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

Grammar-based Compression of RDF Graphs

by
Philip Frerk

Thesis Supervisors:

Prof. Dr. Axel-Cyrille Ngonga Ngomo

Prof. Dr. Stefan Böttcher

Paderborn, May 20, 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 *subject* and *objects* are nodes and *predicate* is an edge 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 accessible for queries. 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.

evtl Beispiele
knowledge
s

The currently most popular compressor for RDF data is HDT [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 of the well known is GraphRePair [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. Storage

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

2. Transmission

3. Processing/Consumption

The idea is that one can make all these use cases easier / possible by compression, especially if one is dealing with very large amounts of data. 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 tasks 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.

k nodes
hreiben

Blank nodes are normally used when a node does not get a URI, but is still necessary to represent a statement. Such a node is often used to formulate more complex logical statements.

2.1.1 Ontology

hen

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 accessible for queries. The idea behind it is to make RDF's extensive representation more compact, thus reducing memory size. Usually, in many plain text formats for RDF (e.g. N-Triples [1]) each triple is written down after another. 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.

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

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:

$$\begin{aligned}(s, p_1, o_{11}), ..., (s, p_1, o_{1_{n1}}), (s, p_2, o_{2_1}), ...,\\ (s, p_2, o_{2_{n2}}), ..., (s, p_k, o_{k_{n_k}})\end{aligned}$$

In HDT the triples are then grouped by s.

$$\begin{aligned}s \rightarrow [(p_1, (o_{1_1}, ..., o_{1_{n_1}})), (p_2, (o_{2_1}, ..., o_{2_{n_2}})), ...,\\ (p_k, (o_{k_{n_k}}))]\end{aligned}$$

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.

First, all URIs and literals (which are usually quite long) are mapped to unique IDs (integers). From now on these IDs will be used. One can see the triples with the IDs in Fig. 2.1 on the left ('Plain Triples').

The above mentioned grouped representation is called 'Compact Triples' and can be seen in Fig. 2.1 in the middle. In the array 'Predicates' one can see 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 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.

dict compr erkä

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.

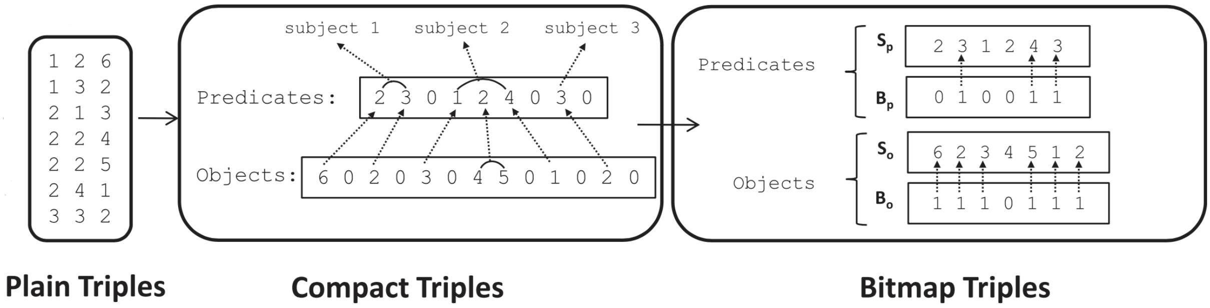


Figure 2.1: Three different representations of triples in HDT, figure from [FMPG$^+$13].

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 one can usually 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.

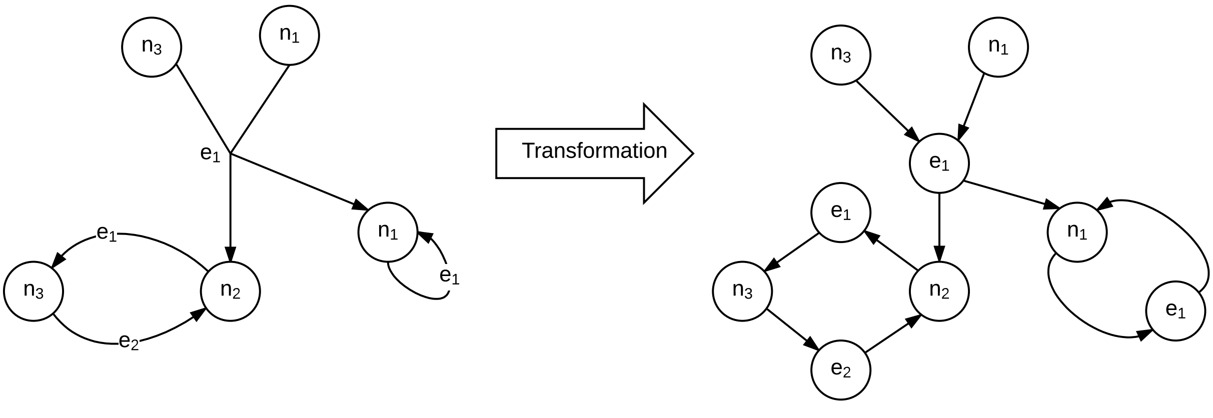


Figure 2.2: Transformation of a labeled graph (each edge is replaced by a new node having the label of the replaced edge).

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'. 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 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.

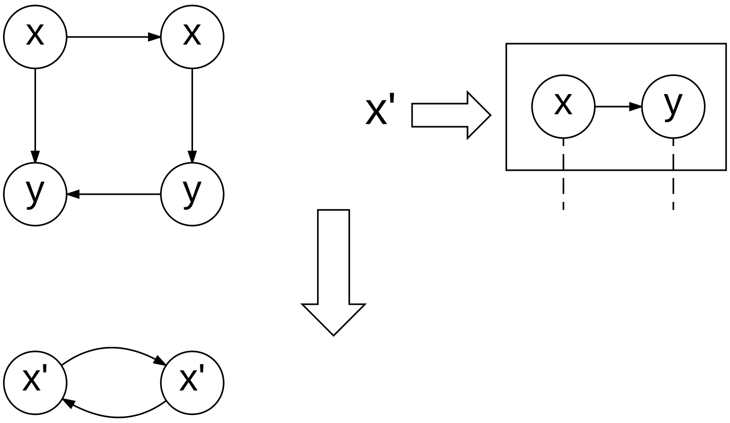


Figure 2.3: Replacement of two occurrences of the digram $X \rightarrow Y$ by nodes with the label X'.

