Fachhochschule
Münster University of
Applied Sciences

Fachbereich Elektrotechnik und Informatik

Bachelorarbeit

Konzeption und Implementierung eines Frameworks zur Verarbeitung großer semantischer Netze

Carsten Hibbeler

Betreuer / Prüfer
Prof. Dr. rer. nat. Nikolaus Wulff

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit sowie die zugehörige Implementierung selbstständig verfasst und unter ausschließlicher Verwendung der angegebenen Quellen und Hilfsmittel erstellt zu haben.

Münster, den 6. August 2010

Carsten Hibbeler

Danksagung

Mein besonderer Dank gilt meinen Eltern und meiner Schwester, auf die ich mich zu jeder Zeit verlassen kann und die hierdurch wesentlich zum Erfolg dieser Arbeit beigetragen haben.

Desweiteren bedanke ich mich bei Herrn Professor Dr. Nikolaus Wulff für seine Unterstützung und die Betreuung meiner Arbeit.

Nicht zuletzt geht mein Dank an meine Kollegen der Xdot GmbH für das kreativitätsfördernde und sehr positive Arbeitsklima.

Darüber hinaus bedanke ich mich bei allen anderen, hier nicht im Einzelnen genannten Personen, die mich bei der Anfertigung dieser Arbeit in unterschiedlicher Weise unterstützt haben.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Ziel

Zur Bearbeitung von textuellen Inhalten und der Strukturierung von Informationen kommen vielfach Ontologien zum Einsatz. Sie werden verwendet, um Wissen in maschinenlesbarer Form zu repräsentieren. Dabei wird der Begriff Ontologie für verschiedene Formen der Wissensrepräsentation verwendet, die sich in ihrem Aufbau der Komplexität und der Art des enthaltenen Wissens unterscheiden. Eine Form von Ontologien sind semantische Netze.

Die grundlegende Motivation für diese Bachelorarbeit stammt aus dem Einsatz semantischer Netze bei der xdot GmbH. Für verschiedene Bereiche des Information Retrival werden hier große semantische Netze mit mehreren hunderttausend bis zu einigen Millionen Begriffen und Relationen eingesetzt. Zur Verarbeitung solcher Netze kommen häufig graphentheoretische Algorithmen zum Einsatz. Die praktische Umsetzung solcher Algorithmen hat sich jedoch mit den zurzeit bei der xdot GmbH eingesetzten Technologien als problematisch erwiesen. Hierfür sind zwei Faktoren verantwortlich. Zum einen bieten diese Technologien wenig Unterstützung bei der Umsetzung von Algorithmen der Graphentheorie. Zum anderen führen die bisherigen Umsetzungen durch die Größe der Netze zu sehr hohem Speicherverbauch und langen Ladezeiten.

Ziel dieser Arbeit ist es, über die Konzeption und Implementierung eines Frameworks eine Lösung für diese Problematiken bereitzustellen. Das Framework soll eine API bieten, die sich an den Konzepten der Graphentheorie orientiert und in der Lage ist, semantischen Netzen mit mehreren Millionen Begriffen und Relationen performant zu verarbeiten. Der Speicherverbrauch, der zur Verarbeitung benötigt wird, muss dabei möglichst gering gehalten werden.

1.2 Struktur der Arbeit

Kapitel 2 gibt zunächst einen Überblick über Modelle zur Repräsentation von Semantik und vermittelt die grundlegenden Technologien und Standards, die für diese Arbeit relevant sind. Das Kapitel 3 beschäftigt sich mit der Verarbeitung semantischer Netze durch Graph-Algorithmen, wie sie bei der xdot GmbH stattfindet. Wo sich hierbei die Problematik ergibt, wird im Kapitel 4 erläutert. Die daraus folgenden Anforderungen an das zu entwickelnde Framework werden im Kapitel 5 dargestellt. Das Kapitel 6 beschäftigt sich mit möglichen Ansätzen auf Basis existierender Technologien. Die hierauf folgenden Kapitel beschreiben das realisierte Framework. Auf die Hauptkomponenten und grundlegenden Konzepte des Frameworks wird im Kapitel 7 und auf die Details der Implementierung im Kapitel 8 eingegangen. Um darzustellen, inwieweit das implementierte Framework die gestellten Anforderungen erfüllt, werden im Kapitel 9 Ergebnisse von Speicherverbrauchs- und Performancemessungen präsentiert. Die durch das Framework erzielten Ergebnisse werden im Kapitel 10 zusammengefasst. Abschließend wird im Kapitel 11 ein Fazit gezogen und Ausblick auf mögliche Erweiterungen gegeben.

2 Grundlagen

2.1 Repräsentation von Semantik

In diesem Kapitel werden verschiedene Modelle zur formalen Repräsentation von semantischem Wissen vorgestellt. Ziel dabei ist es, ein grundlegendes Verständnis der verschiedenen Modelle zu vermitteln.

2.1.1 Kontrollierte Vokabularien

Ein kontrolliertes Vokabular ist eine sehr einfache Art der Wissensrepräsentation. Es besteht aus einem Wortschatz, in dem jede Bezeichnung innerhalb des Wortschatzes eindeutig einem Begriff zugeordnet ist. Hierdurch werden Homonyme, also Bezeichnungen, die mehreren Begriffen zugeordnet sind, vermieden. Somit muss in einem kontrollierten Vokabular klar sein, ob der Term „Bank“ eine Bezeichnung für ein Geldinstitut oder eine Sitzbank referenziert.

In vielen Fällen soll zusätzlich das Auftreten von Synonymen unterbunden werden. Hierzu wird zu jedem Begriff eine eindeutige oder zumindest eine präferierte Bezeichnung festgelegt. Eine solche Bezeichnung nennt man auch Deskriptor. Durch ein kontrolliertes Vokabular wird eine Übereinkunft über die Semantik von Deskriptoren getroffen. Viele der im Folgenden beschriebenen Repräsentationsformen basieren auf einem kontrollierten Vokabular, um Homonym- und Synonymfreiheit zu gewährleisten.

2.1.2 Taxonomien

Im informationstechnologischen Umfeld bezeichnet eine Taxonomie eine Klassifikation von systematisch geordneten Schlagwörtern, die Typen oder Klassen bezeichnen. Da bei den verwendeten Schlagwörtern vor allem Homonyme unerwünscht sind, werden als Schlagwörter überwiegend Deskriptoren aus einem kontrollierten Vokabular verwendet. Um eine Ordnungsstruktur zu beschreiben, werden die verwendeten Schlagwörter in der Regel, wie in der Abbildung 2.1 dargestellt, in einer hierarchischen Struktur geordnet.

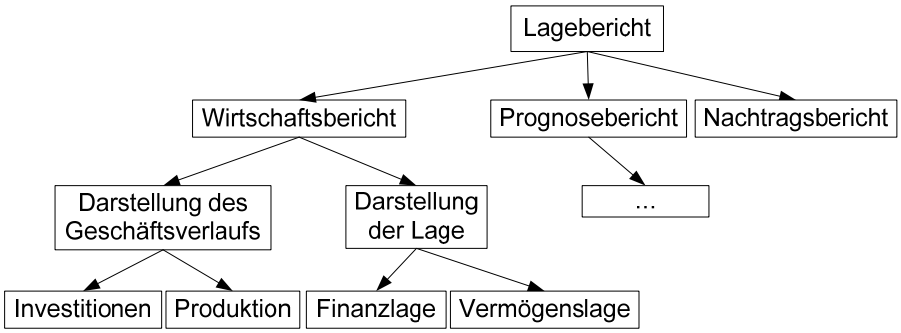

Abbildung 2.1: Beispiel einer Taxonomie

2.1.3 Thesauri

Der Thesaurus ist ein kontrolliertes Vokabular, in dem die enthaltenen Deskriptoren durch semantische Relationen verbunden sind. Hauptbestandteil vieler Thesauri sind Relationen, die Hyponymie (Ober- und Unterbegriffe) und Synonymie beschreiben. Verwendung finden Thesauri oft bei der manuellen oder automatischen Indexierung, wo sie als kontrolliertes Vokabular von Deskriptoren für eine differenzierte Begriffszuordnung sorgen. Auch in vielen aktuellen Textverarbeitungssystemen finden sich Thesauri, die hier dem Nachschlagen von synonymen Bezeichnungen dienen.

2.1.4 Semantische Netze

Ein semantisches Netz bezeichnet allgemein ein formales Modell, in dem Begriffe und deren Bezeichnungen durch Relationen in Beziehung zueinander gesetzt werden. Da sich Modelle wie Thesauri und Taxonomien als semantisches Netz mit einer eingeschränkten Menge an Relationen darstellen lassen, werden semantische Netze in dieser Arbeit als Verallgemeinerung von Modellen wie Thesauri und Taxonomien verstanden. Für die Repräsentation von semantischen Netzen wird meist ein Graph verwendet, in dem die Knoten die Begriffe oder Bezeichnungen und die Kanten die semantischen Relationen darstellen. Ein bekanntes und in der Forschung häufig verwendetes semantisches Netz ist WordNet (siehe [1]). In WordNet werden Begriffe der englischen Sprache durch unterschiedliche semantische Relationen wie Hyponymie, Meronymie (Teil-von-Beziehung) oder auch Antonymie (Gegensatz-von-Beziehung) in Beziehung gesetzt. Diese Arbeit bezieht sich auf semantische Netze wie WordNet, in denen lexikalisch semantische Relationen enthalten sind.

2.1.5 Ontologien

Der Terminus Ontologie wird in verschiedenen Bereichen der Informatik für eine explizite formale Spezifikation einer Konzeptualisierung verwendet. Konzeptualisierung meint dabei ein Modell, welches für eine Domäne relevante Begriffe und deren Beziehungen untereinander beschreibt. Anstatt von Begriffen wird in diesem Zusammenhang oft auch von Konzepten gesprochen. Die gebräuchlichste Definition von Ontologien in der Informatik stammt von T. Gruber. Er definiert eine Ontologie als „An ontology is an explicit specification of a conceptualization“ [2]. Diese Definition ist relativ allgemein gehalten. Somit existiert auch keine einheitliche Meinung darüber, inwieweit die im Vorfeld aufgeführten Modelle zur Wissensrepräsentation hinsichtlich einer Ontologie einzuordnen sind. So werden Thesauri beziehungsweise semantische Netze teilweise als eigenständige Konzepte, Vorläufer oder auch Spezialfälle von Ontologien angesehen. In dieser Arbeit werden unter dem Begriff Ontologie sowohl semantische Netze und Thesauri, als auch strukturell komplexere Konzeptualisierungen verstanden.

Grundlage einer Ontologie bilden Klassen, die in den meisten Fällen hierarchisch geordnet sind. Sie repräsentieren Konzepte, die in ihren Eigenschaften durch Attribute beschrieben werden. Beziehungen zwischen Klassen sind eine spezielle Form von Attributen, deren Wertebereich eine Klasse der Ontologie darstellt.

2.2 Semantic Web Standards

In der Version des Semantic Web geht es darum, den Nutzen und die Userbility des Internets zu verbessern. Dabei ist die zugrundeliegende Idee, Technologien in die Lage zu versetzen, Daten anhand ihrer Bedeutung beziehungsweise die Bedeutung selbst technisch zu verarbeiten. Hierfür sollen die existierenden Inhalte des Internets mit maschinenlesbarer Semantik annotiert werden[3].

Um die Vision des Semantic Web Wirklichkeit werden zu lassen, müssen die verschiedenen Systeme sich über die formale Repräsentation der Semantik einig sein. Dies wiederum macht die Verwendung einheitlicher Standards unerlässlich. Aus diesem Grund werden seit einigen Jahren vom World Wide Web Consortium (W3C) Technologien entwickelt, die Repräsentation und Verarbeitung von Semantik

ermöglichen sollen. Die Abbildung 2.2 illustriert den schichtweisen Aufbau der für das Semantic Web eingesetzten Standards und Technologien.

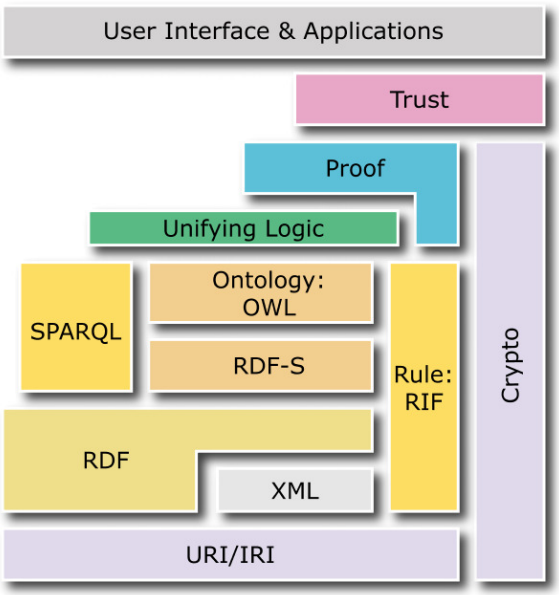

Abbildung 2.2: Semantic Web Stack
Quelle: http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#(24)
(Stand: 27. Juli 2010)

Einige dieser Standards etablieren sich mittlerweile auch in anderen Bereichen, wie dem Informationsmanagement. Hier sorgen sie für ein hohes Maß an Interoperabilität. Zusätzlich entstehen auf Basis dieser Standards eine Vielzahl an Produkten, Tools und Technologien wie zum Beispiel Ontologie-Editoren.

2.3 RDF

Das Resource Description Framework (RDF) bildet einen der Grundbausteine des Semantic Web Stacks. RDF ist ein Datenmodell, das die formelle Beschreibung von Ressourcen ermöglicht. Dabei können Ressourcen alles sein über das man eine Aussage machen kann. Sie können sowohl konkrete Objekte wie Bilder und Webseiten, sowie Städte und Personen der realen Welt als auch abstrakte Begriffe wie Prozesse oder Zustände referenzieren. Eindeutig identifiziert werden Ressourcen in RDF über URIs, wodurch erreicht werden soll, dass jede Ressource weltweit eindeutig identifiziert werden kann. Zusätzlich zu den eindeutigen Ressourcen sieht das Konzept von RDF anonyme Ressourcen vor. Sie werden als blank nodes bezeichnet. Eingesetzt werden blank nodes meist, um andere Ressourcen in nicht näher spezifizierter Weise zu

gruppieren. Um auf eine blank node zu referenzieren, wird ein lokal eindeutiger Identifikator verwendet. Ein weiterer Teil von RDF sind RDF-Literale. Hierbei handelt es sich um Strings, die optional als Wert eines einfachen Datentyps, wie einer Zahl oder eines Datums, deklariert werden können. RDF-Literale, URIs und bank nodes werden in dieser Arbeit auch als RDF-Objekte bezeichnet.

In RDF können "wahre" Aussagen über Ressourcen in Form von sogenannten Statements, auch als RDF-Triple bezeichnet, formuliert werden. Eine Aussage in Form eines Statements besteht immer aus drei Teilen, dem Subjekt, dem Prädikat und dem Objekt. Das Subjekt ist dabei die Ressource, zu der über das Prädikat ein Wert, das Objekt, zugewiesen wird. Das Prädikat eines Statements wird durch eine URI identifiziert, wodurch beliebig viele unterschiedliche Prädikate definiert werden können, ohne dass Mehrdeutigkeiten auftreten. Das Objekt als Wert des Prädikats kann eine Ressource, aber auch ein RDF-Literal sein. Die Tabelle 2.1 zeigt drei Statements aus Subjekt, Prädikat und Objekt. In dieser Tabelle wird für das Prädikat eine Kurzschreibweise verwendet die später in diesem Kapitel noch erläutert wird.

Subjekt	Prädikat	Objekt
http://example.org/doc/doc4711	dc:creator	http://example.com/persons/Muster
http://example.org/doc/doc4711	dc:title	„RDF Example“
http://example.org/persons/Muster	foaf:name	„Max Muster“

Tabelle 2.1: Drei Statements aus Subjekt, Prädikat und Objekt.

Eine Menge von Statements werden als RDF-Graph bezeichnet (siehe [4]). Die Bezeichnung RDF-Graph bezieht sich dabei auf den Graph, der entsteht, wenn Statements als gerichtete Kante zwischen Subjekten und Objekten verstanden werden. Diese intuitive Sichtweise wird häufig auch zur grafischen Darstellung von Statements verwendet. Ein Beispiel hierfür zeigt Abbildung 2.3.

