University of Zurich^UZH

A Framework for Creating Sequences of Versioned Knowledge Graphs from Wikidata

Thesis November 6, 2019

Corina Rüegg
of Zurich ZH, Switzerland

Student-ID: 15-701-709
corina.rueegg@uzh.ch

Advisor: **Daniele Dell'Aglio**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to thank my advisor, Dr. Daniele Dell'Aglio, for his support and guidance throughout the process of writing this thesis. He provided me with valuable feedback during the meetings and with reviewing the chapters.

Further, I thank Romana Pernischová and Matthias Baumgartner for giving feedback on the requirements of the framework and supplying relevant papers.

Finally, I would like to thank Professor Abraham Bernstein for letting me write the bachelor thesis at the Dynamic and Distributed Information Systems Group and therefore, being able to work on the interesting topic of knowledge graphs.

Zusammenfassung

Die Evolution von Wissensgraphen, sogenannten Knowledge Graphs, stösst in der Forschung auf zunehmendes Interesse. Aus diesem Grund bildet diese Thesis dafür eine Grundlage, indem sie ein Framework entwickelt, welches historische Versionen des Wikidata Wissensgraphen nachbildet. Um Ressourcen zu sparen extrahiert das Framework einen Subgraphen aus dem Original und generiert die ursprünglichen Versionen basierend darauf. Mögliche Verzerrungen durch verschiedene Sampling Methoden werden analysiert und stimmen mit den Ergebnissen aus bisherigen Studien überein. Darüber hinaus beschreibt diese Arbeit die verschiedenen Arten von Revisionen und wie diese rückgängig gemacht werden können um eine Sequenz von früheren Versionen des Subgraphen zu erhalten. Schlussendlich wird aufgezeigt, wie diese Versionen in einem Standard RDF Format zurückgegeben werden.

Abstract

Studying the evolution of knowledge graphs has become an important topic of current research. For that purpose, this thesis provides a foundation by contributing a framework that creates historic snapshots of the Wikidata knowledge graph. In order to save resources, the framework extracts a sample out of the original graph and generates the snapshots based on that sample. The behavior and biases of different traversal-based sampling techniques are analyzed and they agree with previous observations by related work on sampling. This work further describes the types of revisions and how they can be undone in order to create a sequence of versions of the sampled graph in earlier stages of its history. Finally, it demonstrates how the snapshots are returned in a standard RDF format.

Table of Contents

1 Introduction

1.1 Motivation

Knowledge graphs have become ubiquitous in our daily lives. We interact with them in various ways, although we do not obviously recognize them. While surfing through the internet we get product suggestions through recommender systems. When doing a Google search not only links but also a box with auxiliary information and context is displayed to the user. Personal digital assistants are used to answer questions or to complete tasks the user assigned to it. Hiding behind all these interactions are knowledge graphs which provide applications with their linked data.

Google coined the term "Knowledge Graph"[1] in 2012. Today, there exist many different knowledge graphs, commercial as well as open-source. Among the openly available ones, the most prominent graphs are DBpedia, Freebase, YAGO, OpenCyc and Wikidata. These graphs arrange knowledge as networks with nodes describing things and edges expressing the relations between them. This thesis focuses on the Wikidata knowledge graph. Wikidata is a free collaborative knowledge base where users can contribute and edit data. In a comparative study between different free knowledge graphs Wikidata has shown to be one of the most complete graphs and it also stands out in other quality aspects like trustworthiness, accuracy, accessibility and relevancy [Färber et al., 2017].

Knowledge graphs, as well as knowledge itself, naturally evolve over time. New knowledge is added, already existing knowledge may be updated or has to be corrected in cases of wrong or unspecific information. Analyzing the evolution of knowledge graphs can provide researchers with new discoveries. They are for example interested of how evolution affects the services implemented on top of the knowledge graphs [Pernischová, 2019] or in the prediction of the occurrence or recurrence time of facts (in this context, fact means an edge describing a relation) [Trivedi et al., 2017].

To be able to conduct research on knowledge graph evolution, all the changes a graph experiences need to be kept track of. This can either happen by storing old versions of the knowledge graph or by providing a change history which collects each of the revisions individually. The maintainers of Wikidata offer both data sets. Nevertheless, there are several difficulties when working with old versions of Wikidata. First, they are

[1] *https://www.blog.google/products/search/introducing-knowledge-graph-things-not/*

not available at regular intervals and second, they may contain changes over a too long time period. Further, processing the whole Wikidata knowledge graph would require a high amount of resources such as memory and time.

1.2 Description of Work

This motivation leads us to the goal of this thesis which is the design and implementation of a framework for creating sequences of versioned knowledge graphs (snapshots) from Wikidata. The framework first extracts a sample from the current Wikidata graph to save resources. To this objective, this work discusses and implements different sampling techniques. The framework then accesses the revisions for the sampled graph from the change history provided by Wikidata. It has to undo these revisions in their correct order to generate a sequence of snapshots of how the sample graph looked like back in time. Such an approach helps to overcome the obstacles discussed before as we are aware of each individual revision. In contrast, old data dumps only contain summarized revisions over a long time period. Furthermore, working with samples saves a lot of computing resources. Therefore, this thesis should provide a resource-efficient, precise tool to generate a sequence of historical snapshots from the Wikidata knowledge graph.

1.3 Outline

Chapter 2 gives an introduction into the Semantic Web with attention to Wikidata and the way Wikidata structures its data. The third chapter analyzes different graph sampling techniques by comparing related work. In Chapter 4 the frameworks requirements are discussed, followed by the description of the implementation in Chapter 5. The implementation chapter is divided into an overview of the framework and the implementation of each the sampling, the undoing of the revisions and the returning of the snapshots. Chapter 6 then discusses the limitations of the framework, whereas Chapter 7 contains enhancements and ideas for future work. Finally, Chapter 8 concludes this thesis with a critical reflection and a short summary.

2

Wikidata and the Semantic Web

In order to pave the way for implementing a versioning tool for Wikidata, we first have to take a look at the foundation concepts of the Semantic Web and how Wikidata organizes its data. The first section discusses the key concepts of the Semantic Web while the second section introduces the collaborative knowledge base Wikidata which is based on these concepts. The final section then discusses the Wikidata data model.

2.1 Introduction to the Semantic Web

The vision of the Semantic Web is to make web data machine readable [Berners-Lee et al., 2001]. This way, it is possible to integrate data across websites and query not only for keywords but also contextual information. Semantic Web follows three main design principles:

- Labeled graphs model objects as nodes and edges as relations between those objects. Resource Description Framework (RDF) [Miller and Manola, 2004] is used to formalize such logic statements.
- Uniform Resource Identifiers (URIs) identify data items and their relations.
- Ontologies formally describe the semantics of the data.

RDF uses resources, properties, statements and graphs as concepts. A resource describes an object with an URI which unambiguously refers to that object. Properties (again declared with URIs) are used to express relations between resources. Statements then specify a claim about an object. This claim is handled as a triple consisting of a resource, a property and a value. A value, in turn, may be either another resource or a literal (e.g. numbers, strings, dates). Such statements can further be illustrated as graphs with the property as a label for the edge which is directed from the subject of a statement to the object. The object may then be the subject of another statement. This linking of data results in a knowledge graph which can be shared across different applications. A simple RDF statement extracted from the Wikidata knowledge base is represented in Figure 2.1 as a simple graph with two resource nodes and a directed edge describing the relationship between them.

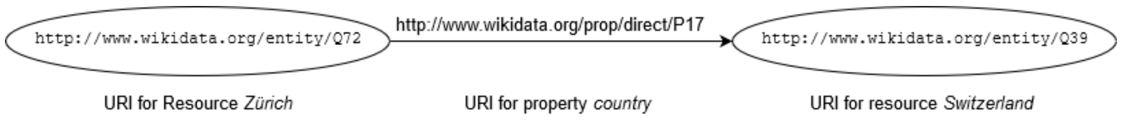

http://www.wikidata.org/entity/Q72
http://www.wikidata.org/prop/direct/P17
http://www.wikidata.org/entity/Q39
URI for Resource *Zürich*
URI for property *country*
URI for resource *Switzerland*

Figure 2.1: A Simple RDF Statement

The statement in Figure 2.1 is expressed with full triple notation which means that each URI has to be written out completely. Since such a notation would result in many repetitions and long statements, there is a way to abbreviate URI references with qualified names. A qualified name consists of a prefix, a colon and a local name. The prefix is assigned to a namespace URI. Therefore, resources and properties starting with the same namespace can share the corresponding prefix. The shorthand notation of our simple RDF statement consequently results in:

```
PREFIX wd: <http://www.wikidata.org/entity/> .
PREFIX wdt: <http://www.wikidata.org/prop/direct/> .
wd:Q72 wdt:P17 wd:Q39 .
```

2.1.1 Querying the Semantic Web

To make use of the information represented in RDF, we need to be able to access relevant data parts. For this purpose a query language is needed. SPARQL is the W3C recommendation query language for RDF [Seaborne and Prud'hommeaux, 2008]. SPARQL queries normally consist of a set of triple patterns, namely a basic graph pattern. Such patterns are similar to RDF, but each of the subject, property and object may be substituted with a variable. A variable is assigned a "?" at the beginning. A graph pattern for the RDF statement in Figure 2.1 that introduces a variable for the object has therefore the following structure: *wd:Q72 wdt:P17 ?country.*

Such a basic graph pattern can match with triples from the actual RDF database if the triples are equivalent except for the variable which may be substituted with possibly multiple results. When running above pattern over the Wikidata SPARQL endpoint, it returns all triples where *wd:Q72* is the subject and *wdt:P17* is the property. In our example graph pattern, the endpoint returns only one matching triple: *wd:Q72 wdt:P17 wd:Q39.*

Nevertheless, for a complete SPARQL query the prefixes as well as the particular variable(s) that we want to be returned in the result need to be defined. Therefore, the complete query for above graph pattern may be written as:

```
PREFIX wd: <http://www.wikidata.org/entity/> .
PREFIX wdt: <http://www.wikidata.org/prop/direct/> .
SELECT ?country
WHERE {
   wd:Q72 wdt:P17 ?country .
}
```

Of course, SPARQL provides functionalities for writing more complex queries. The SPARQL W3C recommendation page[1] provides a list of those functionalities and possible query patterns. But even complex queries build up on the basis of finding triples matching a specified graph pattern.

2.2 Wikidata

Wikidata[2] is the free collaborative knowledge base of the Wikimedia Foundation[3]. Since its launch in October 2012, Wikidata has become one of the largest open data collections [Malyshev et al., 2018]. To make Wikidata applicable to data analysis and query mechanisms, it follows the principles of Semantic Web technologies. As part of the Semantic Web, Wikidata integrates data not only from Wikipedia but also many other external sources resulting in a wide range of general and specific knowledge [Erxleben et al., 2014]. At the current time, Wikidata contains over 65 million items[4] and more than 808 million statements[5].
Before discussing the data model, it is important to mention several design decisions defining Wikidata [Vrandečić and Krötzsch, 2014]:

- Open Editing and Community Control: The data as well as the schema of the data itself are managed and controlled by the community.

- Plurality and Secondary Data: Often data can not be represented as the one ultimate truth. There has to be the possibility that one statement was true only for a certain range of time or we may want to store one or more sources to support our statement. Wikidata has been designed to deal with such plurality.

- Multilingual Data: Wikidata is a multi-lingual project and supports over several hundred different languages including dialects like Swiss German. While some data values as numbers and coordinates are shared over languages, others as labels and descriptions can be translated and displayed in any language supported by the software.

- Easy Access: Wikidata dumps are accessible in several formats including JSON, Turtle and N-Triples. Therefore, the data can easily be used by external applications.

- Continuous Evolution: Since Wikidata is a collaborative project, it evolves continuously with its growing community.

[1] *https://www.w3.org/TR/rdf-sparql-query/*
[2] *https://www.wikidata.org/wiki/Wikidata:Main_Page*
[3] *https://wikimediafoundation.org/*
[4] *http://www.wikidata.org/wiki/Wikidata:Statistics*, statistic based on snapshot from 2019-10-28
[5] *https://tools.wmflabs.org/wikidata-todo/stats.php*, statistic based on snapshot from 2019-10-28

2.3 The Wikidata Data Model

This section presents the Wikidata data structure and the way this data is represented in RDF. Wikidata handles data about things/resources described by Wikimedia articles such as Wikipedia, Wikivoyage, Wikisource, Wikiquote, or Wikimedia Commons. Its main objective is to store the data and make it available in any language. An **Item** defines what an article is about. There exist over hundred Wikipedia articles for the item *Zürich* in several languages; in contrast to Wikidata which handles *Zürich* as a single item supporting different languages and pointing with sitelinks to each of these articles.

Each item gets supplied with various pieces of information forming the **ItemDescription**. An ItemDescription contains some kind of basic information, normally sitelinks to the corresponding page on a Wikimedia site. Furthermore, it consists of labels, descriptions and aliases in diverse languages and a set of **Statements** about the item. A statement is composed of a claim (e.g. *Zürich* has a *population* of *414,215*) and a list of references supporting that claim. Figure 2.2 illustrates an example of a part of a Wikidata item and its most important terms.

An essential part of an item are the aforementioned statements, they define detailed characteristics and commonly consist of property-value pairs. In the Zürich example, population is the property and the value is a number. Properties store a **PropertyDescription** which defines the datatype they may take as input for values. Possible data types are strings, numbers, coordinates, other items or properties and time values. A full list of all data types and their values is available on the Mediawiki page[6]. As in RDF, unique identifiers are used to determine entities like items and properties. Taking the item from Figure 2.2, Internationalized Resource Identifiers (IRIs) are used to declare the *Douglas Adams* item as Q42 and the property *educated at* as P69. Nevertheless, Wikidata differs to classic RDF datasets in making use of reification. RDF reification is applied when there is the necessity to make statements about statements. In Wikidata, this happens in the following cases:

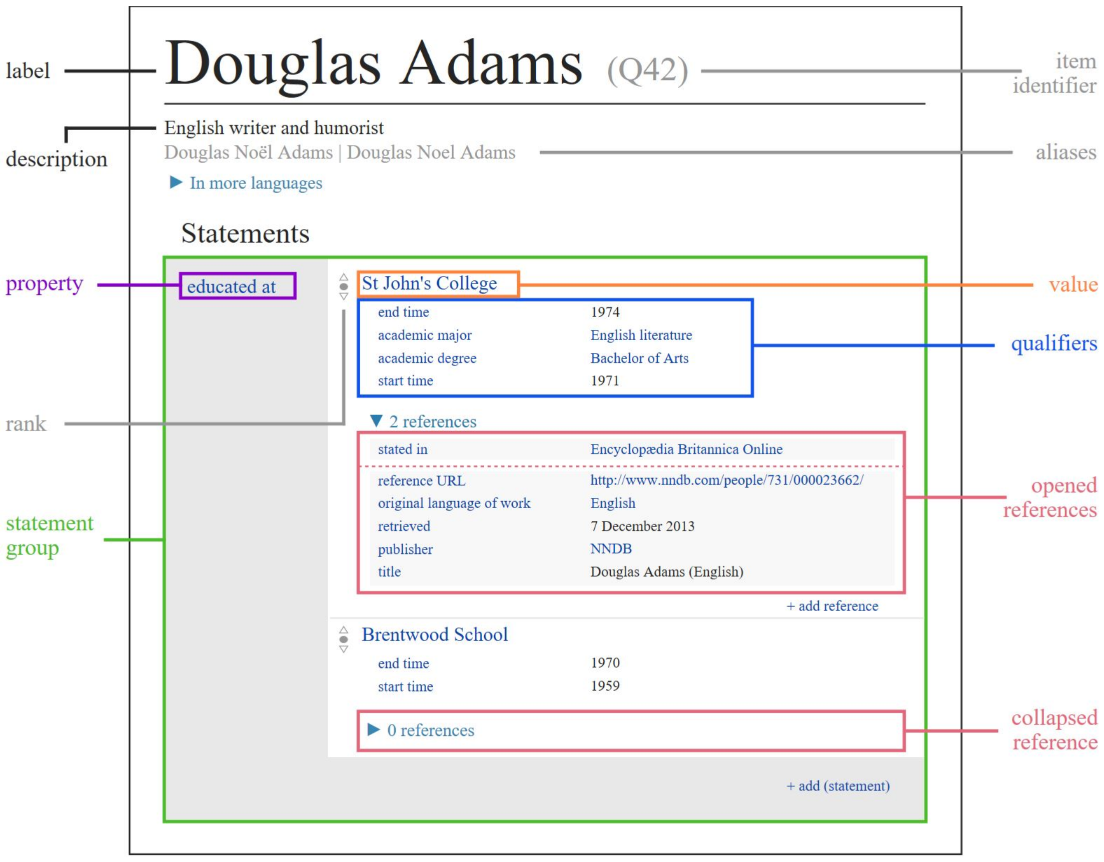
- Statements may themselves be the subject of property-value pairs. Such additional pairs are called qualifiers and are used to add context to a claim. In Figure 2.2, qualifiers add auxiliary information like start time, end time and kind of degree to the main statement.