2.3.2 GraphRePair

This chapter deals with GraphRePair, the compressor from [MP18].

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. $att : E \rightarrow V^+$ is the attachment mapping assigning start and end nodes to all edges. $lab : E \rightarrow \Sigma$ 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]

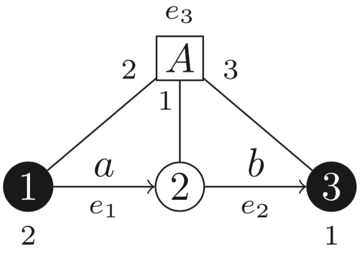


Figure 2.4: A hyper graph as defined in [MP18]. The black nodes 1 and 3 (consider the numbers within the nodes) are external nodes whereas the white node 2 is internal. e_3 (with the label A) is a hyper edge while e_1 and e_2 (labeled with a and b, respectively) are normal edges.

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

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

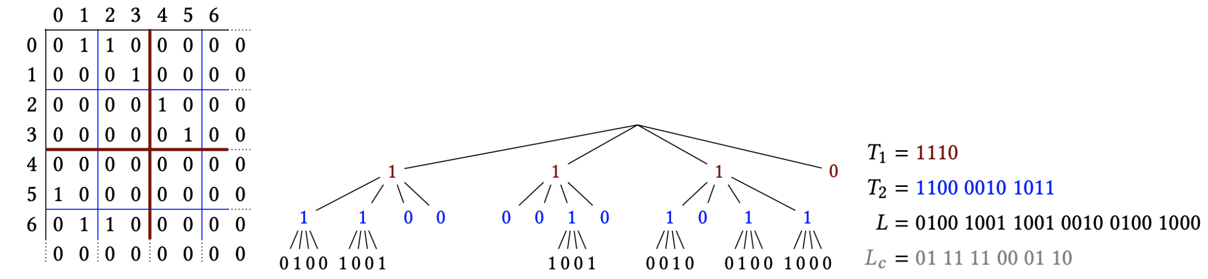
Digram occurrences are concrete instances of a digram.
Let g be a hyper graph and d be a digram with the two edges e_1^d, e_2^d. Let $o = \{e_1, e_2\} \subseteq E_g$ and let V_o be the set of nodes incident with edges in o. Then o is an occurrence of d in g if there exits a bijection $b : V_o \rightarrow V_d$ 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



(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.