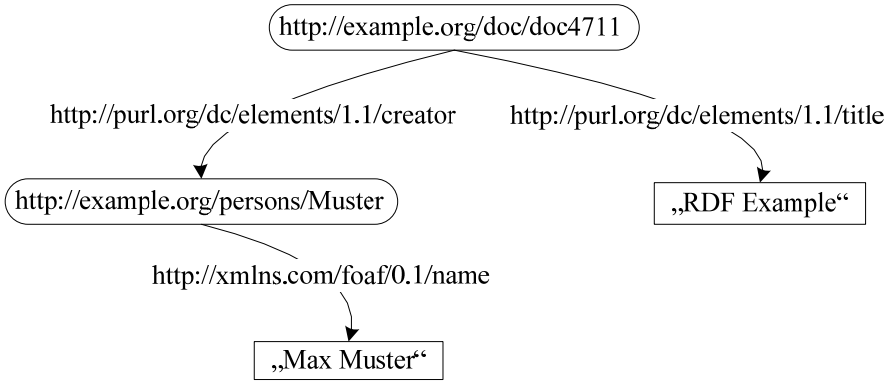

Abbildung 2.3: Ein RDF-Graph dargestellt als gerichteter Graph

RDF ist ein reines Datenmodell und nicht an eine spezielle Darstellungsform gebunden. Zur Darstellung eines RDF-Graph stehen verschiedene Arten der Serialisierung zur Verfügung. Eine bekannte Form RDF zu serialisieren ist RDF/XML. Bei dieser Form der Serialisierung kann der XML Namensraum Mechanismus dazu genutzt werden, eine verkürzte Schreibweise zu erreichen.

```
<?xml version="1.0" enconding="UTF-16"?>
<rdf:RDF
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
       xmlns:foaf="http://xmlns.com/foaf/0.1/"
       xmlns:dc="http://purl.org/dc/elements/1.1/">

       <rdf:Description rdf:about="http://example.org/doc/doc4711">
              <dc:title>RDF Example</dc:title>
              <dc:creator rdf:resource="http://example.org/persons/Muster"/>
       </rdf:Description>

       <rdf:Description rdf:about="http://example.org/persons/Muster">
            <foaf:name>Max Muster</foaf:name>
        </rdf:Description>

</rdf:RDF>
```

Listing 2.1: RDF-Graph im RDF/XML Format dargestellt.

In dieser Arbeit werden verschiedene Vokabulare verwendet. Aus Platzgründen wird zur Darstellung hierbei wird die Kurzschreibweise mit dem Präfix des Namensraumes anstelle der vollständigen URI verwendet. Tabelle 2.2 zeigt die verwendeten Namensräume und ihren Präfix.

Vokabular	Präfix	Namensraum
RDF	rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
RDF Schema	rdfs	http://www.w3.org/2000/01/rdf-schema#
OWL	owl	http://www.w3.org/2002/07/owl#
SKOS	skos	http://www.w3.org/2004/02/skos/core#
Dublin Core	dc	http://purl.org/dc/elements/1.1/
foaf project	foaf	http://xmlns.com/foaf/0.1/"

Tabelle 2.2: Präfix und Namensraum verschiedener Vokabulare

2.4 RDF-Schema

Eine in RDF beschriebene Menge an Statements ist alleine nicht viel mehr als eine Liste von Objekt-Attribut-Wert Trippeln. Aus Verwendung dieser ergeben sich keine logischen Konsequenzen, da ihre Semantik nicht formal definiert ist. Sie sind eine sehr schwache Wissensrepräsentation. An diesem Punkt setzt RDF-Schema kurz RDFS an. Die Idee hinter RDFS ist, eine Möglichkeit zu schaffen, einfache Ontologien formalisieren zu können. Um dies zu ermöglichen, definiert RDFS ein Vokabular mit festgelegter Semantik zur Beschreibung von Klassen und Eigenschaften.

Als formale Sprache zur Definition des RDFS Vokabulars wird RDF benutzt, folglich sind die zur Modellierung durch RDFS genutzten Entitäten RDF Ressourcen. Dabei wird RDFS rekursiv definiert. Das bedeutet, dass RDFS selbst durch RDFS beschrieben wird. Abbildung 2.4 zeigt einen Teil des RDFS Schema als gerichteten Graph.

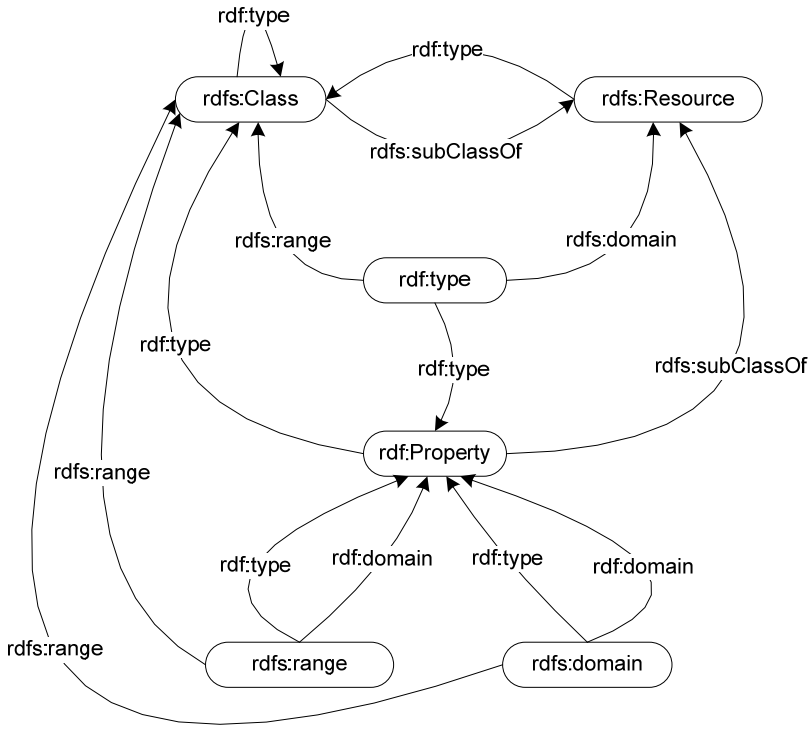

Abbildung 2.4: Teil des RDFS-Schema als gerichteter Graph

Als Klasse aller RDF-Ressourcen wird in RDFS die Klasse *rdfs:Resource* definiert. Alle RDF-Ressourcen sind Instanzen von *rdfs:Resource*.

Zur Umsetzung des Klassenkonzepts existiert die Ressource *rdfs:Class*. Um eine Ressource als Klasse auszuweisen, wird sie als Instanz von *rdfs:Class* definiert. Dies geschieht über die RDF-Eigenschaft *rdf:type*. Listing 2.2 zeigt die Definition einer RDFS Klasse mit der URI *http://example.org/Document.*

```
<rdf:Description rdf:about="http://example.org/schema/Document">
      <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdf:Description>
```

Listing 2.2: Definition einer RDFS-Klasse.

Um eine Vererbungshierarchie zu modellieren, steht die Eigenschaft *rdfs:subClassOf* zur Verfügung. Über diese transitive Eigenschaft kann festgelegt werden, dass eine Klasse Unterklasse einer Klasse ist. Ein Beispiel hierfür zeigt Listing 2.3.

```
<rdf:Description rdf:about="http://example.org/schema/WordDocument">
       <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
       <rdfs:subClassOf rdf:resource="http://example.org/schema/Document"/>
</rdf:Description>
```

Listing 2.3: Definition einer Klasse als Unterklasse.

Klassen und Eigenschaften werden durch RDFS getrennt voneinander modelliert, das heißt, dass Eigenschaften nicht wie zum Beispiel bei der Objekt orientierten Programmierung zur Klasse gehören, sondern zunächst global definiert werden.

Eigenschaften werden ähnlich wie Klassen definiert. Hierzu steht die spezielle Klasse *rdf:Property* zur Verfügung. Sie ist Basisklasse aller Eigenschaften. Ressourcen vom Typ *rdf:Property* werden im Folgendem als Property bezeichnet. Um für eine Property einen Definitionsbereich festzulegen wird *rdfs:domain* verwendet. Der Wertebereich wird über *rdfs:range* eingeschränkt.

Auch bei Eigenschaften existiert die Möglichkeit, eine transitive Vererbungshierarchie zu definieren. Dies geschieht über die Property *rdfs:subPropertyOf,* die eine Spezialisierung einer Property definiert.

Mit RDF-Schema ist man in der Lage, Klassen und ihre Properties sowie Vererbungshierarchien dieser zu modellieren und somit einfache Ontologien formal zu beschreiben. Da RDFS selber über RDF formalisiert wird, ist die Trennung von Schema und den Instanzen der im Schema definierten Klassen zwar aus konzeptioneller Sicht gegeben, aus technischer Sicht werden beide Teile jedoch weitgehend identisch behandelt. Abbildung 2.5 zeigt die konzeptionelle Trennung zwischen Schema und Instanzen, der gestrichelte Pfeil deutet die Verbindung zwischen dem Prädikat „related“ und der beschreibenden Property Ressource an.

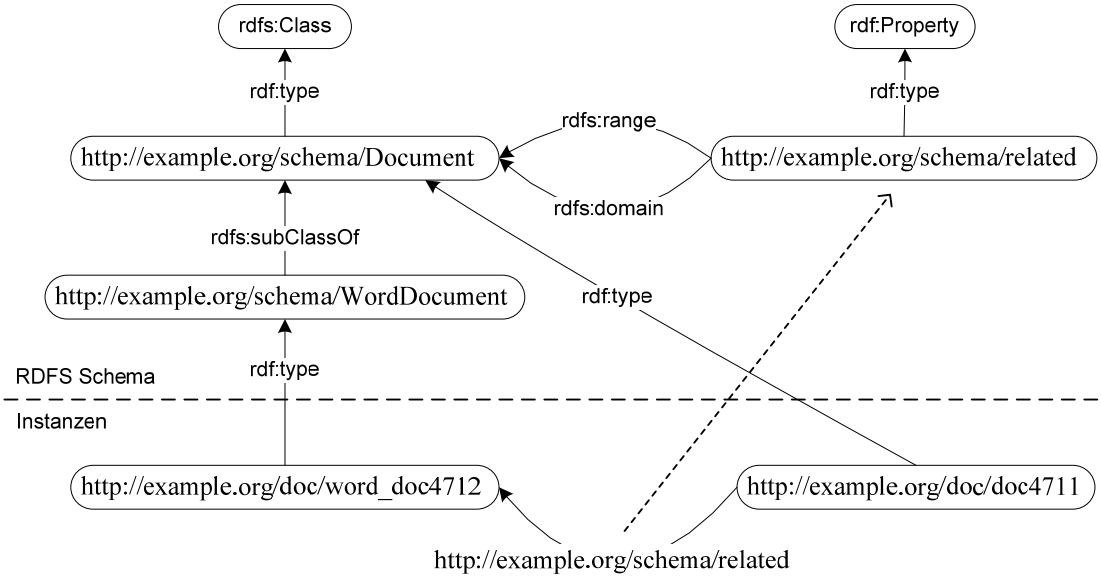

Abbildung 2.5: Konzeptionelle Trennung zwischen Schema und Instanzen

2.5 OWL

Die Ausdrucksstärke von RDFS ist sehr begrenzt und eignet sich damit nur eingeschränkt für komplexe Ontologien. Aus diesem Grund existiert mit OWL eine weitere auf RDF aufsetzende Ontologie Beschreibungssprache. Im Unterschied zu RDFS ermöglicht OWL eine mächtigere Beschreibung von Properties und erlaubt hierdurch formelle Beschreibungen, die mit RDFS nicht modellierbar sind. So können OWL Properties zum Beispiel als symmetrisch, transitiv, funktional oder zueinander invers deklariert werden. Da die Ausdruckskraft von OWL im Bereich semantischer Netze meist nicht benötigt wird, wird an dieser Stelle auf eine detaillierte Darstellung von OWL verzichtet.

2.6 SKOS

Das Simple Knowledge Organization System kurz SKOS ist eine vom W3C im Kontext des Semantic Web entwickelte formale Sprache. Sie dient zur Repräsentation von Struktur und Inhalt einfacher auf kontrollierten Vokabularien aufbauender Wissensstrukturen wie Glossaren, Thesauri oder Taxonomien. SKOS setzt auf RDF auf und definiert ein Vokabular aus Klassen und Properties. Es ermöglicht Konzepte zu definieren, welche über Literale beschriftet und mit semantischen Relationen in

Beziehung zueinander gesetzt werden können. Zusätzlich existieren Properties zur Dokumentation des enthaltenen Wissens. Im Folgenden werden die für diese Arbeiten relevanten Elemente des Vokabulars erläutert.

Für die Formalisierung von Ressourcen, die ein abstraktes gedankliches Konzept darstellen, existiert die Klasse *skos:Concept*. Sie ist das fundamentalste Element in SKOS. Über *skos:prefLabel, skos:altLabel* und *skos:hiddenLabel* können, den Ressourcen vom Typ *skos:Concept*, Bezeichnungen zugewiesen werden. Dabei ist *skos:prefLabel* die präferierte und *skos:altLabel* eine alternative Bezeichnung. Die Property *skos:hiddenLabel* ist ebenfalls alternativ, jedoch explizit nicht für die Anzeige gedacht. Neben der Festlegung von Bezeichnungen gehört die Formalisierung von semantischen Relationen zwischen den Konzepten zur Hauptaufgabe von SKOS. Um eine direkte hierarchische Beziehung zwischen zwei Konzepten zu formalisieren, existieren die Properties *skos:broader* und *skos:narrower***.** Sie bringen zum Ausdruck, dass ein Konzept in Bezug zu einem anderen Konzept "allgemeiner" beziehungsweise "spezifischer" ist. Beide Properties sind invers zu einander und nicht transitiv. Für die Formalisierung einer assoziativen nicht hierarchischen Beziehung zwischen zwei Konzepten existiert die symmetrische Property *skos:related*.

2.7 RDF-Repositories

Mit RDF und den auf RDF aufbauenden Standards wie OWL oder SKOS können Ontologien beliebiger Größe formalisiert werden. Um solche Ontologien zu speichern, abzufragen und zu verarbeiten werden RDF-Repositories eingesetzt. RDF- Repositories, häufig auch als Triple-Stores bezeichnet, sind dafür optimiert sehr große Mengen an RDF Statements zu verarbeiten. Zur Speicherung der Statements werden verschiedene Strategien eingesetzt. Ein großer Teil der RDF-Repositories verwendet hierfür im Hintergrund ein gewöhnliches RDBMS[1]. Dies hat den Vorteil, dass existierende Features wie Transaktionen genutzt werden können. Andere RDF- Repositories nutzen eigene dateibasierte Speichertechniken, oder setzen auf nicht rationalen Datenbanken auf. Zum Zugriff auf RDF unterstützen viele RDF Repositories mittlerweile die RDF Anfragesprachen SPARQL, welche seit Januar 2008 W3C Recommendation ist.

[1] Relational Database Management System

2.8 Begriffsdefinition Graphentheorie

Diese Arbeit beschäftigt sich unter anderem mit Konzepten der Graphentheorie. Als Grundlage hierfür werden in diesem Kapitel wichtige Begriffe der Graphentheorie definiert. Die Definitionen erfolgen weitgehend anlog zu [5], wobei die Terminologie stellenweise abweichen kann.

Ungerichteter Graph

Ein *ungerichteter Graph* $G = (V, E, \gamma)$ besteht aus zwei disjunkten Mengen V,E und einer Abbildung γ. Die Elemente der endlichen[2] nicht leeren Menge V werden als *Knoten*, die Elemente der Menge E als *Kanten* bezeichnet. Die Abbildung γ ordnet jeder Kante $e \in E$ ihre *Endknoten* zu. Die Endknoten $\gamma(e)$ sind dabei eine ein- oder zweielementige Teilmenge von V, es gilt also $1 \leq |\gamma(e)| \leq 2$.

