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.)

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, May 31, 2019

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.

Ort, Datum	Unterschrift

Abstract. We present a full documentation of the Paderborn University Computer Science thesis template (UPB-CS-TT) and how to use it. This document also serves as a demonstrator to show what documents UPB-CS-TT produces. Have fun!

Contents

1 Introduction

1.1 Motivation

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 then 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.
A currently popular compressor for RDF data 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 compression 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.

1.2 General Idea

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

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

1. Storage
2. Transmission
3. Processing/Consumption

The idea is to make all these use cases easier / 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 above all can be transferred faster, which occurs frequently in reality and can become a problem due to possibly slow transfer speeds .

here possibly co
crete numbers

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.

1.3 Thesis Structure

In Ch. 2 the necessary fundamentals are presented. It will mainly be about the two compressors HDT and GRP.
Ch. 3 deals with the approaches for the different aspects of the thesis. There, they will only be presented in a theoretical manner whereas in Ch. 4 the implementation details will be presented.
In Ch. 5, the approaches to the different tasks will be evaluated using real RDF data in order to confirm the theoretical hypotheses stated before.
Finally, the results of the thesis are summarized in Ch. 6 and the 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 infinite and 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 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 the Semantic Web, vocabularies define the concepts and relationships (also called "terms") that describe and represent a problem area. Vocabularies are used to classify the terms that can be used in a particular application, characterize possible relationships, and define possible restrictions on the use of these terms. In practice, vocabularies can be very complex (with several thousand terms) or very simple (describing only one or two concepts).
There is no clear distinction between what is called "vocabularies" and "ontologies". The trend is to use the word "ontology" for more complex and possibly quite formal collections of terms, while "vocabulary" is used when such strict formalism is not necessarily used or only in a very loose sense. Vocabularies are the building blocks for inference techniques in the Semantic Web.
ssen Web Ontology Language (OWL)

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 [1]) 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. Imagine for example some triples that 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. So 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' means that from now on a new subject comes. 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 means that from then on a new predicate comes. 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

[1] 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.

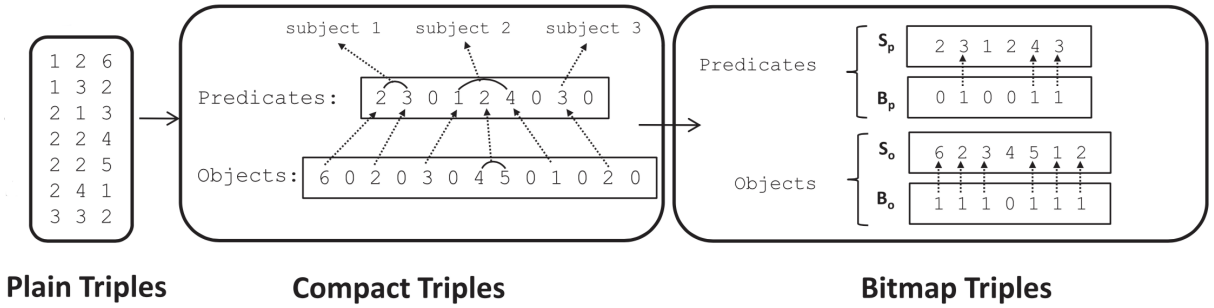


Figure 2.1: Three different representations of triples in HDT, figure from [FMPG+13].

2.3 Grammar-based Graph Compression

In this section we will discuss two different approaches to grammar-based graph compression. First, [Dü16] is introduced and briefly explained why the approach is less suitable for RDF. Then [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.
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'. Thus 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

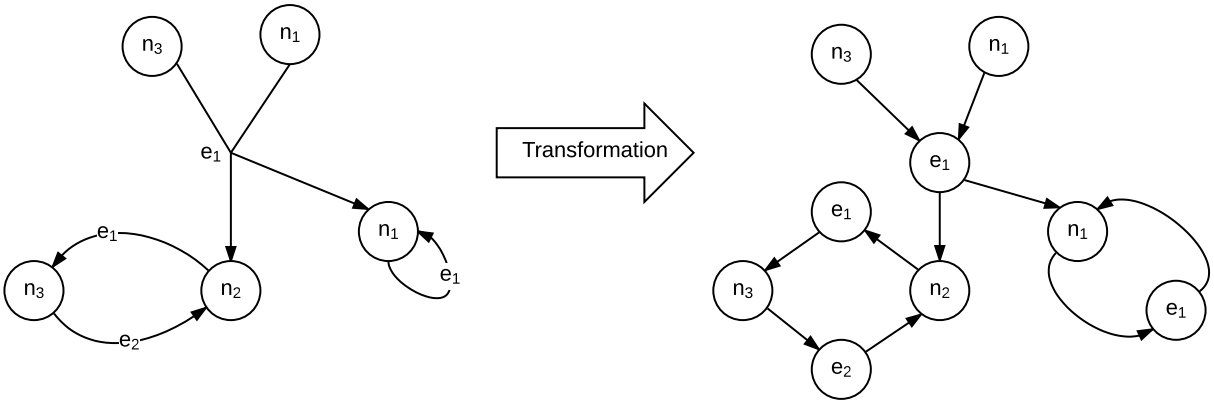


Figure 2.2: Transformation of a labeled graph (each edge is replaced by a new node having the label of the replaced edge).

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.

In addition, 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. In addition, 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 GraphRePair, which is presented in chapter 2.3.2.

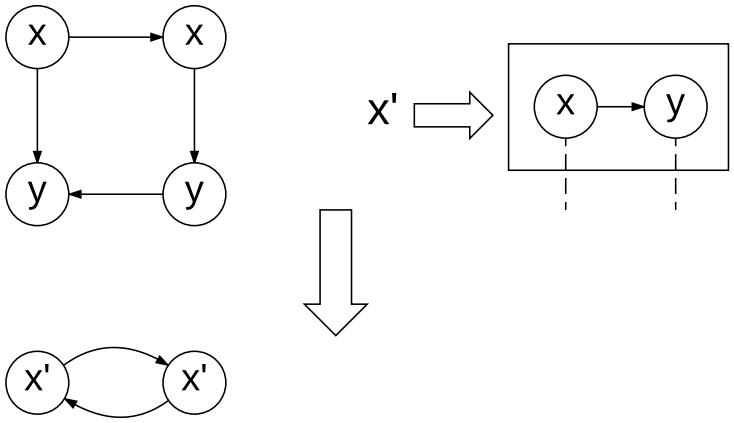


Figure 2.3: Replacement of two occurrences of the digram $X \to Y$ by nodes with the label X'.

2.3.2 GraphRePair

This chapter deals with GRP, the compressor from [MP18]. It firstly presents some foundations and afterwards 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.

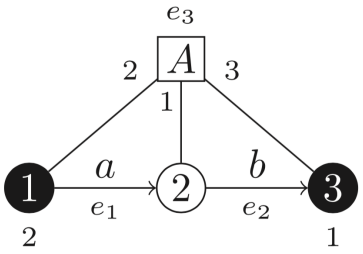


Figure 2.4: A hyper graph as defined in [MP18]. The black nodes 1 and 3 (consider the numbers within the nodes) are external nodes whereas the white node 2 is internal. e_3 (with the label A) is a hyper edge while e_1 and e_2 (labeled with a and b, respectively) are normal edges.

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

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. $M^* = M^+ \cup \{\epsilon\}$ is similar to M^+, but it also includes 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. Conditions 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]

Algorithm

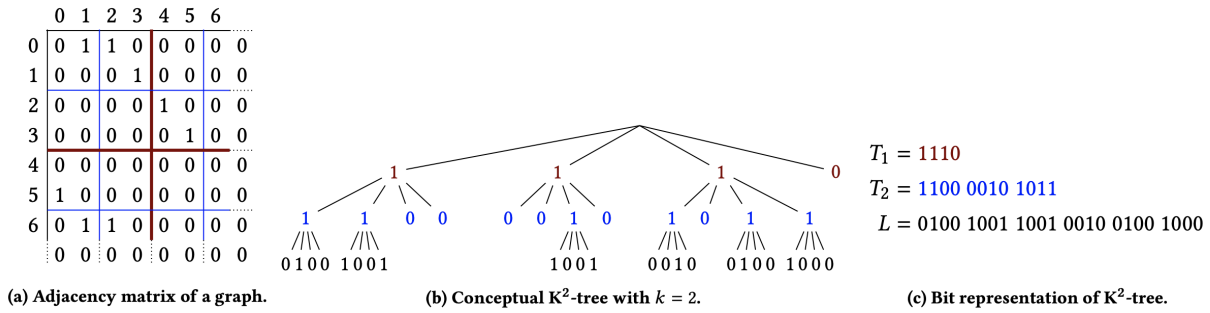
Algorithm 1 is the main routine of GraphRePair. The algorithm take a graph as input and returns a grammar whereas N is the set of non terminals, P is the set of productions and $S \in P$ is the start production. It maintains a list of digram occurrences in g. As long as this list contains multiple occurrences for one digram the loop will continue. In the loop, the most frequent digram is found and 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]

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

```
1: N, P ← ∅
2: S ← g
3: L(d) ← 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:     mfd ← most frequent digram
6:     A ← new non terminal for mfd
7:     Replace every occurrence of mfd in g
8:     N ← N ∪ {A}
9:     P ← P ∪ {A → mfd}
10:    Update the occurrence list L
11: return Grammar (N, P, S)
```

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. 2.5. 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. Then it is partitioned into k^2 equally large partitions ($k = 2$ here).
The tree's root represents the whole matrix and its child order correspond 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. 2.5 (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. 2.5 (c). T_1 and T_2 encode the first and second level and L encodes the level of the leaves.