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 $g = (V, E, att, lab, ext)$)

```
$N, P \leftarrow \emptyset$
$S \leftarrow g$
$L(d) \leftarrow$ list of non-overlapping occurrences of every digram $d$ appearing in $g$
while $|L(d)| > 1$ for at least one digram $d$ do
    $mfd \leftarrow$ most frequent digram
    $A \leftarrow$ new non terminal for $mfd$
    Replace every occurrence of $mfd$ in $g$
    $N \leftarrow N \cup \{A\}$
    $P \leftarrow P \cup \{A \rightarrow mfd\}$
    Update the occurrence list $L$
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 procedures continues until there is zero-leaf for each path of the tree.

Afterwards, the tree is represented by bit-strings.

Since the graph has also edge labels, an adjacency matrix and its corresponding tree will be created for each of those labels.

einfachere Grafi
oder alles erklä

The remaining productions of the grammar are encodes 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

3 Approach

This chapter contains the solution approaches to the parts of the thesis. Firstly, a formal model of an RDF 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 RDF Compressor Model

This section introduces a formal model for an RDF compressor, that can be applied to both HDT and GRP and is illustrated in Fig. 3.1.
Let $C \in \{HDT, GRP\}$ be a compressor. C takes an RDF file *in* as input and produces an output $out = \{out_{graph}, out_{dict}\}$, which contains the compressed RDF graph and the compressed dictionary. out_{graph} and out_{dict} can each be a single file or set of files.
Let m be a single file or a set of files. Then $|m|$ is defined as the size which is measured in bytes.

3.1.1 Compression Ratio

One of the key performance indicators for a compressor C is its compression ratio (CR_C). CR_C depends on the input data *in* and is defined as follows:

$$CR_C(o) = \frac{|o|}{|in|} \text{ with } o \in \{out, out_{graph}, out_{dict}\}$$

Sometimes $CR(o)$ is used instead of $CR_C(o)$ if it is clear from the context, which compressor C is considered.
CR_C is in the $(0, \infty)$ interval, since the compressed data can potentially be arbitrarily large. Obviously, the compressed data cannot have a size of 0 or less.
CR 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. 3.1, *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 compressor is compared, their input has to have the same format.

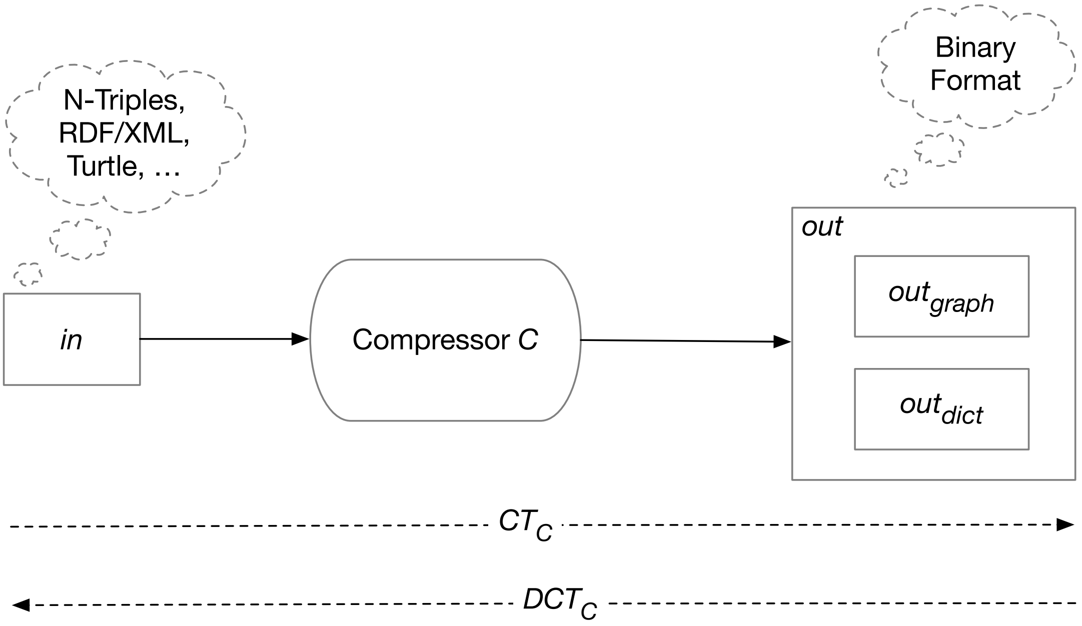


Figure 3.1: Visualization of the Compressor Model.

3.1.2 (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.

3.1.3 How GRP and HDT fit in the Model

HDT is compressor made for RDF and therefore produces both out_{graph} and out_{dict}. GRP only produces out_{graph}. In order to compare CR_{HDT} and CR_{GRP} in a fair way we use the same out_{dict} for both algorithms.

3.2 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.

3.2.1 Relation Between Structure of Data and Compression Ratio

First these features are considered for HDT. Fig. 3.2 is shown again. There one can see 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.

stimmmt anschein
nicht

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

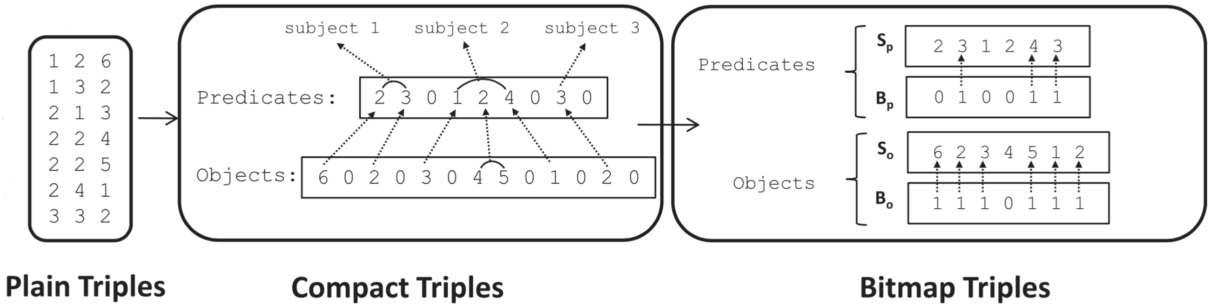


Figure 3.2: Three different representations of triples in HDT

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.

3.2.2 Dictionary Size

en ob sectoin
,passt nicht

As already explained in Ch. 2, HDT divides an RDF file into its header, dictionary and triples component. This is partially also true for GRP except that GRP does not create a header. But it also assigns an ID to every URI or literal and then only works with these IDs. Unfortunately the authors of [MP18] did not work on efficient storing of the dictionary. In GRP the size for the dictionary component is just ignored. Therefore we have to add this size in order to compare GRP with HDT in a fair way. To achieve that we just add the same size to the compressed size of GRP that HDT would need to store the dictionary.

3.3 Compression Improvements