Gerichteter Graph

Ein *gerichteter Graph* $G = (V, E, \alpha, \omega)$ besteht aus zwei disjunkten Mengen V,E und zwei Abbildungen α und ω. Die Elemente der nicht leeren Menge V und E werden wiederum als Knoten und Kanten bezeichnet. Die beiden Abbildungen $\alpha: E \to V$ und $\omega: E \to V$ ordnen jeder Kante $e \in E$ einen *Anfangsknoten* $\alpha(e)$ und einen *Endknoten* $\omega(e)$ zu, man sagt auch, die Kante e ist *von* $\alpha(e)$ *nach* $\omega(e)$ *gerichtet*. Zur Unterscheidung werden die Kanten eines gerichteten Graphen auch als *gerichtete Kanten* bezeichnet.

Inzidenz, Adjazenz

Inzidenz beschreibt eine Beziehung zwischen Knoten und Kanten eines Graphen.

In einem ungerichteten Graph $G = (V, E, \gamma)$ heißt ein Knoten $v \in V$ und eine Kante $e \in E$ *inzident* wenn $v \in \gamma(e)$.

[2] Die Graphentheorie befasst sich auch mit unendlichen Graphen, diese sind jedoch für diese Arbeit nicht relevant

Für einen gerichteten Graph $G = (V, E, \alpha, \omega)$ wird ein Knoten $v \in V$ und eine Kante $e \in E$ als *inzident* bezeichnet, wenn $v \in \{\alpha(e), \omega(e)\}$.

Im Gegensatz zur Inzidenz beschreibt Adjazenz eine Beziehung zwischen den Knoten eines Graphen. Für einen gerichteten oder ungerichteten Graph G mit der Knotenmenge V und der Kantenmenge E heißen zwei Knoten $u, v \in V$ *adjazent* oder *benachbart,* wenn eine Kante $e \in E$ existiert, die sowohl mit u als auch mit v inzident ist. Man sagt, die Kante $e \in E$ *verbindet* die beiden Knoten u, v.

Grad, Nachbarn, Nachfolger, Vorgänger

Im Folgenden werden eine Reihe von Notationen und Bezeichnungen definiert, die hilfreich für die Beschreibung von Adjazenz- und Inzidenzbeziehungen ausgehend von einem Knoten sind.

Sei $G = (V, E, \gamma)$ ein ungerichteter Graph, dann verwenden wir die Notation:

- $\delta(v) \coloneqq \{e \in E : v \in \gamma(e)\}$ für die Menge aller zu v inzidenten Kanten.
- $N(v) \coloneqq \{u \in V : \gamma(e) = \{u, v\} \, und \, e \in E\}$ für *Nachbarn* von v.
- $g(v) \coloneqq \sum_{e \in E : v \in \gamma(e)} (3 - |\gamma(e)|)$ für den *Grad* von v.

Sei weiter $G = (V, E, \alpha, \omega)$ ein gerichteter Graph dann verwenden wir die Notation:

- $\delta^+(v) \coloneqq \{e \in E : \alpha(e) = v\}$ für die von v *ausgehenden Kanten*.
- $\delta^-(v) \coloneqq \{e \in E : \omega(e) = v\}$ für die in v *mündenden Kanten*.
- $N^+(v) \coloneqq \{\omega(e) : r \in \delta^+(v)\}$ für die *Nachfolger* von v.
- $N^-(v) \coloneqq \{\alpha(e) : r \in \delta^-(v)\}$ für die *Vorgänger* von v.
- $g^+(v) \coloneqq |\delta^+(v)|$ für den *Außengrad* von v.
- $g^-(v) \coloneqq |\delta^-(v)|$ für den *Innengrad* von v.
- $g(v) \coloneqq g^+(v) + g^-(v)$ für den *Grad* von v.

Schlinge, isolierter Knoten, parallele Kante, einfacher Graph

In einem ungerichteten Graph $G = (V, E, \gamma)$ heißt eine Kante $e \in E$ *Schlinge,* wenn $|\gamma(e)| = 1$. Weiter werden zwei Kanten $e, e' \in E$ mit $e \neq e'$ als *parallel* bezeichnet, wenn $\gamma(e) = \gamma(e')$.

In einem gerichteten Graph $G = (V, E, \alpha, \omega)$ spricht man von einer Schlinge, wenn für eine Kante $e \in E$ gilt, dass $\alpha(e) = \omega(e)$. Zwei Kanten $e, e' \in E$ mit $e \neq e'$ heißen parallel, wenn $\alpha(e) = \alpha(e')$ und $\omega(e) = \omega(e')$.

Ein Knoten $v \in V$ eines gerichteten oder ungerichteten Graphen wird als *isolierter Knoten* bezeichnet, wenn der Grad von v null ist.

Von einem *einfachen gerichteten Graphen* beziehungsweise *einfachen ungerichteten Graphen* spricht man, wenn der Graph keine Schlingen und *parallele Kanten* enthält. Die in dieser Arbeit verwendeten Definitionen eines ungerichteten und eines gerichteten Graphen erlauben parallele Kanten. Für solche Graphen wird häufig auch die Bezeichnung *Multigraph* verwendet.

Inverse Kante, Inverser Graph

Sei $G = (V, E, \alpha, \omega)$ ein gerichteter Graph. Zwei gerichtete Kanten $e, e' \in E$ heißen *inverse Kanten* wenn $\alpha(e) = \omega(e')$ und $\omega(e) = a(e')$. Um die Inverse einer Kante $e \in E$ zu bezeichnen, wird auch die Notation e^{-1} verwendet. Ein zu G *inverser Graph* ist der Graph, der entsteht, wenn alle Kanten $e \in E$ durch ihre inversen Kanten ersetzt werden.

Weg, Pfad, Zyklus, Kreis

Für einen Graph G mit der Knotenmenge V und der Kantenmenge E ist ein **Weg** eine endliche Folge $W = (v_0, e_1, v_1, \dots, e_k, v_k)$ mit $k \geq 0$. Hierbei sind $v_0, \dots, v_k \in V$ Knoten und $e_0, \dots, e_k \in E$ Kanten von G. Ist e_i eine gerichtete Kante, so gilt für $i = 1, \dots, k$ dass v_{i-1} der Anfangsknoten und v_i der Endknoten von e_i ist. Für eine ungerichtete Kante e_i sind v_i, v_{i-1} die Knoten, die e_i verbindet wobei wiederum für $i = 1, \dots, k$ gilt. Der Knoten v_0 wird als *Startknoten* und der Knoten v_k als *Endknoten*

bezeichnet. Ein *Pfad* ist ein Weg, bei dem alle Knoten in W nur einmal enthalten sind und somit gilt, dass $v_i \neq v_j$ wenn $i \neq j$.

Hat ein Weg einen identischen Start- und Endknoten, so wird er als *Zyklus* bezeichnet. Ist ein Weg ein Zyklus, in dem alle Knoten bis auf Start- und Endknoten, nur einmal enthalten sind, wird von einem *Kreis* gesprochen.

Traversierung

Unter dem Begriff *Traversierung* wird hier ein Verfahren verstanden, bei dem alle Knoten und Kanten eines Graphen in einer systematischen Reihenfolge durchlaufen werden können. Man spricht auch davon, dass die Knoten eines Graphen bei einer Traversierung *besucht* oder *abgearbeitet* werden.

3 Anwendung semantischer Netze

Dieses Kapitel gibt einen Überblick über die xdot-spezifischen Anwendungsbereiche semantischer Netze bei denen Graph-Algorithmen zum Einsatz kommen. Es wird ein kurzer Überblick über eingesetzte Technologien und das WikiNet als Beispiel für ein großes semantisches Netz vorgestellt. Darauffolgend wird die Relevanz von Graph-Algorithmen bei der Verarbeitung von semantischen Netzen im Allgemeinen und anhand von zwei Szenarien aus der Praxis der xdot GmbH dargelegt. Im letzten Teil dieses Kapitels werden gemeinsame Eigenschaften, die für diese Form der Verarbeitung gefunden wurden, zusammengefasst.

3.1 Anwendungsbereiche

Die xdot GmbH entwickelt auf Projektbasis Produkte im Bereich Webtechnologien und Wissensmanagement. Mit dem Programm xfriend wird zusätzlich ein eigenes Information Retrival System vertrieben. Das im Rahmen dieser Arbeit entwickelte Framework soll sowohl in xfriend als auch in aktuellen und zukünftigen Projekten Verwendung finden. Im Bereich semantische Technologien werden bei der xdot GmbH überwiegend die im Semantic Web Stack spezifizierten Technologien verwendet.

Als Repository und Framework für die Verarbeitung kommt Sesame zum Einsatz. Bei Sesame handelt es sich um ein relativ weit verbreitetes in Java implementiertes RDF-Repository. Es wurde ursprünglich von der Firma Aduna[3] entwickelt und steht unter einer BSD-ähnlichen Lizenz. Einer der Vorteile von Sesame ist seine Flexibilität und Erweiterbarkeit. So ist es möglich, unterschiedliche Strategien zum Speichern der Statements zu verwenden. Sesame enthält standardmäßig Komponenten, um die Statements im Speicher (*MemoryStore*) in einem eigenen Dateiformat (*NativeStore*) oder in einem RDBMS (*RdbmsStore*) zu speichern. Bei der xdot GmbH kommt hauptsächlich der *NativeStore* zum Einsatz. Der Vorteile des *NativeStore* ist, dass keine zusätzliche Datenbank zur Verfügung stehen muss.

Semantische Netze enthalten Wissen über Begriffe und Relationen zwischen diesen. Dieses Wissen wird bei der xdot GmbH für die Verarbeitung von textuellen Inhalten (Computerlinguistik) eingesetzt, um zum Beispiel einen Text nach seinem Inhalt zu

3 http://www.aduna-software.com

klassifizieren. Zusätzlich werden diese Netze zur Strukturierung und Annotation von Inhalten verwendet. Diese Annotationen können daraufhin zum Beispiel zur Navigation, Visualisierung oder Query expansion[4] eingesetzt werden.

Eine Schwierigkeit stellt in der Praxis die manuelle Erstellung semantischer Netze dar. Wird für eine spezielle Domäne ein semantisches Netz benötigt, so besteht meist noch die Möglichkeit, dieses manuell zu erstellen und zu pflegen, da in der Regel nur eine begrenzte Anzahl an Begriffen benötigt wird. Anders ist es, wenn die Domäne im Vorfeld nicht bekannt ist oder der Anwendungsfall ein sehr allgemeines semantisches Netz voraussetzt. Hier werden semantische Netze mit sehr vielen Begriffen benötigt. Solch semantische Netze manuell zu erstellen, ist kaum noch zu bewerkstelligen. Aus diesem Grund werden automatische oder semi-automatische Verfahren zur Erstellung eingesetzt. Automatisch oder semi-automatisch erstellte semantische Netze können Millionen Konzepte und Relationen enthalten.

3.2 WikiNet

Das WikiNet dient in dieser Arbeit als Beispiel für ein großes semantisches Netz. Es wurde zum Großteil automatisch aus der deutschen Wikipedia extrahiert und entspricht formal einem Thesaurus. Die einzelnen Seiten der Wikipedia bilden in diesem Fall die Begriffe beziehungsweise die Konzepte. Diese Konzepte sind wiederum über Relationen verbunden, die aus den Links zwischen den einzelnen Seiten und dem Kategorien-System der Wikipedia abgeleitet werden. Jedem Konzept wird durch die Property *skos:prefLabel* eine eindeutige Bezeichnung zugeordnet. Zusätzlich existieren synonyme Bezeichnungen, die über die Property *skos:altLabel* zugeordnet werden. In Beziehung zueinander stehen die Konzepte über die Relationen *skos:narrower* und *skos:broader* sowie *skos:related*. Die Abbildung 3.1 zeigt einen Ausschnitts aus dem WikiNet als gerichteter Graph, die URIs sind gekürzt.

[4] Mit Query expansion wird die Erweiterung einer Schlagwort Suchanfrage um zum Beispiel Synonyme Schlagworte bezeichnet.

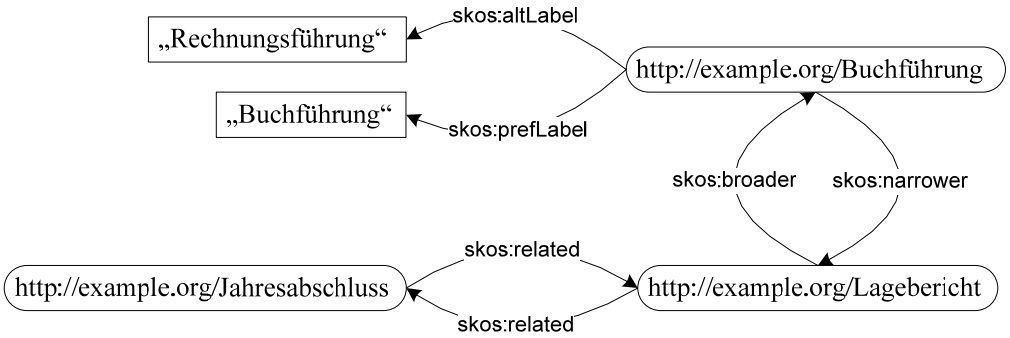

Abbildung 3.1: Ausschnitt aus dem WikiNet als gerichteter Graph

Die Tabelle 3.1 und Tabelle 3.2 zeigt eine Übersicht der Ressourcen und Statements des WikiNet. In Tabelle 3.1 wird dafür die Anzahl der Statements mit einem bestimmten Prädikat aufgelistet. In Tabelle 3.2 hingegen wird die Anzahl der Ressourcen eines bestimmten Typs dargestellt.

Prädikat	Anzahl Statements
skos:related	4.023.361
skos:narrower	3.262.705
skos:broader	3.262.705
skos:prefLabel	2.821.114
skos:altLabel	2.642.438
rdf:type	2.821.185
Andere	187
Alle	18.833.695

Tabelle 3.1: Anzahl Statements pro Prädikat

Type	Anzahl Ressourcen
skos:Concept	2.821.114
RDF Literale	5.277.670
Andere	50
Alle	8.098.834

Tabelle 3.2: Anzahl Statements pro Ressource Typ

3.3 Szenarien

Im Folgenden wird anhand von zwei Szenarien beschrieben, wie große semantische Netze bei der xdot GmbH eingesetzt beziehungsweise verarbeitet werden.

3.3.1 Hierarchie zur Kategorisierung

Die hierarchische Struktur, die durch die SKOS *skos:broader* und *skos:narrower* Relationen des WikiNet gebildet wird, soll zum Einordnen von Dokumenten genutzt werden. Zusätzlich soll diese Hierarchie dem Benutzer zur Navigation präsentiert werden. Da es möglich sein soll, dass ein Konzept mehr als einem Oberkonzept untergeordnet ist, handelt es sich um eine Polyhierarchie. Sie hat dabei die Struktur eines gerichteten azyklischen Graphen. Die Kanten dieses Graphen sind die *skos:narrower* Relationen.

Um eine sinnvolle hierarchische Einordnung und spätere Navigation zu ermöglichen, muss diese Hierarchie einige Anforderungen erfüllen. Zunächst muss sie konsistent sein. Im Falle einer Polyhierarchie bedeutet dies, dass der durch sie beschriebene Graph keine Zyklen enthalten darf (azyklisch). Weiterhin muss gewährleistet sein, dass alle im semantischen Netz enthaltenen Konzepte in die Hierarchie eingeordnet sind und keine abgetrennten Teilhierarchien existieren. Auf den Graph bezogen heißt das, dass alle Konzepte mit der Hierarchie Wurzel über einen Pfad verbunden sein müssen.

Diese Voraussetzungen sind bei dem WikiNet nicht gegeben. Für die weitere Qualitätsverbesserung und Anpassung ist es nötig, dass das WikiNet auch manuell bearbeitet werden kann. Hierbei müssen sowohl einzelne Konzepte und Relationen als auch ganze Teilgraphen gelöscht oder hinzugefügt werden können. Durch diese manuellen Änderungen kann wiederum schnell die geforderte Struktur zerstört werden. Da es durch die Größe des WikiNet jedoch nicht einfach möglich ist, die gesamte Struktur zu visualisieren, kann der Bearbeiter solche Fehler oft nicht selbst erkennen. Aus diesem Grund muss die hierarchische Struktur permanent im Hintergrund auf Konsistenz geprüft werden. Somit muss die Laufzeit hierfür zuständigen der Algorithmen möglichst gering sein.