(a) Adjacency matrix of a graph. (b) Conceptual K^2-tree with $k = 2$. (c) Bit representation of K^2-tree.

Figure 2.5: An adjacency matrix, its k^2 tree representation and the tree's bit representation.

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 to the adjacency matrix.
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. Essentially this is a way of displaying numbers as a bit-string in an efficient way (similar to a Huffman Code). To realize that, the right hand side of the components of the production have to represented by numbers, e.g. the edges labels, the node IDs etc. [MP18]

uer erklären

Implementation

GRP has been implemented by the authors of [MP18] in the Scala language [2]. It has to been seen as a proof of concept implementation, i.e., it has quite a high run time for bigger graphs. The 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.

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

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. That step is called compression. Decompression can be described as tasking out as an input and producing in.

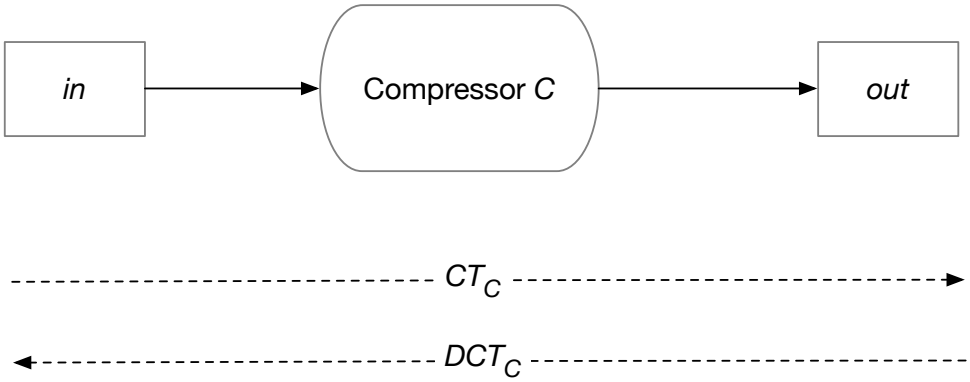


Figure 3.1: Visualization of the General Compressor Model.

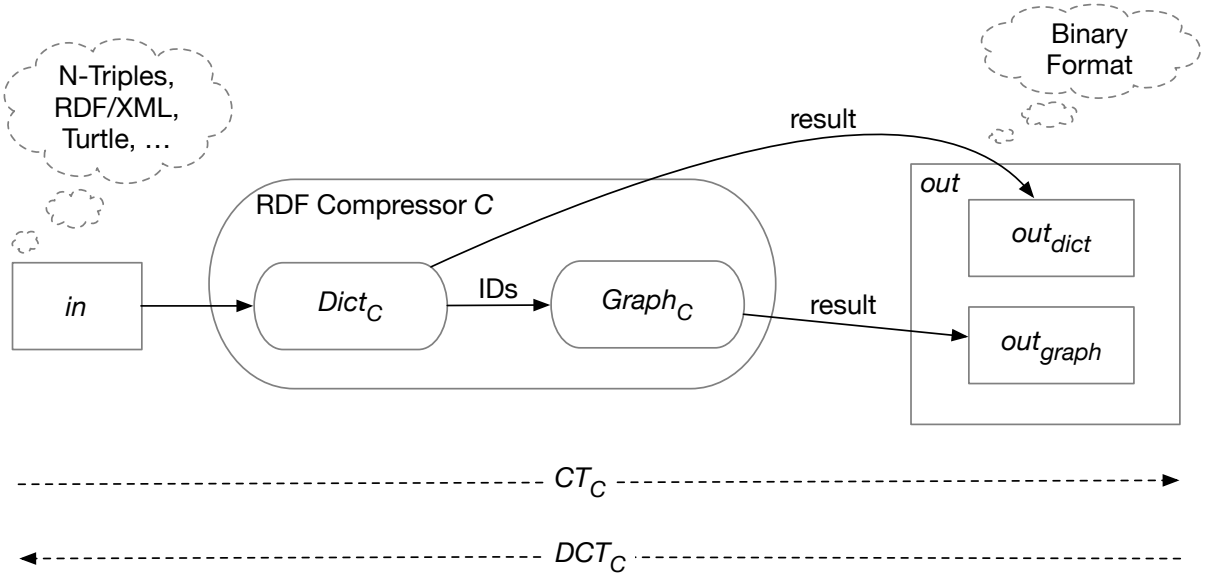


Figure 3.2: Visualization of the RDF Compressor Model.

3.1.2 RDF Compressor Model

This section 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 (whereas $|out| = |out_{dict}| + |out_{graph}|$). out_{graph} and out_{dict} can each be a single file or a set of files.

3.1.3 Existing RDF Compressors

Table 3.1 shows an overview of the RDF compressors discussed in this thesis and in which way they fulfill the model. 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}.

	$Dict_C$	$Graph_C$
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 the following two sections, key performance indicators for general and RDF compressors are introduced. Those indicators will be measured in to determine the overall performance for 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} and 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 dictionary or graph compression.
Sometimes CR is used instead of CR_C if it is clear from the context, which compressor C is considered.
If C is an RDF compressor, then CR_C is not always measured with respect to the whole output *out*, but sometimes only with respect to out_{graph} or out_{dict}. This is due to the fact that in some cases it is interesting to only consider the compression of the dictionary or the graph.
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 depends 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 section, the two existing compressors - HDT and GRP - will be compared. Therefore, the features of the compressors and their applicability to certain properties of RDF graphs are discussed.
The question is whether there are certain properties/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 Relation Between Structure of Data and Compression Ratio

First these features are considered for HDT. Fig. 3.3 is shown again. There, it is noticeable that the size of the data becomes smaller if there are only a few subjects. This is the case because the bit-array B_p contains a 1 every time a new subject is considered. For example, if there is only one subject, then B_p consists only of zeros.

stimmt anschein nicht

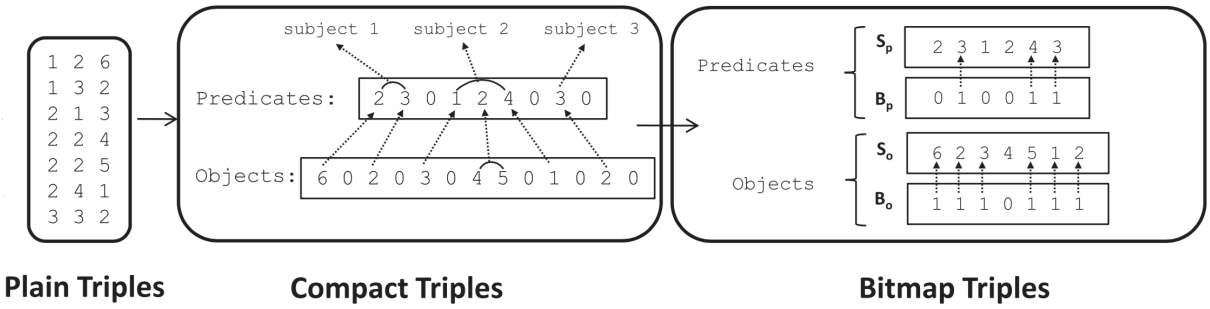


Figure 3.3: Three different representations of triples in HDT, figure from [FMPG+13].

For GRP that 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 will 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.

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. Therefore, Ch. 3.4.1 will mainly focus on improving $Graph_{GRP}$. But Ch. 3.4.2 is about the improvement of $Dict_{HDT}$ which is used for 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. 3.3.1 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.
This chapter is about elaborating the theoretical concepts of OWL and investigating how they can be used for grammar-based compression.
Let

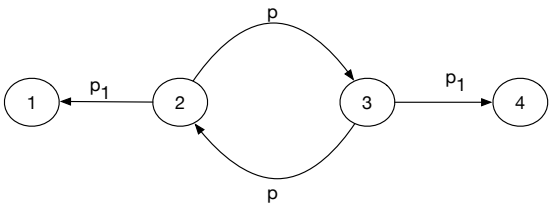
$$elr = \frac{\text{number of different edge labels}}{\text{number of edges}}$$

(edge label ratio) be the ratio of the edge labels or properties to the total number of edges of the graph.
Generally it can be said that GRP can compress a graph better if elr is lower, because then there it is more likely that there is much redundancy in the graph. However, if the graph structure becomes unfavorable for GRP, the compression ratio may still be worse at a lower value for elr.