First comparisons of GRP and HDT (see Ch. 5.2) showed that GRP achieves a better compression ratio. Therefore, improving the compression ratio will mainly focus on improving GRP, although parts of this work can be used to improve HDT as well.

3.3.1 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. 3.2.1 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

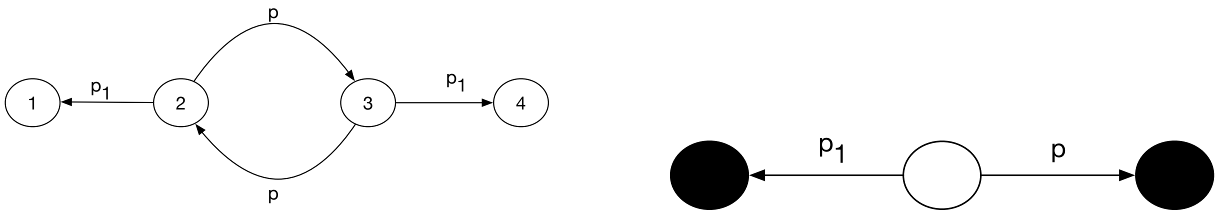
$$elr = \frac{\text{number of different edge labels}}{\text{number of edges}}$$

(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

besseres 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 one make *elr* smaller by adding it, which can lead to a better compression ratio. At the same time one should not get an unfavorable structure. The procedure is illustrated in Fig. 3.3a. 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. 3.3b 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. 3.3a

Figure 3.3: Visualization of the benefits of adding symmetric edges to the graph. p is symmetric, p_1 is not symmetric.

Inverse Properties

A 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. One can 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

[1] The prefix `owl:` stands for...

those edges will not really make the graph's structure more complex since all the nodes have been connected before.

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 3, as illustrated in Fig. 3.4. The shown graph shall again be seen as some sub graph. The approach is to remove the triple $(1, p, n)$. This is sensible, as it gives the nodes 1 and n 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. 3.4).

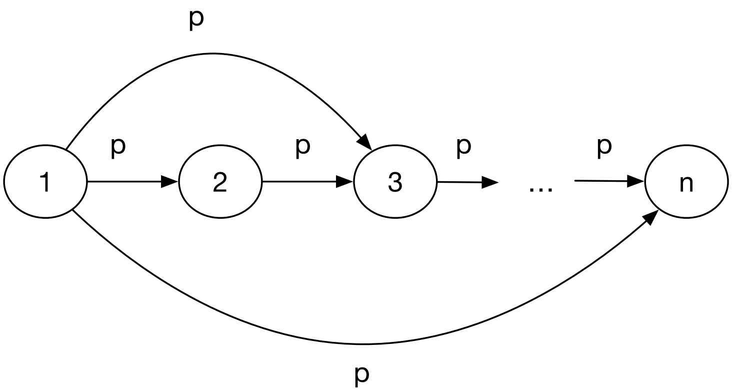


Figure 3.4: A sub graph with the transitive predicate p.

Equal Properties

In OWL there are the predicates `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 more complexity, because not only properties can be the same but single nodes can be marked as the same as well.

3.3.2 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. One can take advantage of certain features of the dictionary to achieve that.

As mentioned above, GRP does not have its own method for compressing the dictionary. We have therefore taken the compression method from HDT, and applied it in GRP to ensure a fair comparison.

uer erklären

HDT has a fairly mature mechanism for compressing the dictionary.

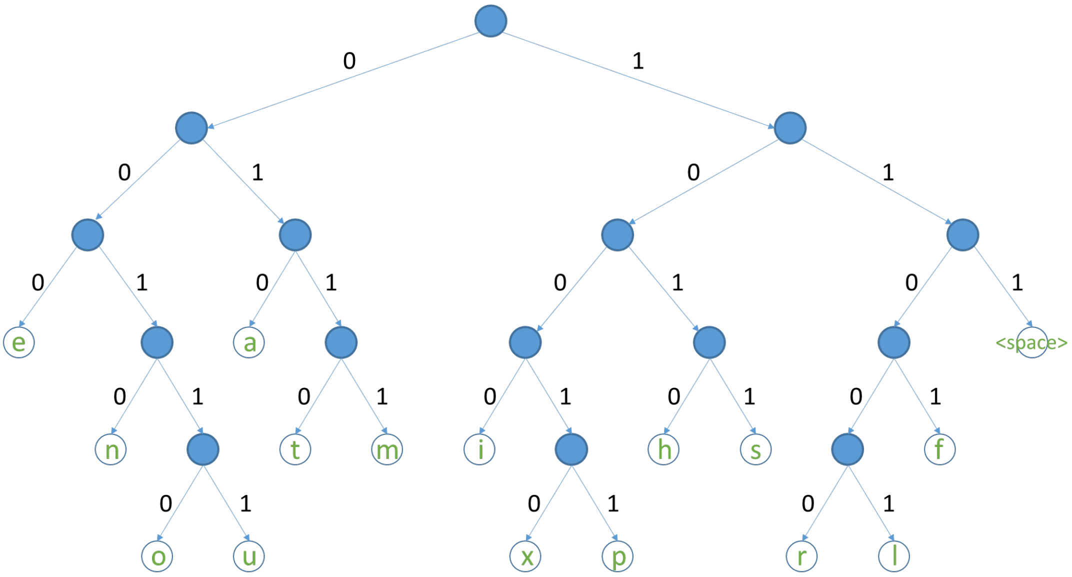


Figure 3.5: An example of a Huffman Tree.

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 the text is converted into a binary format. 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. 3.5. 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].
This tree must then be stored in addition to the compressed data so that the original data can be recovered.The details of the process are shown in Ch. 4.2.3.

Blank Nodes

The same blank node can occur in different triples. In order for it to be referenced uniquely, it 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 long strings are assigned for the blank nodes, which are quite long. They also have no common prefixes, which makes the HDT dictionary compression ineffective again.
To improve compression, one can reassign the IDs of the blank nodes. For example one can use numbers from 1 to n (n = number of blank nodes) to have 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. HDT must

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

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.

weniger details

4

Implementation

This chapter will explain how the different approaches of Ch. 3 have been implemented.

4.1 GRP vs HDT

In this section, it will be explained how the comparison between HDT and GRP is implemented.

4.1.1 Synthetic Data Creation

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.

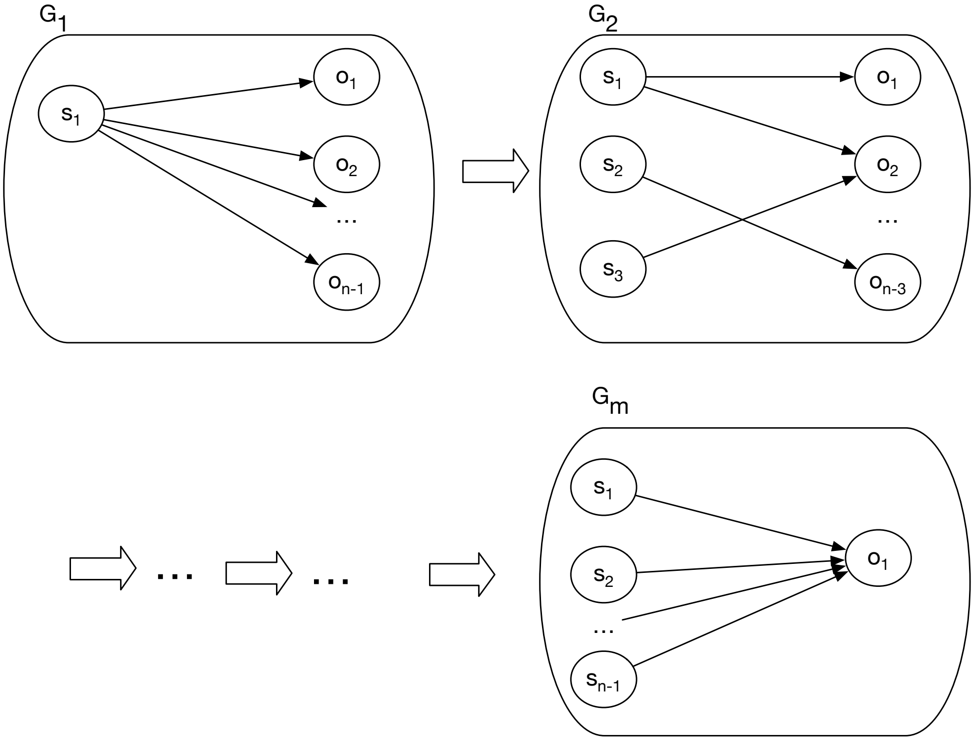
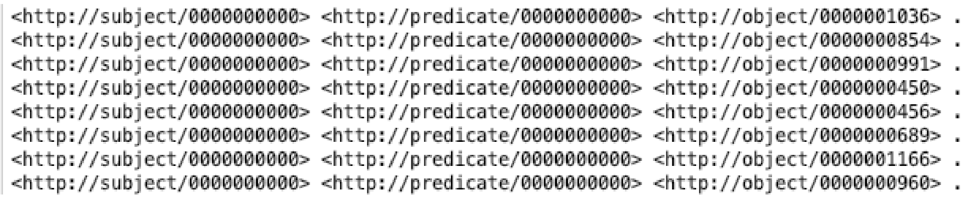


Figure 4.1: Step-by-step transition from hub pattern to authority pattern. The number of nodes n is the same for each graph. The number of edges is also the same for each graph.