- Wikidata stores supplementary information to some data values. They can be considered as compound objects. For example, time values not only store date and time, but also precision, timezone and the type of calendar such as Gregorian.

- To support the claims made by statements, Wikidata provides the possibility to store references. As can be seen in Figure 2.2, references also may consist of multiple property-value pairs specifying the provenance of the reference in question.

Wikidata solves the reification problem by using *n-ary relations.* N-ary relations implement additional, intermediate nodes and therefore can capture relations between

[6]*https://www.mediawiki.org/wiki/Wikibase/DataModel*



Douglas Adams (Q42)

English writer and humorist

Douglas Noël Adams | Douglas Noel Adams

▶ In more languages

Statements

educated at

St John's College

end time	1974
academic major	English literature
academic degree	Bachelor of Arts
start time	1971

▼ 2 references

stated in	Encyclopædia Britannica Online
reference URL	http://www.nndb.com/people/731/000023662/
original language of work	English
retrieved	7 December 2013
publisher	NNDB
title	Douglas Adams (English)

+ add reference

Brentwood School

end time	1970
start time	1959

▶ 0 references

+ add (statement)

Figure 2.2: Wikidata Item with Important Terms[7]

more than two entities. Wikidata handles such complex relations by adding intermediate statement, value and reference nodes. About a tenth of all triples in Wikidata are instantiations of just statement nodes, therefore leading to a significant amount of overhead [Färber et al., 2017]. This shows that Wikidata extensively uses reification, distinguishing it from many other openly available knowledge graphs.

Figure 2.3 illustrates this approach. The statement in Figure 2.3 claims that *Germany* has a *speed limit* of *100 km/h*. An additional qualifier specifies the context of this limit narrowing down the validity of the main statement for *paved roads outside of settlements*. The simplest case in the illustration is wdt:P3086 (*speed limit*) which connects the item *Germany* directly to a simplified version of the value (the number *100*). This way, properties with the namespace prefix wdt (which stands for http://www.wikidata.org/prop/direct/) always behave like normal RDF triples returning such simplified values with no context information.

As can be seen in both Figures 2.2 and 2.3, Wikidata also provides a statement rank which is some type of built-in annotation. Ranks are important for filtering mechanisms when more than one statement exists for one property. Ranks can be normal

[7]Source: *https://www.wikidata.org/wiki/Wikidata:Introduction*

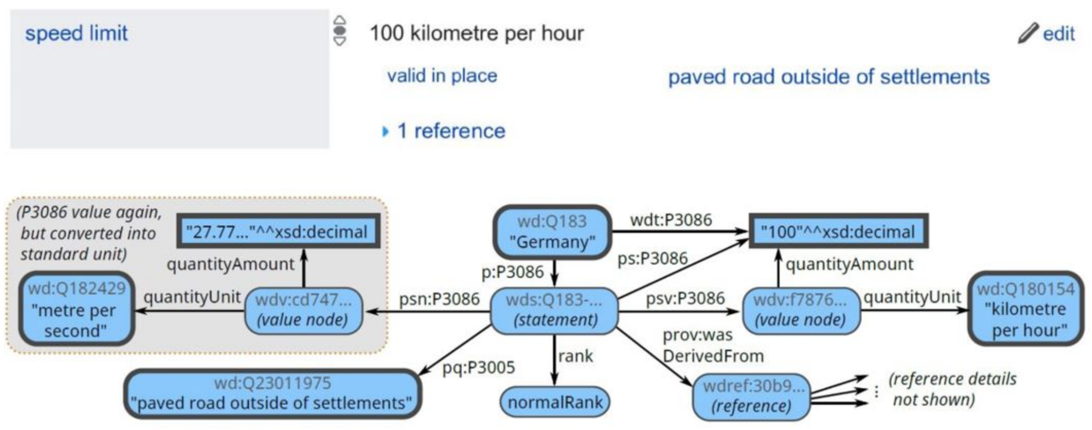

speed limit
100 kilometre per hour
edit
valid in place
paved road outside of settlements
1 reference
(P3086 value again, but converted into standard unit)
"27.77..."^^xsd:decimal
quantityAmount
wd:Q182429 "metre per second"
quantityUnit
wdv:cd747... (value node)
psn:P3086
wd:Q183 "Germany"
p:P3086
wdt:P3086
ps:P3086
"100"^^xsd:decimal
quantityAmount
wds:Q183-... (statement)
psv:P3086
wdv:f7876... (value node)
quantityUnit
wd:Q180154 "kilometre per hour"
prov:was DerivedFrom
rank
pq:P3005
wd:Q23011975 "paved road outside of settlements"
normalRank
wdref:30b9... (reference)
(reference details not shown)

Figure 2.3: Wikidata Statement and its RDF Graph Representation [Malyshev et al., 2018]

(default), preferred or deprecated. Preferred ranks are commonly used to mark the most up-to-date statement. The rank as well as qualifiers and references can be retrieved by following p:P3086 to the statement node. The statement node is declared by the prefix wds (for http://www.wikidata.org/entity/statement/), the item identifier Q183 and a Universally Unique Identifier (UUID) defined by Wikidata for every statement. From the statement node, one can follow ps:P3086 again for a simplified version of the value or psv:P3086 for the complete compound value. Depending on the type of the value node (e.g. GlobecoordinateValue, TimeValue), corresponding properties add the auxiliary information. In Figure 2.3, a QuantityValue node further specifies the quantityAmount as a decimal number of *100* and the quantityUnit as *kilometre per hour*. A full list of all properties for value nodes is available online[8]. These value properties belong to the OWL ontology of Wikibase[9]. When following the statement node to psn:P3068, a normalised version of the value can be accessed. The pq prefix (for http://www.wikidata.org/prop/qualifier/) is further used to access the qualifiers. In the example, pq:P3005 (*valid in place*) adds context information for the validity of the main statement. Finally, through prov:wasDerivedFrom, one obtains the reference node which further links to the reference details.

Next to property-value statements, Wikidata offers the possibility to create statements that contain no value or some values. Statements with no value indicate that a value simply does not exist (e.g. to state that a person has no children) and has not been forgotten or left out. Statements with some values refer to statements that have a value for a certain property, but the value is not specified or unknown (e.g. a person has an unknown date of birth).

A statement with only one qualifier rapidly results in a comparatively complex RDF graph as can be seen in Figure 2.3. The Wikidata data model produces graphs with

[8] *https://www.wikidata.org/wiki/Special:ListDatatypes*
[9] *http://wikiba.se/ontology#*

many, often redundant, triples. Redundant triples arise for example for normalized values. A normalized value may be stated with the psn: prefix after a statement node and may occur a second time after the value node for property quantityNormalized. Furthermore, for all intermediate nodes like statement, value and reference nodes, an additional triple declares the type of these nodes. These data model design decisions were made to simplify query mechanisms [Malyshev et al., 2018]. Therefore, a wide range of queries are possible, from simple to complicated.

3

Graph Sampling

Graph or network sampling is a crucial part when dealing with large, real-world networks. To analyze and create snapshots of the Wikidata knowledge graph, it is beneficial to first extract a sample to deal with the massive data volume. The motivation behind sampling is execution efficiency. In order to save resources such as time and memory, it is computationally more efficient to further process a small, but representative sample from the original graph.

In statistics, sampling methods are techniques used to select members out of a target population group. A representative sample accurately reflects the characteristics of a larger population. With a representative sample, one can draw conclusions from the sample over the population. Appropriate estimations for the population can be achieved if the sample size is large enough and members are randomly selected. Since networks depend on two different elements (nodes and links), simple random selection may not be suitable: many network properties depend on how nodes and links are interwoven [Lee et al., 2006]. This node-link dependence makes graph sampling a challenging task.

There exist many papers presenting different approaches for reducing the graph size while still obtaining a "good" sample. Such papers not only differ in the sampling algorithms but also in the way they define such a representative sample. This is not surprising since graphs are characterized by many different properties. Being representative can mean plenty of things depending on which graph metrics are of interest.

There is an important trade-off between the complexity of the sampling algorithm and the complexity and size of the network [Ahmed et al., 2013]. For example, one could formulate the problem of sampling as a minimization problem to reduce the distance between the sampled and the original graph [Hu and Lau, 2013]. Such complex algorithms can easily ruin our primary purpose of saving resources. Thus, simpler sampling methods are generally preferred over complex ones. This chapter discusses different sampling techniques from recent years.

3.1 Classes of Sampling Algorithms

Sampling techniques can be categorized into two groups, namely random sampling and topology-based sampling. Random sampling is based on either selecting nodes or edges at random while topology-based sampling builds up on the existing topology of the original graph. This section gives an overview over both groups and introduces the most common methods.

3.1.1 Random Sampling

In classic node sampling (NS), nodes are selected independently and uniformly at random from the original graph G [Ahmed et al., 2013]. Therefore, for a fraction ϕ of nodes required to be in the sampled subgraph G_s, each node is sampled independently with a probability of ϕ. After all the nodes V_s are sampled, all edges among $V_s \in G$ are added to the edge sample set E_s, resulting in an induced subgraph.

In contrast to selecting nodes, one can as well select edges independently and uniformly at random. Classic edge sampling (ES) makes use of this strategy [Ahmed et al., 2013]. Each time an edge is selected to be in G_s, both nodes incident to that edge are included in the sample as well. No additional edges are added to G_s than the ones chosen in the random edge selection process. Hence, the resulting subgraph is only partially induced.

3.1.2 Topology-Based Sampling

Topology-based sampling methods start by randomly selecting a node and then recursively visit one, some or all of its neighbors. This category can further be classified into two subcategories: graph traversal techniques and random walks [Kurant et al., 2011].

In graph traversal techniques, nodes are visited exactly once, no encounter with previously sampled nodes is intended. The main difference between the algorithms of this subcategory is the order in which they visit the nodes and the number of neighbors that are selected to be in the sample. Breadth-First-Search (BFS) first visits all successor nodes of the starting node, before moving onto the next level of depth. Depth-First-Search (DFS) explores each branch to the greatest extent possible and first samples the leaf nodes. Other examples of graph traversal techniques include Forest Fire Sampling and Snowball Sampling.

In random walks, the next node to add to the sample is selected uniformly at random among the neighbors of the current node. In contrast to graph traversal techniques, random walks generally allow revisiting nodes. Therefore, random walks may try to sample the same node more than once.

3.2 Discussion of Related Work

This section discusses different sampling strategies. An overview of related work is given by Tables 3.1 to 3.3. The papers summarized in the tables study the problem of network sampling. Nevertheless, they differ in various aspects like sampling algorithms, against which graph properties they compare the samples, as well as the type of networks itself. It is therefore difficult to conclude which sampling technique provides the best results. Still, papers generally agree on some sampling techniques to generate unsatisfactory samples.

Classic random sampling techniques suffer from numerous drawbacks. For example, node sampling does not retain the properties of graphs with power-law degree distributions [Stumpf et al., 2005]. Also, it seems reasonable that the original level of connectivity will be lost, since only edges between sampled nodes are kept. Edge sampling techniques have similar limitations and fail to preserve many graph properties. Samples obtained from ES result to be too sparse and have a bias towards high-degree nodes [Leskovec and Faloutsos, 2006][Lee et al., 2006]. Considering these drawbacks, many researchers have studied topology-based sampling methods. An advantage of such methods is the generation of connected topologies, even when the sample size is very small. Differently from random sampling, they take better into account the interdependence of nodes and edges and they usually perform better than simple NS and ES. Therefore, the rest of this section focuses mainly on related work for traversal-based sampling.

Breadth-First-Search has been widely used for graph sampling since it produces connected samples and returns a full view of a particular region of the graph. Unfortunately, BFS is biased towards high-degree nodes and thus, BFS samples may underestimate low-degree nodes by two orders of magnitude [Gjoka et al., 2010]. To correct the degree bias of BFS, one of the papers proposes a correction procedure that leads to an unbiased estimation of the degree distribution [Kurant et al., 2011]. Another drawback of this technique is that the sample may have different topological features opposed to the graph as a whole since only some small area is represented. In contrast to BFS, Depth-First-Search starts sampling from the last discovered node and as a consequence, samples nodes from the periphery of the graph [Doerr and Blenn, 2013]. This results in adding leaf nodes with low degrees first into the sample. Hence, this algorithm drastically underestimates the average node degree. Random-First-Search (RFS) is an alternative for BFS and DFS. RFS randomly selects the next node from among the list of discovered, but not yet sampled nodes. This method is quite similar to a random walk without revisiting and has been shown to perform better than its two relatives [Doerr and Blenn, 2013].

A variant of BFS is Snowball Sampling (SBS). According to the classic definition by Goodman, SBS is similar to BFS. But in contrast to BFS which samples all neighbors of a node, SBS randomly samples exactly a fixed fraction of n neighbors [Goodman, 1961]. If these neighbors have not been visited before, they are added to the queue to process next. SBS has been a popular method in sociology studies for research on hidden populations [Hu and Lau, 2013]. This is due to the fact that every person (node) can name a number n of his friends (neighboring nodes) in an iterating fashion. Thus, one obtains a sample of people with similar interests useful in such studies. Similar to BFS, this sampling method retains the network connectivity. Shortcomings of SBS are the negative trait to pick hubs (nodes with degrees greatly above average) in short intervals and the boundary bias. Last mentioned bias results from the problem that the nodes sampled on the last round are missing a large number of neighbors [Lee et al., 2006]. Mostly affected from this boundary bias is however BFS since it samples all neighbors for each node. Respondent-driven sampling (RDS) is an approach to overcome these biases and supplies SBS with a correction procedure [Heckathorn, 1997].