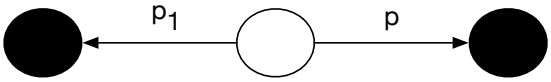
Symmetric Properties

eres Bsp

There is a class in [owl] called `owl:SymmetricProperty` [1] which expresses that a certain other predicate 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 adding it, which can lead to a better compression ratio. At the same time this will not result in an unfavorable structure. The procedure is illustrated in Fig. 3.4a. That graph shall be seen as a sub graph of a much larger graph. Here the predicate p is symmetric, so the edge from node 3 to 2 was added, p_1 is not symmetric. Due to the addition, the digram of Fig. 3.4b can now be found twice, whereas it was previously found only once. These two occurrences overlap and therefore cannot be applied both. However, it may be that one of the two occurrences cannot be replaced because the nodes involved are still connected to other nodes that are not shown in this figure. So the addition increases the probability that the digram can actually be replaced. At the same time the degree of nodes 2 and 3 is increased by one. But this should not really decrease the chance of finding other digrams in the graph, since 2 and 3 have already been connected before.



(a) A sub graph to which the edge from node 3 to 2 was added.



(b) The digram that can be found twice in the graph of Fig. 3.4a

Figure 3.4: Visualization of the benefits of adding symmetric edges to the graph. p is symmetric, p_1 is not symmetric.

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

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. The approach is similar to the symmetric properties. It is reasonable to argue in a similar way for adding those edges to graph here. It can decrease *elr* if there are many occurrences of a few inverse properties. Also, adding those edges will not really make the graph's structure more complex since all the nodes were already connected before.

Transitive Properties

hier noch ein bs
wo nach entfern
mehr digramme
gefunden werde

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 3, as illustrated in Fig. 3.5. Such a path, with the length of at least two, is called *transitive path* from now on.
The shown graph shall again be seen as some sub graph. The approach is to remove all triples (i, j) if a *transitive path* from i to j exists. 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 (they are not shown in Fig. 3.5).
The opposing approach is to add an edge from i to j if there is a *transitive path* from i to j. But a strong argument against this approach is that it would dramatically increase the number of edges, because for each pair (x, y) (with $x, y \in \{1, ..., n\}$ and distance between x and y greater than 1), an edge (x, y) 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.

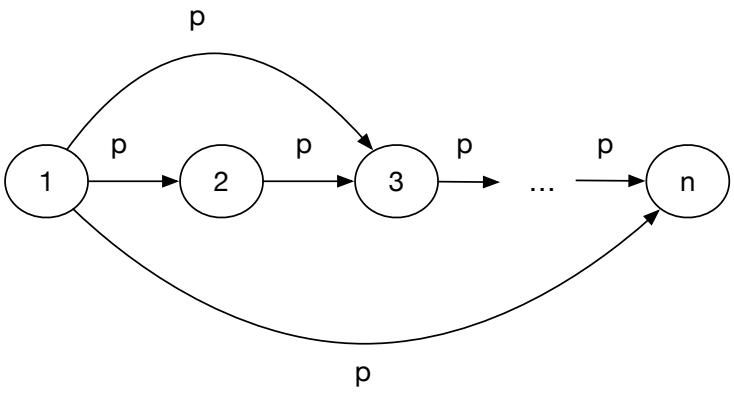


Figure 3.5: A sub graph with the transitive predicate p.

Equal Properties

In OWL there are the properties `owl:equivalentProperty` and `owl:sameAs`. The first one denotes that two properties are equivalent, but is 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

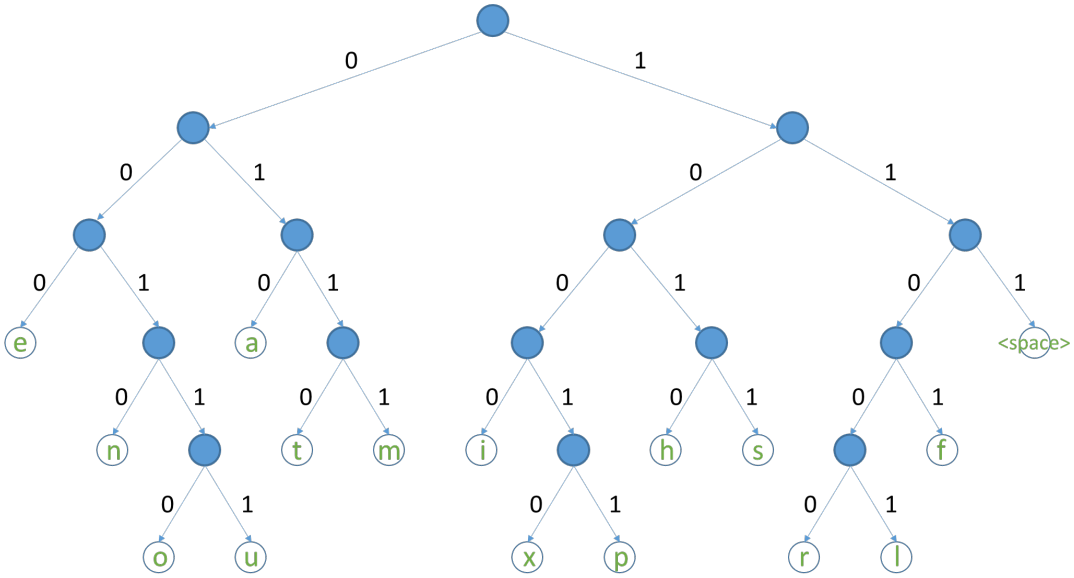


Figure 3.6: An example of a Huffman Tree.

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 a compression output. First results in Ch. 5 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 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 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. 3.6. 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 procedures 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 [2] (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 the HDT dictionary compression 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.

[2] 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 section, it will be explained how the comparison between HDT and GRP is implemented.

neu machen

As has just explained, HDT can compress a graph that is similar to a hub pattern (few subjects, many objects) very well. So it gets worse the further away the graph is from this pattern. This corresponds to the authority pattern, where there are few objects but many subjects.

The task now is to create a series of RDF graphs that first correspond to the hub pattern and then continue to change in the direction of the authority pattern. This is illustrated in Fig. 4.1. In this scenario all graphs (G_1 to G_m) have the same size, i.e. the same number of nodes and edges. G_1 has only one subject connected to all objects. G_2 then has two subjects more and correspondingly 2 objects less. This goes on and on 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.

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. Since the RDF graph also has the same number of triples, the files are of the same size. Since the evaluation compares the compression ratios for the different RDF files, it is important that all files are of the same size to ensure a fair comparison.

A section of such a file (for $G1$) is shown in Fig. 4.2. In that example, there is only one distinct predicate for all triples. The number of predicates is always one at first. The amount of predicates has a similar effect on both compressors and is therefore omitted at first. But in Ch. **??** that effect will be discussed in more detail.

Apart from that, blank nodes and literals are not used here. For both compressors, blank nodes and literals are being handled analogously to URI nodes. Therefore they are not needed at this point, in order to show the behavior of HDT and GRP in the above described scenario.

