Benchmarking HDT and GraphRePair

Philip Frerk

May 4, 2019

In the thesis we want evaluate the GraphRePair graph compression approach by using HDT as its benchmark. Fig. 1 shows the general aspects of a compression process applied to a RDF graph. p_{input} and p_{alg} are benchmark parameters, this means they can be changed during the evaluation. In contrast, m_{output} and $m_{runtime}$ are measures which we want to evaluate.

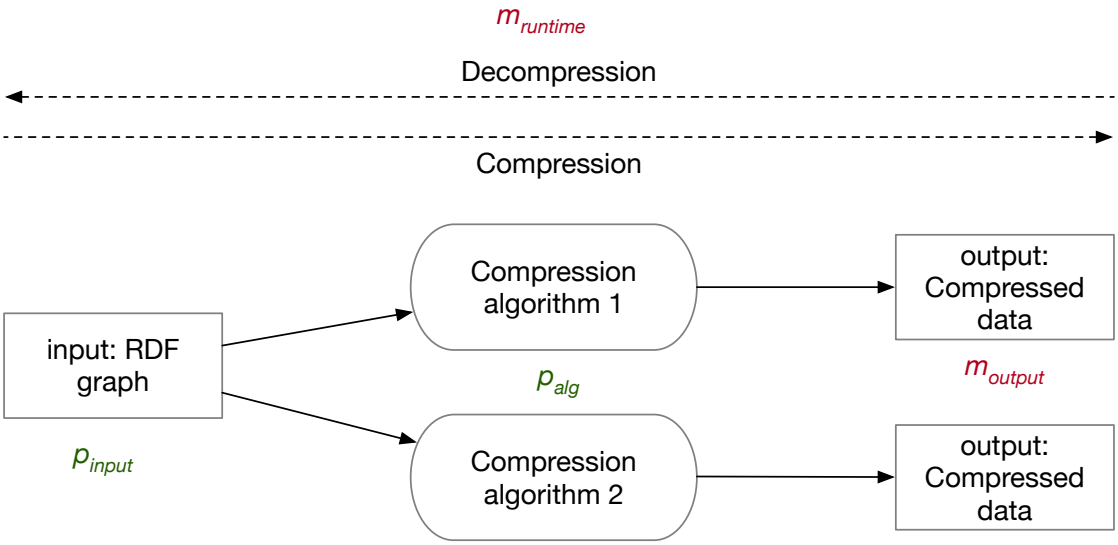


Figure 1: The different aspects of a compression process. p_{input}, p_{alg} are parameters that can be changed. $m_{output}, m_{runtime}$ are measures of the algorithm's performance.

We want to analyze the influence of different values for the parameters on the output measures. The following list explains the parameters and measures.

- (p_{input}) RDF graph: Parameters of the input data
  - format (ttl, xml, ...)
  - #edges, #nodes, average node degree,...

  - percentage of literals
  - percentage of blank nodes
  - 'HDT pattern similarity' (star pattern):one or several node(s) that appear in many triples as subject(s)
    (HDT benefits from this pattern)
  - 'GraphRePair pattern similarity': More difficult, many occurrences of one or several digrams appear
    (GraphRePair benefits from this pattern)

- (p_{alg}) Compression algorithm: Parameters of the algorithms
  - hardware environment
  - multi threading / single threading

- (m_{output}) Compressed data:
  The size of the compressed data (similarly the compression ratio) is the core measurement of a compression algorithm.
  - size of output file compared the size of the original file

- ($m_{runtime}$) (De)-Compression
  Obviously the run time of the compression is very important. Additionally, the decompression is interesting, because sometimes the user wants to decompress the data since some operations might not be possible to do with the compressed data.
  - run time

Generating Artificial RDF Data

For the evaluation of compression algorithms, real data is often used to show that the algorithms work well in reality. In a direct comparison of two algorithms (benchmark), real data is not well suited, because here we want to investigate which algorithm works better for which parameter values. Therefore, it is necessary to generate artificial data that precisely fulfill these parameter values. A big task of the thesis will be the generation of such data.

Generating a single RDF Graph

We will now discuss the generation of some different input parameters (p_{input}).