Forest Fire Sampling (FFS) makes use of a partial BFS as well as it samples only a fraction of neighbors. The algorithm starts by randomly selecting a starting node and then "burning" a random number of its outgoing edges. The edges together with their incident nodes are added to the sample and the process continues recursively. Leskovic and Faloutsos showed that FFS yields good samples that accurately match many properties of the original graph [Leskovec and Faloutsos, 2006]. The random number of edges to be burned at each node is generated by a geometric distribution with mean $p_f/(1-p_f)$. For best results, the authors suggest 0.7 as a value for p_f which results in an average of 2.33 sampled edges per node.

The most popular and simplest version of random walk topology-based sampling methods is Random Walk Sampling (RW). RW is basically working like SBS, but with $n = 1$. Therefore, at each node only one outgoing edge with its incident node is sampled. Some papers additionally include a fly-black probability. With this probability, a neighbor is visited only with a certain probability or else, the initial node is visited again, therefore allowing to sample more neighbors. A typically used fly-back probability is 0.15 [Leskovec and Faloutsos, 2006]. However, in contrast to SBS, RW can visit edges and nodes again whereas SBS does not allow revisiting. A drawback of RW is the bias towards high degree nodes and densely connected parts of the graph [Leskovec and Faloutsos, 2006][Gjoka et al., 2010].

Metropolis-Hastings Random Walk (MHRW) was designed to avoid this high-degree bias. It modifies the probabilities to move to a neighbor to achieve a uniform degree distribution. Initially, it starts like Random Walk by selecting the next candidate node w uniformly at random from among the neighbors of node v. Then, a uniformly random number $0 \leq p \leq 1$ is generated. If $p \leq \frac{degree_v}{degree_w}$, then w is the next node to be in the sample. Otherwise, w is rejected and the algorithm stays at v giving other incident nodes a chance to be selected. This way, a neighboring node with a lower degree than the current one will always be sampled and some of the nodes with higher degrees are rejected. Hence, samples generated by MHRW lead to very accurate degree distributions [Gjoka et al., 2010].

Gjoka et al. further propose Re-Weighted Random Walk (RWRW) which is supposed to lead to accurate estimations of degree distributions as well. This method is not further considered here as it consists simply of a Random Walk with a subsequent correction procedure. Since MHRW rejects sampling many neighboring nodes, it only slowly diffuses over the network which consecutively may result in poor estimation accuracy [Lee et al., 2012]. Lee et al. therefore came up with another sampling method called Metropolis-Hastings algorithm with delayed acceptance (MHDA). Normal MHRW may return to nodes it has already visited before. MHDA remembers the previous nodes and increases the probability to move to one of the other neighbors. MHDA is supposed to lead to unbiased graph sampling and smaller variance than MHRW at almost no additional costs.

Another version of a random walk based algorithm that may eliminate the bias of RW is Frontier Sampling (FS). FS first samples m randomly selected seed nodes. From the list of seed nodes S, a node v is selected with a probability of $P_v = \frac{degree_v}{\sum_{u \in S} degree_u}$. One of the outgoing edges of v is then selected uniformly at random and added together with its incident node w to the sample. The set of seed nodes S gets updated by replacing

v with the newly sampled node w and the process continues by selecting the next node out of S. FS estimates have shown to be consistently more accurate than those of RW and furthermore, FS is expected to maintain robust in the presence of disconnected or loosely connected components [Ribeiro and Towsley, 2010]. Wang et al. as well proved in experiments that both FS and MHRW keep the degree distribution well. But in contrast to the statement of Ribeiro and Towsley, they conclude that FS (as well as MHRW) works better for tightly connected graphs [Wang et al., 2011].

Krishnamurthy et al. take a completely different approach to generate a graph sample. They use reduction techniques to not destroy already existing graph properties. The idea is to delete edges or nodes or to contract two incident nodes and therefore ending up with a removed edge and a merged node. Among the examined methods, DHYB-0.8 performed best [Krishnamurthy et al., 2005]. This algorithm removes with a probability of 0.8 a random edge incident to a randomly selected node or, with a probability of 0.2 removes a randomly selected edge. Unfortunately, the authors only received good results for reducing the size of the graph up to 70% in terms of nodes, whereas Leskovic and Faloutsos were able to generate samples that properly matched the properties of the whole graph for sample sizes down to 15%.

A promising sampling method belonging to the family of random sampling is called totally induced edge sampling (TIES) [Ahmed et al., 2013]. Initially, TIES chooses uniformly at random edges and adds the nodes incident to them to the sample. Next, for all edges from the original graph, it checks if its incident nodes are already sampled. If this is true, then it adds that edge to the sample. Therefore, the sample contains all possible edges between the nodes sampled in the initial phase. The authors claim that every graph sampling method naturally produces subgraphs with underestimated degrees since only a subset of a nodes neighbors may be collected. They refer to this as the downward bias. Since edge sampling results in a bias towards high degree nodes, Ahmed et al. conclude that this upward bias helps to offset the downward bias of the sampled degree distribution. A problem of edge sampling nevertheless is the missing connectivity of the sample. The TIES second step, the induction, helps to recover much of the connectivity and therefore increasing the local clustering in the sample. Ahmed et al. were able to produce good sampling results with TIES, outperforming even FFS. Indeed, one of the main advantages of TIES is that it samples the network sequentially. Since many graphs do not fit into main memory, sampling from large networks requires many random disk accesses which may lead to high I/O costs. Unlike topology-based methods, TIES does not have to explore a nodes neighbors and therefore leading to huge amounts of disk accesses. The sequential fashion in which TIES samples a graph is much more cost-saving and may therefore be a suitable sampling algorithm for large networks.

The discussion of related work shows that many different algorithms have been studied, but there is a lack of a fair comparison between them. The decision on how to sample a graph finally depends on the specific application and what requirements to the sample are necessary.

Reference	**[Doerr and Blenn, 2013]**	**[Leskovec and Faloutsos, 2006]**	**[Gjoka et al., 2010]**
Sampling Algorithm(s)	Breadth-First-Search (BFS), Depth-First-Search (DFS), Random-First-Search (RFS), all without revisiting	Random Edge Sampling (ES); Random Node Sampling (NS); Exploration: Random Walk (RW), Random Jump (RJ), Forest Fire (FFS)	RW, BFS, Metropolis-Hastings Random Walk (MHRW), Re-Weighted Random Walk (RWRW)
Algorithm Input	100 random starting nodes	Exploration: random starting node; FF: burning probability	28 uniformly random initial nodes
Network Type	Social Network	Large networks, directed	Social Network, undirected
Graph Properties/Metrics analyzed	Assortativity, Avg degree, Correlation, Diameter, Density, Power-law exponent	Full set of graph properties containing 9 different distributions and 5 single measures	Node degree, Relative size of sampled nodes from specific region to actual region size
Discovered Bias	BFS overestimates high-degree nodes while DFS underestimates them. This results also in a bias for density and power-law exponent.	RW, RJ are biased towards high-degree nodes and densely connected parts. The slope of ES degree distribution is too steep, samples are sparsely connected.	BFS and RW are biased towards high degree nodes and underestimate low-degree nodes by two orders of magnitude. BFS densely covers only some specific region.
Findings	BFS and DFS generally perform poorly. Good estimates are only held for sample sizes of more than 20-30% of the full network size. RFS performs significantly more accurate and converges to the correct value faster.	FFS generally yields good samples (with burning probability of 0.7). Good samples that match the properties of the real graph are obtained for sample sizes down to 15%.	The authors find that MHRW and RWRW perform very well, they estimate the two distributions of interest almost identically to the true uniform sample they used to validate the samples against.

Table 3.1: Sampling Methods studied by Related Work.

Reference	**[Ribeiro and Towsley, 2010]**	**[Wang et al., 2011]**	**[Lee et al., 2006]**
Sampling Algorithm(s)	Frontier Sampling (FS)	BFS, MHRW, FS	ES, NS, Snowball Sampling (SBS)
Algorithm Input	m uniformly random starting nodes	Randomly selected seed node (one for each BFS and MHRW, a set for FS)	SBS: randomly selected starting node
Network Type	Complex networks, directed, with labeled vertices and edges	Large-scale social networks, directed	Scale-free, real-world networks, undirected
Graph Properties/Metrics analyzed	Assortativity, Degree distribution, Clustering coefficient	Degree distribution, Clustering coefficient	Degree and betweenness centrality (BC) distribution, Avg path length, Assortativity, Clustering coefficient
Discovered Bias	When original graph is disconnected/loosely connected, Random Walks may get ”trapped” inside a subgraph whose properties differ from those of the whole graph.	BFS is biased towards high-degree nodes. Clustering coefficient strongly depends on the node degree. Therefore, BFS is also biased towards a larger average clustering coefficient.	NS and ES both overestimate degree and BC exponents. ES additionally underestimates clustering coefficient. SBS underestimates degree and BC exponents and also assortativity.
Findings	Results show that FS consistently outperforms Random Walk in estimating the above mentioned graph properties. FS is robust to disconnected/loosely connected components.	MHRW and FS perform well in keeping the node degree distribution. Regarding clustering coefficient, both methods perform better in tightly connected graphs with FS converging faster to the true value.	The biases stated above seem to hold for scale-free networks in general. A sampling method should be chosen depending on the property we are interested in. For example if assortativity is of interest, both ES and NS are suitable.

Table 3.2: Sampling Methods studied by Related Work.

Reference	**[Ahmed et al., 2011]**	**[Krishnamurthy et al., 2005]**	**[Lee et al., 2012]**
Sampling Algorithm(s)	ES, NS, FFS, totally induced edge sampling (TIES)	Deletion and contraction methods	MHRW with delayed acceptance (MHDA)
Algorithm Input	FFS: Randomly selected seed node	-	Randomly selected seed node
Network Type	Real-world networks, undirected, sparse	Internet topology graph, undirected	Real-world networks, undirected
Graph Properties/Metrics analyzed	Degree distribution, Path length, Clustering coefficient, Size of connected components	Average degree and deviation, Degree distribution, Spectral analysis	Degree distribution, Largest connected component
Discovered Bias	ES is biased towards high-degree nodes. NS results in too sparsely connected nodes. Any sampling algorithm naturally underestimates the degrees in the degree distribution (since only a subset is sampled).	The authors claim that with such graph reduction methods they do not destroy existing graph properties, in contrast to constructive methods which have to reproduce the original properties.	Normal MHRW can get stuck at a node and therefore only slowly diffuses over the space. This behavior can lead to a reduced estimation accuracy.
Findings	TIES generally outperforms the other algorithms. The upward bias from the edge sampling process offsets the downward bias of the sampled degree distribution. The induction step helps to recover much of the connectivity in the sample.	DHYB-0.8 (remove random edge incident to randomly selected node with probability 0.8, else remove random edge) performs best among the examined methods. Good results were obtained by reducing the graph up to 70% in terms of nodes.	MHDA is supposed to lead to unbiased samples while also achieving higher sampling efficiency than MHRW.

Table 3.3: Sampling Methods studied by Related Work.

4

Requirements

This chapter gives an overview of what the framework should be capable of. The following sections analyze the requirements for each of the tasks of sampling, creating the snapshots and of displaying the results. The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" used for the requirements can be interpreted as described in RFC 2119 [Bradner, 1997].

4.1 Requirements for Sampling

Because of the sheer volume of the Wikidata knowledge graph, the framework implemented during this thesis is based on sampling. Processing the whole knowledge graph would result in requiring huge amounts of resources like memory and time. Consequently, researchers rely on sampling techniques to study smaller subgraphs that have similar properties as the graph they were derived from.

As discussed in Chapter 3, sampling from large graphs is possible in various ways. There is no perfect solution, but one can conclude some appropriate approaches from related work. For example, topology-based sampling methods result in better (e.g. in terms of connectivity) samples than random edge or node sampling and should therefore be preferred (see Section 3.2). Regarding the sampling process, the framework focuses on the following requirements:

- **RS-1**: The framework MUST extract a sample set of nodes and edges out of the Wikidata knowledge graph.
- **RS-2**: The sample set MUST be connected. Hence, the number of connected components in the sample must be one.
- **RS-3**: The framework MUST take into account constraints the user sets on the graph sampling. These constraints include the number of vertices to be in the sample and the setting of a seed node from where the sampling algorithm starts collecting further nodes and edges.

- **RS-4** The framework MAY take as input two seed nodes that create overlapping samples and therefore are connected. This requirement includes finding a path between the seed nodes.
- **RS-5**: The framework is RECOMMENDED to create samples having similar graph properties as the graph they are derived from.
- **RS-6**: The system SHOULD give the user a choice among several sampling techniques to choose from.

4.2 Requirements for Creating Snapshots

After collecting a connected subgraph sample from Wikidata, the actual core functionality of the framework consists of building snapshots. In this stage, a current sample version is available and the framework has to create previous versions of that sample graph. Hence, it has to access revision data and undo the changes generating a series of snapshots while going back in time.

Wikidata offers possibilities to get old versions of the Wikidata knowledge graph. For example, it makes old versions of the whole data dumps[1] available. Though, these dumps are not available at regular intervals. Wikidata also provides a SPARQL endpoint to query the Wikidata edit history[2]. This history query service allows to query the full state of the Wikidata knowledge graph after any revision. However, we are not interested in getting snapshots of the whole graph but only from our sample graph in order to save resources. Besides, this query service only offers data from the creation of Wikidata in 2012 until July 1st 2018.

What the framework needs to create those snapshots is the actual revision data of the triples in the sampled subgraph. It could request this revision data from the Mediawiki API[3]. The framework created during this thesis, nevertheless, accesses the revision data from a database provided by the Dynamic and Distributed Information Systems Group (DDIS). The following requirements refer to the process of creating snapshots:

- **RC-1**: The framework MUST take into account constraints that the user sets on the graph. These constraints include the number of changes in each snapshot and a timestamp until when to create the snapshots.
- **RC-2**: Each snapshot SHOULD be equipped with a time interval pointing out the range of time in which these changes applied or in which that snapshot was valid.
- **RC-3**: The framework MUST implement the undoing of changes in the correct order. In this way, historic versions of the graph can be reconstructed with high precision.

[1] *https://www.wikidata.org/wiki/Wikidata:Database_download#Old_JSON_and_RDF_dumps*
[2] *https://www.wikidata.org/wiki/Wikidata:History_Query_Service*
[3] *https://www.mediawiki.org/wiki/API:Revisions*

- **RC-4**: The framework MUST handle the undoing of changes correctly. Therefore, it must implement different undoing events (e.g. creating a statement, removing a statement, updating a value) accurately.

4.3 Requirements for Displaying the Results

After having created the historic snapshots of the Wikidata sample graph, they have to be presented to the user in a reasonable way. The following bullet points list up the requirements regarding this issue:

- **RF-1**: The framework MUST return the current sample and the corresponding snapshots in a standard RDF format. There SHOULD be several formats available to possibly choose from.
- **RF-2**: The framework SHOULD offer the possibility to return the results in a format natively supported by NetworkX. The chosen format SHOULD NOT loose any data or information during the transformation process.

5

Implementation

With the knowledge about related work on sampling techniques, the way Wikidata structures its data in RDF and the requirements defined in Chapter 4, we can move forward to the practical realization of the framework that creates snapshots of sampled Wikidata knowledge. This chapter starts with an overview over the framework, before explaining each part of it in the following sections.

5.1 Framework Overview

The framework addressed in this thesis extracts a sample out of the Wikidata knowledge graph and creates snapshots of that sample by going back in time and undoing revisions. Figure 5.1 displays a diagram with the data flows necessary to fulfill these tasks.