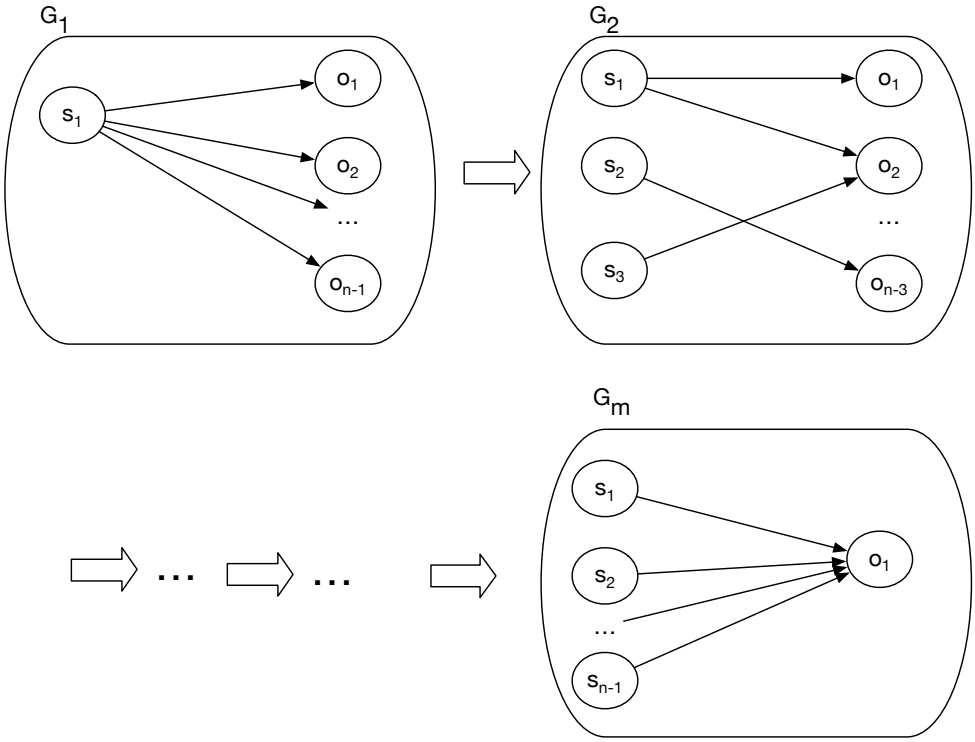


Figure 4.1: Step-by-step transition from hub pattern to authority pattern. The number of nodes n is the same for each graph. The number of edges is also the same for each graph.

```
<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 about 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 RDF researchers community. It contains data about their conferences and workshops.

DBPedia

DBPedia [2] is an RDF version of the knowledge from Wikipedia. It contains many different data files of a quite big size, with some of them having hundreds of millions of triples. Also, DBPedia includes an ontology that is used for the evaluation.

Wordnet

Wordnet contains knowledge about the English language. Words that have the same meaning are grouped in to the same "synset". Those sets are the nodes in an RDF graph and relations between them are edges/properties.

Nuts-RDF

The NUTS (Nomenclature of Territorial Units for Statistics) [3] 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.

4.2.2 Ontology Knowledge

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

Gathering Relevant Properties

First, all relevant 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 [4] are given. Now it is possible to check in the Wikidata ontology whether the properties are symmetric or inverse, because there the information is given.
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 antonymy, which is like an opposite-relation between words. Therefore, the property can be seen as symmetric.

se bei wordnet

After all symmetric properties have been found (they are shown in Ch. 5), manipulations to the graph have to be executed.

[1] http://www.scholarlydata.org/dumps/
[2] https://wiki.dbpedia.org/Downloads2015-04
[3] http://nuts.geovocab.org/
[4] https://www.wikidata.org/wiki/Wikidata:Main_Page

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 executed. That delete is shown in Listing 4.2.

```
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.

Here it has to be taken care that not to delete both directions, this can be done by using a filter.

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.

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.

```
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.

```
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.

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 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 now 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 with a set of desired properties). Analogously to the other approach, we gather all triples e is

```
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.

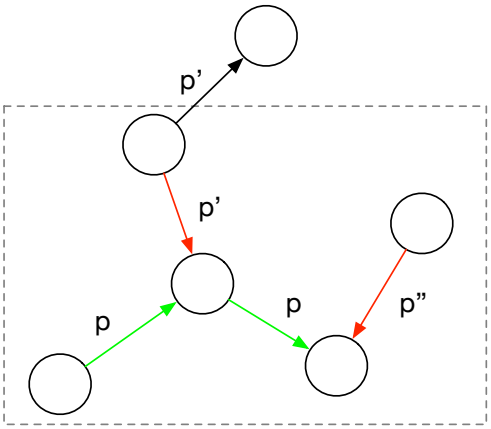


Figure 4.3: Extraction of a sub graph (marked by dashed line).

part of and then add triples which the newly gathered entities $e_1, ..., e_n$ are part of. That step is repeated a few times until some defined number of triples has been reached. The result graph is called e-based subgraph.

4.2.3 Dictionary Improvements

For the dictionary improvements, HDT's dictionary compression is used as a basis. Therefore, the HDT-Java code [5] 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 Algorithm 2 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's character in case the current node is 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 2 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)
```

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

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.
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 directly 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

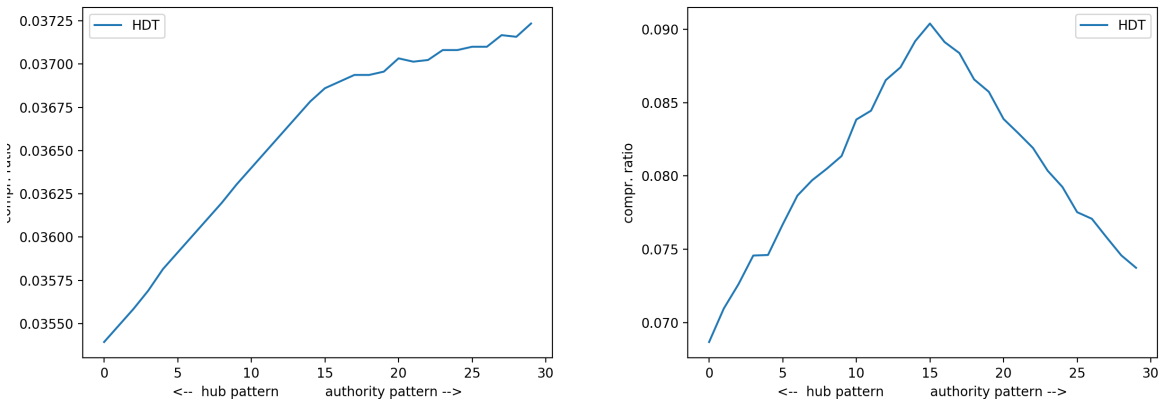
This chapter is about the evaluation of the several approaches presented in Ch. 3 and 4. For the following evaluations HDT-Java 2.0 [1] (the currently newest version) has been used. For GRP the implementation presented in Ch. 2.3.2 has been used. The evaluated datasets are the ones introduced in Ch. 4.2.1.

5.1 GRP vs HDT

alles nicht
wegen neuer
ntnis

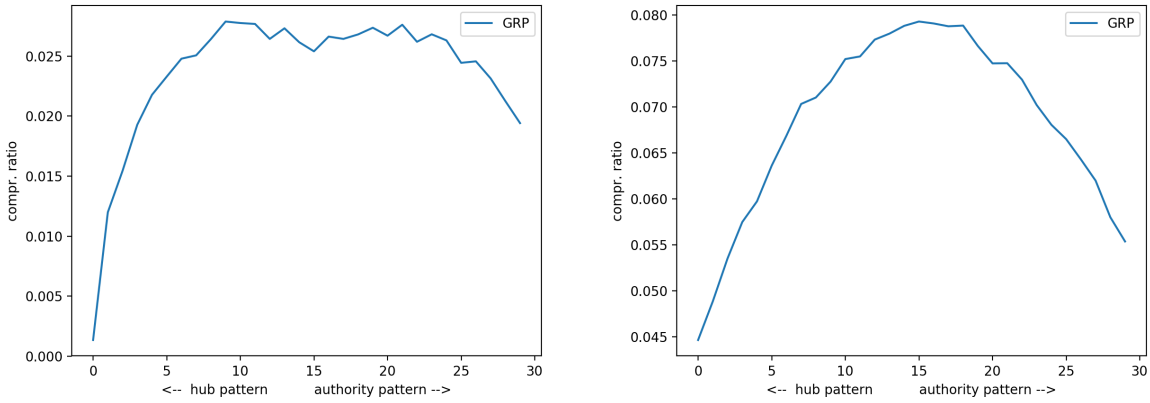
Fig. 5.1a shows $CR_{Graph_{HDT}}$. As expected, $CR_{Graph_{HDT}}$ gets higher the more similar the graph is to the authority pattern. In general it can be said that this effect is quite small. There is only a distance of 0.002 between the minimum and the maximum. This small effect can also be seen by looking at Fig. 5.1b. It can be seen that the size of the dictionary (out_{dict}) has a much bigger effect, since $CR_{HDT}(out)$ is much larger than $CR_{Graph_{HDT}}$ and the curve behavior from Fig. 5.1a is no longer recognizable. It is noticeable that $|out_{dict}|$ is bigger when the graph is further away from the star pattern. $Dict_{HDT}$ seems to be more inefficient when there are about as many subject as objects.
Next, the compression ratio of GRP ($CR_{Graph_{GRP}}$) is considered, which is presented in Fig. 5.2a. Here, it is noticeable that GRP has a better compression ratio if the graph is more similar to the star pattern (hub or authority pattern). This property of GRP has also been mentioned in [MP18]. It can also be seen that the effect is higher on $CR_{Graph_{GRP}}$ than on $CR_{Graph_{HDT}}$ (standard deviation is twice as high for $CR_{Graph_{GRP}}$ as for $CR_{Graph_{HDT}}$).
When Fig. 5.2b is considered, it can be seen that this curve behaves almost exactly like the one from Fig. 5.1b, since $|out_{dict}|$ accounts for most of $|out|$.
Finally, Fig. 5.3a and Fig. 5.3b show CR_{Graph_C} and CR_C, respectively for GRP and HDT. Since both use $Dict_{HDT}$ to compress the dictionary, the curves in Fig. 5.3b are very similar. However, it becomes clear that GRP compresses better than HDT. In Fig. 5.3a, $CR_{Graph_{HDT}}$ is 31 times higher than $CR_{Graph_{GRP}}$ on average. Of course, this factor becomes much smaller in Fig. 5.3b because the dictionary accounts for most of the memory size. Here, the CR_{HDT} is on average 1.8 times as high as CR_{GRP}
It would be possible to argue that only one distinct predicate was used in that scenario and this is beneficial for GRP, as it gets worse as the number of predicates increases. Therefore a further evaluation is made in Fig. 5.4, where 1000 distinct predicates have been used. That is

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



(a) $CR_{Graph_{HDT}}$. (b) CR_{HDT}.