- format: trivial
- # edges / # nodes: trivial
- average node degree: for each node generate random number of incident edges: standard deviation of random number has to be defined

- percentage of literals: Let n be the number of nodes. Then we are evaluating $\frac{l}{n} \in [0,1]$ where l is the absolute number of literals. So we have to fix n first and then l

- percentage of blank nodes: Let n be the number of nodes. Then we are evaluating $\frac{b}{n} \in [0,1]$ where b is the absolute number of blank nodes. So we have to fix n first and then b

- 'HDT pattern similarity': Determine number of 'frequent subject nodes' (FSN). Then determine average number of triples a single FSN appears in (as subject)

- 'GraphRePair pattern similarity': Determine class of digram (many different such classes exist). Then determine number of occurrences.

  The goal is to define a property $p_{GPR} \in [0,1]$ for an RDF graph, such that the higher p_{GPR} is, the higher the graph can theoretically be compressed by GraphRePair. p_{GPR} should be defined in such a way that we can generate a partially random RDF graph with a pre-determined value for p_{GPR}. We can then construct a plot with p_{GPR} on the horizontal axis and the compression ratio on the vertical axis (for different graphs with different values for p_{GPR}) and analyze if our hypothesis about the impact of p_{GPR} is correct.

  A possible way of defining p_{GPR} is the following:

$$p_{GPR} = \frac{\sum_{d \in \text{digrams with } |occ(d)|>1} (|occ(d)|)}{n} \in [0,1]$$

  Here we count the sum of all occurrences of digrams with more than one occurrence and finally divide the sum by n which is the number of nodes of the graph. $occ(d)$ is defined as the set of all occurrences of the digram d.

  We only count occurrences of digrams with more than one occurrence because only digrams with multiple occurrences can have a positive impact on the compression ratio (in most cases).

  A further intention of the definition of p_{GPR} is that this property should not influence the compression ratio of HDT. That could be achieved by choosing digram types that HDT does not benefit from. It must be investigated what those types are.

  In order to make the definition of p_{GPR} clearer, we will look at an example from the GraphRePair paper.

  In Fig. 2 you see an uncompressed graph with 8 nodes. In Fig. 3 you see the compressed version of the graph and its two digrams A and B. A is a 'simple' digram as it contains no other digrams, but B contains one occurrence of A. Therefore we have to count one occurrence of A for each occurrence of B. The value of p_{GPR} is the following in this case:

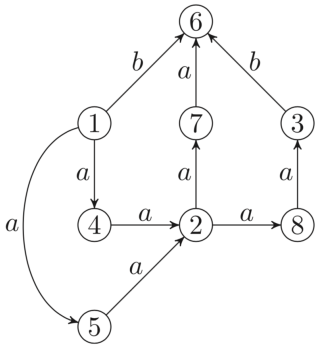


Figure 2: An uncompressed graph.

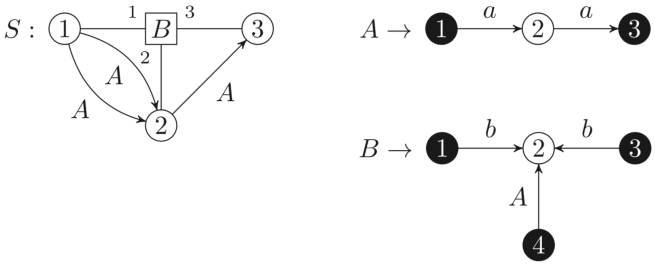


Figure 3: The compressed graph from Fig. 2 with the digrams A and B.

$$p_{GPR} = \frac{\sum_{d \in \text{digrams with } |occ(d)|>1}(|occ(d)|)}{n} = \frac{|occ(A)|}{8} = \frac{4}{8}$$

According to the definition of p_{GPR} the one occurrence of B is not considered (but the one occurrence of A in B is counted).

There are now two directions of handling the property p_{GPR}:

1. Generate a new graph with a pre-defined value for p_{GPR}
2. Compute the value for p_{GPR} of a given graph

The first direction seems to be the easier one. We can define a set of digrams for which we want to generate multiple occurrences. Then we generate a graph triple by triple. For some triples we just use random values, for others we insert occurrences of one of the defined digrams. Furthermore we want to make sure that the different occurrences do not overlap, as this would decrease the potential compression ratio. Due to the randomness it will happen that we do not precisely achieve the pre-defined p_{GPR}-value, but an approximation should work.