```
<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. 3.3. 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.

4.2.2 Ontology Knowledge

This chapter is about how to manipulate the RDF graphs to match the properties of Ch. 3.3.1. For this the query language SPARQL [spa] is used.

Overall 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. 4.3 shows how it is implemented. First, the original graph or sub graph (the reason for sub graphs is explained below) is given to GRP. That part is called in_1 in Fig. 4.3 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 called in_2 which is then compressed by GRP and that delivers out_2.

Finally, $|out_1|$ and $|out_2|$ have to be compared. Here, the output sizes have to be compared instead of the compression ratios, because the ratio of out_2 would be relative to in_2. But a comparison to the original RDF graph (in_1) is needed in order to evaluate whether the manipulations delivered an improvement.

The 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.

In this way one sees first, what the manipulations bring individually and in the end also, what they all achieve in combination.

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:

[1] http://www.scholarlydata.org/dumps/

[2] https://wiki.dbpedia.org/Downloads2015-04

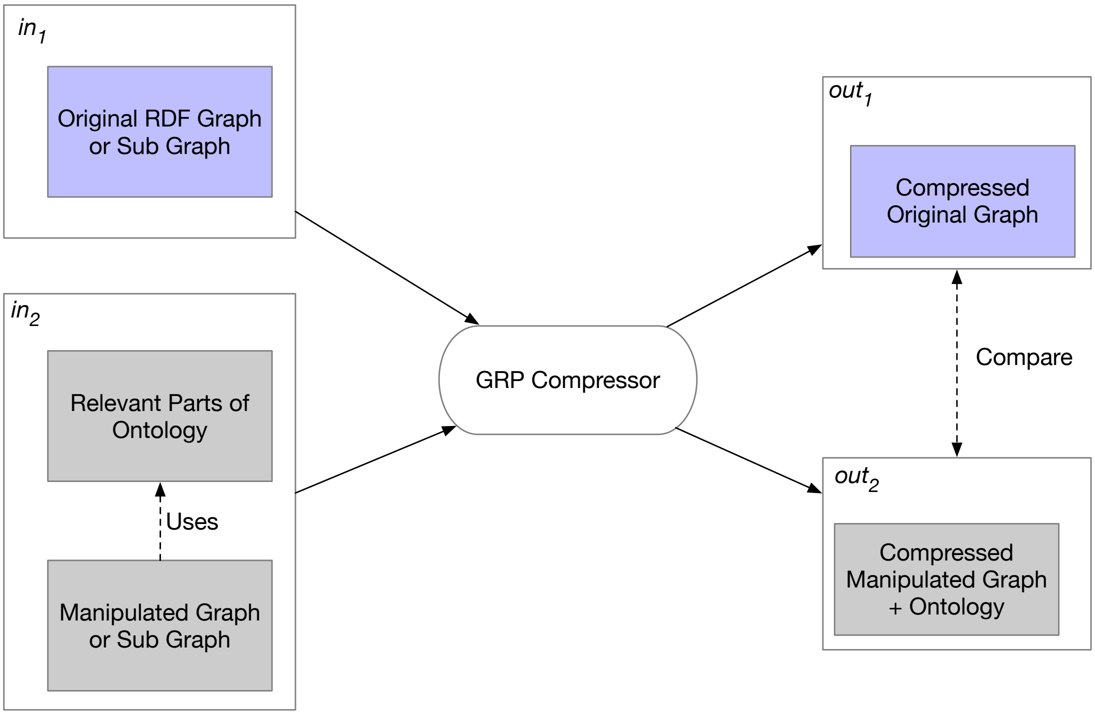

in$_1$
Original RDF Graph or Sub Graph
in$_2$
Relevant Parts of Ontology
Uses
Manipulated Graph or Sub Graph
GRP Compressor
out$_1$
Compressed Original Graph
Compare
out$_2$
Compressed Manipulated Graph + Ontology

Figure 4.3: Overall process of applying ontology knowledge and comparing the compression results.

```
<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 [3] are given. Now one can 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.

inverse bei word

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

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.
That update has to be executed for each symmetric property p. In the case where one wants to remove the second, a delete update would have to executed. That delete is shown in Listing 4.2. Here one has to be careful not to delete both directions, this can be done by using a filter.

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

```
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.

```
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 one wants to remove triples 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 one wants to remove the triple (s, p, o) 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 can not have a big impact on the compression ratio. However, one still wants 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.

```
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.

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. 4.4. 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.

4.2.3 Dictionary Improvements

For the dictionary improvements, HDT's dictionary compression is used as a basis. Therefore, the HDT-Java code [4] has been extended.

Literals

As already mentioned in Ch. 3.3.2, the 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 `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.

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

```
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.

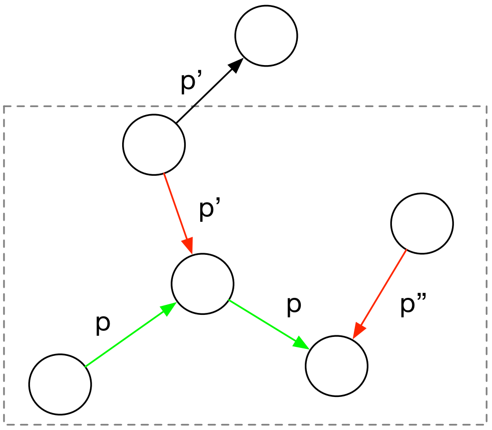


Figure 4.4: Extraction of a sub graph (marked by dashed line).

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 with 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 and 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)

```
if node is leaf then
    writeBit(1)
    writeCharacter(node.character)
else
    writeBit(0)
    EncodeNode(node.leftChild)
    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 one does not have to save the Huffman tree and does not have to calculate it oneself, 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 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.

5

Evaluation

This chapter is about the evaluation of the several approaches presented in Ch. 3 and 4.

5.1 Experimental Setup

For the following evaluations HDT-Java 2.0 [1] (the currently newest version) has been used. For GRP the implementation mentioned in [MP18] has been used (written in Scala). It is not open source and has been given to us by the authors of [MP18].