3.3.2 Kategorisierung

Für die Kategorisierung von Texten in eine Hierarchie, wie im vorhergehenden Kapitel beschrieben, verwendet die xdot GmbH einen Algorithmus, der auf Aktivierungsausbreitung (spreading activation) basiert. Die grundlegende Funktionsweise dieses Algorithmus wird im Folgenden näher erläutert.

Zunächst wird der zu klassifizierende Text nach Termen durchsucht, die in dem semantischen Netz als Label vorkommen. Sind alle Labels im Text gefunden, erhält jedes Konzept, das durch einen der gefundenen Labels bezeichnet wird, eine Grundaktivierung. Im nächsten Schritt wird diese Aktivierung an die über die skos:*broader* Relation verbundenen Konzepte weitergegeben. Hierbei verstärken sich aufeinandertreffende Aktivierungen gegenseitig. In welchem Umfang sich eine Aktivierung in der Hierarchie ausbreitet, richtet sich nach dem Wert der Grundaktivierung und nach der Tiefe, in der sich die jeweiligen Konzepte in der Hierarchie befinden. Werden viele Labels in einem Text gefunden, können schnell einige tausend Konzepte aktiviert werden. Die eigentliche Kategorisierung wird im Anschluss durch das Ranking berechnet, dass sich durch die Aktivierung der Konzepte ergibt.

In den meisten Fällen findet die Kategorisierung in einem Batch-Prozess parallel zu einer Ressourcen-hungrigen Indexierung, zum Beispiel von Textdokumenten, statt. Aufgrund dessen sind Anforderungen an die Kategorisierung, neben einer guten Einordnung, vor allem ressourcenschonendes Arbeiten.

3.4 Graph Algorithmen

Bei der Verarbeitung semantischer Netze kommen häufig Algorithmen der Graphentheorie zur Anwendung. Der Grund hierfür liegt in der Tatsache, dass in einem semantischen Netz das Wissen zu einem großen Teil implizit in der Struktur des Netzes verborgen ist. Ein beliebiger Pfad zwischen zwei Konzepten stellt eine semantische Assoziation zwischen diesen beiden dar (siehe [6]) und beinhaltet somit Wissen zu beiden Konzepten. Die Semantik der Assoziation wird dabei durch die einzelnen Elemente des Pfades bestimmt. Hierdurch wird das Wissen über ein Konzept durch alle Konzepte und Relationen der zusammenhängenden Komponenten definiert, in der das Konzept zu finden ist.

Die Suche nach bestimmten semantischen Assoziationen beziehungsweise Pfaden ist für viele Anwendungsfälle von großer Bedeutung. Beispiele hierfür sind Abfragen wie:

- "*Welcher Pfad ist der kürzeste Pfad eines Konzeptes in einer skos:broader Hierarchie zum Wurzelknoten?"*
- "*Gibt es einen Pfad, der zwei Konzepte über skos:related Relationen verbindet und wie lang ist dieser Pfad?*"
- *"Wie nahe oder wie ähnlich sind sich zwei Konzepte?"*[5]

Solche Pfad-Abfragen sind auch für andere Ontologie-Typen von Interesse. Beispiele hierfür finden sich in [7] und [6]. Die Abfrage von Pfaden im RDF-Graph wird durch die Sesame API jedoch nicht direkt unterstützt. In [7] wird beschrieben, dass, wenn nicht zumindest die Länge des Pfades bekannt ist, dies sowohl für SeRQL[6], als auch für sechs weitere RDF-Anfragesprachen zutrifft. Dies gilt auch für die vom W3C empfohlene Abfragesprache SPARQL. In [6] wird daher eine Erweiterung für SPARQL, namens SPARQLeR, vorgestellt, die die Abfrage von durch Pfade definierte, semantische Assoziationen ermöglicht. Verwendet werden hierfür Pfadvariablen, für die über sogenannte "path patterns" Bedingungen festgelegt werden können.

Neben der Suche nach bestimmten Pfaden, die sich über eine Breitesuche realisieren lässt, benötigen einige Algorithmen komplexere Möglichkeiten der Traversierung. So muss im Szenario aus Kapitel 3.3.1 sichergestellt werden, dass der durch die Relationen *skos:broader* und *skos:narrower* gebildete Graph azyklisch ist. Um dies zu gewährleisten, kann eine Tiefensuche eingesetzt werden, die alle Kanten des Graphen klassifiziert (siehe [5]). Bei dem im Szenario aus Kapitel 3.3.2 beschriebenen Algorithmus breitet sich die Aktivierung iterativ aus, ausgehend von den im Graphen verstreuten Knoten über inzidenten Kanten. Um Wissen aus einem semantischen Netz zu gewinnen, sind auch Algorithmen interessant, die zur Analyse komplexer Netzwerke eingesetzt werden. So wird in [8] der PageRank Algorithmus (siehe [9]) erfolgreich dazu verwendet, Wörter eines Textes zu disambiguieren[7].

[5] Details zur Definition von semantischer Nähe und Ähnlichkeit finden sich in [14].

[6] SeRQL ist neben SPARQL eine RDF Abfragesprache die von Sesame unterstützt wird.

[7] Bei der Disambiguierung werden Wörter eindeutig ihrer Bedeutung zugeordnet.

3.5 Gemeinsamkeiten der Verfahren

Bei Betrachtung der beschriebenen Verfahren und Algorithmen zeigen sich gemeinsame Eigenschaften hinsichtlich der Verarbeitung durch Graph-Algorithmen. Eine Eigenschaft der beschriebenen Algorithmen ist, dass bei der Verarbeitung nur auf Statements mit einem oder sehr wenigen unterschiedlichen Prädikaten zugegriffen wird. Um dies zu verdeutlichen, wird das Szenario aus Kapitel 3.3.1 aufgegriffen. Hier soll eine bestimmte Hierarchie sichergestellt werden, die allein durch die Statements mit den Prädikaten *skos:broader* und *skos:narrower* definiert werden. Da *skos:broader* und *skos:narrower* zueinander inverse Properties sind, reicht zur vollständigen Beschreibung der Hierarchie eine dieser beiden Properties aus. Der Graph, der bei diesem Anwendungsfall benötigt wird, besteht somit nur aus Statements mit identischem Prädikat. An diesem Beispiel ist weiter ersichtlich, dass der Teilgraph, auf dem die Algorithmen arbeiten, meist in großem Umfang verarbeitet werden muss. Dies trifft auch auf Pfad-Abfragen zu. Denn auch hier sind meist nur bestimmte semantische Assoziationen, und somit Prädikate, von Interesse, wobei bei der eigentlichen Suche wiederum große Teile des Graphen betrachtet werden müssen.

Typisch für die Verarbeitung ist auch, dass hauptsächlich die Eigenschaften des durch die RDF-Statements gebildeten Graphen benötigen werden. Soll zum Beispiel der kürzeste Pfad zwischen zwei Kantenobjekten gefunden werden, reicht die Information welche Knoten über welche Kanten verbunden sind, um einen Pfad zu ermitteln. Die Objekte[8], die hinter den Knoten und Kanten stehen, werden erst für den gefundenen Pfad benötigt. Ein weiteres Beispiel ist der im Kapitel 3.3.1 zur Konsistenzprüfung nötige Test, ob ein Graph azyklisch ist. Auch hier sind die konkreten Objekte erst für gefundene Zyklen von Interesse. Im Umkehrschluss bedeutet dies, dass die konkreten Objekte, die hinter Knoten und Kanten dieses Graphen stehen, häufig nicht benötigt werden.

Die gefundenen Gemeinsamkeiten bei der Verarbeitung semantischer Netze durch Graph-Algorithmen lassen sich wie folgt zusammenfassen:

- *Die einzelnen Algorithmen arbeiten meist auf Teilgraphen aus wenigen Properties.*

[8] Mit Objekten sind hier die RDF Objekte URIs, Literale, Statements und blank nodes gemeint.

- *Ein Teilgraph muss in großem Umfang beziehungsweise vollständig verarbeitet werden.*
- *Die Verarbeitung geschieht meist in Form einer Traversierung.*
- *Die Struktur der Teilgraphen steht im Vordergrund.*

4 Problembeschreibung

Mit den zurzeit bei der xdot GmbH eingesetzten Technologien scheitert eine praxistaugliche Umsetzung von Graph-Algorithmen für große semantische Netze häufig an sehr hohem Speicherverbrauch und langen Ladezeiten. Der Grund für den hohen Speicherverbrauch ist, dass große Teile des RDF-Graphen im Speicher gehalten werden. Dies ist erforderlich, da Zwischenergebnisse durch die Algorithmen benötigt werden, und nur so auch die nötige Performance erreicht werden kann. Hier sind es nicht allein die Sesame Literal Objekte, die viel Speicher belegen. Ein großer Teil wird auch dadurch belegt, dass URI Objekte in Sesame intern durch den zugehörigen URI-String repräsentiert werden.

Neben dem hohen Speicherverbrauch stellen die langen Ladezeiten, die beim Zugriff auf eine große Menge an Statements entstehen, ein Problem dar. Sie entstehen dadurch, dass der *NativeStore* von Sesame auf das schnelle Beantworten von vielen einzelnen Anfragen zum Beispiel über SPARQL optimiert ist. Dies hat zur Folge, dass auch bei einem sequentiellen Zugriff pro Statement mehrere I/O Operationen auf dem externen Speicher durchgeführt werden müssen. Da der Zugriff auf externe Speicher wie Festplatten, SSDs oder auch CDs im Verhältnis sehr langsam ist, wird entsprechend viel Zeit benötigt. Auf dem Testsystem brauchte das Laden aller *skos:narrower* Statements mehr als zwei Minuten. Auch dieses kann im praktischen Einsatz sehr problematisch sein.

Neben dem *NativeStore* bietet Sesame auch noch den *MemoryStore*. Zur Laufzeit hält er den kompletten RDF-Graph im Speicher, wobei er gleichzeitig persistent auf einem externen Speicher liegen kann. Durch das Dateiformat, das der *MemoryStore* verwendet, wird der RDF-Graph schneller als beim *NativeStore* in den Speicher geladen. Zur Lösung der Problematik kann er jedoch nicht weiter helfen, da auch bei dem *MemoryStore* alle RDF-Objekte im Speicher gehalten werden und somit auch hier viel Speicher benötigt wird. Zudem lädt der *MemoryStore* immer den gesamten RDF-Graphen in den Speicher.

Das WikiNet ist das zurzeit größte durch die xdot GmbH eingesetzte semantische Netz. Es ist jedoch geplant, ein semantisches Netz mit der gleichen Technik aus der englischen Wikipedia zu erstellen. Da die englische Wikipedia deutlich größer ist als die deutsche Version, wird dieses semantische Netz etwa um den Faktor drei größer als

das beschriebene WikiNet. Hierdurch wird die hier beschriebene Problematik noch verschärft.

Eine weitere Problemstellung, die auch bei kleineren semantischen Netzen besteht, ist, dass die Umsetzung von Algorithmen durch die Sesame API nicht direkt unterstützt wird. Die von Sesame unterstützten Abfragesprachen SeRQL und SPARQL bieten, wie im Kapitel 3.4 bereits beschrieben, keine Möglichkeit Pfade flexibler Länge abzufragen. Neben den Anfragesprachen können über die Methode *getStatements()* der Sesame API Statements direkt abgefragt werden. Diese Methode erwartet die Parameter Subjekt, Prädikat und Objekt und liefert passende Statements. Wird als Wert für Subjekt, Prädikat oder Objekt *null* angegeben steht dies für einen beliebigen Wert (siehe [10]). Hierüber ist es möglich, die Nachbarn eines Konzeptes abzufragen. Allerdings besteht hierbei die beschriebene Problematik, dass dieser Zugriff durch die I/O Operationen langsam ist. Zudem müssen alle weiteren grundlegenden Graph- Zugriffe über diese Methode umgesetzt werden. Um die Umsetzung speziell von Graph-Algorithmen zu unterstützen, wäre eine API wünschenswert, die sich mehr auf eine graphentheoretische Sicht fokussiert und bereits Grundlegende Graph Algorithmen bereitstellt.

5 Anforderungen an das Framework

Die in den vorherigen Kapiteln beschriebenen Szenarien und Aufgabenstellungen lassen sich aufgrund der beschriebenen Problematik für große semantische Netze, wenn überhaupt, nur umständlich und mit hohem Ressourcenaufwand umsetzen. Ziel dieser Arbeit ist es, hierfür eine Lösung in Form eines Frameworks zu entwickeln.

Die Anforderungen, die an dieses Framework gestellt werden, werden nicht alleine durch das technische Problem, sondern auch durch die Rahmenbedingungen bestimmt, die durch das geplante Einsatzgebiet sowie die bisher eingesetzten Technologien der xdot GmbH vorgegeben sind.

Da die beschriebenen Szenarien, insbesondere auch das von der xdot GmbH entwickelte, kommerziell Information Retrieval System xfriend betreffen, muss bei dem Einsatz von Drittsoftware sowohl die Lizenz der Software als auch eine sinnvolle Integration in das bestehende Produkt und die existierenden Technologien berücksichtigt werden.

Neben der Festlegung auf Java als Programmiersprache ist die weitreichendste Anforderung, die es im Vorfeld auf technologischer Seite zu berücksichtigen gilt, dass Sesame als RDF-Repository weiter genutzt werden soll, da es sich im praktischen Alltag bewährt hat. Das Framework soll aus diesem Grund auf Sesame als Repository aufsetzen und die gegebene Funktionalität erweitern und sich möglichst einfach in bereits existierende Anwendungen integrieren lassen.

Hauptaufgabe des Frameworks ist es, die praxistaugliche Verarbeitung von großen semantischen Netzen durch Graph-Algorithmen zu ermöglichen. Um die Umsetzung der Algorithmen zu vereinfachen, erweitert das Framework die Sesame API funktional um eine API, die sich an graphentheoretischen Konzepten orientiert. Ein Schwerpunkt liegt dabei auf Möglichkeiten zur Traversierung des RDF-Graph, um eine flexible und einfache Realisierung von Pfad-Abfragen zu ermöglichen.

Als nichtfunktionale Anforderungen kommt hinzu, dass bei der Verarbeitung großer semantischer Netze der Speicherverbrauch gering gehalten werden muss. Als Referenz für ein großes semantisches Netz dient das WikiNet. Ein semantisches Netz dieser Größe muss sich problemlos auf einem 32-Bit-System verarbeiten lassen. Um eine praxistaugliche Umsetzung von Algorithmen zu ermöglichen, ist zudem auch die

Performance ein wichtiger Faktor. Da gerade in performancekritischen Anwendungsfällen meist Wissen aus den semantischen Netzen extrahiert wird, muss die Verarbeitung für den lesenden Zugriff optimiert sein.

6 Ansätze

In diesem Kapitel werden zwei Ansätze in Form von existierenden Technologien beschrieben, die als mögliche Grundlage für das zu entwickelnde Framework in Betracht gezogen wurden.

6.1 Graphen-Datenbanken

Die Verarbeitung großer Graph- und Netzwerkstrukturen hat, insbesondere in den letzten Jahren, mit der Entstehung großer sozialer Netze und der Idee des Semantic Web immer mehr an Bedeutung gewonnen. Werden solche vernetzten Daten in klassischen RDBMS repräsentiert, führen komplexere Abfragen häufig zu kostspieligen Join Operationen. Aus diesem Grund wurden sogenannte Graph-Datenbanken entwickelt, die speziell auf die Verarbeitung von stark vernetzten Daten ausgelegt sind. Beispiele solcher Graph-Datenbanken sind HyperGraphDB[9], AllegroGraph[10] oder Neo4J[11].

Graph-Datenbanken adressieren prinzipiell die hier beschriebenen Probleme. Der Einsatz einer Graph-Datenbank als Basis zur Lösung der hier beschriebenen Problematik stellt jedoch aus verschiedenen Gründen keine zufriedenstellenden Ansatz dar. Zum einen stehen die meisten Java basierten Graph-Datenbanken unter einer kommerziellen oder einer "starken" Copyleft Lizenz, was die Integration in eigene kommerzielle Software ausschließt. Zum anderen sollen die semantischen Netze, die verarbeitet werden, weiter in Sesame gespeichert werden. Der Einsatz einer Graph-Datenbank würde somit zu einer redundanten Datenhaltung führen.