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



(a) $CR_{Graph_{GRP}}$. (b) CR_{GRP}.

Figure 5.2: The compression ratios for GRP without and with dictionary sizes.

a quite high number considering the number of triples (1199) compared to real RDF data . It is visible that the compression ratios are now higher for both compressors, but still CR_{GRP} is always smaller than CR_{HDT}. CR_{HDT} is still 1.7 times higher on average. So, the increasing number of predicates has a similar effect on both algorithms.

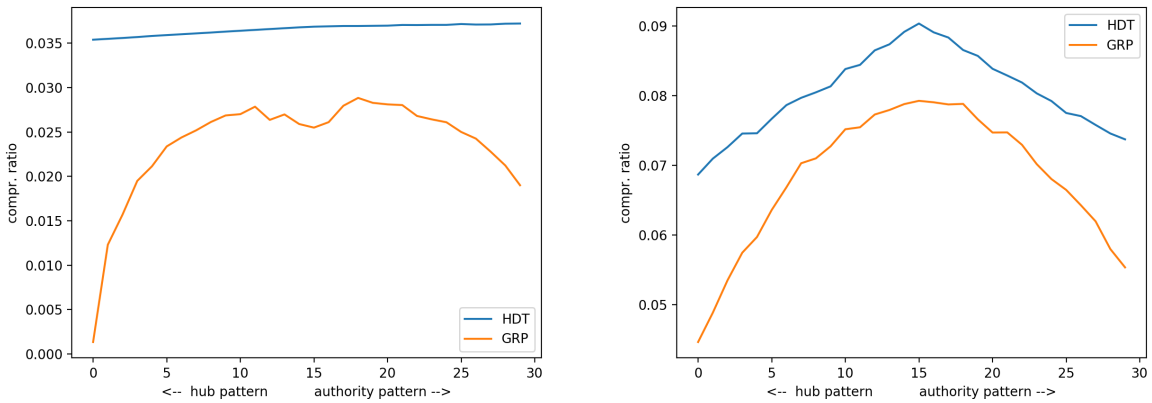
belegen

figure passt nicl

Apart from the compression ratio, the run time is also important for the overall performance. Fig. 5.5 shows the average value of CT for both compressors, respectively. For this 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 auch messen

It can be seen that CT_{GRP} is significantly higher than CT_{HDT}. It is on average ca. 48 times as high.
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



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

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

Figure 5.3: The compression ratios for GRP and HDT without and with dictionary sizes.

time. So they are not comparable in terms of quality. Unfortunately, it is not possible say at this point whether a more professional implementation of GRP will also be slower than HDT. In addition it can be noticed that CT_{GRP} fluctuates more than CT_{HDT}. CT_{GRP} has a standard deviation of about 134, while CT_{HDT} only has a standard deviation of about 7. One reason for this is that GRP, in contrast to HDT, is non-deterministic because of the partly random search order of the graph. On the other hand, the high deviations are also a confirmation of the above mentioned hypothesis that the behavior of GRP depends more on the structure of the input data than HDT does.

5.2 Compression Improvements

This chapter is about evaluating the compression improvements. It starts with applying ontology knowledge and then continues with dictionary compression improvement. In contrast to Ch. 5.1, real data will be evaluated here to see how much influence the improvements have.

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 size of the output out_{graph} will be measured. Since these approaches have the intention to improve the ability for GRP to produce a smaller grammar, the features of that grammar will be presented in more detail. That is because after the encoding of the grammar it might happen that despite the fact that the grammar is smaller after the manipulations that effect is not visible in the value of $|out_{graph}|$.

Occurrence of Properties

Now, it will be presented how much of those symmetric/inverse/transitive properties exist in real data. For that we will use the datasets presented in Ch. 4.2.1.
Fig. 5.6 shows how often the relevant properties occur in real datasets (DBPedia and Wordnet). (Relative amount $= \frac{\text{number of occurrences}}{\text{number of triples}}$). It can be noticed that the amounts are quite low, especially for symmetric and inverse properties. This supports the hypothesis from Ch. 4.2.2

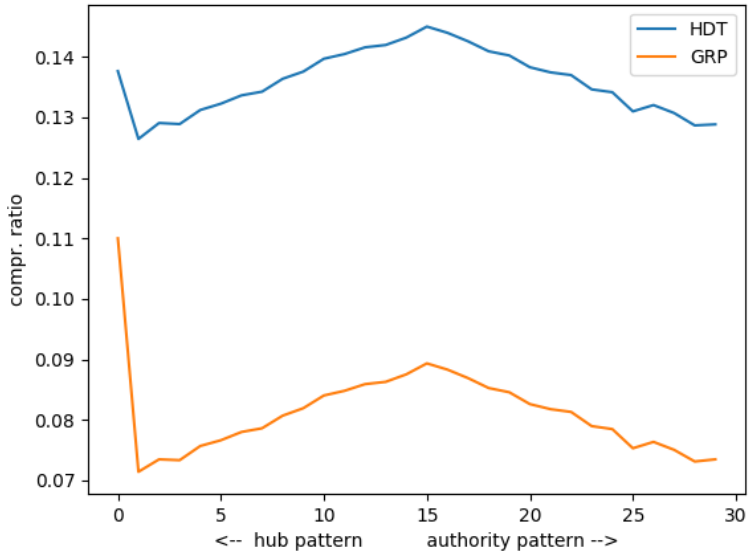


Figure 5.4: CR_{HDT} and CR_{GRP}. Graphs have now 1000 distinct predicates.

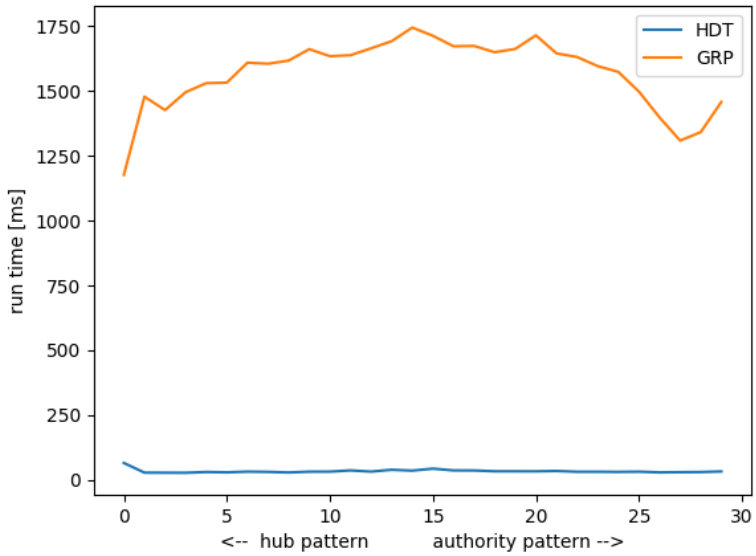


Figure 5.5: CT_{HDT} and CT_{GRP} (Average run time of 100 consecutive executions).

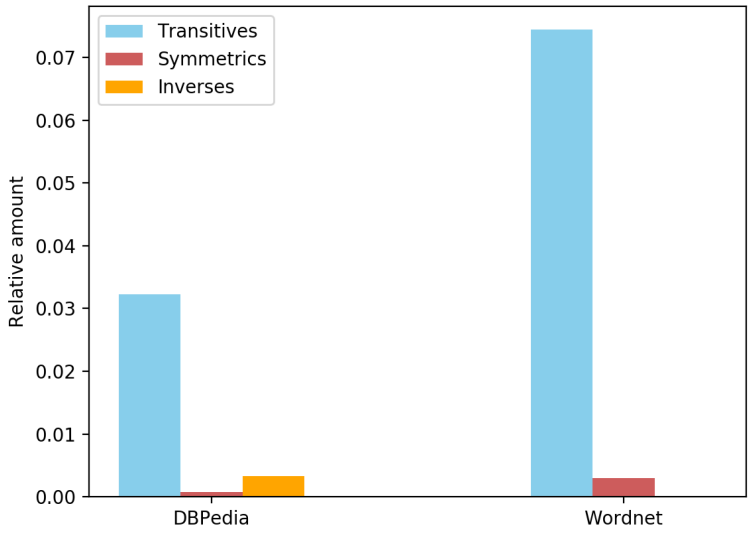


Figure 5.6: Relative amount of transitive/symmetric/inverse properties in real datasets.

that building sub graphs with higher amounts of relevant properties can be necessary to observe a real effect of the data manipulations. Nevertheless, first the normal datasets will be used to evaluate the manipulations.

diwe zeigen, interessante zu selten eten

Therefore, for the following evaluations of Ch. 5.2.1, we use sub graphs as described in Ch. 4.2.2. More precisely, we use graphs that contain half triples with the relevant properties and the other half triples connected with them. By doing that, the effects will be significant and it will be clear whether they lead to a better or worse compression ratio.

