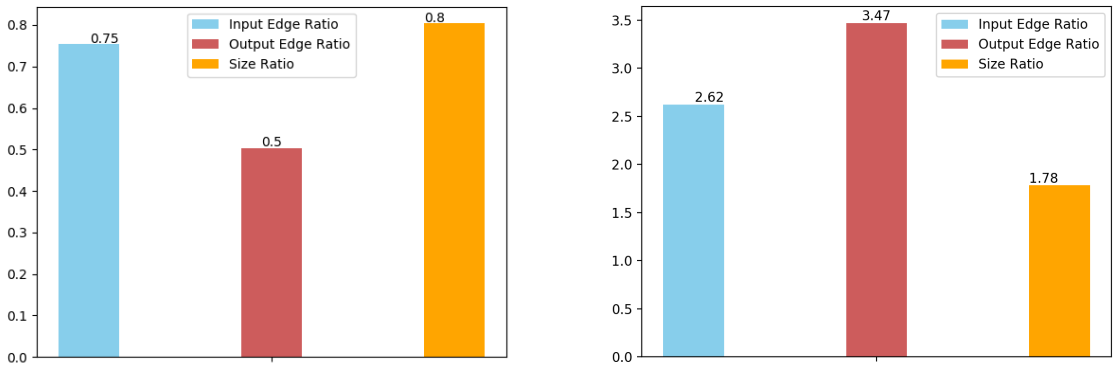


Figure 5.11: Results for adding inverse properties, on the grammar level (IER, OER) and on the file size level (SR). In contrast to Fig. 5.10a and 5.10b, the input graph in_1 contains only one direction for each triple with an inverse property.

Transitive Properties

The final evaluated ontology-based manipulation is adding or removing triples with transitive properties. The problem with the transitive case is that it is more restrictive than the others (symmetric or inverse). It is therefore more difficult to find data where this manipulation can be executed. In DB_{tra}, there are no *transitive paths* (defined in Ch. 3.4.1). Thus, only WN_{tra} will be used here, since it contains *transitive paths*.
First, the approach suggested in Ch. 3.4.1, to remove *direct transitive paths*, is evaluated. Unfortunately, those edges do not exist in Wordnet. To circumvent that problem, an artificial graph has been created in the following way: The original graph has been divided into two halves (via its list of triples). In the first half, all possible *direct transitive paths* are added. Afterwards, the manipulated first half is merged with the untouched second half again. This gives us a graph in which both adding and removing *direct transitive paths* is possible. So, both approaches can be evaluated.
The results for removing *direct transitive paths* can be seen in Fig. 5.12a. IER is about 0.75, as several edges have been removed. OER is about 0.5, which means that the removals have delivered a significant improvement on the grammar level. It could be argued that the improvement is only due to the reduced size of the graph. But this is not the case, since $OER < IER$ holds. Hence, due to the removed *direct transitive paths*, $Graph_{GRP}$ could indeed find more digram occurrences, as stated in Ch. 3.4.1. On the file size level, the improved compression is also visible, because $SR \approx 0.88$ is less than 1. Again, the improvement is not as good as on the grammar level.



(a) Results for removing transitive properties.

(b) Results for adding transitive properties.

Figure 5.12: Results for deleting and adding transitive properties, on the grammar level (IER, OER) and on the file size level (SR).

Next, the opposing approach (adding *direct transitive paths*) is evaluated. The results are illustrated in Fig. 5.12b. Obviously, $IER > 1$ holds, since many edges have been added. $OER \approx 3.5$ is significantly high, which means that adding *direct transitive paths* has resulted in a much larger result, on the grammar level. In contrast to Fig. 5.12a, $IER < OER$ holds here, which supports the hypothesis that adding *direct transitive paths* makes $Graph_{GRP}$ find less digram occurrences (see Ch. 3.4.1). On the file size level, $SR \approx 1.8$ is higher than 1, but the result is not as bad as on the grammar level.
Thus, it can be concluded that removing *direct transitive paths* results in a better compression, and adding them delivers a worse compression. Although *direct transitive paths* do not exist in

the considered datasets, they still can exist in other datasets. The evaluation results have shown that removing *direct transitive paths* leads to a significant improvement.