First, the user must feed the application with all the necessary information. This includes the choice of a specific sampling technique, the size of the sample in terms of the number of nodes, a seed node from where the sampling technique starts, a past timestamp until when to create the snapshots, the number of changes in each snapshot and finally, the output format as well as the database credentials. All this information can be entered or changed in the framework's configuration file named *config.ini*.

Provided with these details, the framework accesses a MySQL[1] database made available by DDIS. This database stores the Wikidata data as triples with an ID for each subject, property and object. Such a database schema makes it relatively straightforward to sample a subset of the knowledge graph with traversal-based methods. Simply get the current node, in this case the current subject, before selecting randomly a certain number of its neighbors, the objects. The sampling is done with the help of NetworkX[2], a Python package for the creation and manipulation of networks. Sections 5.2 to 5.4 will address the topic of sampling with NetworkX in more detail.

After having collected the nodes and edges belonging to a connected Wikidata sample graph, the historic snapshots must be generated. This is the actual core process of the framework. For this task, the university provides a database which stores the Wikidata revision data. The revision data is retrieved for each subject, property tuple in the

[1] *https://www.mysql.com/*
[2] *https://networkx.github.io/*

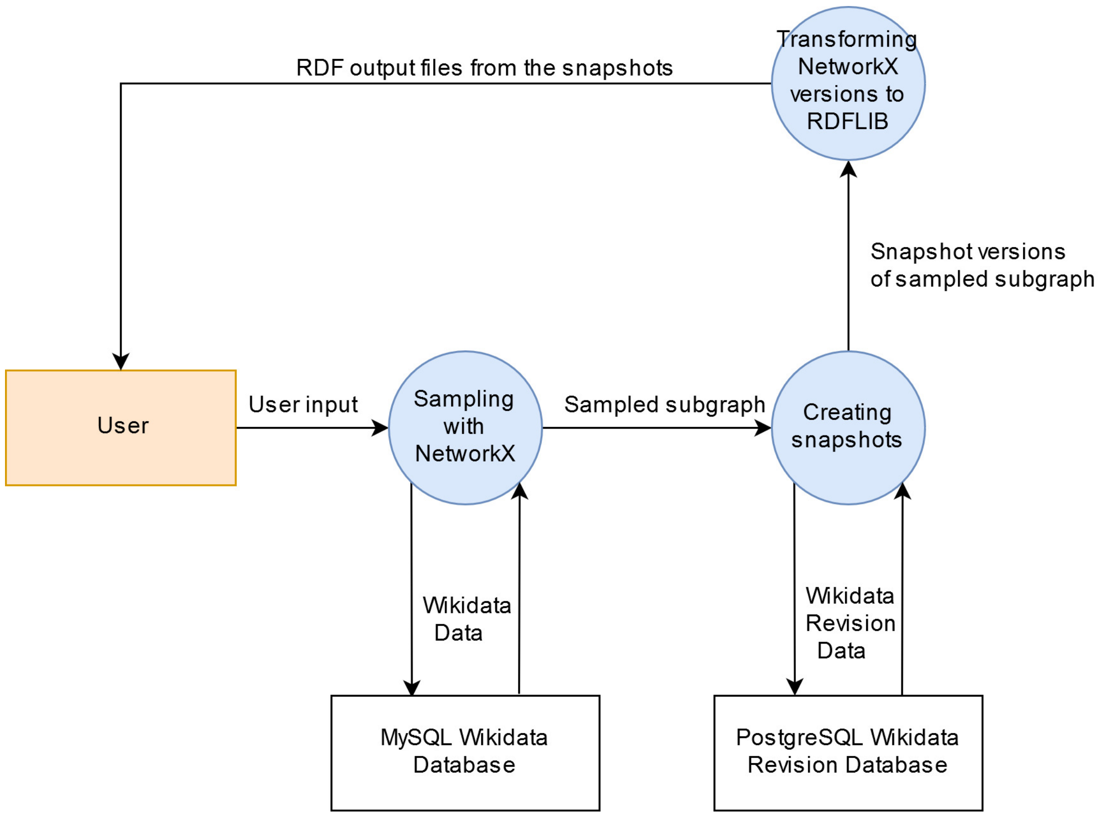

Transforming NetworkX versions to RDFLIB
RDF output files from the snapshots
Snapshot versions of sampled subgraph
User
User input
Sampling with NetworkX
Sampled subgraph
Creating snapshots
Wikidata Data
Wikidata Revision Data
MySQL Wikidata Database
PostgreSQL Wikidata Revision Database

Figure 5.1: Data Flow Diagram of Framework

sample from this database. Therefore, the framework can undo the changes for each triple by going back in time and ending up with several snapshots of what the knowledge graph sample looked like back then. This part of the implementation is explained in more detail in Section 5.5.

The last part of the framework addresses the formatting of the output which are the initial sample and its following snapshots. The output must certainly be returned in a standard RDF format. This can be done by transforming the sample graph from NetworkX to RDFLib[3], a Python package for working with RDF. With RDFLib it is possible to serialize the graph data into various formats. Section 5.6 describes further how data is represented by RDFLib and which formats to return RDF documents the framework supports.

5.2 Sampling Prerequisites

This section addresses the prerequisites for the sampling part of the framework. Section 5.2.1 introduces NetworkX which is used by the framework for the actual sampling

[3] *https://github.com/RDFLib/rdflib*

	directed	multiple edges
Graph()		
DiGraph()	✓	
MultiGraph()		✓
MultiDiGraph()	✓	✓

Table 5.1: NetworkX Graph Types

process as discussed in Section 5.3. The schema of the database from which the Wikidata data is retrieved is discussed by Section 5.2.2.

5.2.1 NetworkX

The framework is written in the Python[4] programming language. Therefore, the usage of NetworkX, a Python package for creating and manipulating networks, comes handy when dealing with graph sampling. Apart from that, NetworkX also provides algorithms to study various graph properties for graph analysis. To generate and store the Wikidata data, the framework first creates an empty NetworkX graph with no nodes an no edges. The package provides different types of graphs. Table 5.1 lists these four types according to the two properties they differ.

The framework must certainly store the data in either DiGraph or MultiDiGraph since the direction of the edges is crucial when working with RDF data. These two directed types of graphs offer additional functions specific to directed edges such as calculating the in- or out-degree or getting the predecessor or successor of a node. MultiGraph or MultiDiGraph allow to add more than one edge between any pair of nodes. In Wikidata, multiple edges between subject and object nodes are possible which is why it is logical to initialize a MultiDiGraph for doing the sampling. Unfortunately, there is one drawback when working with NetworkX graphs that are directed and/or allow multiple edges: many of the algorithms provided by NetworkX are not defined for these types. This includes for example calculating the number of components or the clustering coefficient. Nevertheless, it is possible to convert MultiDiGraph into the undirected standard Graph when analyzing of such properties is of interest.

Having initialized a NetworkX MultiDiGraph, the sampling can start by adding nodes and edges to it. NetworkX allows nodes to be any kind of hashable objects. This prerequisite is fulfilled since nodes will be filled with objects of type string. Nodes and edges are added to the MultiDiGraph using the NetworkX functions *add_node()* and *add_edge()* to add one node at a time or add an edge between two nodes specified as parameters. Additionally, it is possible to add attributes to them. This is necessary since the type (e.g quantity, globecoordinate, time) or the language tag of the object must be stored as well to be able to transform the sample graph into a standard RDF format with corresponding namespaces and language tags later on.

[4] *https://www.python.org/*

5.2.2 MySQL Database Schema

For the sampling part, the framework accesses Wikidata knowledge stored on a DDIS MySQL database server. MySQL is an open-source relational database management system. Figure 5.2 shows the schema of the database used for extracting the samples.

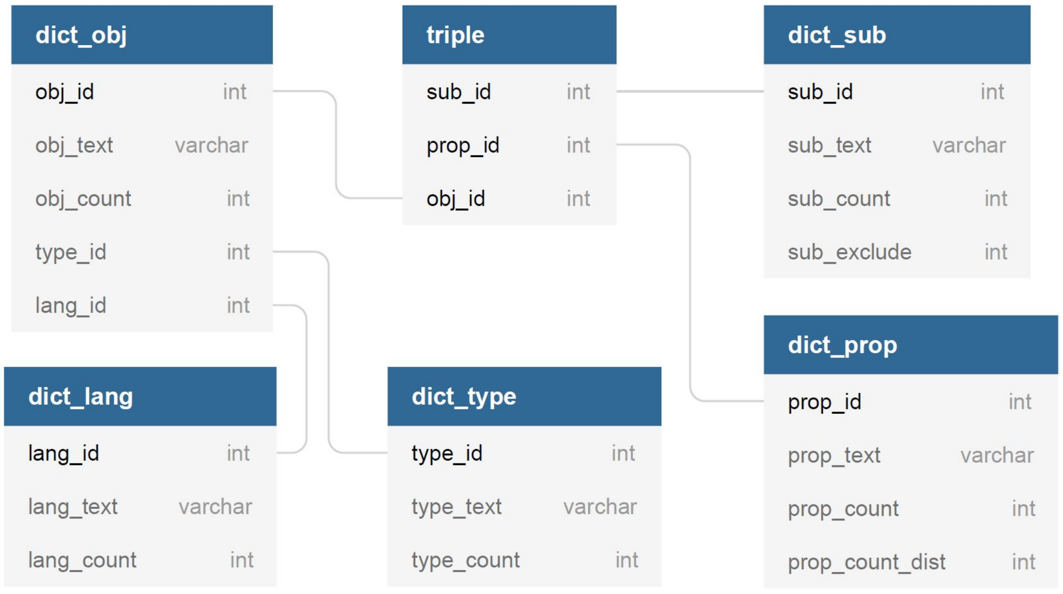


Figure 5.2: MySQL Database Schema for Retrieving Wikidata Knowledge

The data is stored in a structure called dictionary tables. The table called *triple* stores the Wikidata data in triple form with subject, property and object similar to RDF. But it stores each of them with identification numbers. Therefore, one has to query the tables *dict_sub*, *dict_prop* and *dict_obj* with the corresponding id to get the actual content for each subject, property and object. The contents are stored for every table in the column ending with "*_text*".

The table *dict_sub* consists not only of Wikidata items but also of properties since properties as well can be the subject of Wikidata triples (e.g. for defining a properties constraints). Statement nodes, the ones beginning with an item identifier such as Q183 followed by a UUID, are not contained in *dict_sub*. This results in having no information about qualifier statements or references supporting a statement.

The table containing all the properties called *dict_prop* contains only properties of the P-type, as for example P31. Therefore, only the properties defined in the Wikidata property namespace[5] are available. Properties referring to other namespaces outside of Wikidata like label[6] or description[7] are not stored in *dict_prop*.

[5] *https://www.wikidata.org/wiki/Wikidata:List_of_properties*
[6] namespace: *http://www.w3.org/2000/01/rdf-schema#*
[7] namespace: *http://schema.org/*

Dict_obj refers to the table that consists of additional information for the object IDs. Objects can have one of six different *type_ids* which are again specified further in the table *dict_type*. These types are strings, wikibase-entities, quantity, globecoordinate, monolingualtext and time. Objects of type monolingualtext also contain a *lang_id* referring to the table *dict_lang* which specifies the language tags.

This database schema forms the initial situation for the actual sampling process which is discussed in Section 5.3. Following from the description of the schema, the database contains only direct statements for properties defined in the Wikidata property namespace. Taking again the Wikidata statement from Figure 2.3, the database stores only the triple *Q183–P3086–"100"^^xsd:decimal*, and statement nodes are not part of this MySQL database.

5.3 Sampling Process

This section addresses the frameworks first task: the extracting of a sample out of the Wikidata knowledge graph.

Towards sampling, the most important requirement is the extraction of a graph sample that is connected. Hence, sampling techniques like random node or edge sampling are not suitable. First, they do not sample connected graphs and second, they have been shown to produce samples which fail to preserve many graph properties (as discussed in Section 3.2). Due to these limitations and the requirement of getting connected samples, topology-based sampling methods are optimal for the sampling part of the framework. I chose several of these sampling algorithms for implementation. Among the selected ones are BFS, FFS, SBS, RFS, RW and MHRW.

This section explains the sampling process based on a more detailed description of FFS in Section 5.3.2 and MHRW in Section 5.3.3. FFS serves as a representative of traversal-based methods, whilst MHRW takes this position for random walks. But first, Section 5.3.1 clarifies the general process of adding a neighboring node with its edge to the sample graph.

5.3.1 Sampling a Neighboring Node and Edge

Nearly all topology-based sampling methods have in common that they move forward to a neighboring node, or rather a successor node (since the Wikidata graph is directed), of a current node. The successor and the edge leading to it are sampled and the current node will now be one of previously sampled successor nodes to continue this process. Because all of the algorithms applied in the framework share this idea of adding successor nodes, Figure 5.3 summarizes this process.

While sampling neighboring nodes of a current node, the framework first sends a query to the table *triple* with the *subject_id* of the current node. Depending on the algorithm, all or a certain number of (random) triples are returned. The objects of these triples represent the neighboring nodes of the current subject node. Nevertheless, since the table *triple* stores only IDs referring to other tables containing the actual values, we need some