Evaluation Process

Now, the overall process of applying ontology knowledge and evaluating whether it results in a better compression is explained. The several parts of the process will be discussed in the next sections. Fig. 5.7 shows how it is implemented. 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. Then the manipulated graph is given to GRP. In addition, the relevant ontology triples have to be compressed and stored as well. Otherwise, the original graph could not be restored. Both graphs together are merged together into one graph called in_2 which is then compressed by GRP and that delivers 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. Finally, it is also possible to execute the process for one graph in which all different manipulations have been applied. This way, it is noticeable what the manipulations bring individually and in the end also, what they all achieve in combination.

e das gemacht?

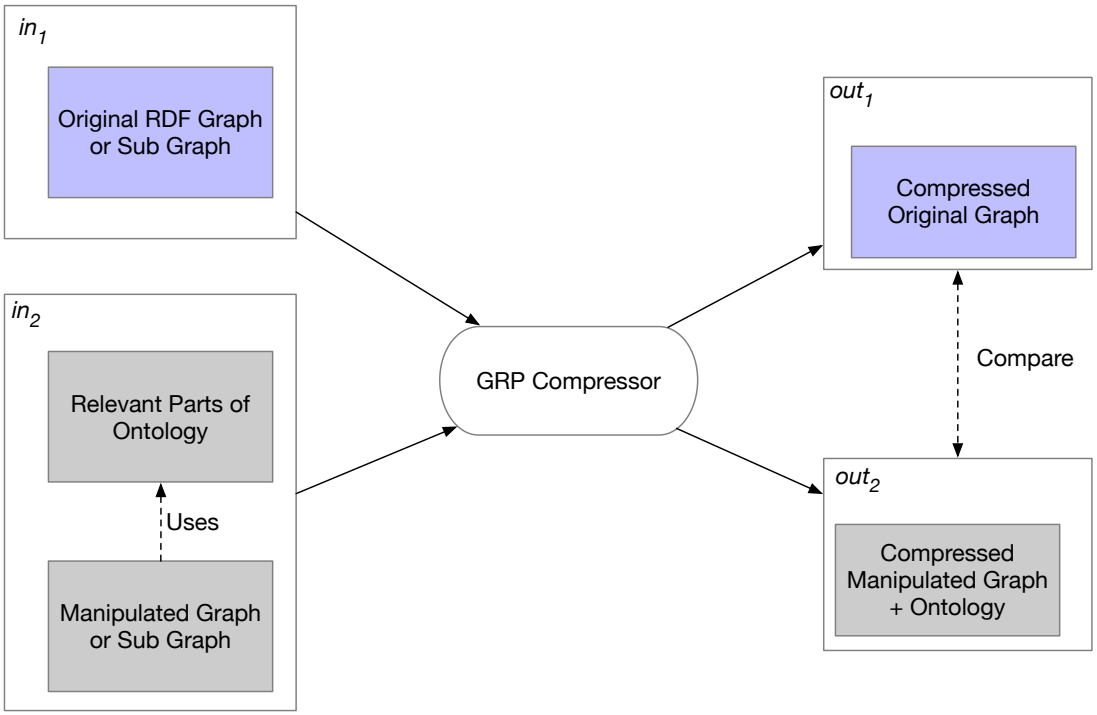


Figure 5.7: Overall process of applying ontology knowledge and comparing the compression results.

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 by enabling $Graph_{GRP}$ to produce a smaller grammar, it is interesting to not only compare out_1 and out_2 at the file size level, but also in terms of their grammar/graph sizes. The most significant metric for the size of a grammar/graph is its number of edges. In order to compare out_1 and out_2 with respect to their grammar/graph edge amounts, the metrics Input Edge Ratio (IER) and Output Edge Ratio (OER) are defined:

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

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

IER shows how many edges have been added or removed by the manipulations 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 re-using CR_{GRP} as defined in Ch. 3.1.2. For out_1, CR_{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:

$$SR = \frac{|out_2|}{|out_1|}$$

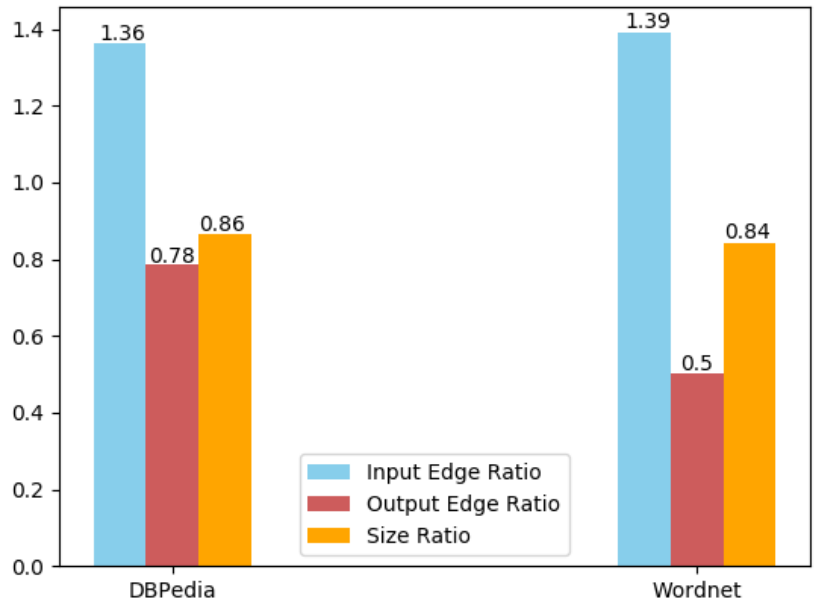


Figure 5.8: Results for adding symmetric properties, on the grammar level (IER, OER) and on the file size level (SR).

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 real amount necessary 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. 5.8 illustrates the results. 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 $Dict_{GRP}$ and even brings a significant improvement on the grammar level as OER is about 0.8 for DBPedia, and about 0.5 for Wordnet.
When SR is considered, it can be seen that $SR < 1$ 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 2.3.2, that the encoding does not always work well, is supported by these results.
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. 5.9). There, it is possible to see 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 $Graph_{GRP}$. However, 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 graph out_2 from Fig. 5.9 is taken as the input in_1

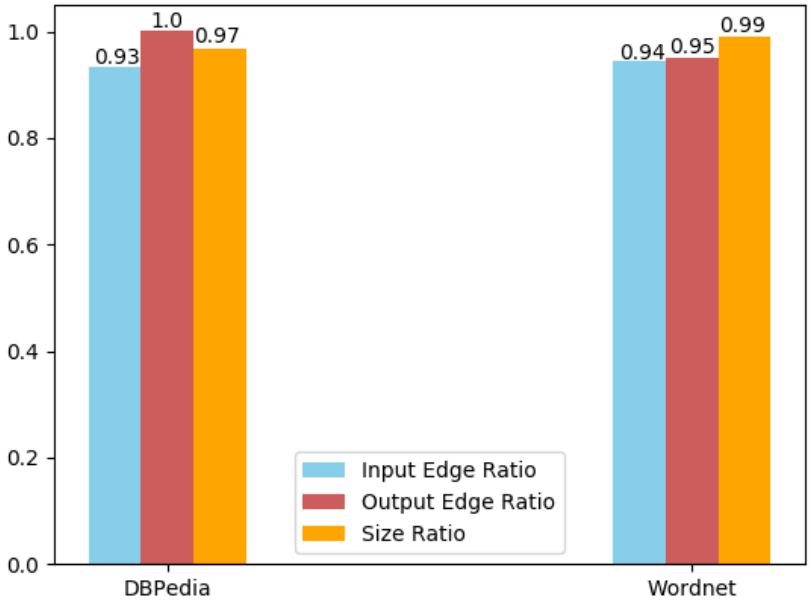


Figure 5.9: Results for deleting symmetric properties, on the grammar level (IER, OER) and on the file size level (SR).

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, since in_1 is a graph where all symmetric edges have been removed before hand.
The evaluation results are illustrated in Fig. 5.10. Of course, $IER > 1$ is true and IER is bigger than in Fig. 5.8, since now a higher amount of edges has been added. The fact that both OER and SR are lower than 1, support 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 grammar.