5.2.2 Dictionary Improvements

In this chapter, the approaches from Ch. 4.2.3, which are intended to improve the dictionary compressor $Dict_{HDT}$ (which is used by HDT and GRP), are evaluated. First, the Huffman compression of literals is evaluated and afterwards, the improvements regarding blank node IDs are discussed.

Dataset Overview

Tab. 5.6 shows the RDF files which are used for evaluating the Huffman improvements. ELR and SPS are not shown here, since they are not relevant for the dictionary compression. Moreover, the datasets used in Ch. 5.2.2 (with the names DF0-DF9) are the same as in Tab. 5.2 and are therefore not shown in Tab. 5.6.
Tab. 5.7 shows the files used for the blank node improvements. Here, also the number of triples with blank nodes is presented (includes triples with at least one blank node).

Short Name	Long Name	#Triples	#Resources
AB_EN	long-abstracts_en_small	10000	10000
AB_BG	long-abstracts-en-uris_bg_small	10000	10000
AB_FR	long-abstracts-en-uris_fr_small	10000	10000
AB_RU	long-abstracts-en-uris_ru_small	10000	10000

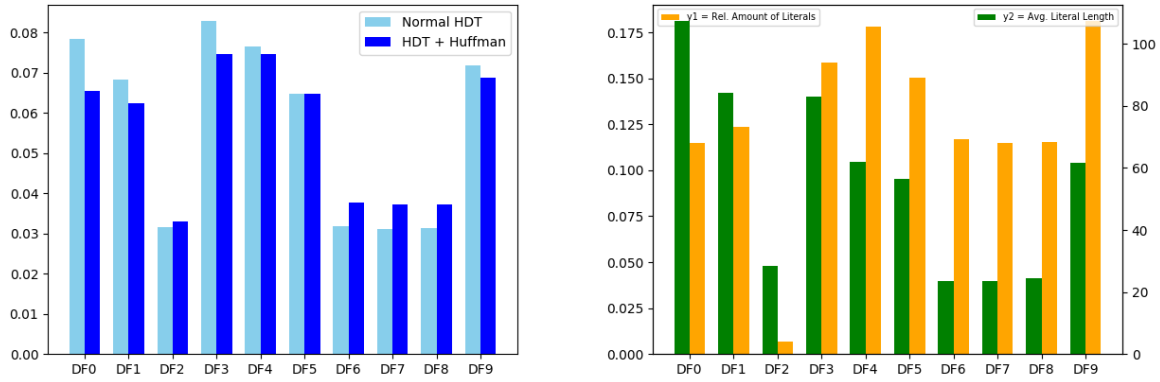
Table 5.6: RDF files for the Huffman evaluations. URLs to the datasets are in Ch. 4.2.1.

Short Name	Long Name	#Triples	#Resources	#Triples with blank nodes
GC1	DEE_geometry	213	12	211
GC2	PL1_geometry	357	12	355
GC3	TR2_geometry	280	14	278

Table 5.7: RDF files for the blank node evaluations. URLs to the datasets are in Ch. 4.2.1.