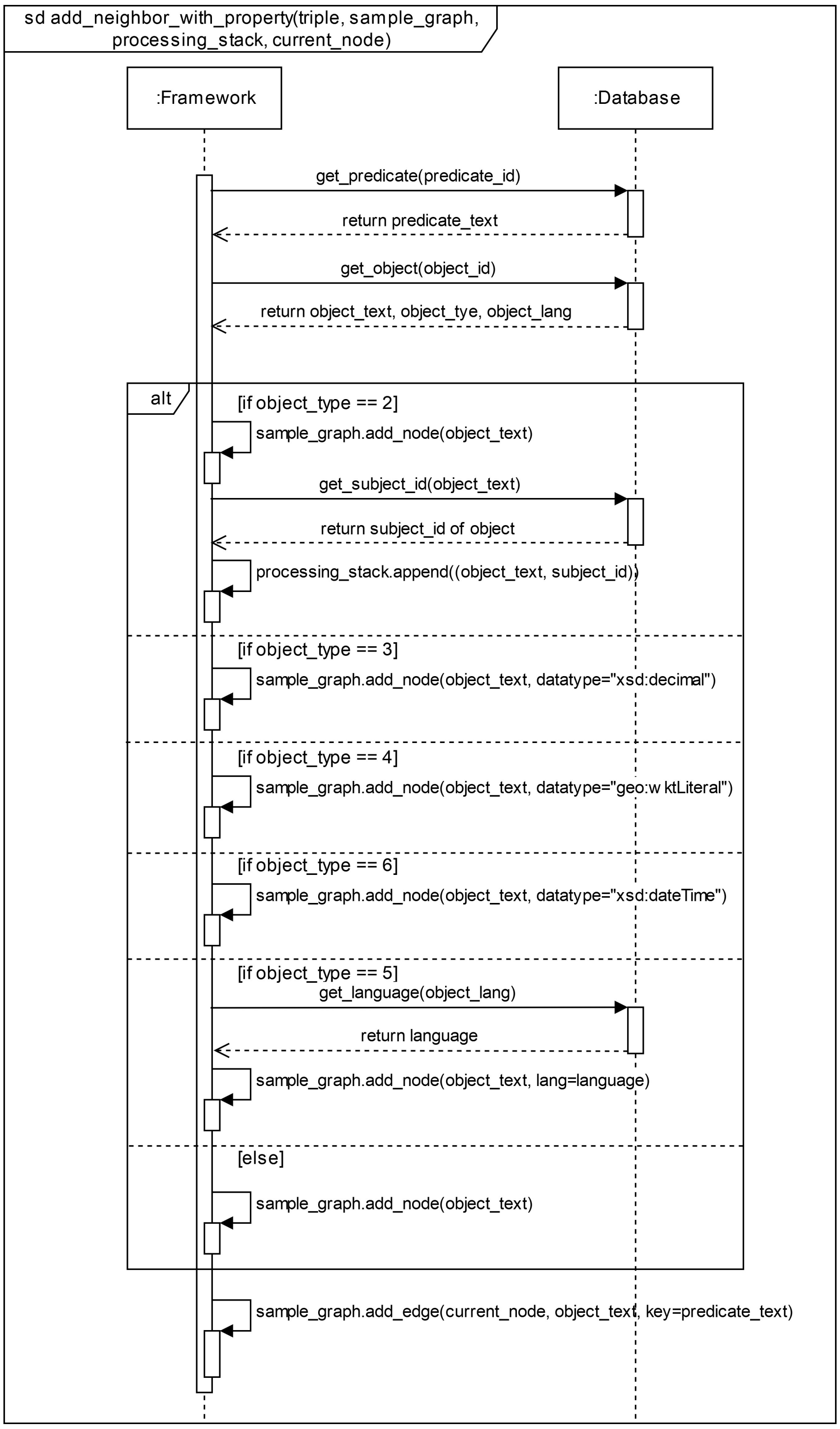

sd add_neighbor_with_property(triple, sample_graph, processing_stack, current_node)
:Framework
:Database
get_predicate(predicate_id)
return predicate_text
get_object(object_id)
return object_text, object_tye, object_lang
alt
[if object_type == 2]
sample_graph.add_node(object_text)
get_subject_id(object_text)
return subject_id of object
processing_stack.append((object_text, subject_id))
[if object_type == 3]
sample_graph.add_node(object_text, datatype="xsd:decimal")
[if object_type == 4]
sample_graph.add_node(object_text, datatype="geo:w ktLiteral")
[if object_type == 6]
sample_graph.add_node(object_text, datatype="xsd:dateTime")
[if object_type == 5]
get_language(object_lang)
return language
sample_graph.add_node(object_text, lang=language)
[else]
sample_graph.add_node(object_text)
sample_graph.add_edge(current_node, object_text, key=predicate_text)

Figure 5.3: Sequence Diagram of Sampling a Neighboring Node with Property Edge

more information to add such a successor node with the corresponding property edge to the sample. As can be seen in Figure 5.3, the framework has to query first for the actual values for the property as well as the object. For the property, the method returns just the property text. For the object, it returns as well the object text but also object type and, if available, object language. Then, the *add_neighbor* method takes into account the object type and differentiate between six ways to handle the adding of the object to the sample graph.

When the database returns for the object type *5*, the object is of *monolingualtext* type. For such types the framework sends a query to the table *dict_lang* to get for the language id the corresponding language tag. After the tag is known, the object can be added to the sample with an additional attribute ”lang” for the language tag. Object type number *3* corresponds to type *quantity*, *4* to type *globecoordinate* and *6* refers to type *dateTime*. For these three types, the system needs no additional query. It is enough to just add the node to the sample graph and pass along the data type via the datatype attribute in the *add_node* method. If the object type is *2*, then the object is an entity such as an item or a property. Only objects of this type can be future subject nodes and be processed for their successor nodes. All the other node types are sink nodes, meaning that they do not possess any outgoing edges. Therefore, the algorithm only adds entity nodes to a stack from which it pops off the next node to continue the sampling with. Also, for such nodes, the framework needs to send another query to the database: this is because the id of an entity is not the same for subject and object. Hence, a query which looks for the *subject_id* in the table *dict_sub* with the object text equal to the subject text is necessary.

The last else clause displayed in Figure 5.3 refers to objects containing simple strings. They can just be added to the sample graph with no additional attribute values. Finally, this method adds the edge to the sample by setting parameters for current node, successor node and the type of property as an attribute called ”key”. This way, the key attribute not only stores the type of the property but also distinguishes multiple edges between a pair of nodes.

5.3.2 Forest Fire Sampling

Figure 5.4 shows the sequence diagram of the Forest Fire sampling method. The algorithm takes as input a seed node and the number of nodes to be in the sample graph. Then, it initializes a NetworkX MultiDiGraph and retrieves the id of the subject from the database. Furthermore, it adds the seed node to the MultiDiGraph and a list called *processing_stack*. The FFS method enters a while-loop which stops when the sample contains enough nodes. From the *processing_stack* a node is popped off to be the next node to collect successor nodes from. The *processing_stack* works like a First-In First-Out (FIFO) queue. This way, the algorithm ensures that nodes detected first are also processed first. As discussed in Section 3.2, FFS samples only a fraction of neighboring nodes. It generates this fraction from a geometric distribution as proposed by Leskovic

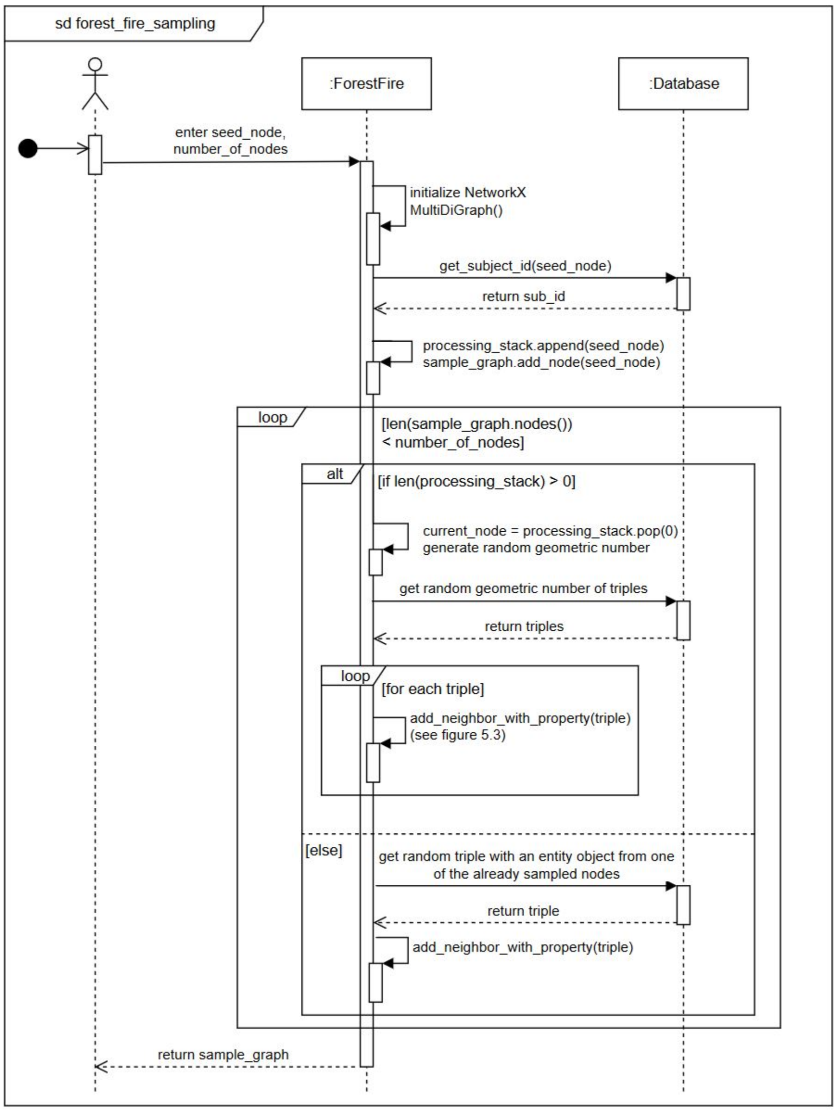

sd forest_fire_sampling
:ForestFire
:Database
enter seed_node, number_of_nodes
initialize NetworkX MultiDiGraph()
get_subject_id(seed_node)
return sub_id
processing_stack.append(seed_node)
sample_graph.add_node(seed_node)
loop
[len(sample_graph.nodes()) < number_of_nodes]
alt
[if len(processing_stack) > 0]
current_node = processing_stack.pop(0)
generate random geometric number
get random geometric number of triples
return triples
loop
[for each triple]
add_neighbor_with_property(triple)
(see figure 5.3)
[else]
get random triple with an entity object from one of the already sampled nodes
return triple
add_neighbor_with_property(triple)
return sample_graph

Figure 5.4: Sequence Diagram of Forest Fire Sampling

et al. To do so, the framework uses the Python package NumPy[8] which provides the feature *numpy.random.geometric*[9] to draw samples from a geometric distribution. The framework then queries for that generated number of random triples with the subject id equal to the current node id that is being processed. In a for-loop, each of these triples is then passed to the *add_neighbor_with_property* function as explained in Figure 5.3. Therefore, the algorithm adds for each subject node a geometrically generated amount of neighbors to the sample with their corresponding property edges. If the successor node

[8] *https://numpy.org/*
[9] *https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.random.geometric.html*

is of type entity, it will get pushed to the *processing_stack* to eventually be processed in a later loop as subject node to collect neighbors from. Unfortunately, since many nodes in the Wikidata knowledge graph are not entity nodes like items or properties, the "fire may go out", meaning that no further entity nodes are available on the *processing_stack*. To handle such cases, I modified the original FFS algorithm and added an alternative clause: if *processing_stack* gets empty, the algorithm searches in the database for a triple with an already sampled subject node and an object that is an entity object not yet sampled. This way, the *processing_stack* is filled again, the sample graph stays connected and the algorithm traverses the network in a yet unexplored direction.

5.3.3 Metropolis-Hastings Random Walk

To also illustrate a random walk based sampling algorithm, Figure 5.5 shows a sequence diagram of the MHRW method. This method starts similar like FFS with a user entering a parent node from where the walk starts and the number of nodes to be in the sample graph. Then, it initializes a NetworkX MultiDiGraph to be the sample graph and queries the database for some more information about the parent node. This includes the id and the degree, as well as the triples containing neighbors of the parent node. The MHRW algorithm adds the parent node to the sample graph and the triples to *neighbor_list*, a list that contains all the triples with subject id corresponding to the node id of the node from where the next neighboring node will be sampled. The algorithm then enters a while-loop until the number of nodes in the sample is smaller than the number of nodes as specified in the configuration file. If there are triples in the *neighbor_list*, randomly select one of them and get the neighboring child node information from the database with the help of the object id. The information to be returned includes the actual content, the type and the degree. Furthermore, the method generates a random number between 0 and 1. If such a random number is smaller than the minimum value of either 1 or the division of the parent degree by the child degree, then the algorithm adds the child node with its corresponding property edge to the sample. Like this, neighboring nodes with a smaller degree than the parent node are always sampled and some of the nodes with higher degrees are rejected. This is the actual core idea of the MHRW to correct the bias of samples towards high degree nodes. If the neighboring node added to the sample is of type entity, then the role of the child node switches to the one of the parent node, the *neighbor_list* gets cleared and filled with the triples belonging to our former child node. Should the algorithm reject a node or add a literal node to the sample, it simply continues to randomly select another triple from the list.

5.3.4 Other Sampling Techniques

The other sampling algorithms the frameworks contains are BFS, SBS, RFS and RW. The implementations of BFS and SBS are very similar to the FFS algorithm. The difference lies in the number of neighbors they sample. While BFS samples all the successors of a node and therefore provides a complete view of a part of the knowledge graph, SBS is user-dependent and lets the user decide how many neighboring nodes

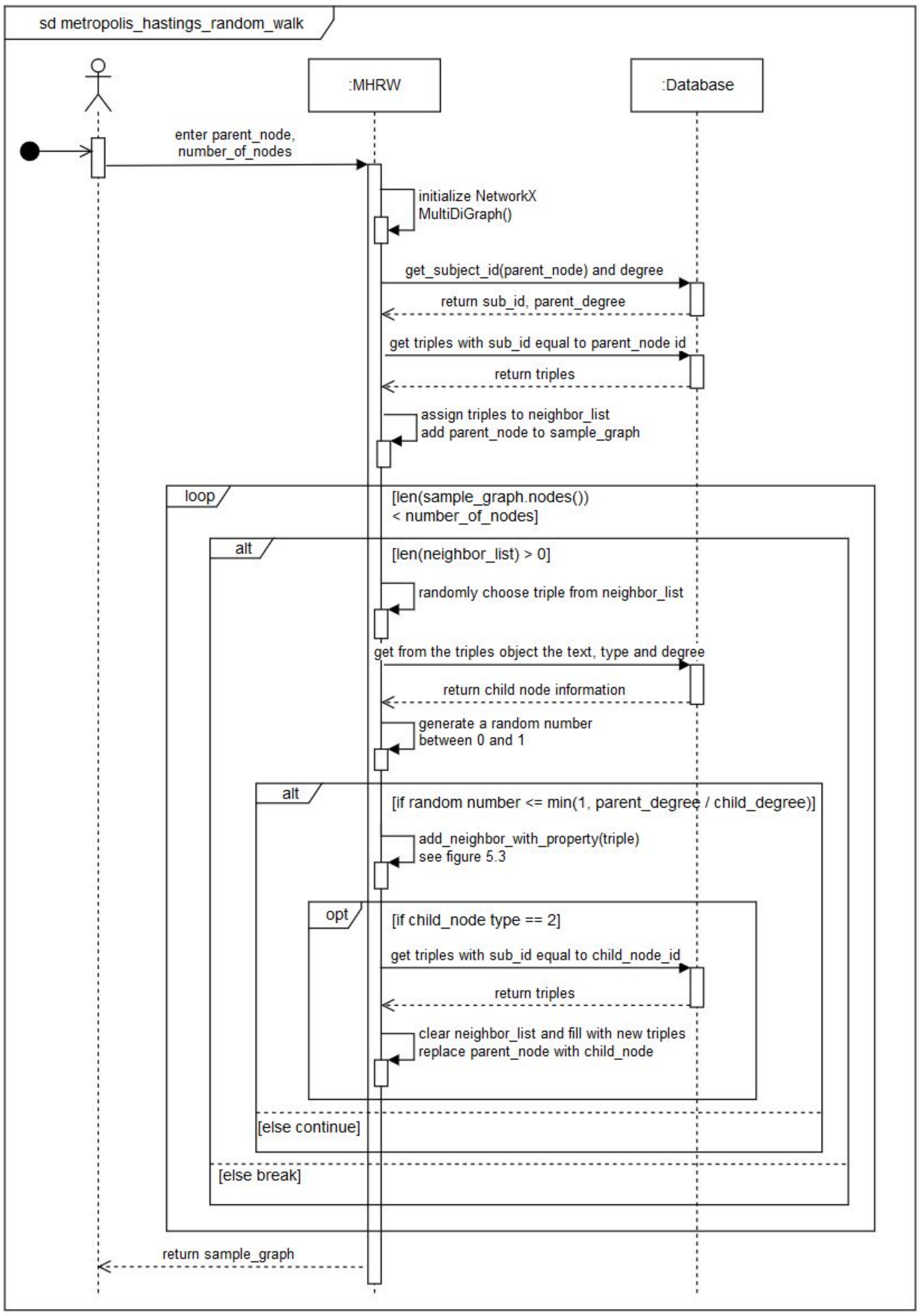

sd metropolis_hastings_random_walk
:MHRW
:Database
enter parent_node, number_of_nodes
initialize NetworkX MultiDiGraph()
get_subject_id(parent_node) and degree
return sub_id, parent_degree
get triples with sub_id equal to parent_node id
return triples
assign triples to neighbor_list
add parent_node to sample_graph
loop
[len(sample_graph.nodes()) < number_of_nodes]
alt
[len(neighbor_list) > 0]
randomly choose triple from neighbor_list
get from the triples object the text, type and degree
return child node information
generate a random number between 0 and 1
alt
[if random number <= min(1, parent_degree / child_degree)]
add_neighbor_with_property(triple)
see figure 5.3
opt
[if child_node type == 2]
get triples with sub_id equal to child_node_id
return triples
clear neighbor_list and fill with new triples
replace parent_node with child_node
[else continue]
[else break]
return sample_graph

Figure 5.5: Sequence Diagram of Metropolis Hastings Random Walk

should be sampled for each entity node (by setting this number in the *config.ini* file). RFS starts as well from a seed node and collects for each entity node sampled all the triples with neighbors in a list. To add a node to the sample, the RFS algorithm randomly selects a triple from that list and executes the *add_node_with_property* function with it. If the newly added object node is of type entity, it again adds its neighbors to the list to eventually be selected later. In case the object was of another type, the algorithm simply continues with another randomly selected triple from the list.