Inverse Properties

Transitive Properties

The next 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 DBPedia, we could not find a graph in which *transitive paths* exist (defined in Ch. 3.4.1). Thus, only Wordnet will be used here, since it contains *transitive paths* .
First, the approach suggested in Ch. 3.4.1, to remove transitive edges (i, j) if there exists a *transitive path* from i to j, is evaluated. Unfortunately, those edges do not exist in Wordnet. To solve 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, transitive edges are added whenever a *transitive path* exists between two nodes. Afterwards, the manipulated first half is merged with the untouched second half again. This gives us a graph in which both adding and removing transitive edges is possible. So, both approaches can be evaluated.

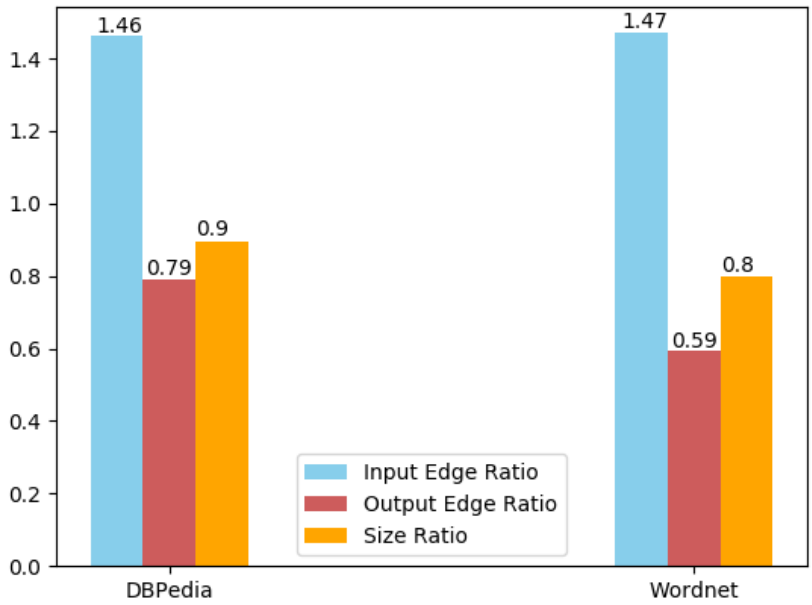


Figure 5.10: Results for adding symmetric properties, on the grammar level (IER, OER) and on the file size level (SR). In contrast to Fig. 5.8, the input graph in_1 contains only one direction for each triple with a symmetric property.

The results for removing transitive edges can be seen in Fig. 5.11. Since only Wordnet is evaluated here, Fig. 5.11 only consists of one figure, which contains all metrics. 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 transitive edges, $Graph_{GRP}$ could indeed find more patterns, as stated in Ch. 3.4.1. On the file size level, the improved compression is also visible, because $SR \approx 0.88$ is also less than 1. Again, the improvement is not as good as on the grammar level.

Next, the opposing approach (adding transitive edges) is evaluated. The results are illustrated in Fig. 5.12. Obviously, $IER > 1$ holds, since edges have been added. $OER \approx 3.5$ is significantly high, which means that adding transitive edges has resulted in a much larger end result, on the grammar level. In contrast to Fig. 5.11, $IER < OER$ holds here, but that supports the hypothesis that adding transitive edges makes $Graph_{GRP}$ find less digrams (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 transitive edges results in a better compression, and adding them delivers a worse compression.

5.2.2 Dictionary Improvements

At this point about the evaluation of the approaches from Ch. 4.2.3. First, the Huffman compression of literals is evaluated and then the compression of blank node IDs.

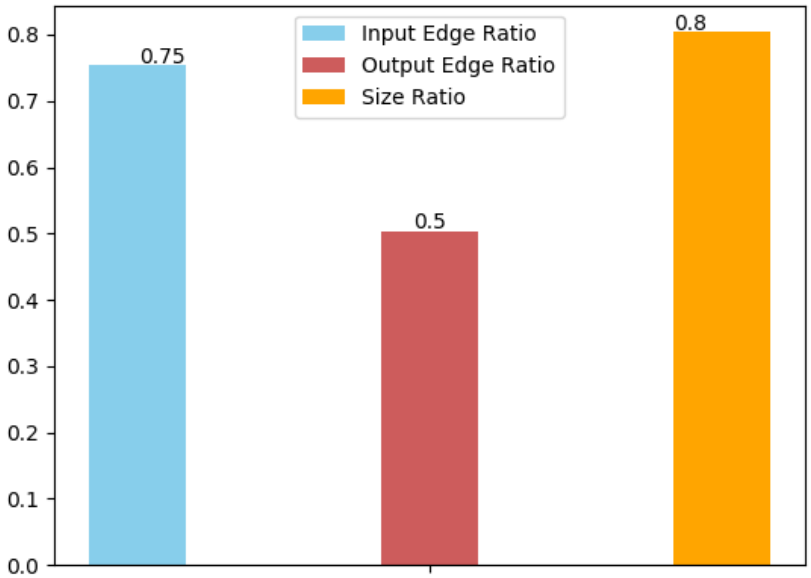


Figure 5.11: IER, OER and SR for removing transitive edges in Wordnet.

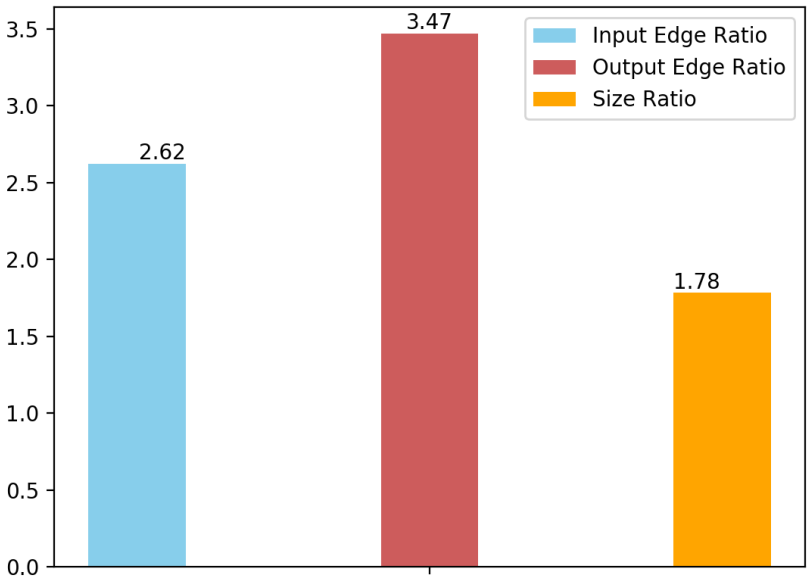


Figure 5.12: IER, OER and SR for adding transitive edges in Wordnet.

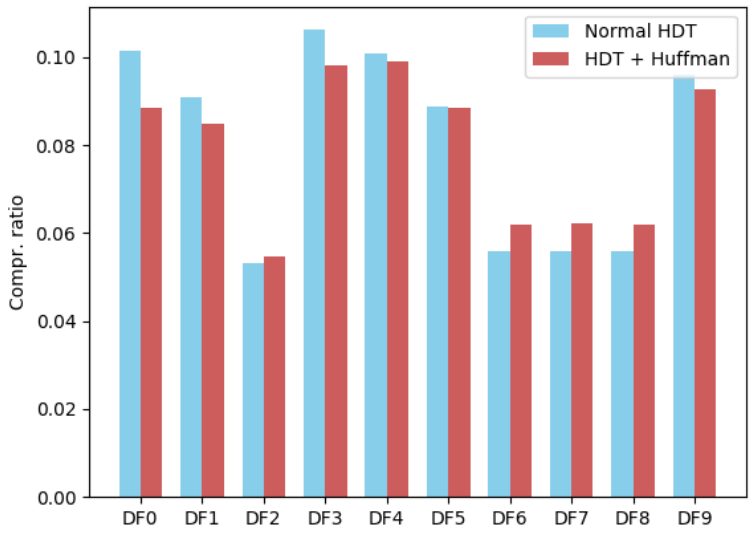
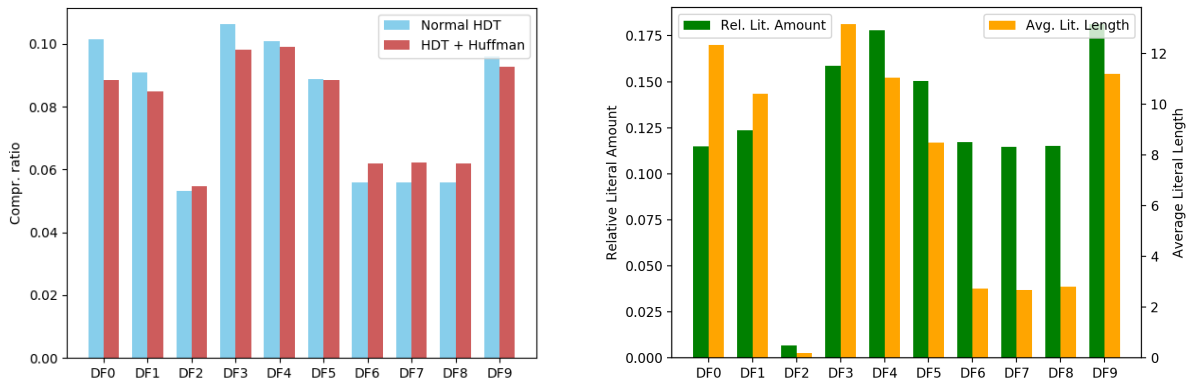


Figure 5.13: $CR(out_{dict})$. Comparison between Normal HDT and HDT + Huffman.