Literals

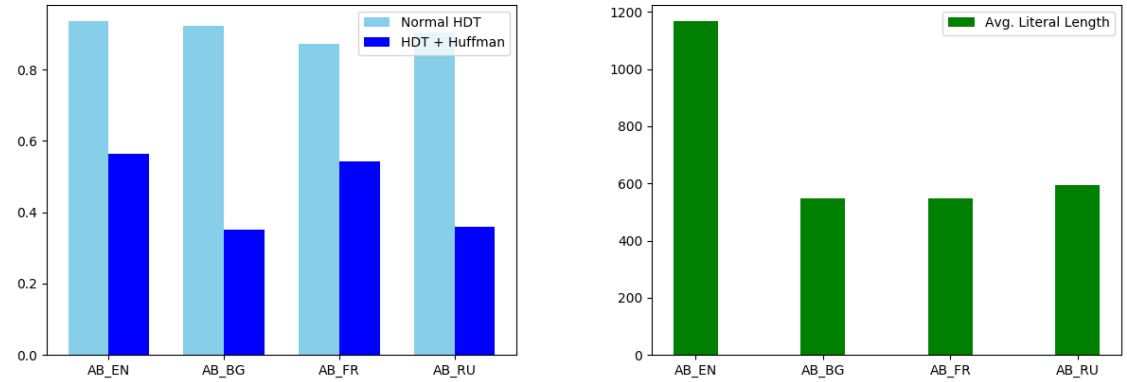
As already mentioned, a self-generated Huffman code is used to compress the literals of an RDF graph, as it is likely to achieve a better compression than with the prefix-based compression of HDT.
Thus, the results of the evaluation are presented. First, the Semantic Web Dog Food data set is used. This is well suited for an evaluation, since both literals and URIS are included here. Fig. 5.13a shows $CR_{Dict_{HDT}}$ for a subset of the data (the same files as in Fig. 5.4b).
It can be seen that the usage of the Huffman code only brings a small improvement in some cases. In other cases, Normal HDT even compresses better than HDT with Huffman.
This is because the data does not have a very high proportion of literals. Also, literals are rather short here. This can be seen in Fig. 5.13b showing the relative literal amount ($\frac{\#literals}{\#triples}$) and the average length of a literal. The proportion of literals in the triples is never higher than 17.5% and the highest average literal length is about 12. Hence, these are single words rather than whole texts. Only in cases where both the proportion and the literal length are relatively high, an improvement can be seen (e.g. DF9).



(a) $CR(out_{dict})$ for Normal HDT and HDT + Huffman (b) Relative amounts of literals and average literal length.

Figure 5.13: Compression ratios (5.13a) and features of the same files (5.13b) from Semantic Web Dog Food.

Next, the DBPedia data set is considered, because it has a different structure of data. Here, graphs are considered which contain the abstracts of Wikipedia, because these are longer texts. In addition, there are abstracts in different languages, so it can be seen whether some languages are better suited for Huffman than others.
Fig. 5.14a shows $CR_{Dict_{HDT}}$ for the abstract files. The abbreviations stand for the languages the abstracts are written in. It can be seen that Huffman significantly improves $CR_{Dict_{HDT}}$ here. The reason for this can be seen in Fig. 5.14b, where the average literal length is illustrated. The relative literal amount is not shown this time, since it is 100% for all files. The average literal length varies between languages, but is generally much higher than in Semantic Web Dog Food. Therefore, the improvement is much greater here.
Another phenomenon can also be seen here: Although the English file contains the longest literals by far, the improvement by Huffman here is not as big as, for example, in the Bulgarian file. In the English case, the Huffman code is less effective, because there are fewer different characters than in the other languages. Huffman is generally more efficient on large alphabets, according to [Sha10].