The implementation of RW is similar to a very simplified version of MHRW. RW just walks from one node to its successor node without considering any degrees. If the successor node is a literal node, the algorithm adds this node to the sample and continues sampling neighbors from the current subject node. In case the next node is an entity node, it adds this node as well but the sampling now continues from that freshly sampled successor node. Similar to the RW proposed by another work, I implemented RW with a backtracking probability [Leskovec and Faloutsos, 2006]. They propose to restart the random walk from its initial node with a probability of 0.15 after every sampled node. With this approach the authors want to ensure exploration into other directions of the network. While implementing, I tested this variant and did not get satisfactory results: the algorithm very quickly slows down, even for sampling only 1000 nodes. This could have been expected, since restarting from the initial node with a quite high probability makes it difficult to explore more distant regions of the network. As a result, I decided to change the proposed technique by keeping the probability, but restarting from a random (already sampled) entity node instead of the initial one. This way, the original idea to explore other directions is maintained, but the algorithm is much less likely to get stuck in some initial network part.

The next section compares these implemented sampling techniques by various important graph metrics and their execution time.

5.4 Comparison of the Sampling Techniques

In this section, the implemented sampling techniques are compared to each other. For the comparison, I chose the mostly used graph metrics as applied in the related work mentioned in Tables 3.1 to 3.3. Average degree, degree distribution, assortativity, clustering coefficient and average shortest path belong to important graph metrics and are discussed in this section. I also include the execution time as a metric to compare the algorithms. Table 5.2 lists the machine specifications of the notebook I used to execute the algorithms.

Operating System	Windows 10 Pro 64 Bit
RAM	8 GB
CPU	Intel Core i7-5500U @ 2.40GHz

Table 5.2: Machine Specification

Tables 5.3 to 5.5 list the elected graph metric results for the sampling techniques, for

1'000 sampled nodes, 10'000 sampled nodes and 100'000 sampled nodes. From these metrics, there are several conclusions to draw. But first, the metrics itself are shortly explained.

5.4.1 Graph Metrics

The *average degree* refers to the average of the sum of both in-degrees and out-degrees for the nodes in the sampled network.

The *degree distribution* is the distribution of the degrees over the whole network. The plots of the degree distributions in Figure 5.6 show the distributions separately for each in-degrees and out-degrees.

The concept of *assortativity* measures the extent to which nodes tend to connect to other nodes of similar or of opposite sort [Newman, 2002]. In this comparison, I measure the assortativity degree (which is the most common form of assortativity). The values it can take range from $-1 \leq \rho \leq 1$ and therefore, from disassortative to assortative. A network is assortative if high-degree nodes are generally connected to other high-degree nodes and low-degree nodes are rather connected to low-degree nodes. Disassortativity means that high-degree nodes are more likely to have low-degree node neighbors as well as the other way around, low-degree nodes tend to connect to high-degree nodes.

The *clustering coefficient* is a graph metric that analyzes to which extent nodes tend to cluster together [Watts and Strogatz, 2011]. If many neighbors of a specific node are connected to each other as well, the clustering coefficient is high. If these neighbors are poorly connected, then this measure is low. It is therefore a measure of the degree to which neighbors of a node are neighbors of themselves as well. The value the clustering coefficient can take ranges from 1 (neighborhood is fully connected) to 0 (no connections between neighboring nodes themselves).

The *average shortest path length* measures the mean shortest number of edges that have to be followed to connect any two pair of nodes in the network. The average path length correlates with the edge density of a network [Smith, 2007]. The more densely a network is connected, the shorter is the average path length.

5.4.2 Comparison of the Graph Metrics

Analyzing the Tables 5.3 to 5.5, there are several things worth noting. Regarding execution time, all the algorithms slow down when the number of nodes they are sampling increases. The slowing down in comparison to the growth of the sample is between 1.75 times (in case of BFS) to 8.8 times (in case of FFS) when looking at the samples with 1'000 nodes to the ones with 100'000 nodes. Also, the tables indicate that the more neighbors for each node an algorithm samples, the faster it is. This becomes apparent on the decreasing execution times from FFS, SBS and BFS. RW and RFS, both algorithms which only add one neighbor at a time, need in most cases significantly more execution time.

According to the assortativity, one can observe that the Wikidata knowledge graph is rather disassortative because entity nodes with high degrees are connected to many

Graph Metrics for 1'000 Nodes					
Sampling technique	Execution Time	Avg Degree	Assortativity	Clustering Coefficient	Avg Shortest Path
FFS	10.0 sec	2.61	-0.104	0.031	9.07
SBS (n=5)	7.6 sec	2.60	-0.194	0.021	7.36
SBS (n=10)	7.5 sec	2.79	-0.325	0.032	5.67
BFS	4.8 sec	2.13	-0.687	0.008	2.55
RFS	19.1 sec	2.12	-0.268	0.007	6.35
RW	24.6 sec	2.59	-0.176	0.0284	7.89
MHRW	329.6 sec	2.38	-0.363	0.038	21.48

Table 5.3: Measures for Samples with 1'000 Nodes

Graph Metrics for 10'000 Nodes					
Sampling technique	Execution Time	Avg Degree	Assortativity	Clustering Coefficient	Avg Shortest Path
FFS	159.5 sec	3.1	-0.032	0.040	7.85
SBS (n=5)	95.3 sec	3.03	-0.032	0.031	7.48
SBS (n=10)	85.7 sec	2.87	-0.066	0.024	6.56
BFS	58.4 sec	2.56	-0.345	0.039	3.94
RFS	169.7 sec	2.32	-0.280	0.005	5.15
RW	371.4 sec	2.91	-0.070	0.030	7.04

Table 5.4: Measures for Samples with 10'000 Nodes

Graph Metrics for 100'000 Nodes				
Sampling technique	Execution Time	Avg Degree	Assortativity	Clustering Coefficient
FFS	8824 sec	4.15	-0.055	0.052
SBS (n=5)	1254 sec	3.55	-0.027	0.042
BFS	841 sec	2.824	-0.431	0.023
RFS	4251 sec	2.73	-0.090	0.014
RW	11494 sec	3.18	-0.043	0.035

Table 5.5: Measures for Samples with 100'000 Nodes

low-degree literal nodes. This is also confirmed by the negative assortativity numbers listed in the Tables 5.3 to 5.5. The algorithms, nevertheless, show great differences in that graph metric as well. In general, it can be stated that the more neighbors an algorithm samples, the more disassortative its samples are. This is especially true in case of BFS which samples every neighbor. Also, this assumption agrees with the findings of related work which has shown that especially samples collected with BFS tend to be more disassortative than the original networks [Lee et al., 2006].

The clustering coefficient measured for all algorithms and sample sizes tend to be adjacent to zero. Therefore, a nodes neighbors are very loosely connected between themselves. This may not be surprising since the Wikidata knowledge graph is very sparse. Although all the clustering coefficient numbers are quite low, a negative correlation with the average degree can be observed: The lower the average degree, the lower the clustering coefficient. This behavior is consistent with empirical observation of related work [Bloznelis, 2013][Wang et al., 2011].

The average degree seems to result in too small measures for RFS and BFS. For BFS, this can be explained because of the boundary bias: the nodes sampled in the last round are missing a large number of neighbors [Lee et al., 2006]. Therefore, BFS is biased towards a too small average degree. In case of RFS, this sampling technique results in very sparsely connected sample graphs and hence, low average degree because this technique may spread out over the network quickly.

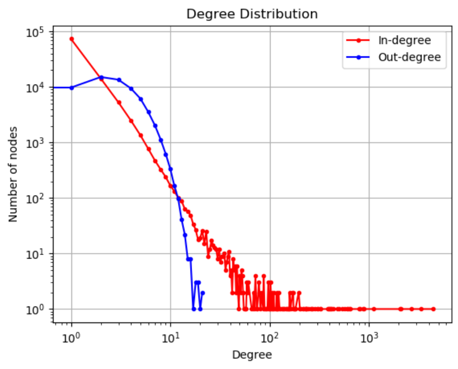
The numbers for the average shortest path length in the tables are quite different. Average path length decreases as average degree increases [Lee et al., 2006]. This is true for some of the measurements in the tables, but certainly not for BFS. Since the average degree for BFS is strongly biased (boundary bias), above general rule does not hold in this case. On the contrary, it is only logical that BFS has the smallest values for the average shortest path because it densely samples all nodes of a specific part of the graph.

Only Table 5.3 contains measures for MHRW. I chose to discard this technique for sampling more than 1'000 nodes since it performed very poorly in terms of execution time. Because many neighboring nodes are rejected when using this technique, MHRW suffers from a very slow diffusion over the network [Lee et al., 2012]. If the current subject node has a much lower degree than all its entity successor nodes, this algorithm may also get stuck. Therefore, I recommend not to apply this technique for Wikidata sampling.

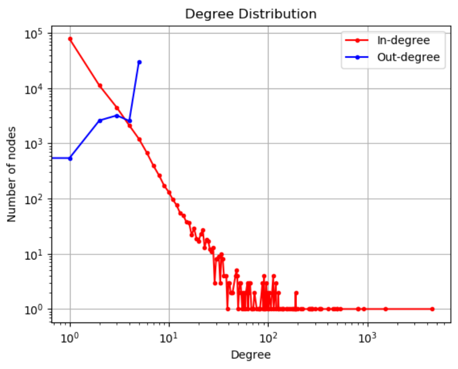
Figure 5.6 illustrates the degree distributions of FFS, SBS ($n = 5$), BFS, RFS and RW for each in-, and out-degrees. The distributions of the in-degrees are quite similar for all the techniques, except for RFS which contains less high-degree outliers.

The difference in the plots lies mainly in the distributions of the out-degrees. For FFS, this distribution follows the geometric distribution as proposed by related work [Leskovec and Faloutsos, 2006]. Most of the nodes in the sample have an out-degree of 2 or 3, as can be seen at the peek of the curve. After that, the curve falls steeply and only very few nodes have an out-degree around 20. The out-degree distribution of SBS looks a bit peculiar since it suddenly stops at degree 5. This is because this algorithm always selects 5 (or less if not as many are available) neighbors to add to the sample. The out-degree distributions for BFS, RFS and RW look similar except for the number of sampled low-degree nodes. BFS samples much less low-degree nodes than the other ones. This is consistent with the findings in related work: BFS underestimates low-degree nodes. Furthermore, these tree plots show an oversampling of high-degree out-nodes compared to in-degree out-nodes. From this can be concluded that these three algorithms tend to be biased towards high-degree nodes which is as well consistent with the findings of related work listed in Tables 3.1 to 3.3.

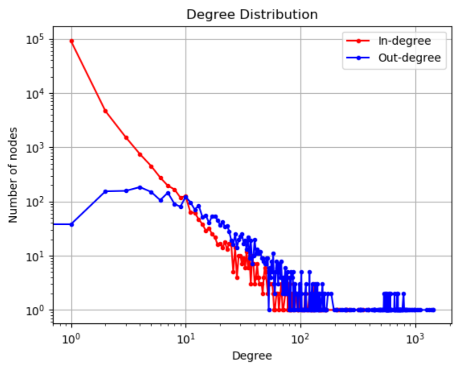
The following bullet points evaluate if the requirements regarding the sampling of the data defined in Section 4.1 are fulfilled:



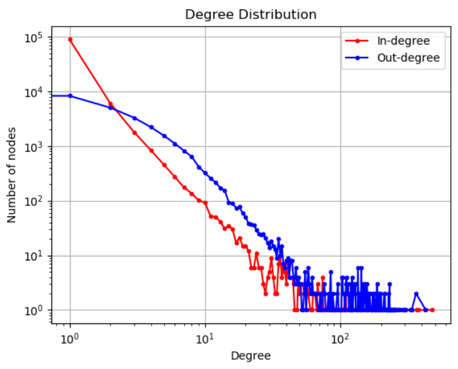
(a) Forest Fire Sampling



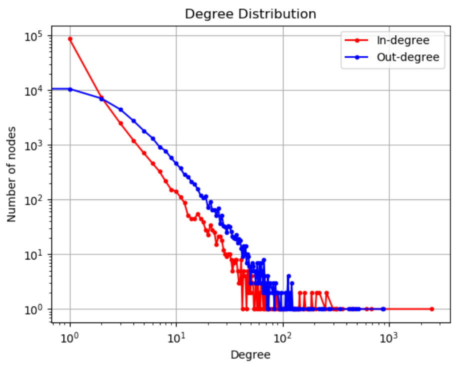
(b) Snowball Sampling ($n = 5$)



(c) Breadth First Sampling



(d) Random First Sampling



(e) Random Walk

Figure 5.6: Degree Distribution of Different Sampling Techniques

- **RS-1**: The framework is able to extract node and edge data from a database provided by the DDIS group. With this data, a sample graph of the Wikidata knowledge graph can be recreated. Requirement RS-1 is therefore fulfilled.
- **RS-2**: The framework makes use of only traversal-based sampling techniques and random walks. This ensures having samples that are connected. RS-2 is therefore fulfilled.
- **RS-3**: In the configuration file *config.ini*, the user can set the number of nodes that the sample has to contain and a seed node from where the sampling algorithm starts. This fulfills requirement RS-3.
- **RS-4** Unfortunately, the framework is not able to find paths between two entered seed nodes and creating an overlapping sample. With the schema of the database, it is difficult to find a (shortest) path between nodes. One way to achieve this requirement would possibly be to keep an eye out for every node sampled if the other node searched for is one of its neighbors and then follow the corresponding edge to it. Nevertheless, such an approach would slow down the computation since every nodes' neighbors have to be checked while sampling. RS-4 is therefore not fulfilled.
- **RS-5**: This requirement can hardly been evaluated since computing graph metrics for the whole Wikidata knowledge graph would consume way too much resources. The comparison between the techniques may give the reader hints about which techniques are rather suitable.
- **RS-6**: The user can set the sampling technique in the configuration file. Among the techniques available are FFS, SBS, BFS, RFS, RW and MHRW. Requirement RS-6 is therefore fulfilled.