Literals

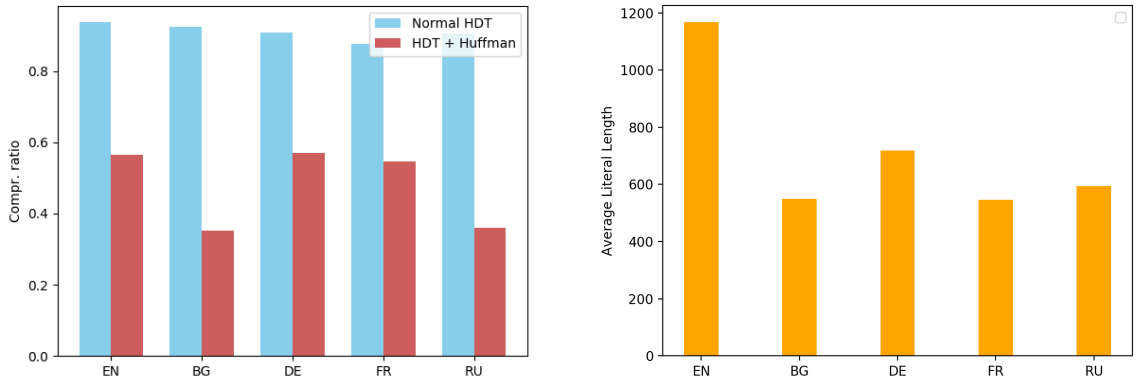
As already mentioned, a self-generated Huffman code was used to compress the literals of an RDF graph, as it is likely to get a better compression than the with the prefix-based compression of HDT.
Now the results of the evaluation are presented. First, the Semantic Web Dog Food data set is used. This is well suited for an evaluation, since both literals and URIS are included as objects. Fig. 5.13 shows $CR(out_{dict})$ for a subset of the data (DF0 - DF9).
It can be seen that the use of the Huffman code only brings a small improvement in some cases. In some cases Normal HDT even compresses better than HDT with Huffman.
This is because these data do not have a very high proportion of literals and literals are also rather short. This can be seen in Fig. 5.14, where Fig.5.14a lists $CR(out_{dict})$ again and Fig.5.14b shows the relative proportion of literals 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. So 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. with DF9).
Now 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. 5.15a shows $CR(out_{dict})$ for the abstract files. The abbreviations stand for the languages in which the abstracts are written. It can be seen that Huffman significantly improves $CR(out_{dict})$ here. The reason for this can be seen in Fig. 5.15b, where the average literal length is illustrated. The relative proportion of literals 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 by far the longest literals, 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



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

Figure 5.14: Relation between literal percentage and literal length (right) and $CR(out_{dict})$ (left) for Semantic Web Dog Food Data.

than in the other languages. Huffman is generally more efficient on large alphabets, according to [Sha10].



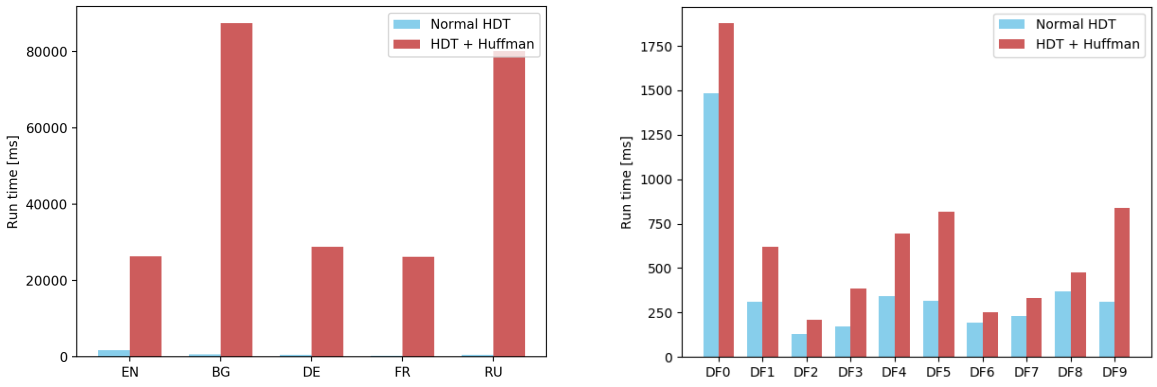
(a) $CR(out_{dict})$ for Normal HDT and HDT + Huffman. (b) Average literal length.

Figure 5.15: Relation between literal length (b) and compression ratios (a) for DBPedia Abstracts (Abbreviations denote the languages the abstracts are written in).

evtl Formel ab wann sich huffm lohnt

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. 5.16a shows CT for the compression of the DBPedia abstract data. Here, CT has been increased very much. This is because HDT can normally compress very little with this data because of the many long literals and is therefore finished quite quickly. With Huffman the data can be compressed much better and it takes quite a long time.

Fig. 5.16b shows CT for the Semantic Web Dog Food data, where the run times are only slightly longer, because Huffman hardly improves the compression.
So it can be said that the use 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 5.16: Run times (CT) for Normal HDT and HDT + Huffman.

Blank Nodes

This chapter evaluates the two approaches mentioned in Ch. 3.4.2 to store blank node IDs more efficiently. For that, the Nuts-RDF dataset is used as it contains solely blank nodes as subjects and objects. It is therefore prone to show the inefficiency of the prefix-based compression of $Dict_{HDT}$ with respect to blank nodes. Fig. 5.17 shows the results for three different files from Nuts-RDF. 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 brings a significant improvement as it delivers a compression ratio between 0.76 and 0.89. So, the compression is now really reducing the size of the files. Omitting the blank node IDs results in 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.

5.3 Final Result

evtl einen DB-
a Datensatz
nen, wo HDT
er besser war
GRP wegen
netry jetzt
er ist

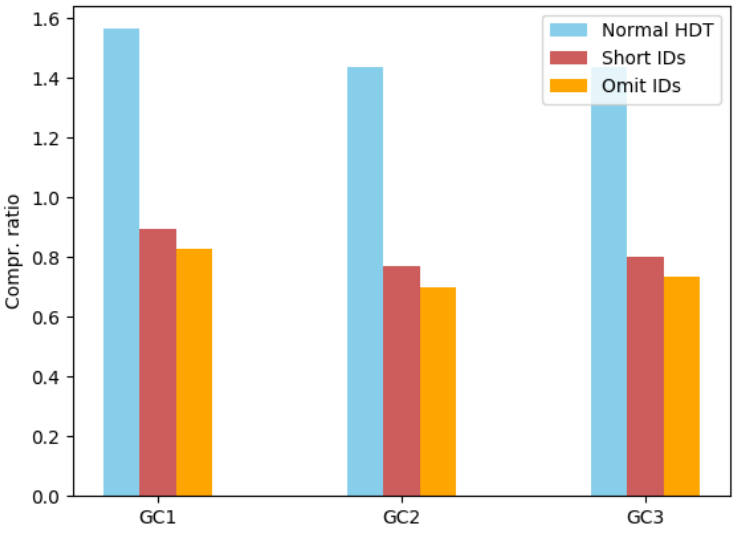


Figure 5.17: Results for Blank Node ID improvements for Geo Coordinates (GC) files. **Normal HDT** uses long IDs. **Short IDs** uses integer IDs. **Omit IDs** does not store the IDs.

6 Summary and Future Work

This last chapter concludes the thesis by first summarizing its most important results in a compact form. Secondly, it is then discussed what future work might be interesting to do in this area.

6.1 Summary

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 compressor 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 by equal properties or entities, respectively. This would enable $Graph_{GRP}$ to produce a better compression.

- 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 good to have a separate compression technique for each of those data types. That would probably result in a better compression ratio.

List of Figures

List of Acronyms

RDF	Resource Description Framework
HDT	Header Dictionary Triples
GRP	GraphRePair
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
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.

[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.

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

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

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

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