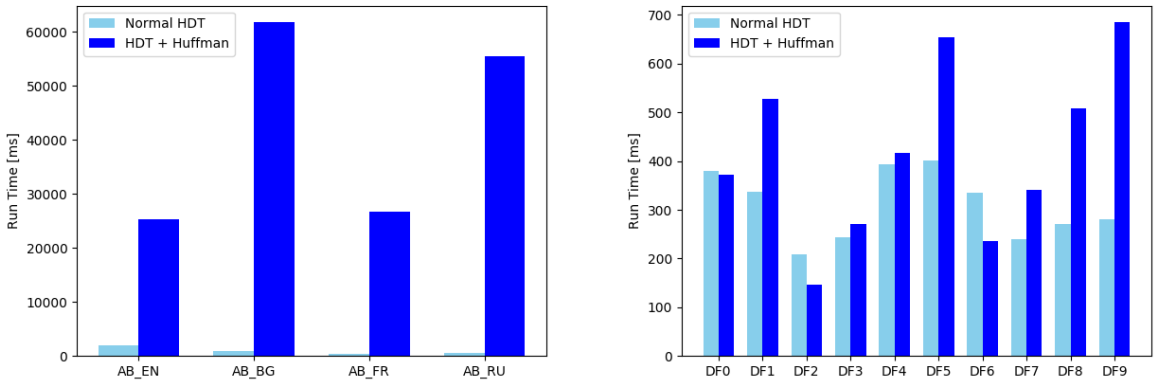


(a) $CR(out_{dict})$ for Normal HDT and HDT + Huffman.

(b) Average literal length.

Figure 5.14: Compression ratios (5.14a) and features of the same files (5.14b) from DBPedia.

As mentioned above, saving the Huffman code means almost no additional storage effort. The average fraction of the Huffman code is about 0.1% and was therefore not displayed in the visualizations.
Since the calculation of the Huffman code increases the runtime of the whole compression, it is now considered how big this effect is. Fig. 5.15a shows CT for the compression of the DBPedia abstract data. Here, CT has been increased very much. This is because $Dict_{HDT}$ can normally compress very little with this data, due to the numerous long literals and is therefore finished quite quickly. With Huffman, the data can be compressed much better and it takes quite a long time. Here, especially the creation of the Huffman tree is expensive.
Fig. 5.15b shows CT for the Semantic Web Dog Food data, where the run times are only slightly longer, because Huffman hardly improves the compression.
Generally, it can be said that the usage of a standard Huffman code could be worthwhile, because of the otherwise high runtime. At this point, however, the evaluation with such a standard code was omitted, since such existing codes do not contain all the special characters that occur in RDF data.



(a) CT for DBPedia abstracts.

(b) CT for Semantic Web Dog Food files.

Figure 5.15: Run times (CT) for Normal HDT and HDT + Huffman.

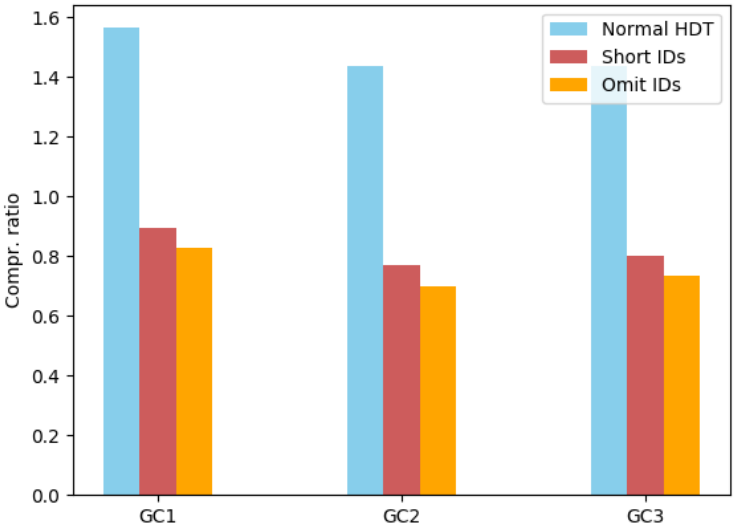


Figure 5.16: Results for Blank Node ID improvements for Geo Coordinates (GC) files. **Normal HDT** uses long IDs. **Short IDs** uses integer IDs. **Omit IDs** does not store the IDs at all.