6.2 Graph Frameworks

Eine weitere Möglichkeit Graphen zu verarbeiten sind Frameworks, die auf die Verarbeitung von Graphen im Speicher ausgelegt sind. Solche Graph libraries oder Graph Frameworks bieten Datenstrukturen und Algorithmen für Graphen unterschiedlichen Typs und orientieren sich dabei an den Konzepten der Graphentheorie. Somit beinhalten sie bereits einen Teil der für das Framework

[9] http://www.kobrix.com/hgdb.jsp

[10] http://www.franz.com/agraph/allegrograph/

[11] http://neo4j.org/

geforderten Funktionalität. Aus diesem Grund wurde evaluiert, ob die Anforderungen an das zu entwickelnde Framework unter der Zuhilfenahme eines Graph Frameworks realisiert werden können.

Bei der Recherche nach geeigneten Frameworks wurden zwei mögliche Kandidaten ermittelt. Es handelt sich um die Frameworks Jung[12] und JGraphT[13]. Beide Frameworks sind in Java implementiert und stehen unter geeigneter Opensource Lizenz. Als interessanter weiterer Kandidat sei hier gephi[14] erwähnt. Gephi fokussiert sich auf die Visualisierung und Analyse großer Netzwerke und Graphen. Leider steht gephi unter GPL Lizenz und kann somit nicht zur Realisierung des Frameworks genutzt werden.

Grundlage der Evaluierung der beiden Frameworks Jung und JGraphT war die Frage, wie kompakt sich das im Kapitel 3.2 beschriebene WikiNet beziehungsweise Teile aus diesem durch die Frameworks als Graph repräsentieren lassen. Hierfür wurden einfache Testprogramme implementiert und der Speicherverbrauch ermittelt.

Vom grundlegenden API Konzept sind sich die Frameworks sehr ähnlich. Ein Graph wird jeweils durch ein Java Interface der Form *Graph<V,E>* deklariert. Hier kann man erkennen, dass beide Frameworks keine[15] Anforderungen an die Java Klassen stellen, die in dem Graph als Knoten und Kanten verwendet werden. Über Java Generics können die Graphen vom Nutzer der API bei der Implementierung typisiert werden. Für die Repräsentation eines RDF-Graph können hier die Sesame Objekte *Value*[16] und *Statement* als Typen für Knoten und Kanten verwendet werden. Da diese Objekte jedoch, wie bereits beschrieben, zu viel Speicher benötigen, wurde ein anderer Ansatz zur Repräsentation evaluiert.

Grundlage dieses Ansatzes ist es, jedem *Value* Objekt aus Sesame eine eindeutige Integer ID zuzuordnen. Hierauf aufbauend wurden anstatt der *Value* Objekte die zugeordneten Integer ID Objekte als Knoten verwendet. Jedes Statement wurde durch eine Kante repräsentiert. Für die Kanten wurde die Klasse *IDEdge* verwendet. Das einzige Attribut dieser Klasse war die primitive Integer ID des Prädikats. Für Jung

[12] http://jung.sourceforge.net/index.html

[13] http://www.jgrapht.org/

[14] http://gephi.org/

[15] Abgesehen von einer korrekten Implementierung der equals() und hashCode() Methoden.

[16] Das Interface Value wird von Sesame URIs, blank nodes und Literale implementiert.

wurde ein Graph vom Typ *DirectedSparseGraph<Integer, IDEdge>* und für JGraphT ein Graph vom Typ *DefaultDirectedGraph<Integer, IDEdge>* instanziiert.

Um abschätzen zu können, inwieweit sich eine solche Repräsentation hinsichtlich des Speicherverbrauchs als Lösungsansatz eignet, wurde jeweils ein Graph für alle Statements mit dem Prädikat *skos:narrower* des WikiNet erstellt. Der Graph, der sich hierdurch ergibt, hat etwa 3,2 Mio. Kanten und etwa 1 Mio. Knoten. Getestet wurde sowohl mit dem JDK 6 32-Bit als auch mit der JDK 6 64-Bit. Weitere Details zur Testumgebung finden sich im Kapitel 9.

Abbildung 6.1 zeigt die Ergebnisse der Speichermessung. Trotz minimalistischem Ansatz entsteht ein relativ hoher Speicherverbrauch. Der Grund hierfür liegt in den Verwaltungsstrukturen, die zum Speichern der Knoten und Kanten verwendet werden. Die Nutzdaten belegen nur einen geringen Teil des Speichers. Beispielsweise werden bei Jung in der 32-Bit JVM über 500 MB des Heap allein durch die Java *HashMap* Klasse und ihre Einträge belegt[17].

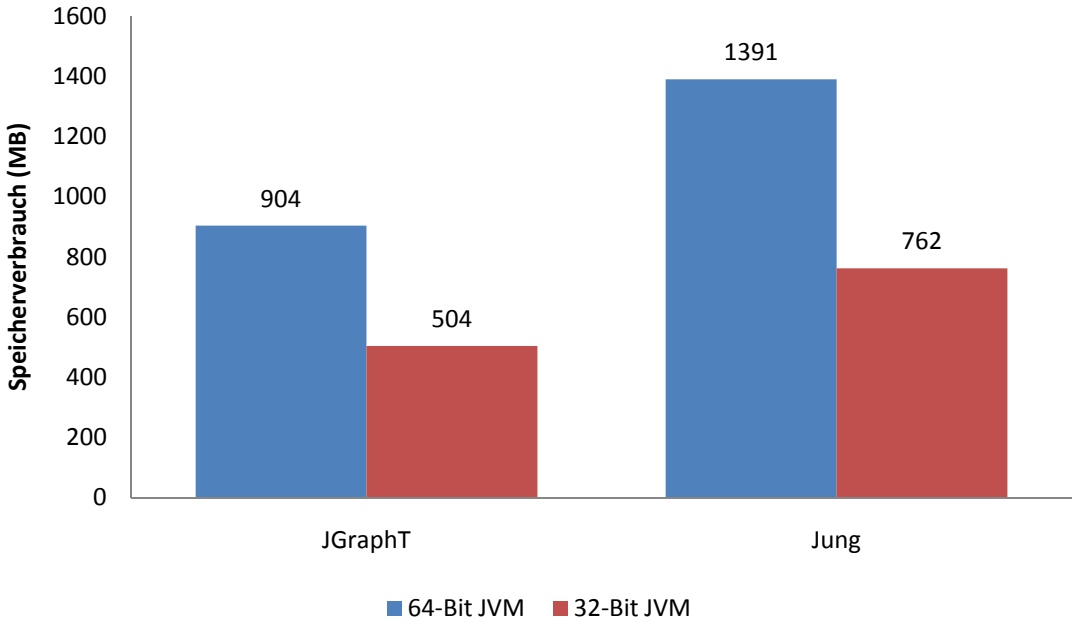

Abbildung 6.1: Speicherverbrauch WikiNet Narrower Graph

[17] Ermittelt mit dem Tool "VisualVM" aus dem Sun JDK 1.6

Für Anwendungsfälle, wie der der Kategorisierung aus Kapitel 3.3.2, kann der ermittelte Speicherverbrauch insbesondere auf einer 32-Bit JVM bereits als problematisch angesehen werden. Werden zusätzlich zu den *skos:narrower* auch die *skos:related* Relationen des WikiNet benötigt, wird bereits ein großer Teil des maximal möglichen Speichers einer 32-Bit JVM belegt[18].

Die Speichermessungen von Jung und JGraphT haben gezeigt, dass die Repräsentation von großen Graphen mit mehreren Millionen Kanten durch sie generell möglich ist. Besonders für eine Anwendung, in der die Verarbeitung von RDF als Graph lediglich einen kleinen Teil der Gesamtfunktionalität ausmacht, ist der Speicherverbrauch jedoch weiterhin problematisch. Der große Overhead, der bei beiden Frameworks durch die verwendeten Verwaltungsstrukturen wie Hash-Maps entsteht, lässt jedoch vermuten, dass eine deutlich kompaktere Repräsentation möglich ist. Aus diesem Grund wurde anstatt der beiden Frameworks ein eigener Ansatz zur Repräsentation großer Graphen realisiert.

[18] Je nach System liegt der real verwendbare Speicher zwischen 1,4 und 3 GB.

7 Framework Entwurf

Auf Grundlage der Anforderungen wurden zwei Hauptkomponenten des Frameworks identifiziert. Zum einen die Graph-Komponente, die für die kompakte Repräsentation und algorithmische Verarbeitung von Graphen zuständig ist. Zum anderen die Graph Sail-Komponente, welche die Graph-Komponente nutzt, um in Sesame gespeicherte RDF-Statements als Graph zu repräsentieren. In diesem Kapitel wird eine Übersicht über die Architektur des implementierten Frameworks gegeben. Dafür werden die beiden Hauptkomponenten des Frameworks und die durch sie realisierten grundlegenden Konzepte und Verfahren beschrieben.

7.1 Graph Sail-Komponente

Die Graph Sail-Komponente ist der zentrale Zugriffpunkt der Framework API. Sie erweitert die normale Sesame API um die Möglichkeit, die in Sesame enthaltenen Statements als Graph zu repräsentieren. Dabei besteht die Hauptaufgabe in dem Aufbau und der Verwaltung der Graphen. Zur Repräsentation der Graphen nutzt sie die Graph-Komponente. Um die Ladezeiten beim Aufbau der Graphen zu beschleunigen, realisiert sie zudem einen persistenten Cache.

7.1.1 Konsistenz des RDF-Graph Modells

Ein gerichteter Graph ist eine anschauliche und intuitiv verständliche Form einen RDF-Graph darzustellen. Wie in [11] beschrieben, stellt ein gerichteter Graph jedoch kein formal korrektes Modell für das RDF-Datenmodell dar. Das Problem liegt in der Tatsache, dass eine RDF-Ressource sowohl als Kante als auch als Knoten existieren kann. Das bedeutet, es können Ressourcen existieren, die als Prädikat in einem Statement verwendet werden. Dies ist zum Beispiel der Fall, wenn in RDFS eine Ressource vom Typ *rdf:Property* definiert und verwendet wird. Wird der RDF-Graph als gerichteter Graph repräsentiert, so ist die Ressource einer Property ein Knoten, die Verwendung als Prädikat dagegen eine Kante. Da in einem mathematisch korrekten Graph die Menge der Knoten und Kanten disjunkt sind, bedeutet dieses, dass die Beziehung eines Prädikates zu seiner Definition, der Property, sich nicht korrekt durch einen solchen Graphen darstellen lässt. Der gestrichelte Pfeil zwischen dem "related"

Prädikat und dem zugehörigen Property in der Abbildung 2.5 zur konzeptionellen Trennung von Struktur und Instanzen verdeutlicht dies.

Diese Inkonsistenz, die sich bei der Repräsentation von RDF als gerichteter Graph ergibt, stellt für die Graphen des Frameworks prinzipiell kein grundsätzliches Problem dar. Das Framework soll lediglich eine „Sicht“ auf das darunter liegende RDF-Modell als Graph bieten. Das konsistente RDF-Modell bleibt weiter in Sesame. Nachteil der Repräsentation von RDF als gerichteter Graph ist jedoch die im Graph selber nicht formalisierte Verbindung zwischen Prädikat und der Semantik des Prädikats, welche durch das zugehörige Property gegeben ist. Da sich jedoch die meisten Anwendungsfälle, für die das Framework gedacht ist, auf bestimmte Properties mit fester Semantik beziehen, besteht die Möglichkeit, diese Semantik auf anderem Weg in die Verarbeitung einzubeziehen. Durch die Repräsentation von RDF als gerichteter Graph entstehen im konkreten Anwendungsfall somit keine Einschränkungen. Aus diesem Grund wurde für die Entwicklung des Frameworks auf die Realisierung eines formal korrekten Graph-Modells für RDF verzichtet.

7.1.2 RDF-Graph als gerichteter Graph

Ausgangspunkt der Repräsentation eines RDF-Graphen als Graph ist die in [4] beschriebene Darstellung eines RDF-Graph als gerichteter Graph. Wie bereits im Kapitel 2.3 beschrieben, wird bei dieser Modell jedes Statement durch eine gerichtete Kante vom Subjekt zum Objekt dargestellt. Im Folgenden wird dieser gerichtete Graph als theoretische Grundlage für weitere Betrachtungen formal definiert. Dabei wird analog zu [11] vorgegangen.

Zunächst werden Mengenbezeichnungen für die RDF-Objekte definiert. Es wird U als die Menge aller URIs, L als die Menge aller RDF-Literale und B als die Menge aller blank nodes bezeichnet. Zusätzlich ist $R := U \cup B$ die Menge aller Ressourcen.

Wie bekannt, ist jedes Statement ein Triple $(\mathrm{s}, \mathrm{p}, \mathrm{o})$, wobei s das Subjekt, p das Prädikat und o das Objekt ist. Für ein Triple gilt $(s, p, o) \in R \times U \times (R \cup L)$.

Ein RDF-Graph T ist eine Menge an Statements. Die drei Abbildungen $sub: T \to R$, $pred: T \to U$ und $obj: T \to (R \cup L)$ ordnen einer Menge an Statements die vorkommenden Subjekte, Prädikate und Objekte zu.

Da die Knoten und Kanten eines Graphen sich aus graphentheoretischer Sicht nicht auf Objekte beziehen, werden die RDF-Objekte als Labels verstanden, die über Labeling-Funktionen den Knoten und Kanten des Graphen zugeordnet werden. Der Graph eines RDF-Graph ist ein gerichteter Graph $\delta = (V, E, \alpha, \omega)$ zusammen mit zwei Labeling-Funktonen l_V, l_E. Jedes RDF-Objekt, das als Subjekt oder Objekt verwendet wird, ist einem Knoten des Graphen zugeordnet. Die Funktion l_V ordnet einem Knoten das jeweilige RDF-Objekt zu. Die Knoten des Graphen sind gegeben durch

$$V = \{v_x : x \in sub(T) \cup obj(T) \wedge l_v(v_x) = x\}.$$

Für jedes Statement des RDF-Graphen existiert eine Kante. Der Anfangsknoten einer Kante ist der Knoten des Subjekts, der Endknoten ist der Knoten des Objekts. Die Labeling-Funktion l_E ordnet einer Kante das Prädikat des Statements zu. Die Kanten des Graphen sind gegeben durch

$$E = \{e_{s,p,o} : (s,p,o) \in T \wedge \alpha(e_{s,p,o}) = v_s \wedge \omega(e_{s,p,o}) = v_o \wedge l_E(e_{s,p,o}) = p\}.$$

Das in [11] beschriebene Problem eines so definierten Graphen ist, dass die Mengen der Labels für Knoten und Kanten nicht disjunkt sind. Wie im vorhergehenden Kapitel beschrieben, kann dieses Problem jedoch für diese Arbeit vernachlässigt werden.

7.1.3 Prädikat Graph

Der zuvor definierte gerichtete Graph zur Repräsentation eines RDF- Graphen bildet die theoretische Grundlage des im Framework zur Repräsentation verwendeten Konzeptes. Zusätzlich kommt den im Kapitel 3.5 beschriebenen Eigenschaften der Algorithmen und Verfahren eine große Bedeutung zu. Hier sind vor allem zwei Punkte von Interesse. Zum einen, dass die einzelnen Algorithmen meist auf Teilgraphen aus wenigen Properties arbeiten und zum anderen, dass dieser Teilgraph in großem Umfang verarbeitet werden muss. Hieraus ist das Konzept entstanden, dass die Graph Sail-Komponente nicht nur einen Graphen, sondern immer einen Graphen je Property verwaltet. Ein solcher Graph der nur aus Statements mit gleichem Prädikat besteht, wird im Folgenden als Prädikat-Graph bezeichnet.