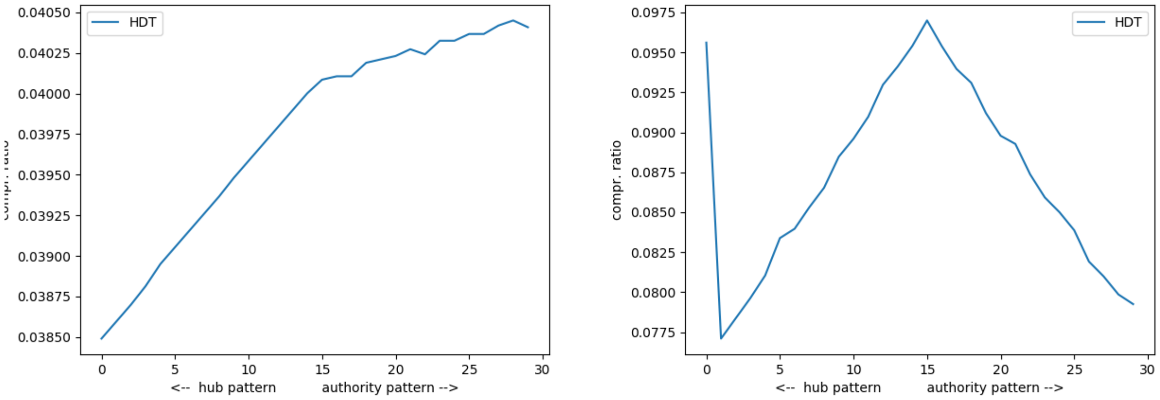
5.2 GRP vs HDT

Fig. 5.1a shows the compression ratio for HDT (without dictionary size). As expected, the ratio 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 has a much bigger effect, since the compression ratio is now much larger and the curve behavior from Fig. 5.1a is no longer recognizable. It is noticeable that the dictionary size gets bigger when the graph is further away from the star pattern. The dictionary implementation of HDT seems to be more inefficient when there are about as many subject as objects.

sorgen, dass
e rechts auch
knick hat

Next the compression ratio of GRP is considered, which is presented in Fig. 5.2a (without dictionary sizes). Here one can see that GRP has a better compression ratio if the graph is more similar to the star pattern (hub order authority pattern). This property of GRP has also been mentioned in [MP18]. It can also be seen that the effect on the compression ratio is bigger for GRP than for HDT (standard deviation is twice as high for GRP as for HDT). A grammar-based compression is therefore more dependent on the structure of the input data.
When Fig. 5.2b is considered, it can be seen that this curve behaves almost exactly like the one from Fig. 5.1b, since the size of the dictionary accounts for most of the compressed data size.
Finally, Fig. 5.3a and Fig.5.3b show the compression ratios for both algorithms. Since both use the same method 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, the ratio of HDT is 31

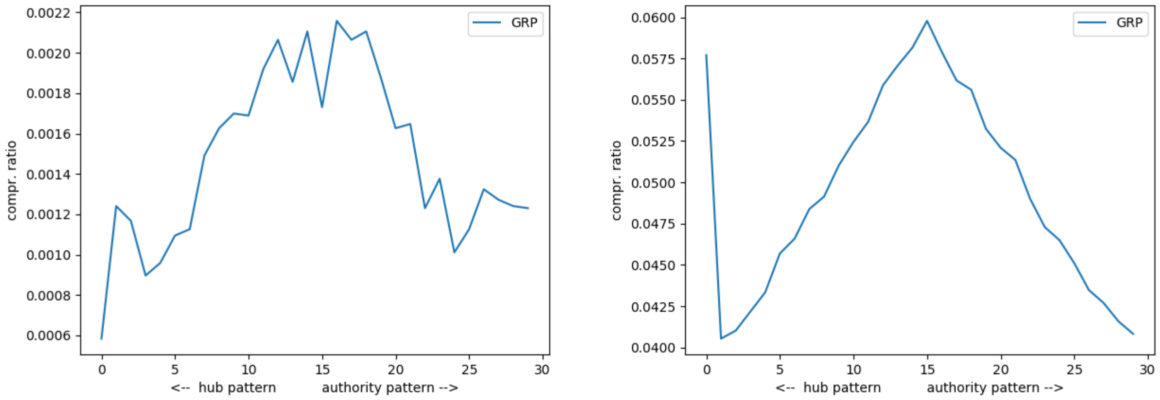
[1] https://github.com/rdfhdt/hdt-java/releases/tag/v2.0



(a) Without dictionary size.

(b) With dictionary size.

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



(a) Without dictionary size.

(b) With dictionary size.

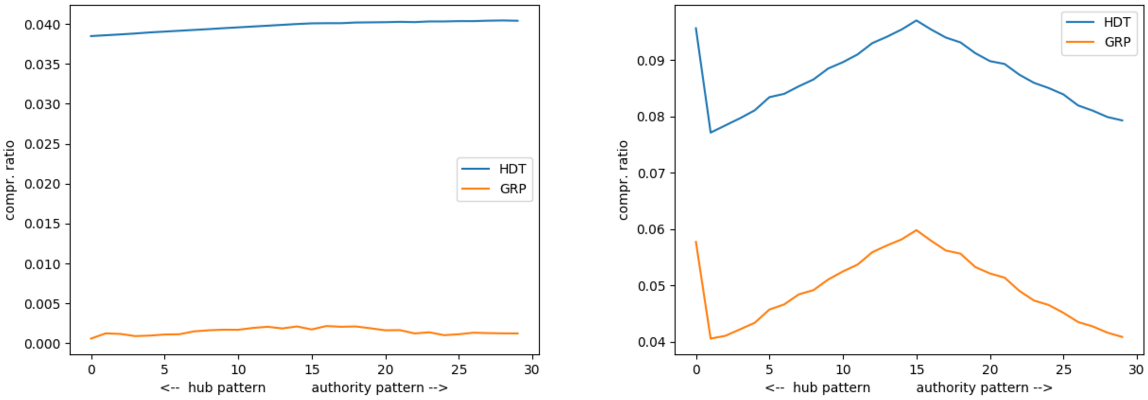
Figure 5.2: The compression ratios for GRP without and with dictionary sizes.

times higher 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 compression ratio of HDT is on average 1.8 times as high as that of GRP.

One could now 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 a quite high number considering the number of triples (1199) compared to real RDF data . One can see that the compression ratios are now higher for both compressors, but still GRP's ratio is always smaller than HDT's. HDT's ratio is still 1.7 times higher on average. So, the increasing number of predicates has a similar effect on both algorithms.

belegen

Apart from the compression ratio, the run time is also important for the overall performance. Fig. 5.5 shows the average run times of the two algorithms. 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 the run time also depends on the current CPU workload of the computer.



(a) Without dictionary size.

(b) With dictionary size.

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

It can be seen that the runtime of GRP is significantly higher than that of GRP. 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 time. So they are not comparable in terms of quality. Unfortunately, one cannot say at this point whether a more professional implementation of GRP will also be slower than HDT.
In addition one can notice that GRP's run time fluctuates more than that of HDT. GRP has a standard deviation of about 134, while 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.