Blank Nodes

This chapter evaluates the two approaches mentioned in Ch. 3.4.2 to store blank node IDs more efficiently. Thus, the Nuts-RDF dataset is used as it contains much more blank nodes than many other datasets. It is therefore prone to show the inefficiency of the prefix-based compression of $Dict_{HDT}$ with respect to blank nodes. Fig. 5.16 shows the results for three different files from Nuts-RDF (see Tab. 5.7). It can be seen that Normal HDT even produces the result $CR_{Dict_{HDT}} > 1$. This is due to the fact that GC1-GC3 are in the turtle format which already reduces redundancy, since it does not store all triples extensively, like N-Triples does. Also, it does not use blank node IDs that are as long as those used by $Dict_{HDT}$. Using short IDs already leads to a significant improvement, as it delivers a compression ratio between 0.76 and 0.89. So, the compression is, in contrast to Normal HDT, really reducing the size of the files. Omitting blank node IDs results in a further slight improvement which is small, because storing the short IDs as numbers from 1 to n costs almost no memory size. Hence, it could be sufficient to use shorter IDs, because omitting IDs would imply more programming effort, as explained in Ch. 4.2.3.

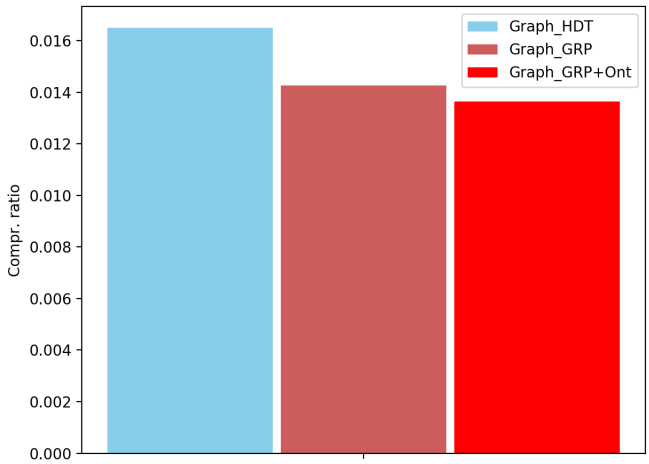


Figure 5.17: Graph compression ratios (CR_{Graph_C}) for $Graph_{HDT}$, $Graph_{GRP}$ and $Graph_{GRP+Ont}$.

5.2.3 Final Results

In the previous chapters, the compression improvements (graph and dictionary) were evaluated individually. This final evaluation chapter compares all improvements together. This makes it possible to see which improvement accounts for which portion of the overall improvement.
In order to realize this, an RDF graph is generated from several graphs. The graph DB_{inv} from Tab. 5.4 serves as a basis, as it contains inverse and symmetric properties. However, there are not many long literals in it. Therefore, the DBPedia abstract text is added for each person in the graph (from the Bulgarian abstract RDF file from Tab. 5.6). Next, blank nodes are added to the graph. Thus, the coordinates (polygons) for each country of the graph are added. The data from Tab. 5.7 is used for that. Finally, this results in a graph on which all improvements have an effect. In Tab. 5.8, the features of that graph can be seen.

#Triples	#Resources	#Literals	#Triples with blank nodes
101193	59565	6058	230

Table 5.8: RDF file for the final evaluations.

Fig. 5.17 shows the graph compression improvements. For the ontology-based improvement of $Graph_{GRP}$ we write $Graph_{GRP+Ont}$. It can be seen that $Graph_{GRP}$ already compresses better than $Graph_{HDT}$. $Graph_{GRP+Ont}$ leads only to a small improvement. This is again due to the encoding problem discussed in Ch. 5.2.1.
Fig. 5.18 shows the results regarding the dictionary compression. By $Dict_{HDT++}$, the improved HDT dictionary compression (literals and blank node IDs) is denoted. Here, the improvement is significant although the used RDF graph does not contain as much literals as the abstract graphs used in Ch. 5.2.2.
Finally, Fig. 5.19 shows both the graph and dictionary compressions in combination. HDT++ denotes the improved HDT (HDT++= $\{Dict_{HDT++}, Graph_{HDT}\}$). GRP++ denotes the ontology-based graph compression improvement combined with the improved dictionary compression (GRP++ = $\{Dict_{HDT++}, Graph_{GRP+Ont}\}$). The horizontal lines drawn through the bars de-