The second direction is completely different. It could be interesting to analyze the p_{GPR}-value for a given graph, e.g. from DBPedia. But this is more difficult,

because in order to count the occurrences of the different digrams we would have to act like the GraphRePair algorithm. One possibility would be to use the algorithm as a white box and note what different digram occurrences it finds.

Generating multiple RDF Graphs

The general idea of the benchmark is that there are certain parameters HDT benefits from (e.g. 'HDT pattern') and others from which GraphRePair benefits (e.g. 'GraphRePair pattern').
The end product should take an RDF graph as input and predict whether it will be better compressed by HDT or GraphRePair. To achieve that we have to compare different parameters directly and compute thresholds for them indicating that above this threshold one of the algorithms works better.
It would also be possible to ask the user whether he rather wants an acceptable runtime and moderate compression rate or a good compression ratio at longer runtime. However, it is likely that HDT in general will have a better runtime as it has been implemented more professionally.
In order to achieve this we must generate RDF graphs that partially fulfills one of the HDT beneficial patterns and at the same time partially fulfills some of the GraphRePair beneficial patterns. Then we can evaluate the measures (compr. ratio and run time) on these graphs and determine the thresholds mentioned above.

New Insights from Maneth Paper

- Storage of dictionary is omitted => we need algorithm for dictionary compression and add the storage to the overall storage
- Node order has marginal input on compr. ratio for RDF
- Comparison only with k^2-tree compressor
- No. of equivalence classes has correlation with compression ratio (two nodes have a similar equiv. class if their neighborhood is similar => repeating sub-structures). Therefore: The lower the number, the better the compr. ratio.
- star pattern has also correlation with compr. ratio
- Graphs that are also tree-like compress well, because in every iteration halves the number of edges around the hub node (Why???)

1 Evaluation Results

The evaluation scenario is the following:

All graphs have the same amount of triples (1199) and nodes (1200). The first graph has only one subject (all other nodes are objects). The next one has 40+1 subjects and so on. In each graph the amount of subjects is 40 more than in the previous graph.
The following results show the compression ratios ($\frac{\text{compr. size}}{\text{orignal size}}$) for the compressors HDT and GraphRePair (GRP) for the scenario described above.
Here is an overview of what the different results show:

- Fig. **??**: Both, 1 predicate, dictionary size included
- Fig. **??**: Both, 1000 predicates, dictionary size included
- Fig. **??**: HDT, 1 predicate, dictionary size not included
- Fig. **??**: GRP, 1 predicate, dictionary size not included

2 Ideas for further work

- (De)-Compr. time evaluation for HDT and GRP (quite trivial, because HDT is way faster, but it is necessary to mention this)
- Improvement of GRP in terms of RDF
  - using symmetric predicates => Are there many? How difficult to implement?
  - Further ideas needed: (e.g. node visiting order has only small effect with RDF)
  - Problem: Code not documented, therefore difficult to extend it.
- Deeper analysis of GRP
  - Where does the low compr. ratio come from? (grammar or k^2)
  - Which features/properties of an RDF graph lead to low/high grammar/k^2 compr. ratio? => E.g. divide compr. ratio in grammar ratio and k^2 ratio, investigate which ratio makes which part of the total ratio
  - Investigate, which kind of digrams are often found in RDF. E.g. star pattern seems well suited for GRP, why is that the case? Which digram does it find here?
- Comparison to a node-based grammar compressor:
  - GRP is an edge-based compressor as it replaces edges. In contrast, Dürksen's algorithm is node-based. It could be investigated which one leads to a higher compr. ratio.
  - "For future work, there are several paths to follow. It would be interesting to consider a RePair compression scheme for graphs that is based on node replacement (NR) graph grammars (see e.g. [48]). NR graph grammars can compress some graphs, e.g. cliques, much stronger than HR grammars. On