Ein gerichteter Prädikat-Graph $G_P = (V_p, E, \alpha, \omega)$, ist ein wie im vorhergehenden Kapitel definierter Graph. Die Labeling-Funktion l_v ist für alle Prädikat-Graphen identisch. Die Knoten V_p von G_P sind gegeben durch

$$V_p = \{v_x : x \in sub(T_p) \cup obj(T_p) \wedge l_v(v_x) = x\}.$$

Die Menge T_p enthält alle Statements mit dem Prädikat p eines RDF-Graphen. Für jede Kante $e_{s,p,o} \in E$ gilt somit das $l_E(e_{s,p,o}) = p$. Hierdurch wird jede Kante eindeutig durch das geordnete Knoten-Paar (v_s, v_o) von Subjekt und Objekt identifiziert. Dies bedeutet, dass in einem Prädikat-Graph keine parallelen Kanten auftreten können. Schlingen können dagegen vorkommen, da in RDF eine Ressource durch ein Prädikat mit sich selbst verbunden sein kann wodurch gilt das $\alpha(e_{s,p,o}) = \omega(e_{s,p,o})$.

Das bis hierher beschriebene Konzept der Prädikat-Graphen basiert ausschließlich auf dem Modell von RDF, in dem ein Statement durch ein Triple an RDF-Objekten repräsentiert wird. Anders ist es bei den nachfolgend vorgestellten Konzepten. Diese beziehen sich auf die Semantik, die im Schema formalisiert ist, oder durch den Benutzer vorgegeben wird. Hier werden symmetrische Properties und Properties, die Inverse einer anderen Property sind, gesondert behandelt.

Symmetrische Properties

Für eine Property P, die als symmetrisch gekennzeichnet ist, gilt $P(A,B) \leftrightarrow P(B,A)$, das heißt, aus einem Statement (A,P,B) kann das Statement (B,P,A) gefolgert werden. Für einen Prädikat-Graphen $G_P = (V, E, \alpha, \omega)$ einer symmetrischen Property muss folglich gelten wenn $e_{s,p,o} \in E$ auch die inverse $e_{s,p,o}^{-1} \in E$. Gilt dies nicht, so ist der Graph nicht semantisch konsistent. Um die semantische Konsistenz zu gewährleisten, müsste beim Löschen oder Hinzufügen von Kanten immer auch die inverse Kante gelöscht beziehungsweise hinzugefügt werden. Anstatt dies dem Benutzer zu überlassen oder die Kante intern automatisch zu löschen, werden Prädikat-Graphen von symmetrischen Properties als ungerichteter Graph $G = (V, E, \gamma)$ repräsentiert. Für diesen Graphen werden die Funktonen l_V, l_E und die Menge der Kanten analog zum gerichteten Graph Prädikat definiert. Den Unterschied macht die Definition der Kanten. Sie sind gegeben durch

$$E = \{e_{s,p,o} : (s,p,o) \in T \wedge \gamma(e_{s,p,o}) = \{v_s, v_o\} \wedge l_E(e_{s,p,o}) = p\}$$

wobei keine parallelen Kanten zugelassen werden. Schlingen sind jedoch auch hier erlaubt.

Konzeptionell kann aus dem ungerichteten Graph, durch Ersetzen jeder ungerichteten Kante durch jeweils zwei inverse gerichtete Kanten, wieder ein gerichteter Prädikat-Graph erstellt werden. In folgenden Kapiteln werden auch die ungerichteten Graphen als Prädikat-Graph bezeichnet.

Ein Beispiel für eine symmetrische Eigenschaft ist *skos:related* (siehe [12]). In Listing 7.1 werden zwei Konzepte über *skos:related* in Beziehung zueinander gesetzt. Wie der ungerichtete Prädikat-Graph für die Property *skos:related* dieser beiden Konzepte aussieht ist in Abbildung 7.1 dargestellt.

```
<rdf:Description rdf:about="http://example.org/Jahresabschluss">
       <skos:related rdf:resource="http://example.org/Lagebericht"/>
</rdf:Description>

<rdf:Description rdf:about="http://example.org/Lagebericht">
       <skos:related rdf:resource="http://example.org/Jahresabschluss"/>
</rdf:Description>
```

Listing 7.1: Zwei Konzepte, die über *skos:related* in Beziehung stehen

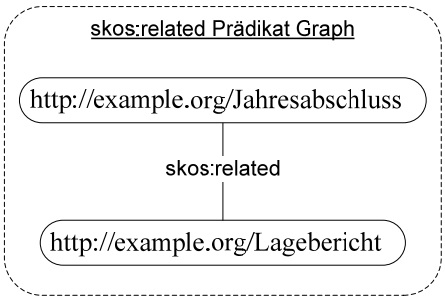

Abbildung 7.1: Ein Prädikat-Graph einer symmetrischen Property

Inverse Properties

Der zweite Typ Properties, der gesondert behandelt wird, sind Properties, die zu einer anderen Property inverse sind. Ist eine Property $P1$ inverse einer Property $P2$ gilt

$P1(A, B) \leftrightarrow P2(B, A)$ das heißt aus einem Statement $(A, P1, B)$ kann das Statement $(B, P2, A)$ gefolgert werden. Der Prädikat-Graph der Property $P1$ ist hierdurch der inverse Graph des Prädikat-Graphen für Property $P2$.

Wie bei symmetrischen Properties ergibt sich auch bei inversen Properties eine spezielle Semantik beim Löschen oder Hinzufügen von Kanten. Hier ist es jedoch nicht der Prädikat-Graph selbst, an dem eine zusätzliche Operation nötig ist, sondern der inverse Prädikat-Graph der inversen Property. Wird beispielsweise eine Kante aus einem solchen Prädikat-Graphen gelöscht, so muss, wenn die semantische Konsistenz erhalten bleiben soll, auch die inverse Kante des inversen Prädikat Graphen gelöscht werden. Das bedeutet, dass die beiden Prädikat-Graphen der inversen Properties durch die Semantik in Beziehung zueinander stehen. Um für diesen Fall die semantische Konsistenz zu wahren, werden die beiden Graphen intern so synchronisiert, dass sich Änderungen an einem der beiden direkt auf den anderen auswirken. Dieses Verhalten ist nicht so gut ersichtlich wie im Falle von symmetrischen Properties. Es wird jedoch in Kauf genommen, da inkonsistente semantisch verbundene Graphen zu Fehlern führen würden. Diese könnten, wenn überhaupt, nur sehr schwer gefunden werden.

Ein Beispiel für zwei zueinander inverse Properties sind *skos:narrower* und *skos:broader* (siehe [12]). In Listing 7.2 werden zwei Konzepte über *skos:narrower* und *skos:broader* in Beziehung zueinander gesetzt. Wie der Prädikat-Graph für diese beiden Konzepte aussieht ist in Abbildung 7.2 dargestellt.

```
<rdf:Description rdf:about="http://example.org/Buchführung">
       <skos:narrower rdf:resource="http://example.org/Lagebericht"/>
</rdf:Description>

<rdf:Description rdf:about="http://example.org/Lagebericht">
       <skos:broader rdf:resource="http://example.org/Buchführung"/>
</rdf:Description>
```

Listing 7.2: Zwei Konzepte die über *skos:narrower* und *skos:broader* in Beziehung stehen

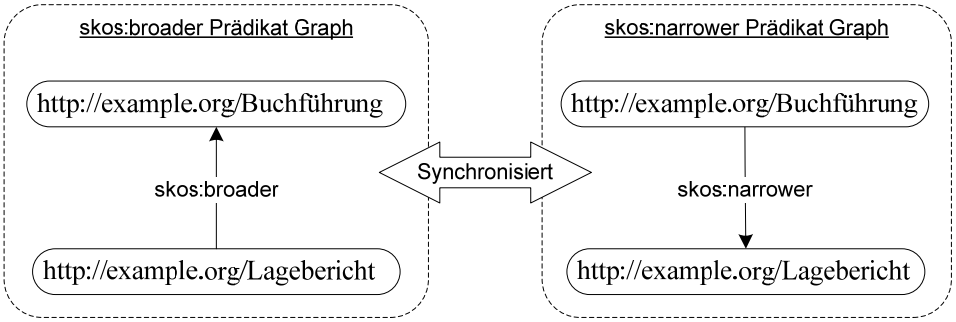

Abbildung 7.2: Zwei synchronisierte Prädikat-Graphen von zwei inversen Properties

Beim Aufbau der Prädikat-Graphen durch das Framework spielt es keine Rolle, ob für ein Statement mit symmetrischem oder inversem Prädikat das zugehörige zweite Statement explizit im Repository vorhanden ist. Sie werden für die entsprechenden Prädikat-Graphen automatisch gefolgert.

Es können jedoch auch Fälle vorkommen, in denen es nicht genügt, einen Prädikat-Graph für sich allein zu verarbeiten. Um diese Fälle zu berücksichtigen, können mehrere Prädikat-Graphen zu einem Graphen vereinigt werden. Der entstehende Graph ist ein sogenannter Mixed Graph der gerichtete und ungerichtete Kanten enthalten kann. Dieser Graph kann dann wie ein einzelner Graph verarbeitet werden. Auf die formelle Definition dieses Graphen sei an dieser Stelle verzichtet. Wichtig zu erwähnen ist jedoch, dass eine gerichtete Kante genau ein Statement und eine ungerichtete Kante zwei inverse Statements repräsentiert.

7.2 Graph-Komponente

Die Aufgaben, die an die Graph-Komponente gestellt werden, decken sich weitgehend mit denen eines Graphen-Frameworks wie zum Beispiel Jung oder JGraphT. Weder die für die Repräsentation von RDF eingesetzten Graphen (siehe Kapitel 7.1.3) noch die benötigten Algorithmen beziehen sich direkt auf RDF. Aus diesem Grund wurde das Graphen-Framework ohne direkten Bezug zu RDF und Sesame entwickelt und enthält somit keine Abhängigkeiten zur Graph Sail-Komponente. Bei vielen Designentscheidungen ist der spezielle Verwendungszweck jedoch berücksichtigt. Bei der Entwicklung der Komponente hat dies den Vorteil eines klar abgegrenzten Aufgabenbereichs. Im Ergebnis stellt die Graph-Komponente allgemeine Funktionalität

zur Verarbeitung von Graphen bereit und kann so prinzipiell auch unabhängig vom restlichen Framework für andere Zwecke eingesetzt werden.

7.2.1 Trennung von Struktur und Daten

Eine Eigenschaft der Verarbeitung semantischer Netze durch Graph-Algorithmen ist, dass hauptsächlich die abstrakte Struktur des RDF-Graphen von Interesse ist. Hiervon ausgehend wird zur Lösung der Speicherproblematik die abstrakte Struktur des Graphen von den durch die Knoten und Kanten repräsentierten Objekten getrennt. Dabei ist es nötig, die Knoten und Kanten des Graphen durch eindeutige Identifikatoren zu identifizieren, um sie den durch sie repräsentierten Objekten zuordnen zu können. Durch dieses Verfahren muss nur noch die Graphstruktur im Speicher gehalten werden.

Um Flexibilität und komfortables Arbeiten mit der Graph API zu ermöglichen, müssen auch die hinter der Graphstruktur stehenden Objekte durch die Methoden der Graph API zur Verfügung gestellt werden. Hierfür wird ein Mapping-Mechanismus genutzt. Dieser bietet Methoden für das Mapping von Objekten zu den Identifikatoren der Knoten und Kanten und umgekehrt. Die Objekte können hierdurch auch "schwergewichtigere" Datencontainer darstellen, da sie erst beim Zugriff in den Speicher geladen werden müssen (lazy-loading) und, wenn nicht mehr benötigt, wieder frei gegeben werden können.

Die beschriebene Mapping-Funktionalität wird an die Stellen der API, an denen ein Zugriff auf Knoten und Kanten erfolgt, weitergegeben. Hierdurch hat der Entwickler die freie Entscheidung entweder die Identifikatoren der Knoten und Kanten oder die dahinterliegenden Objekte zu nutzen. Das ermöglicht die schnelle Verarbeitung der Graphstruktur, wobei der Komfort mit Objekten zu arbeiten erhalten bleibt. Ein Nachteil ist jedoch, dass sich die Anzahl an Methoden in der Graph API deutlich erhöht.

7.2.2 Traversierung

Die Graph-Komponente ist verantwortlich für die algorithmische Verarbeitung der durch die semantischen Netze definierten Graphen. Aufgrund der Annahme, dass eine Traversierung elementarer Bestandteil vieler Algorithmen und Verfahren darstellt, ist sie eine Kernfunktionalität des Frameworks. Im Folgenden werden die verschiedenen

Möglichkeiten zur Traversierung des Graphen, die durch das Framework geboten werden, vorgestellt. Weitere Details zu den beschriebenen Verfahren finden sich in [5].

Breitensuche

Eine der am häufigsten genutzten Traversierungsarten ist die Breitensuche. Bei dieser werden von einem Startknoten zunächst alle Knoten mit der Tiefe 1, also alle benachbarten Knoten, besucht. Sind diese abgearbeitet, werden alle noch nicht besuchten Knoten mit Tiefe 2 besucht. Dies wird für jede Tiefe fortgeführt, wodurch ein Baum aufgespannt wird, der jeden erreichbaren Knoten enthält.

Wird ein Knoten $u \in V$ zum ersten Mal durch eine Breitensuche besucht, ist der Pfad, den die Traversierung vom Startknoten $s \in V$ zu u Knoten genommen hat, der

kürzeste Pfad zwischen s und u. Hierdurch können über die Breitensuche Pfad-Abfragen, wie sie im Kapitel 3.4 beschrieben sind, einfach realisiert werden.

Tiefensuche

Neben der Breitensuche ist die Tiefensuche eine weitere sehr gebräuchliche Art der Traversierung. Hier wird zunächst so weit wie möglich in die Tiefe gesucht. Können keine tiefer liegenden Knoten besucht werden, wird der Weg schrittweise zurückgegangen, bis ein Knoten gefunden wird, der noch nicht besucht wurde. Dies wird als backtracking bezeichnet. Von diesem Knoten wird wiederum in die Tiefe gesucht. Wie bei der Breitensuche wird auch bei der Tiefensuche ein Baum aufgespannt, der alle erreichbaren Knoten enthält.

Das Framework bietet bei der Tiefensuche die Möglichkeit, die Knoten in unterschiedlichen Reihenfolgen zurückzugeben. Bei der Preorder-Reihenfolge werden die Knoten zurückgegeben, wenn sie zum ersten Mal durch die Tiefensuche gefunden werden. Anders bei der Postorder-Reihenfolge, hier wird ein Knoten erst zurückgegeben, wenn alle "tiefer" liegenden adjazenten Knoten besucht wurden.

Neben der einfachen Traversierung bietet das Framework zudem die Möglichkeit, die Kanten eines Graphen in Verbindung mit einer Tiefensuche zu klassifizieren. Hierfür

werden die Knoten bei der Tiefensuche farblich gekennzeichnet. Zu Beginn sind alle Knoten weiß. Wird ein Knoten zum ersten Mal durch die Tiefensuche gefunden, wird er grau. Schlussendlich wird ein Knoten schwarz, wenn alle "tiefer" liegenden adjazenten Knoten besucht wurden. Über diese Farbmarkierung werden die Kanten in Bezug auf den aufgespannten Baum wie folgt klassifiziert:

Wird bei einer Tiefensuche auf einem ungerichteten Graph $G = (V, E, \gamma)$ ausgehend von dem Knoten $u \in V$ über eine Kante $e \in E$ der Knoten $v \in V$ besucht, so ist die Kante:

- eine Baumkante wenn v weiß ist.
- eine Rückwärtskante wenn v grau oder schwarz ist.

Weitgehend analog werden die Kanten eines gerichteten Graphen klassifiziert. Wird ein gerichteter Graph $G = (V, E, \alpha, \omega)$ ausgehend von dem Knoten $u = \alpha(e)$ über eine Kante $e \in E$ der Knoten $\omega(e)$ besucht, so ist die Kante:

- eine *Baumkante,* wenn v weiß ist.
- eine *Rückwärtskante,* wenn v grau ist
- eine *Vorwärtskante* oder *Querkante,* wenn v schwarz ist.