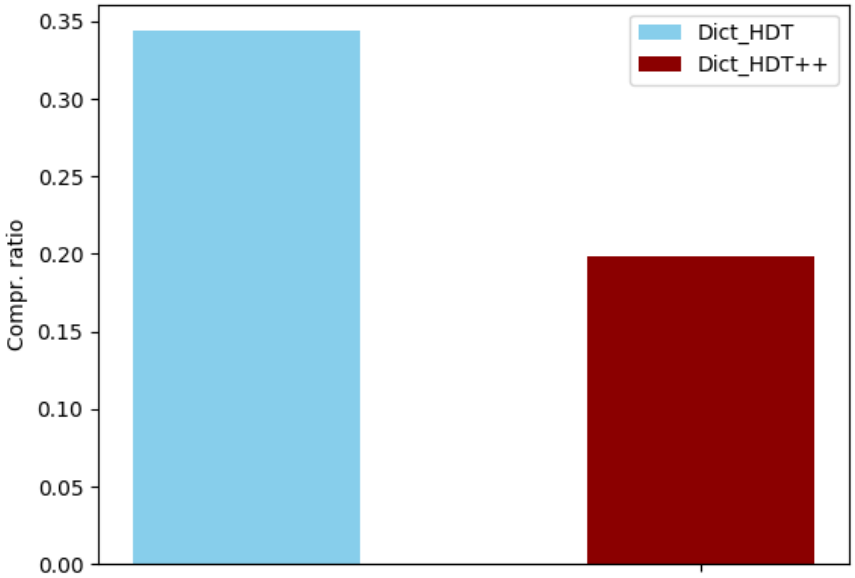


Figure 5.18: Dictionary compression ratios (CR_{Dict_C}) for $Dict_{HDT}$ and $Dict_{HDT++}$.

note the respective values of CR_{Dict_C}. That shows how big the portion of the dictionary is in the compressed data.
Apart from that, the difference between HDT++ and GRP++ is very small. This due to the fact that the graph (out_{graph}) is so much smaller than out_{dict}.
GZip is used a baseline. Since it is a general compressor, it does not have a horizontal line. It is noticeable that GZip achieved a slightly better compression ratio than HDT++ and GRP++. However, as already mentioned in Ch. 1, the data compressed by GZip is not query-able which makes it less usable.
In general, it can be concluded that the improvement compressors (HDT++ and GRP++) deliver a significant difference to HDT and GRP. The resulted compression ratio is almost as low as GZip's ratio. Considering the fact that data compressed by GRP++ or HDT++ is still query-able, it is reasonable to use one of them instead of a general compressor. With further improvements (will be discussed in Ch. 6.2) it could be that HDT++ or GRP++ outperform general compressors in many more cases.
Moreover, it becomes clear that compressing the dictionary has a much bigger impact on CR_C than compressing the graph. So, for future work it would be reasonable to focus on dictionary compression. However, there are cases in which the dictionary is smaller and therefore the graph compression has a higher impact. A further idea is be to compress multiple RDF files, which all have a similar dictionary, together. This way, the influences of the graphs can become bigger.