5.3 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. 5.2, real data will be evaluated here to see how much influence the improvements have.

5.3.1 Ontology Knowledge

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.

Dataset	Symmetric	Inverse	Transitive
DBPedia			
Wordnet			

Table 5.1: The relevant properties for applying ontology knowledge found in real datasets.

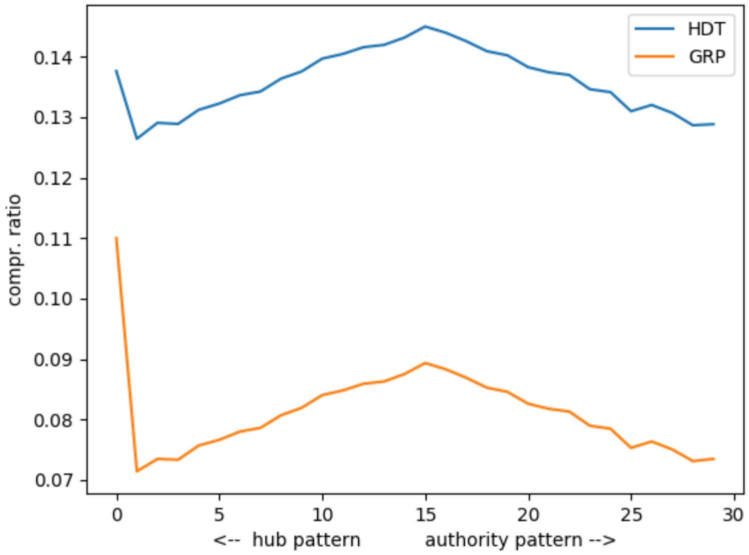


Figure 5.4: The compression ratios for GRP and HDT with dictionary sizes. Graphs have now 1000 distinct predicates.

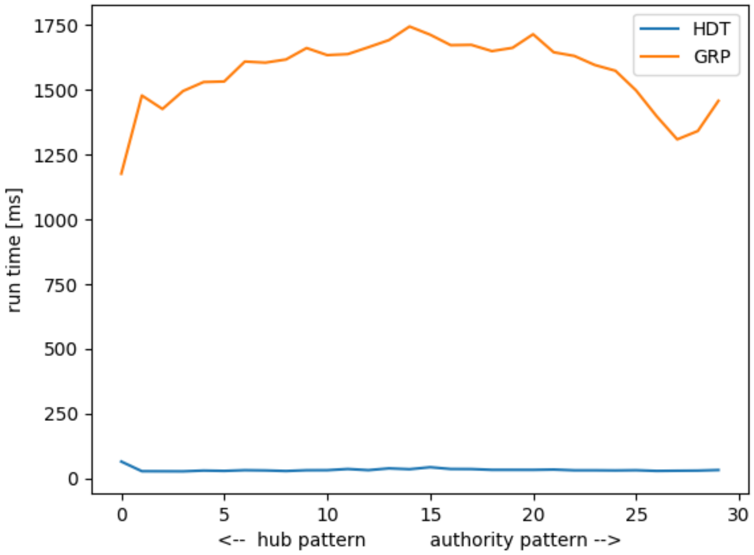


Figure 5.5: Run times of both algorithms (average run time of 100 consecutive executions).

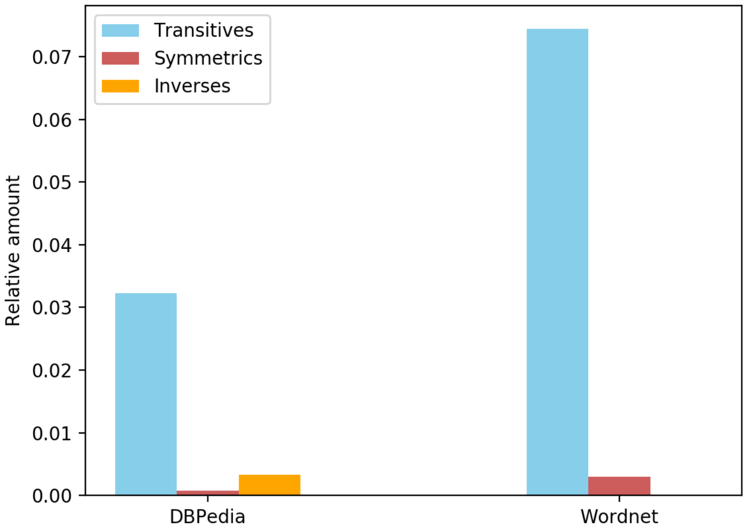


Figure 5.6: Relative amount of transitive/symmetric/inverse properties in real datasets.

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 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.

5.3.2 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. Finally, both approaches combined will be evaluated.

e das gemacht?

Literals

As already mentioned, a self-generated Huffman code was used to compress the literals of an RDF graph, as one hopes for 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. 5.7 shows the compression rates for a subset of the data (DF0 - DF9).

One can see 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. 5.8, where Fig.5.8a lists compression rates again and Fig.5.8b 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).

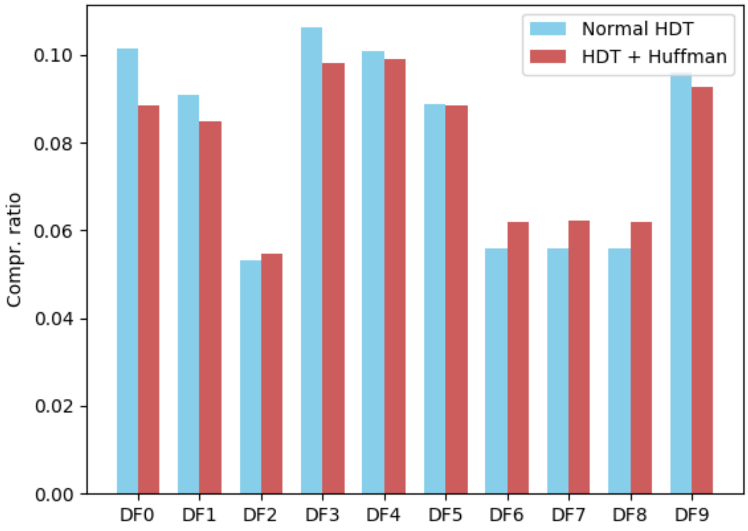


Figure 5.7: Compression ratios for Semantic Weg Dog Food Files. Comparison between Normal HDT and HDT + Huffman.