5.5 Creating Snapshots

This section explains how the revisions are undone to result in historic versions of what the sample graph looked like back in time. The first subsection shortly discusses the database schema from where the revision data is accessed, while the second subsection deals with the creation of earlier versions of the sample graph.

5.5.1 PostgreSQL Revision Database

Figure 5.7 shows the database schema from the PostgreSQL[10] database provided by DDIS. PostgreSQL is an open-source object-relational database management system. The object in that definition refers to additional inheritance features which MySQL does not provide.

[10] *https://www.postgresql.org/*

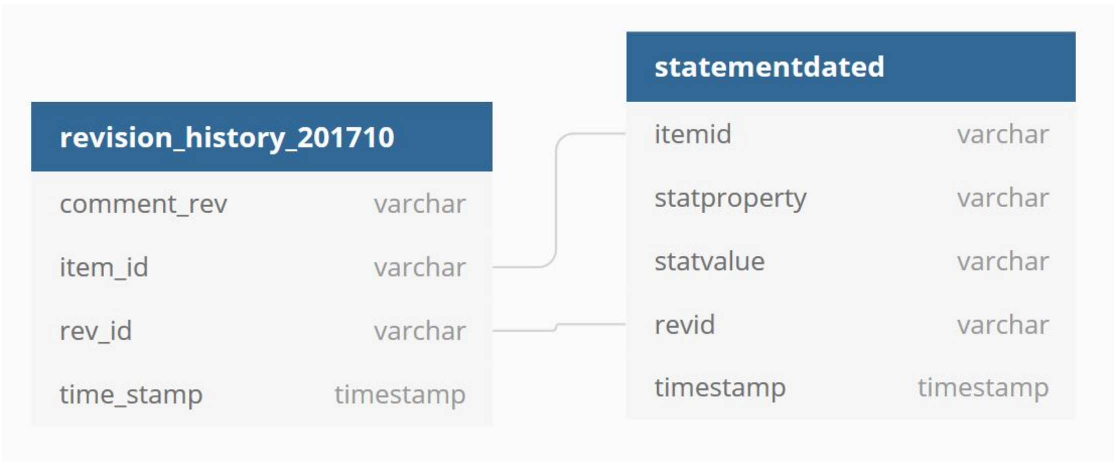


Figure 5.7: PostgreSQL Database Schema for Retrieving the Revision Data

Only the tables and columns necessary for getting the revision data are illustrated in Figure 5.7. The table *statementdated* contains subject-property-object triples expressed as *itemid*, *statproperty* and *statvalue*. Further it contains a revision id called *revid* corresponding to *rev_id* in the other table. Also, a column with the timestamp is available in both tables. The table *revision_history_201710* additionally owns the column *comment_rev*. This column actually states what kind of revision took place. Summarized, the table *statementdated* contains triples that were changed at some point in time, but it is unknown what happened to these triples. To gain knowledge about the actual revision, the framework queries the table *revision_history_201710* with the revision id. The column *comment_rev* then returns the critical information, e.g. if the value of a triple was updated or if a property-value pair was created.

5.5.2 Undoing the Revisions

The undoing of revisions is the actual core part of the framework. During this stage, the historic snapshots of the sampled graph are created by going back in time. As input, the framework needs as information the number of revisions to undo for each snapshot, a timestamp to specify until when to create the snapshots and, obviously, the sampled graph itself. Table 5.6 lists the type of revisions with a short description, an existing example from the column *comment_rev* of the database, as well as a statement of how to undo that type. Most of the revision descriptions from the column *comment_rev* start with the name of the API module that has been requested for this revision, such as wbsetclaim, wbremoveclaims or wbmergeitems. A complete list of these modules is available on the Wikimedia documentation[11].

In a first step, the framework needs to get all the revision ids which may belong to the triples in the sampled graph. For this, the framework executes the following query

[11] *https://www.mediawiki.org/wiki/Wikibase/API/de#API_modules*

API module	**Description**	**Example of *comment_rev***	**Undoing event**
• wbsetclaim-create • wbcreateclaim-create	Creates Wikidata claims	• /* wbsetclaim-create:1||1 */ [[Property:P6]]: [[Q115886]] • /* wbcreateclaim-create:2|*/ [[Property:P281]], 8000	Remove edge
• wbsetclaim-update • wbeditentity-update	Updates a statement/claim	• /* wbsetclaim-update:2||1 */ [[Property:P2046]]: 8,790 hectare • /* wbeditentity-update:0|*/ BOT:Add labels (Upper case)	Query in table *statementdated* for the previous value of the object. Remove edge and create new edge to new node containing the previous value.
wbsetclaimvalue	Sets the value of a Wikidata claim	/* wbsetclaimvalue:1|*/ [[Property:P31]]: [[Q14770218]]	Remove edge
wbremoveclaims-remove	Removes Wikidata claims	/* wbremoveclaims-remove:1|*/ [[Property:P131]]: [[Q39]]	Add claim by adding node and property edge of removed statement
wbmergeitems	Merges multiple items	/* wbmergeitems-from: 0||Q25647097 */	Remove edge of affected triple since it was part of another node.
- (Undo revision)	Undoing a revision and retrieve previous state	/* undo:0||464116012|Chire */	Get the revision that was undone by id and restore it.

Table 5.6: Types of Revisions

for each triple in the sample graph:

```
insert_data = (subject, predicate, timestamp)
sql = "SELECT revid FROM statementdated WHERE itemid = %s " \
      "AND statproperty = %s AND timestamp > %s"
cursor.execute(sql, insert_data)
```

This way, the framework collects not only all the revision ids for the triples in the sample graph, but also the revision ids for subject-property pairs with values not contained in the sample. This is important, for example, in the wbsetclaim-update case. If the revision is an update and the value of the triple therefore reconstructed to its old value, the revision id for that reconstructed triple was collected as well with that first query. In a second query, the framework then retrieves the necessary information from the table *revision_history_2017* by the previously collected revision ids and orders it by time descending:

```
sql_get_history = "SELECT comment_rev, item_id, time_stamp, rev_id " \
                  "FROM revision_history_201710 WHERE rev_id IN " \
                  "%(revision_set)s ORDER BY time_stamp DESC"
cursor.execute(sql_get_history, {'revision_set': tuple(revision_ids), })
```

Hence, a list of revision information ordered from most recently to the user configured timestamp is gained and can then be used further to manipulate the sample graph. For each of these revisions, the framework checks which of the cases listed in Table 5.6 applies and moreover, if the corresponding triple is also present in our sample. If so, the sample graph gets manipulated by undoing that revision. The framework uses the Python package re[12] to read out the necessary parts from the *comment_rev* value of each returned revision information. These parts are of course the actual type of the revision (e.g. wbcreateclaim-create) and in most cases also the property and object value. With the re package, it is possible to extract those values by matching a regular expression.

Having extracted those values, the framework undoes the revision depending on the revision type. In most cases, since we go back in time, the framework has to do the opposite of what the revision states. The simplest types are wbsetclaim-create, wbcreateclaim-create and wbsetclaimvalue. In those cases, the framework must only remove the property edge of the corresponding triple by the NetworkX function *remove_edge(subject, object, key=prop_text)*. By only removing the edge, it is ensured that the object is still part of the sample in case other nodes are connected to it.

When the revision is of type wbremoveclaims-remove, the frameworks adds the object node and the edge to the sample graph. From this follows that the sample graph not only continuously shrinks, but may also gain new triples.

The case of wbsetclaim-update needs more effort. This case informs us that the object of the affected triple was updated (to the new value as mentioned in *comment_rev*), but the previous state of that object is yet unknown. Therefore, another query which returns the previous value is necessary:

[12] *https://docs.python.org/3/library/re.html*

```
data_triple = (subject_text, prop_text, timestamp)
sql = "SELECT statvalue FROM statementdated WHERE itemid = %s AND " \
      "statproperty = %s AND timestamp < %s ORDER BY timestamp DESC" \
      "LIMIT 1"
cursor.execute(sql, data_triple)
```

The old value is looked for in the table *statementdated*. The timestamp in above query refers to the timestamp of the update. So, with setting the timestamp smaller than the timestamp of the update and ordering it by time descending and limiting it to 1, the preceding value is returned. Having retrieved that value, the framework simply removes the edge from the initial subject-object node pair, adds the previous value as node and creates an edge between subject and previous value.

Another revision type is wbmergeitems. This type merges two entity items together. Having a look at the wbmergeitems-example in Table 5.6, the comment on the revision states which item was merged into the affected item in the sample. There exist two types of merges, wbmergeitems-to and wbmergeitems-from. Figure 5.8 illustrates these two cases.

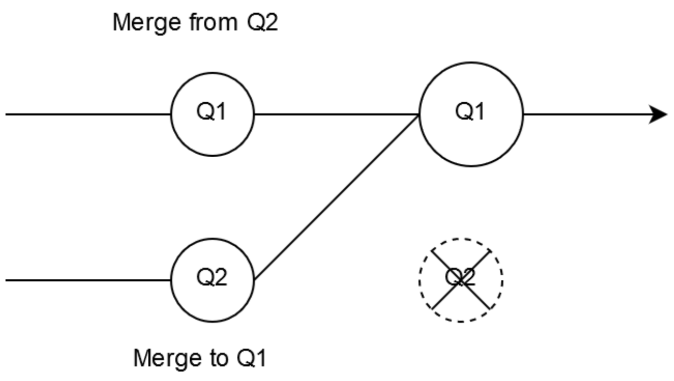


Figure 5.8: Two Cases of Merging

Since we sample our graph from the current state of the Wikidata knowledge graph, we will never retrieve a revision comment with wbmergeitems-to. When node Q2 was merged into node Q1, Q2 can not be part of our sample graph since it does not exist anymore. Therefore, there will not be a revision comment with this type of change and this case can safely be ignored. Nevertheless, the framework needs to handle the case of merging from. When a merging from appears in the list of revisions to undo, this states that the affected triple was part of another node at that point of time. In Figure 5.8, this is illustrated by the former smaller node Q1 which doesn't yet contain the triples of node Q2. Consequently, the framework just has to remove the edge of the affected triple, since it was added from another node which is not (and can not be) part of the sample. This case is actually very similar to the undoing of a simple wbcreateclaim revision, except that another query is needed to get the affected triple in the table *statementdated* by the subject and the revision id.

The last type of revision listed in Table 5.6 is the undoing of a specific revision that happened in the past. Figure 5.9 illustrates this type by a timeline of undoing events. The arrow points in the direction from past to future.

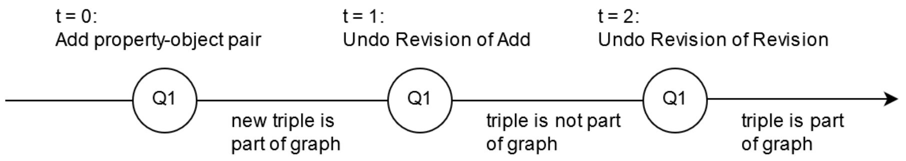


Figure 5.9: Undoing Revision

At time $t = 0$, there is an adding of a property-object pair to the entity node Q1. It can be considered as a wbcreateclaim event. It follows at time $t = 1$ an undo revision of that previous event. The database stores this information by a revision id. An example of such a comment_rev for an undoing event is listed in the revision types Table 5.6. There, the "undo" classifies the type of the revision and the following number 464116012 specifies the revision id of the revision that was undone. When processing the revisions and coming across such a revision type, the framework has to query with that id for the actual revision that took place in an earlier state of the graph. In contrast to all previously described revision types, the framework then has to apply not the opposite, but the exact same revision. This is because we go back in time and the sample graph has to take over the state shortly before $t = 1$. Hence, at time $t = 1$, the framework will add the property-object pair to the knowledge graph. Only later, when reaching time $t = 0$, the edge of that triple will be removed again.

Nevertheless, there also exist cases where the querying for the revision again returns an undo event with another revision id: An undoing of an undoing of a revision, so to speak. This case is illustrated in Figure 5.9 at time $t = 2$. The triple is again part of the graph from that point of time on, since the previous undo revision at time $t = 1$ is invalidated by the newer undo event. Summarized, to get previous snapshots, the framework has to remove the edge of that triple at point $t = 2$, add it at $t = 1$ and remove it again at time $t = 0$.

When going back in time, the framework has to keep track of the number of undoing revisions followed by undoing revisions. If the number of undoing revisions behind each other is odd, like in the simple case of just one undoing event, then the same revision type as at the time of its origin has to be applied. If the number is even, as for the undoing of an undoing, the opposite event of that revision is relevant.

Finally, by applying all the different revision changes described in this section from most recent to going back to a defined timestamp, the framework creates snapshots of earlier versions of the sample graph. The snapshots each contain a specific number of changes. This number can be defined in the configuration file. Therefore, the framework returns snapshots with each the same number of changes, except for the last snapshot which may contain less. Additionally, the snapshot files are named with the timestamp from the oldest change they contain. Hence, the timestamp identifies the date and time of when the snapshot of our sample graph was valid.

The following bullet points evaluate if the requirements regarding the creation of snapshots defined in Section 4.2 are fulfilled:

- **RC-1**: The framework takes into account constraints that the user sets on the graph. In the configuration file, the user can set the number of changes in each snapshot and a timestamp until when the previous versions of the graph should be returned. RC-1 is therefore fulfilled.
- **RC-2**: The output files of the snapshots are named with the date and time indicating the timestamp of when they were valid. RC-2 is therefore fulfilled.
- **RC-3**: The undoing of revisions is applied in the correct order. This is achieved by querying for the revisions directly by their time descending. RC-3 is therefore fulfilled.
- **RC-4**: The framework handles the implementation of the undoing of changes accurately as described in this section.

5.6 Returning the Snapshots

As already mentioned, the framework uses RDFLib, a Python package for working with RDF, to transform the sample graph and its snapshots into an RDF format. Section 5.6.1 explains shortly how the triples from the sampled NetworkX graph are loaded into an RDFLib graph and then serialized into a standard RDF format.

5.6.1 NetworkX to RDFLib

The framework uses the RDFLib package to create an RDFLib graph and serialize it into one of various formats. RDFLib also provides other features such as querying with SPARQL or parsing (reading data from an RDF file into an RDFLib graph). In RDF, nodes are either URI references, blank nodes or literals. To transform the NetworkX graph into an RDFLib graph, the framework has to access each NetworkX triple and load it into the previously initialized RDFLib graph. Of course, namespaces can also be defined with RDFLib. For each triple, the framework defines the subject and the predicate as URI references with the namespace and the actual subject or predicate value. Then, depending on the type of object, the framework adds the triples to the RDFLib graph. For this, the framework gets the datatype (e.g. dateTime, coordinates) and if available the language tag and adds the triples according to them. If the object is again an entity type, then the object is defined as an URI reference. For all other object types, it is represented and added to the graph as a literal. Data types and language tags are passed to this literal as well. This way, the RDFLib graph gets filled with all the triples and their auxiliary information. This is an example of how statements are represented in RDFLib:

```
(rdflib.term.URIRef('http://www.wikidata.org/entity/Q21510865'),
rdflib.term.URIRef('http://www.wikidata.org/prop/direct/P1535'),
rdflib.term.URIRef('http://www.wikidata.org/entity/Q54812269'))
(rdflib.term.URIRef('http://www.wikidata.org/entity/Q52060874'),
rdflib.term.URIRef('http://www.wikidata.org/prop/direct/P1813'),
rdflib.term.Literal('single␣best␣value', lang='en'))
(rdflib.term.URIRef('http://www.wikidata.org/entity/Q20855878'),
rdflib.term.URIRef('http://www.wikidata.org/prop/direct/P571'),
rdflib.term.Literal('+2016-09-24T00:00:00Z',
datatype=rdflib.term.URIRef('http://www.w3.org/2001/XMLSchema#dateTime')))
```

The above example shows three statements: the first with an object of type entity, the second with a literal object having a language tag, and the third with a literal object specifying its type with an additional datatype URI reference. All the subject, predicates and objects are RDFLib types of either URI references or literals. Since the framework accesses only direct statements, it is not concerned with blank nodes.