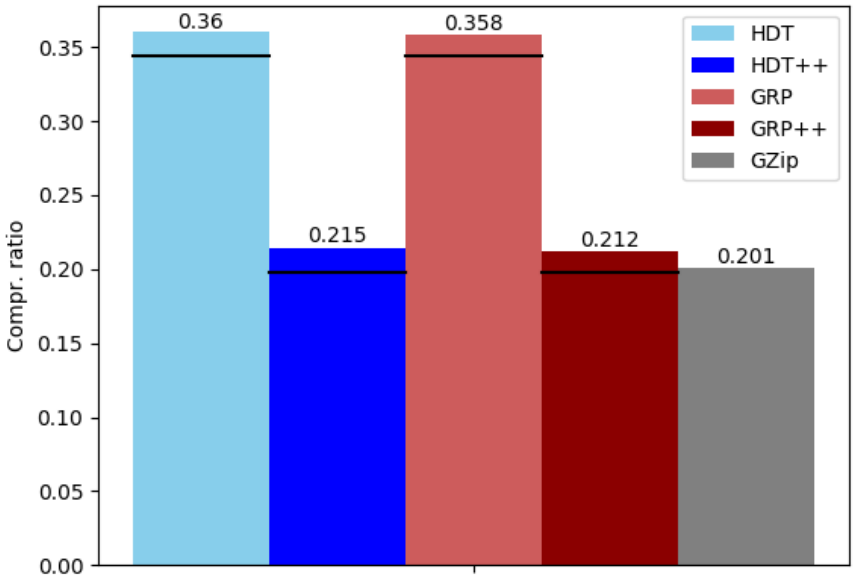


Figure 5.19: Complete compression ratios (CR_C) for all compressors. Horizontal lines mark the values of CR_{Dict_C}. GZip is used as a baseline.

6 Summary and Future Work

This last chapter concludes the thesis by firstly summarizing its most important results in a compact form and secondly, discussing future work in the area of RDF compression.

6.1 Summary

Due to the constantly growing size of the Semantic Web, scalability problems can occur in terms of storage, transmission and consumption of knowledge graphs. At this point, compression can help by reducing the size of the graphs. The focus of this thesis was on storage and transfer, since grammar-based compression, which was considered here, is best suited for these use cases.

The first part of the thesis was to compare GRP with HDT. There, it was shown that GRP achieves a better compression ratio in the vast majority of the used datasets. Moreover, it was shown that the structure of the input data has a much higher effect for GRP than for HDT, because GRP can build much more complex sub structures. In terms of the run time, HDT clearly outperforms GRP. Generally, it can be concluded that GRP is a good alternative to HDT in terms of the compression ratio.

Based on those results, the next task was to improve the compression ratio. Here, the first sub task was to improve the graph component ($Graph_{GRP}$) of GRP by applying knowledge from the ontology of an RDF graph. The results showed that GRP can find much more patterns (digrams) after symmetric and inverse triples were added and transitive triples were removed. The problem was that the grammar encoding method of GRP cannot take full advantage of that improvement.

The second sub task was to improve the dictionary component of HDT ($Dict_{HDT}$) which is also used by GRP. As the results have shown, compressing long literals by using a Huffman Code brings a significant improvement. Also, handling blank node IDs more efficiently results in a much lower compression ratio if the graph contains a high number of blank nodes.

The final evaluation results showed that the dictionary has a much higher effect on the compression ratio than the graph which is an important insight for future research on this topic.

In conclusion, the thesis has shown that grammar-based graph compression is suitable for RDF and also delivers better results than HDT in most cases. Moreover, it was shown that there is much potential in terms of improving the dictionary compression.

6.2 Future Work

The topic of RDF compression covered in this thesis can be divided into two aspects: Graph and dictionary compression. Therefore, these aspects are presented separately in the following list:

- Graph Compression
  - **More sophisticated implementation of GRP:** The GRP implementation used in the thesis is only a proof of concept, i.e., it is not performing well in terms of its run time. Also, the whole graph has to be loaded into the main memory. A faster and more space-efficient variant is needed in order to compete with the more mature HDT.
  - **Node-based compression:** GRP is an edge-based compressor, which means that it replaces edges of a graph by non terminal edges. However, there are also grammar-based compressors which are edge- and node-based, thus also replacing nodes by non terminal nodes. It would be interesting to see whether they can outperform GRP for RDF graphs. Currently, such a compressor is under development as an extension of [Dü16].
  - **Better grammar encoding:** In Ch. 2.3.2, it has already been stated that the grammar encoding method of GRP is not always working well (especially the k^2-tree for storing the start rule). Therefore, a different method can be chosen which maybe delivers better results.
  - **Compress multiple graphs together:** As already mentioned in Ch. 3.4.1, it is possible to compress multiple graphs at once. Then, properties or entities can be replaced by equal properties or entities, respectively. This would enable $Graph_{GRP}$ to produce a better compression.
  - **Query evaluation:** This thesis did not investigate the query run times for GRP. The authors of [MP18] have already proven that neighborhood queries are very slow on graphs compressed by GRP. However, according to [MP18], reachability queries are faster on compressed graphs than on original ones. Therefore, it could be worthwhile to compare those query run times to HDT's query speeds. That could show if GRP is also a good alternative to HDT with regard to reachability queries which can also be done using SPARQL.
- Dictionary Compression
  - **More compression methods for literals:** As explained in Ch. 3.4.2, the thesis focused on string compression via Huffman Coding. But in literals there can be many other data types. It would be sensible to have a separate compression technique for each of those data types. That would probably result in a better compression ratio.
  - **Pre-computed Huffman Trees:** In this thesis, pre-computed Huffman Trees were not used, because they do not contain all special symbols occuring in the data. The problem with self-computed Huffman Trees is that this significantly increases the run time. Hence, it could make sense to pre-compute Huffman Trees (e.g. for DBPedia) including all necessary symbols. Then, the same tree could be used for several files (e.g. all files from DBPedia).
  - **Compress multiple dictionaries together:** This task is similar to compressing multiple graphs together. It can also make a difference with respect to the dictionary.

If multiple RDF files have similar dictionaries (many overlapping URIs or literals) those could be merged into one main dictionary which would then be compressed. Of course, this is no trivial task, because it is necessary to differentiate the different smaller dictionaries in order to be able to restore all original graphs and their respective dictionaries.

List of Figures

List of Tables

List of Acronyms

RDF	Resource Description Framework
HDT	Header Dictionary Triples
GRP	GraphRePair
RDFS	RDF Schema
OWL	Web Ontology Language
CR_C	Compression Ratio for Compressor C
CT_C	Compression Run Time for Compressor C
DCT_C	Decompression Run Time for Compressor C
ELR	Edge Label Ratio
SPS	Star Pattern Similarity
IER	Input Edge Ratio
OER	Output Edge Ratio
SR	Size Ratio

Todo list

Bibliography

[CW84] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Communications*, 32:396–402, 1984.

[Dü16] Matthias Dürksen. Grammar-based Graph Compression. Bachelor's thesis, University of Paderborn, 2016.

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975.

[FMPG+13] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary rdf representation for publication and exchange (hdt). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19:22 – 41, 2013.

[HKRS08] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph, and York Sure. *Semantic Web: Grundlagen*. eXamen.press. Springer, Berlin, 2008.

[LIJ+15] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[MBF+90] George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine J. Miller. Introduction to WordNet: An On-line Lexical Database*. *International Journal of Lexicography*, 3(4):235–244, 12 1990.

[MP18] Sebastian Maneth and Fabian Peternek. Grammar-based graph compression. *Inf. Syst.*, 76:19–45, 2018.

[NGPG16] Andrea Giovanni Nuzzolese, Anna Lisa Gentile, Valentina Presutti, and Aldo Gangemi. Conference linked data: the scholarlydata project. In *Proceedings of ISWC 2016 - Resource Track*, volume 9982 of *Lecture Notes in Computer Science*, pages 150–158. Springer, 2016.

[owl] Owl reference. `https://www.w3.org/TR/owl-ref/`. Accessed: 2019-05-07.

[SDB16] Dr Bhavanari Satyanarayana, Srinivasulu Devanaboina, and Mallikarjun Bhavanari. Star number of a graph. *Research Journal of Science and IT Management-RJSITM (ISSN: 2251-1563)*, 5:18–22, 09 2016.

[Sha10] Mamta Sharma. Compression using huffman coding. 2010.

[spa] Sparql reference. https://www.w3.org/TR/rdf-sparql-query/. Accessed: 2019-05-07.

[Zar05] Jerrold Zar. *Spearman Rank Correlation*, volume 5. 07 2005.