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 you can see if some languages are better suited for Huffman than others.

Fig. 5.9a shows the compression rates for the abstracts. The abbreviations stand for the languages in which the abstracts are written. It can be seen that Huffman significantly improves the compression rates here. The reason for this can be seen in Fig. 5.9b, 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.

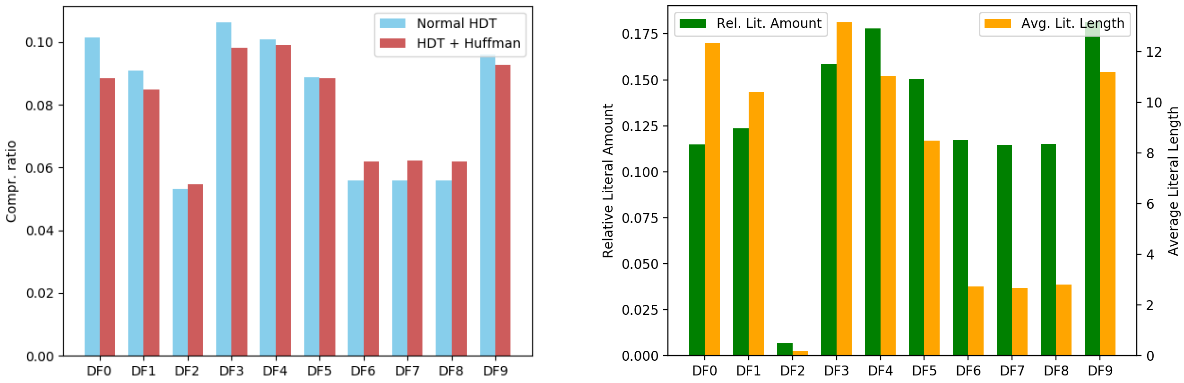
Another phenomenon can also be seen here: Although the English file contains by far the longest literals, the improvement by Huffman here is not as big as, for example, in the Bulgarian file. In the English case, the Huffman code is less effective because there are fewer different characters than in the other languages. Huffman is generally more efficient on large alphabets, according to [Sha10].

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. 5.10a shows the run times for the compression of the DBPedia abstract data. Here the runtime 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 one can compress much better and it takes quite a long time.

In Fig. 5.10b you can see the run times 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



(a) Compression ratios for Normal HDT and HDT + Huffman

(b) Relative amounts of literals and average literal length.

Figure 5.8: Relation between literal percentage and literal length (right) and compression ratios (left) for Semantic Web Dog Food Data.

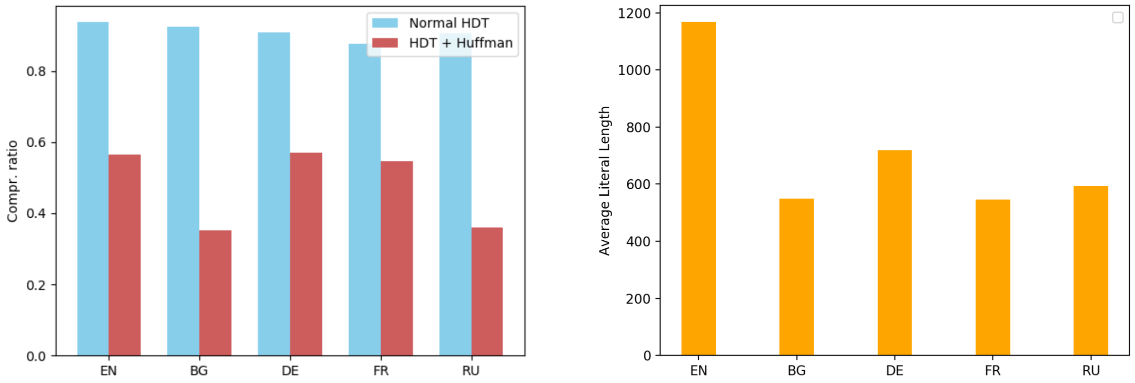
otherwise high runtime. At this point, however, the evaluation with such a standard code was omitted, since such existing codes do not contain all the special characters that occur in RDF data.

Blank Nodes

Literals and Blank Nodes Combined

5.3.3 Ontology and Dictionary

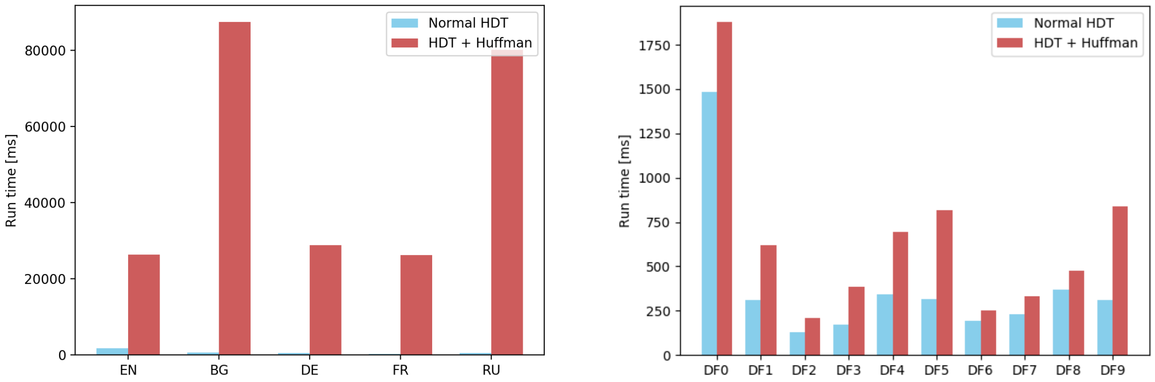
beides anwen-
wahrscheinl. ist
Anteil so ger-
ass man es auf
k nicht sehen



(a) Compression ratios for Normal HDT and HDT + Huffman

(b) Average literal length.

Figure 5.9: Relation between literal length (right) and compression ratios (left) for DBPedia Abstracts (Abbreviations denote the languages the abstracts are written in).



(a) Run times for DBPedia abstracts.

(b) Run times for Semantic Web Dog Food files.

Figure 5.10: Run times for Normal HDT and HDT + Huffman.

6

Summary and Future Work

6.1 Summary

6.2 Future Work

Todo list

Bibliography

[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.