Having all the NetworkX triples represented in an RDFLib graph, the framework can simply serialize the data into various formats with the RDFLib serializers package. The serialize method is passed to the newly created graph with a format parameter defining the output format. According to the RDFLib documentation[13], N-Triples is currently using the most efficient serialization, followed by RDF/XML. The output format can be specified in the configuration file. The framework currently supports the following formats: Turtle[14], Notation 3[15], N-Triples[16], TriG[17], RDF/XML[18] and pretty-xml (an abbreviated RDF/XML syntax).

Furthermore, one of the requirements for the framework is to return the Wikidata graphs in a format natively supported by NetworkX. Therefore, the framework also offers the possibility to output the data in JSON format. By making use of the NetworkX *node_link_data()* function[19], the framework returns a dictionary with node-link formatted data suitable for JSON serialization. With the built-in json package, the dictionary can then be transformed into a JSON file.

The following two bullet points evaluate if the requirements regarding the returning of the data defined in Section 4.3 are fulfilled:

- **RF-1**: The framework is able to return the data in various RDF formats. These formats include Turtle, Notation 3, N-Triples, TriG, XML and pretty-xml. Requirement RF-1 is therefore fulfilled.

- **RF-2**: The framework offers the possibility to return the results in a format natively supported by NetworkX. The chosen format is JSON, it returns the node-link

[13] *https://rdflib.readthedocs.io/en/stable/faq.html*
[14] *https://www.w3.org/TR/turtle/*
[15] *https://www.w3.org/TeamSubmission/n3/*
[16] *https://www.w3.org/2001/sw/RDFCore/ntriples/*
[17] *https://www.w3.org/TR/trig/*
[18] *https://www.w3.org/TR/rdf-syntax-grammar/*
[19] *https://networkx.github.io/documentation/stable/reference/readwrite/generated/networkx.readwrite.json_graph.node_link_data.html*

data such that no information is lost by storing the datatype and language tags as well. Requirement RF-2 is therefore fulfilled as well.

6 Limitations

This chapter discusses some points that may have an impact on the functionality of the framework. The framework currently samples only direct statements. Therefore, a lot of Wikidata information like qualifier statements or references is missing. Furthermore, only properties from the Wikidata namespace are available in the database used for extracting the sample graph. This means that properties referring to other namespaces like label and description will not be part of the sampled graph. This results in sampling from a rather reduced Wikidata graph which is not necessarily a problem but should be considered when working with the framework. Item nodes for example contain lots of labels and descriptions, each for many different languages. This could result in a bias towards such label and description nodes during the sampling because they make up a large part of the neighboring nodes. Even worse, the sampling methods may also be more likely to backtrack more often when biased towards such sink nodes since they do not contain any outgoing edges. This could slow down the sampling process. For this reason, the structure of the database currently used for extracting the sample may even be the wiser choice. Though, users of the framework should be aware of the missing triples.

Another limitation comes from the database used for getting the revision data. At the moment, it stores revisions only until October 2017 and should therefore be updated. Furthermore, it is unknown what some of the earlier revision entries stored in that database have done to the knowledge graph. The column *comment_rev* contains in these cases the string *"no comment"*. Therefore, such unidentified revisions can not be undone in the sampled graph and are currently ignored by the framework.

Regarding the sampling techniques, the MHRW algorithm performs very poorly and should not be used when working with the framework. This algorithm oftentimes rejects nodes to be in the sample and therefore scores badly in terms of execution time. Even worse, when a node has only entity neighbors with a much higher degree than the current node, the MHRW algorithm eventually gets stuck in exploring further nodes since it rejects sampling them. This undesirable behavior of the MHRW is described in related work as well [Lee et al., 2012]. Lee et al. propose the MHDA algorithm to circumvent this problem, mainly by remembering already visited nodes and increasing the probability of moving to a neighbor.

Finally, one of the requirements is not fulfilled by the current version of the framework. This requirement addresses the creation of overlapping samples. Starting from two seed nodes sampling their neighborhood, the framework is recommended to create a sample which has nodes in common for both neighborhoods. To fulfill this requirement, the framework would have to calculate paths between the two seed nodes. Nevertheless, there exists related work regarding this issue. Several papers propose solutions to the problem of finding shortest paths using relational databases [Gao et al., 2011] [Jindal et al., 2015].

7 Future Work

The framework implemented in this work builds a foundation for analyzing the evolution of knowledge graphs as indicated in Section 1.1.

Regarding the sampling process of the framework, it would be interesting to evaluate how the structure of the Wikidata graph affects the sampling. Sampling algorithms that have been proven to result in good samples for e.g. social networks must not necessarily be suited for the knowledge graph sampled in this work. Wikidata consists of many low-degree sink nodes and comparatively little high-degree entity nodes. Hence, studying the impact of this structure on the sampling techniques in more detail could be of further interest. In addition, a sampling technique especially suited for that structure could be evaluated.

If the framework is used for graph evolution studies, then an investigation on the bias introduced from sampling on the snapshots may be of future interest. The question to ask here is if the snapshots are affected by sampling methods that are biased. A sampling technique biased towards high-degree nodes generates samples with an above average amount of such nodes. High-degree nodes may have undergone more changes than low-degree nodes and the series of snapshots would therefore contain too many revisions in contrast to an unbiased sample graph series of snapshots. When analyzing historic versions of a sampled graph this possible behavior should be considered as well.

Apart from that, the framework could be improved by implementing additional functionalities. A possible idea for an auxiliary feature is to extend the sampling with a set of excluding or including conditions, e.g. exclude certain property types from sampling.

Finally, it may be practical for users to instantly see which of the triples have changed in the generated snapshots. For this purpose, the framework could highlight the triples that have been revised in each of the snapshots, e.g. by coloring the affected triples in the output files.

8 Conclusions

This thesis builds the foundation for analyzing the evolution of the Wikidata knowledge graph. Therefore, I implemented a framework that creates a sequence of versioned Wikidata graphs by going back in time and undoing the revisions. This yields in several output files containing the RDF data of the graph in earlier stages of its history. Generating snapshots of the whole Wikidata knowledge graph is impractical due to its sheer size. Hence, the framework first extracts a sample out of the graph to save processing, memory and time resources. The implementation of the framework focuses on three main tasks: the extraction of a sample out of the Wikidata graph, the undoing of its edit history and the returning of snapshots in a standard RDF format.

Sampled graphs are supposed to have similar properties as the original network. Only then, the findings and observations evaluated on the sample can be believed to hold also for the original graph. After comparing and discussing related work on sampling of large graphs, I implemented different sampling techniques. Since the sampled graph should be connected, the framework works with traversal-based sampling algorithms. I found that the biases and behaviors of the implemented techniques correspond to the findings evaluated by related work about sampling (as discussed in Section 3.2). I recommend not to use MHRW for sampling since it performs very badly in terms of execution time. This behavior can be explained because the algorithm rejects many nodes and even may get stuck in cases when the degrees of the neighboring entity nodes are much higher than the degree of the current node.

Regarding the other implemented techniques, most of them fail in preserving one or more graph metrics by being biased. Although BFS extracts a sample graph the fastest, it is evaluated to be too disassortative, having a too small average degree (because of the boundary bias) and to underestimate low-degree nodes. Furthermore, by analyzing the plots in Figure 5.6, I observed that BFS, RFS and RW are biased towards high-degree nodes, whereas RFS additionally results in a too small average degree. Users of the framework should be aware of these limitations.

After having extracted a sample, the framework undoes the different revision types. These types include the creation of a statement, the removal of a statement, the update of a value, the merging of two entity nodes and the undoing of a preceding revision.

Finally, the framework generates snapshot files in a standard RDF format. Currently, it supports Turtle, Notation 3, N-Triples, TriG, RDF/XML and pretty-xml. The snapshots each contain the sampled graph with a specified number of undone revisions and they are generated until reaching a timestamp as declared in the configuration file.

Currently, the framework only samples direct statements since the database used for sampling does not contain information about qualifier statements or references supporting a claim. In future work, such additional information could be included in the frameworks sampling process as well. Also, it may be of interest to implement further features such as highlighting the triples that have been changed in the output files or extending the sampling with a set of excluding or including conditions (e.g. exclude certain property types from sampling).

References

[Ahmed et al., 2011] Ahmed, N., Neville, J., and Kompella, R. (2011). Network sampling via edge-based node selection with graph induction.

[Ahmed et al., 2013] Ahmed, N. K., Neville, J., and Kompella, R. (2013). Network sampling: From static to streaming graphs. *ACM Trans. Knowl. Discov. Data*, 8(2):7:1–7:56.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):28–37.

[Bloznelis, 2013] Bloznelis, M. (2013). Degree and clustering coefficient in sparse random intersection graphs. *The Annals of Applied Probability*, 23(3):1254–1289.

[Bradner, 1997] Bradner, S. O. (1997). Key words for use in RFCs to Indicate Requirement Levels. RFC 2119.

[Doerr and Blenn, 2013] Doerr, C. and Blenn, N. (2013). Metric convergence in social network sampling. In *Proceedings of the 5th ACM Workshop on HotPlanet*, HotPlanet '13, pages 45–50, New York, NY, USA. ACM.

[Erxleben et al., 2014] Erxleben, F., Günther, M., Krötzsch, M., Mendez, J., and Vrandečić, D. (2014). Introducing wikidata to the linked data web. In *Proceedings of the 13th International Semantic Web Conference - Part I*, ISWC '14, pages 50–65, New York, NY, USA. Springer-Verlag New York, Inc.

[Färber et al., 2017] Färber, M., Bartscherer, F., Menne, C., and Rettinger, A. (2017). Linked data quality of dbpedia, freebase, opencyc, wikidata, and yago. *Semantic Web*, 9:1–53.

[Gao et al., 2011] Gao, J., Jin, R., Zhou, J., Yu, J. X., Jiang, X., and Wang, T. (2011). Relational approach for shortest path discovery over large graphs. *Proc. VLDB Endow.*, 5(4):358–369.

[Gjoka et al., 2010] Gjoka, M., Kurant, M., Butts, C. T., and Markopoulou, A. (2010). Walking in facebook: A case study of unbiased sampling of osns. In *2010 Proceedings IEEE INFOCOM*, pages 1–9.

[Goodman, 1961] Goodman, L. A. (1961). Snowball sampling. *Ann. Math. Statist.*, 32(1):148–170.

[Heckathorn, 1997] Heckathorn, D. D. (1997). Respondent-Driven Sampling: A New Approach to the Study of Hidden Populations*. *Social Problems*, 44(2):174–199.

[Hu and Lau, 2013] Hu, P. and Lau, W. (2013). A survey and taxonomy of graph sampling.

[Jindal et al., 2015] Jindal, A., Madden, S., Castellanos, M., and Hsu, M. (2015). Graph analytics using vertica relational database. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1191–1200.

[Krishnamurthy et al., 2005] Krishnamurthy, V., Faloutsos, M., Chrobak, M., Lao, L., Cui, J. H., and Percus, A. G. (2005). Reducing large internet topologies for faster simulations. In *NETWORKING 2005. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications Systems*, pages 328–341, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Kurant et al., 2011] Kurant, M., Markopoulou, A., and Thiran, P. (2011). Towards unbiased bfs sampling. *IEEE Journal on Selected Areas in Communications*, 29:1799–1809.

[Lee et al., 2012] Lee, C.-H., Xu, X., and Young Eun, D. (2012). Beyond random walk and metropolis-hastings samplers: Why you should not backtrack for unbiased graph sampling. *Sigmetrics Performance Evaluation Review - SIGMETRICS*.

[Lee et al., 2006] Lee, S. H., Kim, P.-J., and Jeong, H. (2006). Statistical properties of sampled networks. *Phys. Rev. E*, 73:016102.

[Leskovec and Faloutsos, 2006] Leskovec, J. and Faloutsos, C. (2006). Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 631–636, New York, NY, USA. ACM.

[Malyshev et al., 2018] Malyshev, S., Krötzsch, M., González, L., Gonsior, J., and Bielefeldt, A. (2018). Getting the most out of wikidata: Semantic technology usage in wikipedia's knowledge graph. In *International Semantic Web Conference*.

[Miller and Manola, 2004] Miller, E. and Manola, F. (2004). RDF primer. W3C recommendation, W3C. http://www.w3.org/TR/2004/REC-rdf-primer/.

[Newman, 2002] Newman, M. E. J. (2002). Assortative mixing in networks. *Physical Review Letters*, 89(20).

[Pernischová, 2019] Pernischová, R. (2019). The butterfly effect in knowledge graphs: Predicting the impact of changes in the evolving web of data. In *Doctoral Consortium at ISWC 2019*.

[Ribeiro and Towsley, 2010] Ribeiro, B. F. and Towsley, D. F. (2010). Estimating and sampling graphs with multidimensional random walks. *CoRR*, abs/1002.1751.

[Seaborne and Prud'hommeaux, 2008] Seaborne, A. and Prud'hommeaux, E. (2008). SPARQL query language for RDF. W3C recommendation, W3C. http://www.w3.org/TR/2008/REC-rdf-sparql-query/.

[Smith, 2007] Smith, R. (2007). Average path length in complex networks: Patterns and predictions.

[Stumpf et al., 2005] Stumpf, M., Wiuf, C., and May, R. (2005). Subnets of scale-free networks are not scale-free: Sampling properties of networks. *Proceedings of the National Academy of Sciences of the United States of America*, 102:4221–4.

[Trivedi et al., 2017] Trivedi, R., Dai, H., Wang, Y., and Song, L. (2017). Know-evolve: Deep temporal reasoning for dynamic knowledge graphs. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 3462–3471. JMLR.org.

[Vrandečić and Krötzsch, 2014] Vrandečić, D. and Krötzsch, M. (2014). Wikidata: A free collaborative knowledge base. *Communications of the ACM*, 57:78–85.

[Wang et al., 2011] Wang, T., Chen, Y., Zhang, Z., Xu, T., Jin, L., Hui, P., Deng, B., and Li, X. (2011). Understanding graph sampling algorithms for social network analysis. In *2011 31st International Conference on Distributed Computing Systems Workshops*, pages 123–128.

[Watts and Strogatz, 2011] Watts, D. and Strogatz, S. (2011). *Collective dynamics of 'small-world' networks.*

A

Appendix

A.1 Contents of the CD

The CD has the following content:

- Zusfsg.txt: This file contains a summary of the bachelor thesis in German.
- Abstract.txt: This file contains a summary of the bachelor thesis in English.
- Bachelorarbeit.pdf: This file contains the complete thesis as a PDF.
- code/: This directory contains the source code of the framework.

List of Figures

List of Tables