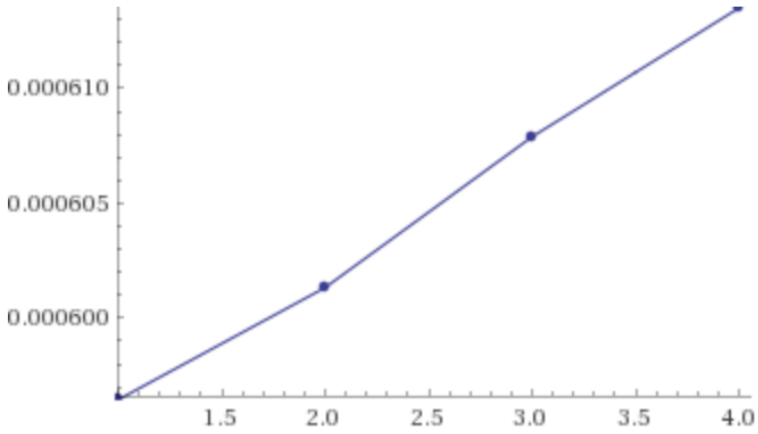


Figure 4: Compr. ratios: Original, inverse, symmetric, transitive

the other hand, rules of NR grammars are more complex due to the additional con- nection relation and thus more expensive to store." [Maneth]

3 OWL

Interesting OWL Constructs

- SymmetricProperty (evtl alles materialsieren), TransitiveProperty (wahrscheinlich alles ent-materialisieren)
- equivalentProperty: Establish mapping from prop to List<Prop>, in all triples where some p in List<Prop> occurs, replace it with prop => more repeating predicates!

4 Results Materialisation

File: mappingbased-properties_en.ttl (first 500k triples)
Symmetric: http://dbpedia.org/ontology/spouse

Transitive:
0 = "http://dbpedia.org/ontology/isPartOf"
1 = "http://dbpedia.org/ontology/province"
2 = "http://dbpedia.org/ontology/locatedInArea"
3 = "http://dbpedia.org/ontology/city"
4 = "http://dbpedia.org/ontology/district"
5 = "http://dbpedia.org/ontology/county"

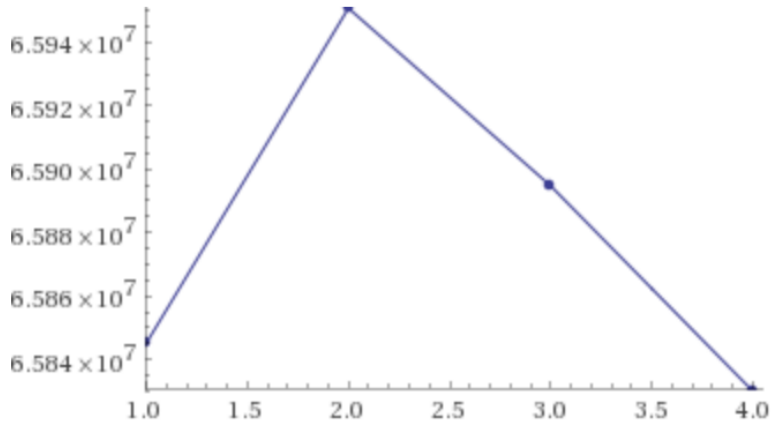


Figure 5: Original file sizes: Original, inverse, symmetric, transitive

6 = "http://dbpedia.org/ontology/settlement"

Inverse:
"http://dbpedia.org/ontology/doctoralStudent" -> "http://dbpedia.org/ontology/doctoralAdvisor"
"http://dbpedia.org/ontology/mother" -> "http://dbpedia.org/ontology/child"
"http://dbpedia.org/ontology/father" -> "http://dbpedia.org/ontology/child"
"http://dbpedia.org/ontology/child" -> "http://dbpedia.org/ontology/mother"
"http://dbpedia.org/ontology/follows" -> "http://dbpedia.org/ontology/followedBy"
"http://dbpedia.org/ontology/followedBy" -> "http://dbpedia.org/ontology/follows"
"http://dbpedia.org/ontology/doctoralAdvisor" -> "http://dbpedia.org/ontology/doctoralStudent"
"http://dbpedia.org/ontology/spouse" -> "http://dbpedia.org/ontology/spouse"

5 Fragen an Michael

- Materialiserung scheint nicht viel zu bringen. Nur ein symmetrisches Prädikat??
  Andere Möglichkeit: Alles entmaterialisieren und dann zeigen dass man weniger speichern muss, auch wenn man die Ontologie abspeichert.
- Blank nodes: Konvertierung von string nach int wahrscheinlich nicht nötig, da in der Impl von dem dict sowieso nur Zahlen gespeichert werden.
  Speichern von Blank node IDs evtl gar nicht nötig??
- Literale: Kompr. mit Huffman; einen Huffman Code für alle Literale