Die wohl nützlichste Information, die diese Klassifikation liefert, ist ob ein Graph kreisfrei ist. Dies ist nämlich genau dann der Fall, wenn keine Kante als *Rückwärtskante* klassifiziert wurde. Wird eine *Rückwärtskante* gefunden, so können über den zur Verfügung stehenden Pfad die Kanten, die den Kreis bilden, ermittelt werden.

8 Implementierungen

Dieses Kapitel widmet sich den Implementierungsdetails, der wichtigen Komponenten und Verfahren des Frameworks.

8.1 Verwendete Hilfsmittel

Das Framework verwendet folgende Bibliotheken:

- Sesame[19] (Lizenz: BSD ähnlichen)
- Simple Logging Facade for Java (SLF4J)[20] (Lizenz: MIT)
- fastutil[21] (Lizenz: GNU Lesser General Public License)
- Apache log4j[22] (Lizenz: Apache License Version 2.0)
- colt[23] (Lizenz: Eigene und GNU Lesser General Public License)

Für wichtige Klassen wurden Unit Tests implementiert. Sie verwenden:

- JUnit[24] (Lizenz: Common Public License)

8.2 Graph-Komponente

8.2.1 Graph Objekt Mapping

Um die Trennung von Graphstruktur und Daten zu erreichen wird, wie im Kapitel 7.2.1 beschrieben, ein Mapping Mechanismus zwischen den Knoten und Kanten der Graphstruktur und den durch sie referenzierten Objekten eingesetzt. Die hierfür notwendigen Identifikatoren sind so ausgelegt, dass sie für alle Graphen eindeutig sind. Als Identifikator eines Knotens dient eine eindeutige Integer ID. Wird eine solche ID in zwei Graphstrukturen als Knoten verwendet, referenzieren beide Knoten auf dasselbe Objekt. Kanten werden eindeutig durch die IDs ihrer inzidenten Knoten und ein zusätzliches Label identifiziert. Hierdurch ist es möglich, parallele Kanten zwischen

[19] http://www.openrdf.org/

[20] http://www.slf4j.org/

[21] http://fastutil.dsi.unimi.it/

[22] http://logging.apache.org/log4j/

[23] http://acs.lbl.gov/software/colt/

[24] http://junit.org/

zwei Knoten zu repräsentieren. Als Vorgriff auf das Kapitel 8.2.3, das die Graph Implementierungen beschreibt, sei hier erwähnt, dass ein einzelner Graph immer für ein Label zuständig ist. Er enthält nur Kanten mit demselben Label.

Zentrales Interface für das Mapping ist das *GraphObjectMapping***.** Dieses Interface stellt das für alle Graphen einheitliche Mapping zur Verfügung. Durch das *GraphObjectMapping* wird jedem Knoten Objekt V_{Obj} die entsprechende Integer ID und jedem Kanten Objekt E_{Obj} das die für gerichtete Kanten geordnete Paar IDs sowie das Label zugeordnet. Ein Label ist dabei ein beliebiges Objekt, dass das Marker Interface *EdgeLabel* implementiert. Zur eindeutigen Referenzierung einer Kante in der API wird vornehmlich das Interface *EdgeIdentifier* verwendet, welches Zugriff auf die IDs und das *EdgeLabel* der Kante bietet. Zusammengefasst ermöglicht das *GraphObjectMapping* das Mapping von:

- $E_{Obj} \rightarrow EdgeLabel$
- $E_{Obj} \rightarrow (ID, ID)$
- $V_{Obj} \rightarrow ID$
- $ID \rightarrow V_{Obj}$

Das *GraphObjectMapping* Interface beinhaltet keine Methoden, um auf die Kanten Objekte zuzugreifen. Stattdessen stellt es für jedes *EdgeLabel* eine eigene Mapping Klasse, das *LabeledEdgeMapping*, bereit. Da ein *LabeledEdgeMapping* immer nur für ein *EdgeLabel* zuständig ist, müssen keine parallelen Kanten berücksichtigt werden und es kann auf ein Kanten Objekt direkt über die IDs zugegriffen werden. Das *LabeledEdgeMapping* ermöglicht somit das Mapping:

- $(ID, ID) \rightarrow E_{Obj}$

Die Mapping Interfaces lassen in vielerlei Hinsicht offen, wie das Mapping konkret auszusehen hat. So ist beispielsweise nicht festgelegt, wo und wie die Objekte für Knoten und Kanten erzeugt werden. Das Mapping kann die Objekte selbst erzeugen, als Adapter zu einem externen System fungieren oder vorsehen, dass die Objekte vom Benutzer erzeugt und dem Graph hinzugefügt werden müssen. Außer einem einfachen Test Mapping, das Strings für Knoten und Kanten erzeugt, sind keine weiteren Implementierungen in der Graph-Komponente vorhanden.

8.2.2 Graphstruktur

Die Datenstruktur, die zur Repräsentation der Graphen im Speicher verwendet wird, bestimmt im Wesentlichen die Performance sowie den Speicherverbrauch. Es kann davon ausgegangen werden, dass in den meisten Anwendungsfällen, für die das Framework eingesetzt werden soll, wenig Änderungen an der Graphstruktur durchgeführt werden. Aus diesem Grund ist die Performance für lesenden Zugriff optimiert. Eine Besonderheit der Datenstruktur ist, dass keine isolierten Knoten gespeichert werden müssen. Der Grund hierfür ist, dass im Hinblick auf RDF der Graph ausschließlich aus Statements, also Kanten, gebildet wird. *Values,* die in keinem Statement vorkommen, sind nicht relevant.

Die Datenstruktur, die als Basis der konkreten Graphen verwendet wird, ähnelt der klassischen Adjazenzlisten-Repräsentation (siehe [5]). Anstelle von Objekten werden primitive IDs zur Repräsentation eines Knoten verwendet. Wie bei der Adjazenzliste wird für die Knoten die Menge der adjazenten Knoten gespeichert. Im Gegensatz zur klassischen Adjazenzliste wird jedoch keine (verkettete) Liste, sondern ein primitives Integer Array, im Folgenden als Adjazenzarrays bezeichnet, zum Speichern der adjazenten Knoten IDs verwendet. Der Zugriff auf die Adjazenzarrays der Knoten wird durch eine Map $adjMap$ ermöglicht. Um den Overhead von Key Objekt für die $adjMap$ Map zu vermeiden, wird sie über eine Datenstruktur aus der *fastutil* Bibliothek realisiert, welche primitive Integer als Keys auf Value Objekte abbildet. Um die Suche nach einer bestimmten Kante zu beschleunigen, werden die IDs der Adjazenzarrays sortiert und eine binäre Suche verwendet.

Die beschriebene Datenstruktur wird durch die abstrakte Klasse *AbstractGraphStructure* implementiert. Von dieser Klasse sind drei verschiedene *GraphStructure* Implementierungen abgeleitet. Nachteil bei dieser Datenstruktur ist, dass das Löschen und Einfügen in die Adjazenzarrays im Vergleich relativ teuer ist, da sie sich nicht, wie zum Beispiel verkettete Listen, dynamisch vergrößern oder verkleinern lassen.

DirectedGraphStructure

Die *DirectedGraphStructure* repräsentiert einen gerichteten Graphen, der weder parallele Kanten noch isolierte Knoten, jedoch Schlingen enthalten kann. Sei $G = (V, E, \alpha, \omega)$ ein solcher Graph und sei weiter V^+ die Menge aller Knoten aus V, die mindestens einen Nachfolger besitzen, dann gilt:

$V^+ := \{\mathrm{v} \in V : g^+(\mathrm{v}) > 0\}$

Für die Repräsentation von G werden die Nachfolger $N^+(v)$ eines Knotens $v \in V^+$ als Map-Eintrag $adjMap(v)$ gespeichert. Insgesamt werden hier $|V^+|$ Map-Einträge und Adjazenzarrays benötigt. Die Gesamtzahl der nötigen Adjazenzarray Einträge ist $|E|$.

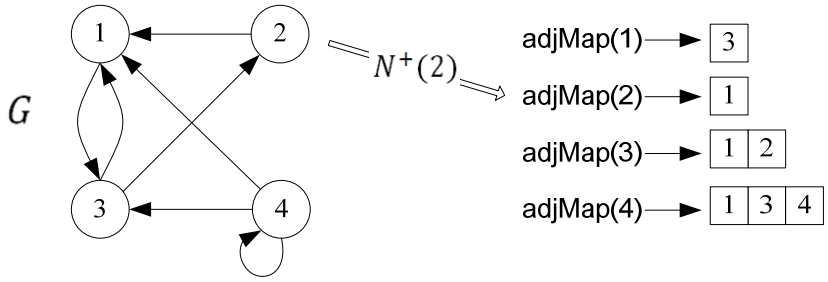

Abbildung 8.1: Konzept der *DirectedGraphStructure*

UndirectedGraphStructure

Die *UndirectedGraphStructure* repräsentiert einen ungerichteten Graph $G = (V, E, \gamma)$ der wiederum keine parallelen Kanten und isolierte Knoten, jedoch Schlingen enthalten kann. Sei weiter V' die Menge aller Knoten aus V, die mindestens einen Nachbarn besitzen, dann gilt:

$V' := \{\mathrm{v} \in V : g(\mathrm{v}) > 0\}$.

Durch die *UndirectedGraphStructure* werden die Nachbarn $N(v)$ eines Knotens $v \in V'$ als Map-Eintrag $adjMap(v)$ gespeichert. Da jeder Endknoten einer Kante jeweils Nachbarn des anderen ist, muss jede Kante doppelt gespeichert werden. Benötigt werden somit $|V'|$ Map-Einträge und Adjazenzarrays und $2|E|$ Adjazenzarray Einträge.

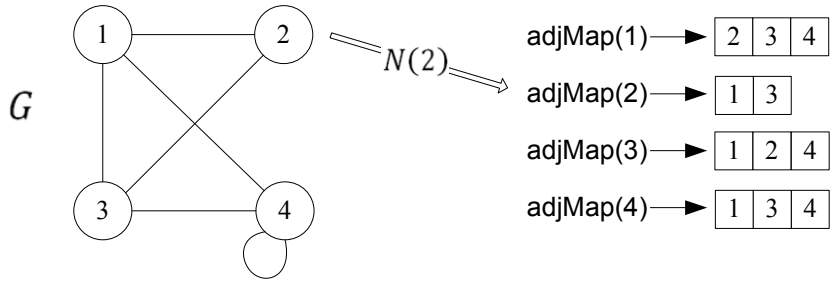

Abbildung 8.2: Konzept der *UndirectedGraphStructure*

BidirectionalGraphStructure

Die *BidirectionalGraphStructure* repräsentiert einen zur *DirectedGraphStructure* analogen gerichteten Graphen $G = (V, E, \alpha, \omega)$, jedoch mit schnellem Zugriff auf die Vorgänger eines Knotens. Der Grund für die zwei unterschiedlichen Implementierungen für gerichtete Graphen ist, dass viele Algorithmen, die nicht auf die Vorgänger eines Knotens zugreifen müssen. In diesem Fall kann eine *DirectedGraphStructure* verwendet werden, die nur die Nachfolger der Knoten explizit speichert.

Für die Realisierung besteht eine *BidirectionalGraphStructure* immer aus einem Paar sich gegenseitig referenzierender *BidirectionalGraphStructure* Instanzen. Zur Erläuterung dieses Konzeptes sprechen wir hier von einer *BidirectionalGraphStructure* und einer referenzierten *BidirectionalGraphStructure*.

Repräsentiert eine *BidirectionalGraphStructure* den Graph G, dann repräsentiert die referenzierte Instanz den zu G inversen Graph GI. Sei $adjMap$ die Map der *BidirectionalGraphStructure* für G und $adjMap^{-1}$ die Map der referenzierten *BidirectionalGraphStructure* für GI. Da die Graphen inverse sind ist die Menge der Knoten V beider Graphen identisch. Hier ist V_G^+ die Menge aller Knoten aus V, die mindestens einen Nachfolger in G besitzen. Unter diesen Voraussetzungen gilt für einen Knoten $v \in V_G^+$ das:

$$adjMap(v) = N_G^+(v)$$

$$adjMap^{-1}(v) = N_G^-(v)$$

Hierbei sind $N_G^+(v)$ und $N_G^-(v)$ die Nachfolger beziehungsweise Vorgänger von v in G. Um direkt auf die Vorgänger eines Knotens für G zuzugreifen, kann somit die referenzierte inverse *BidirectionalGraphStructure* genutzt werden. Dasselbe gilt auch umgekehrt für die referenzierte inverse *BidirectionalGraphStructure,* Abbildung 8.3 verdeutlicht dies.

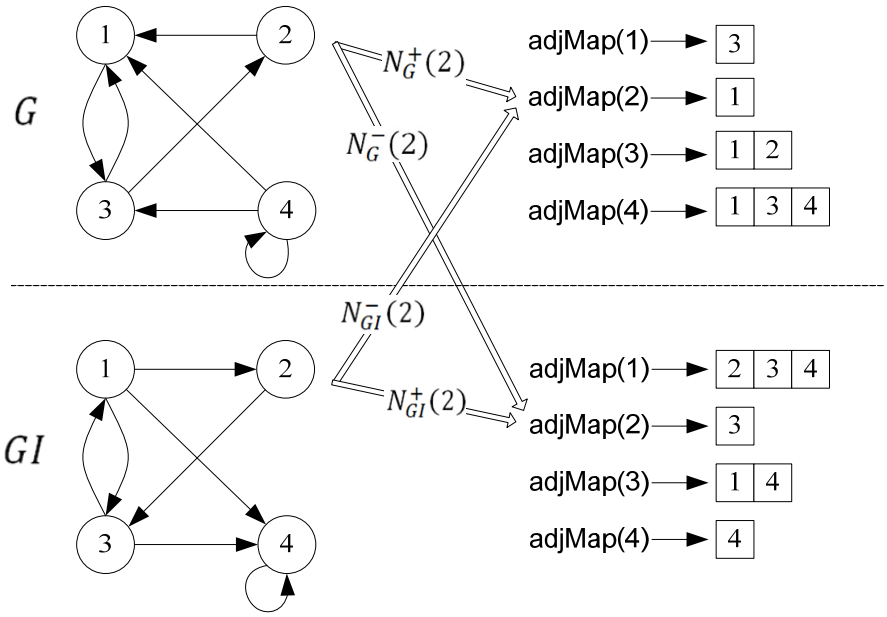

Abbildung 8.3: Konzept der *BidirectionalGraphStructure*

Beim Hinzufügen einer Kante $e \in E$ wird die Kante der *BidirectionalGraphStructure* und die zu e inverse Kante e^{-1} der inversen *BidirectionalGraphStructure* hinzugefügt.

Um einen schnellen Zugriff auf die Vorgänger zu ermöglichen, hätte ebenso eine einzelne *BidirectionalGraphStructure* mit zwei internen Maps verwendet werden können. Das Konzept zweier gegenseitig referenzierenden *BidirectionalGraphStructure* Instanzen wurde im Hinblick auf den im Kapitel 7.1.3 beschriebenen Prädikat-Graph für inverse Properties gewählt. Die beiden inversen Prädikat-Graphen lassen sich durch das Paar *BidirectionalGraphStructure* Instanzen repräsentieren. Dabei ist der Speicherverbrauch äquivalent zur einzelnen Repräsentation der Graphen durch zwei *DirectedGraphStructure* Instanzen. Er ergibt sich aus $|V_G^+|+|V_{GI}^+|$ Map-Einträgen und Adjazenzarrays sowie $2|E|$ nötigen Adjazenzarray Einträgen.

8.2.3 Graphen

Die eigentliche Verarbeitung der Graphen geschieht nicht über die *GraphStructure* Implementierungen, sondern über das *Graph* Interface. Wie bei den evaluierten Graph Frameworks aus Kapitel 6.2 wird eine Typisierung der Graphen durch das generische *Graph* Interface mit zwei Typ-Parametern für Knoten und Kanten erreicht.

Das *Graph* Interface ist so implementiert, dass es einheitlichen Zugriff auf alle durch das Framework darstellbaren Graph Typen ermöglicht. Es kann damit als einheitliches Interface für den Zugriff verwendet werden.