6 Neue Ideen

- Literale UND blank nodes: Datein auswerten und jeweils aufführen wie Groß der rel. Anteil der Literale bzw. blank nodes ist. Bei Literalen auch die durschnittl. Länge angeben, da diese große Auswirkungen hat

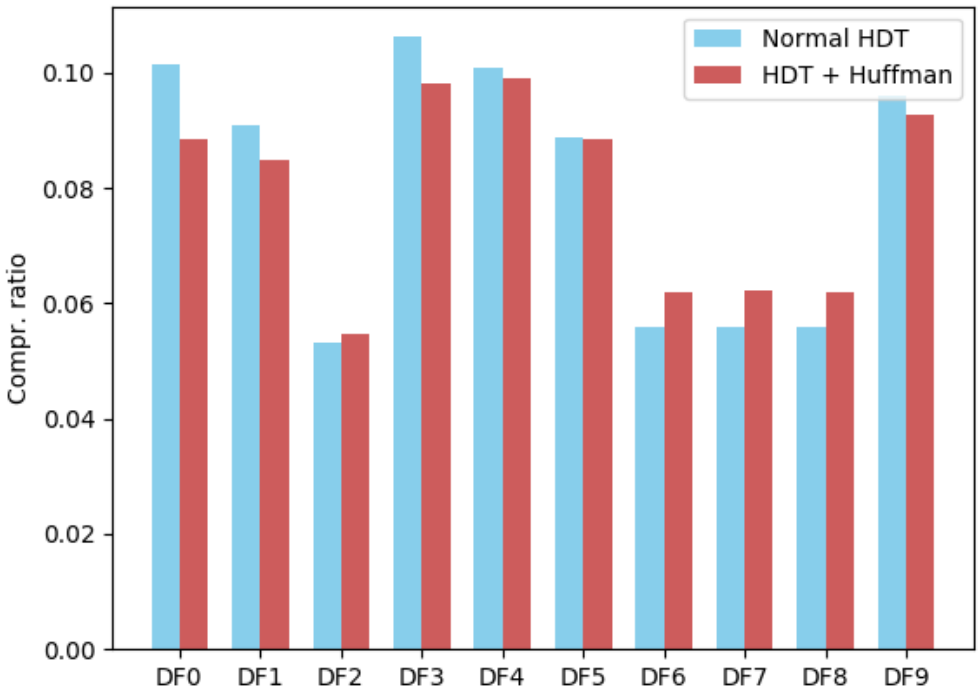
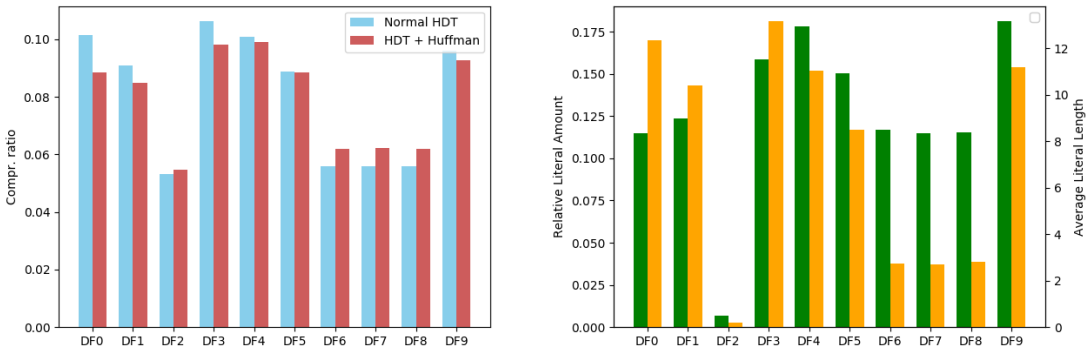


Figure 6

- Aufwand zum Speichern des Huffman Codes
- Laufzeitunterschiede aufführen
- bei HDT symm. Prädkate nutzen um star pattern zu erzeugen
-

7 Results Dictionary Optimization



(a) Compression ratios

(b) Relative amounts of literals

Figure 7: Relation between amount of literals and improvement of compression ratio (Files from Semantic Web Dog Food).