Ein vom *Graph* abgeleitetes Interface ist das *SingleGraph*. Ein *SingleGraph* kann sowohl gerichtet als auch ungerichtet sein. Er repräsentiert einen Graphen, in dem alle Kanten dasselbe *EdgeLabel* besitzen, wodurch parallele Kanten verboten werden. Das *SingleGraph* Interface wird von der Klasse *SingleGraphImpl* implementiert. Sie realisiert typisierte Graphen durch die Kombination des Objekt Mapping mit einer *GraphStructure*.

Beim Erstellen eines *SingleGraphImpl* wird das *EdgeLabel* seiner Kanten sowie das *GraphObjectMapping* übergeben. Über das *GraphObjectMapping* hat der Graph Zugriff auf das für sein *EdgeLabel* zuständige *LabeledEdgeMapping*. Zusätzlich kann optional eine existierende *GraphStructure* im Konstruktor übergeben werden.

Der *MultiGraph* ist ein ebenfalls von *Graph* abgeleitetes Marker Interface für einen Graphen, der aus mehreren *SingleGraph* Instanzen zusammengesetzt ist. Die Implementierung des *MultiGraph* Interfaces ist die *MultiGraphImpl* Klasse. Über sie lässt sich eine Liste von *SingleGraph* Instanzen zu einem Graphen vereinigen. Der durch die Vereinigung entstehende Graph enthält alle Knoten und Kanten der einzelnen Graphen. Somit sind hier auch parallele Kanten erlaubt, die jedoch durch die verschiedenen *EdgeLabel* unterschieden werden können. Das in Abbildung 8.4 dargestellte UML-Klassendiagramm zeigt die Graph Interfaces und Klassen mit den zur Verfügung stehenden Methoden zum Zugriff und für Änderungen.

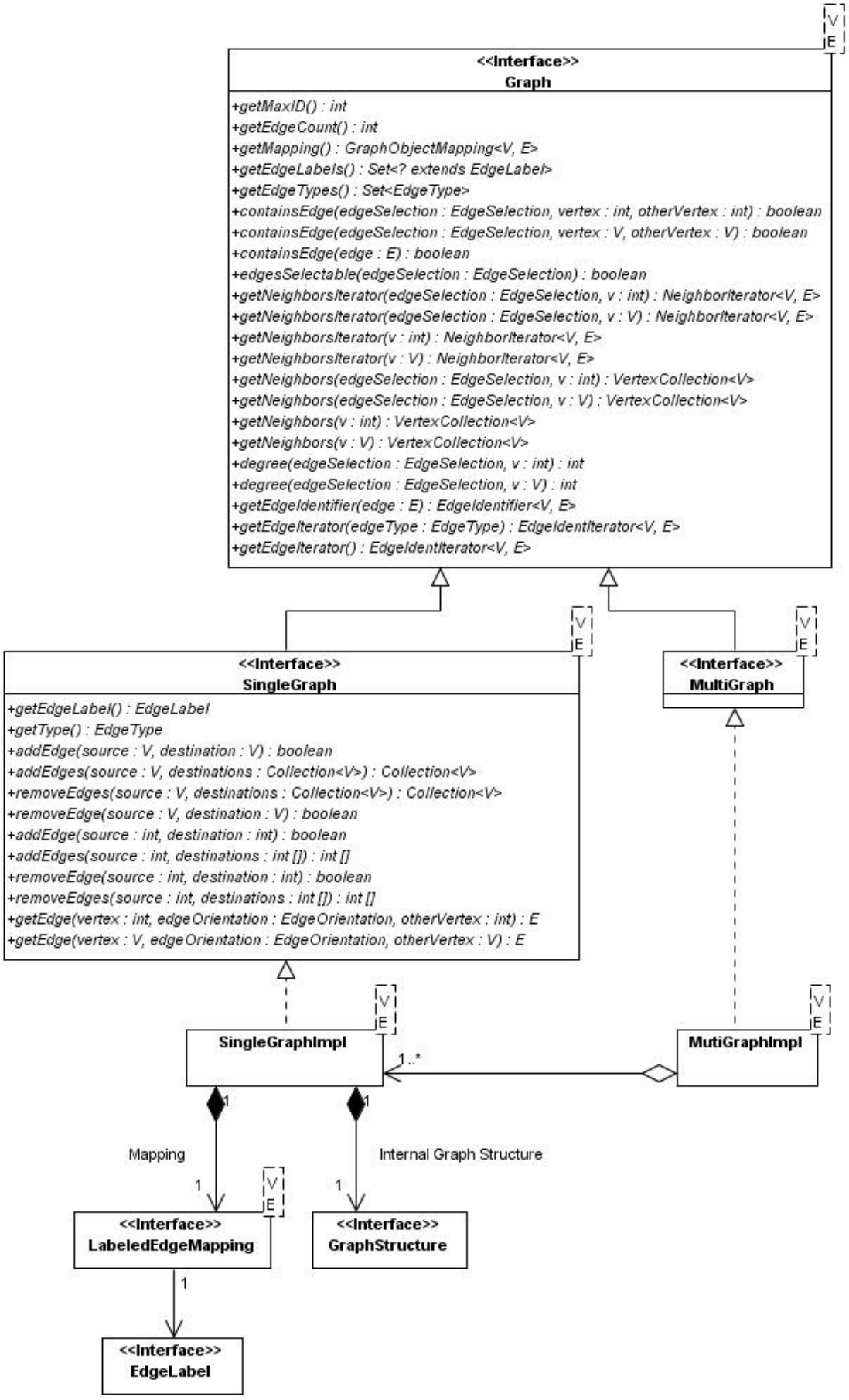

Abbildung 8.4: UML-Klassendiagramm der Graph Interfaces und Klassen

8.2.4 Selektion von Kanten

Wie im vorhergehenden Kapitel erläutert, können durch Instanzen des Interfaces *Graph* unterschiedliche Typen (gerichtet/ungerichtet) von Graphen repräsentiert werden. Beim Zugriff muss es jedoch möglich sein, nur auf bestimmte Typen von Kanten zuzugreifen. Des Weiteren muss es für gerichtete Graphen möglich sein, die Richtung der Kanten mit einzubeziehen, beispielsweise wenn alle Vorgänger eines Knotens benötigt werden.

Um dies zu realisieren wird die Klasse *EdgeSelection* verwendet. Über sie können die Kanten, die bei einem Methodenaufruf beachtet werden sollen, anhand ihrer Orientierung, ihres Typs und ihres Labels eingeschränkt werden. Die Orientierung und der Typ einer Kante werden durch das enum *EdgeOrientation* angegeben. Dieses enum enthält die Konstanten:

OUTGOING, INCOMING, UNDIRECTED

Die *EdgeOrientation* einer Kante ist immer von einem Bezugsknoten aus zu betrachten. Sei G ein Graph mit der Knotenmenge V und der Kantenmenge E. Weiter sei $v \in V$ der Bezugsknoten und $u \in V$ ein zu v adjazenter Knoten, dann ist die Orientierung der Kante $e \in E$ die u, v verbindet:

- *OUTGOING* wenn e eine gerichtete Kante von v nach u ist.
- *INCOMING* wenn e eine gerichtete Kante von u nach v ist.
- *UNDIRECTED* wenn e keine gerichtete Kante ist.

Über die *EdgeOrientation* ist es so möglich, zwischen den verschiedenen Typen von Kanten, die inzident zu einem Knoten sind, zu differenzieren. In der Klasse *EdgeSelection* lassen sich beliebige Kombinationen der *EdgeOrientation* Konstanten spezifizieren. Die Möglichkeit, zusätzlich zur Orientierung auch noch eine Liste von erlaubten *EdgeLabel* anzugeben, ermöglicht es in einem *MultiGraph* die entsprechenden internen *SingleGraph* Instanzen zu selektieren. Zusammengefasst bietet die *EdgeSelection* die Möglichkeit, Kanten durch beliebige Kombination der folgenden drei Eigenschaften zu selektieren:

- Typ der Kante (gerichtet/ungerichtet)
- Orientierung der Kante, ausgehend von einem Bezugsknoten
- Label der Kante

8.2.5 Iteratoren

Neben den *Graph* Methoden, die Java Collections von Graph-Objekten liefern, existieren zum Zugriff auf den Graph verschiedene Iteratoren. Hier muss zunächst darauf hingewiesen werden, dass das bekannte Iterator Pattern zwar Vorlage für die Iteratoren des Framework war, diese sich jedoch in Teilen nicht an das Standard Pattern halten. Grund für diese Abweichungen ist, dass die Iteratoren sehr stark auf Performance ausgelegt sind und so zum Beispiel das Erzeugen von Objekten vermieden werden sollte.

Die Grundidee der Iteratoren ist, dass bei jeder Iteration ein "Schritt" innerhalb des Graphen durchgeführt wird. Jeder Schritt erfolgt dabei von einem Knoten über eine Kante zu einem Knoten. Solange über eine gerichtete Kante nicht gegen ihre Richtung geschritten wird, entspricht ein Schritt einem Weg der Länge eins. Zur Darstellung eines solchen Schrittes wird das Interface *EdgeIdentifier* verwendet. Die Methode *getFromVertexID()* liefert die ID des Knotens, von dem der Schritt erfolgt ist. Die Methode *getEdgeOrientation()* liefert Typ und Orientierung der Kante, über die zu dem Knoten mit der ID *getVertexID()* geschritten wurde.

Der wichtigste Iterator zum direkten Zugriff auf den Graphen ist der *NeighborIterator.* Dieser kann durch das *Graph* Interface erstellt werden. Er iteriert über die benachbarten Knoten eines angegeben Bezugsknotens. Die Methode *getVertexID()* des *EdgeIdentifier* liefert die ID des benachbarten Knotens und *getEdgeOrientation()* in Verbindung mit *getEdgeLabel()* die eindeutige Kante zu diesem Knoten. Um die Kanten einzuschränken, kann bei der Erstellung des *NeighborIterator* ein *EdgeSelection* Objekt übergeben werden. Für den Fall das wiederholt benachbarte Knoten verschiedener Bezugsknoten mit derselben *EdgeSelection* benötigt werden bietet der *NeighborIterator* über die Methode *initVertex()* die Möglichkeit den Iterator auf einen neuen Bezugsknoten zu setzen. Das in Abbildung 8.5 dargestellte UML-Klassendiagramm zeigt die Iterator Interfaces.

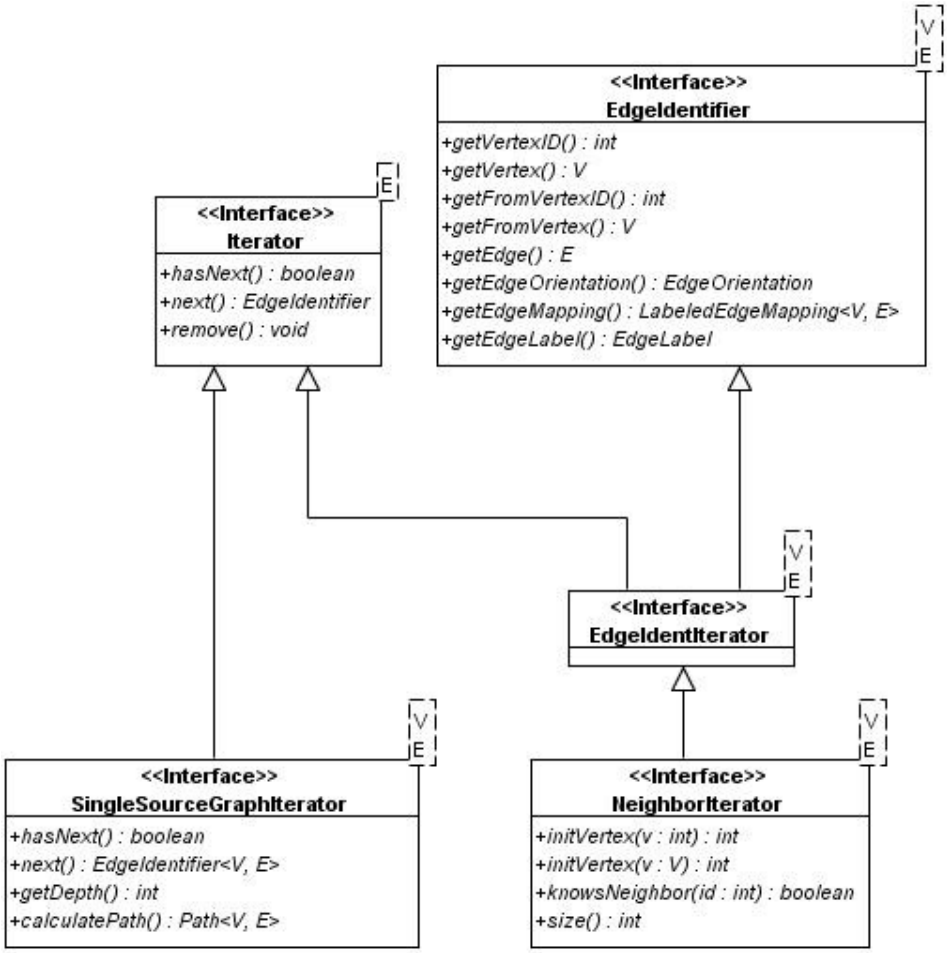

Abbildung 8.5: UML-Klassendiagramm der Iterator Interfaces

8.2.6 Traversierung

Der im vorhergehenden Kapitel vorgestellte *NeighborIterator* entfernt sich von seinem Bezugsknoten immer genau einen Schritt. Weiter entfernte Knoten können nur durch Setzen eines neuen Bezugsknotens erreicht werden. Anders bei der Traversierung. Hier wird der Graph in einer systematischen Reihenfolge durchlaufen. Die Graph-Komponente stellt vier unterschiedlich arbeitende Iteratoren bereit, die eine Traversierung des Graphen ermöglichen. Diese implementieren das Interface *SingleSourceGraphIterator,* das wiederum den Java Iterator erweitert und über *EdgeIdentifier* Objekte iteriert. Neben den Iterator Methoden sind hier vor allem die Methoden *calculatePath()* und *getDepth()* von Interesse, sie liefern den Weg und die Länge des Weges vom Startknoten zum aktuellen Knoten. Der Startknoten wird im Konstruktor der abstrakten Implementierung *AbstractSSGraphIterator* übergeben. Neben dem Startknoten erwartet der Konstruktor eine *EdgeSelection* und optional einen

GraphIterationFilter. Der *GraphIterationFilter* ermöglicht es, die Kanten, die bei der Traversierung benutzt werden, zu filtern. Hierbei ist zu beachten, dass die Iteratoren mit Ausnahme des *EdgeClassificationIterator* standardmäßig einen *SingleUseFilter* verwenden, wodurch jeder Knoten nur einmal besucht wird. Für den Fall, dass ein *GraphIterationFilter* im Konstruktor angegeben wird, wird dieser Standardfilter ersetzt und es obliegt dem Benutzer, das gewünschte Filterverhalten zu definieren.

Durch die *EdgeSelection* kann erreicht werden, dass die Iteratoren gerichtete Kanten gegen ihre Richtung durchlaufen. Dies führt dazu, dass die gelieferten *EdgeIdentifier* sowie *calculatePath()* keine formell der Definition eines Weges entsprechenden Knoten/Kanten-Folgen liefern.

Wie im Kapitel 7.2.2 beschrieben, ermöglicht das Framework sowohl eine Breitensuche als auch eine Tiefensuche. Die Breitensuche wird durch den *BreadthFirstIterator* ermöglicht. Die restlichen *SingleSourceGraphIterator* Klassen arbeiten nach dem Prinzip der Tiefensuche. Der *PreOrderDepthFirstIterator* gibt die Knoten in Preorder und der *PostOrderDepthFirstIterator* in Postorder zurück. Der letzte Iterator ist der *EdgeClassificationIterator.* Er nimmt insoweit eine Sonderstellung ein, als dass er einerseits die Knoten sowohl in Preorder als auch in Postorder zurückgibt. Andererseits werden zusätzlich die in Postorder zurückgegebenen *EdgeIdentifier* durch eine von vier Konstanten des enum *EdgeClass* klassifiziert. Die enum Konstanten der *EdgeClass* haben dabei folgende Bedeutung:

- *EdgeClass.TREE* = Baumkante
- *EdgeClass.BACK* = Rückwärtskante
- *EdgeClass.FORWARD_OR_CROSS* = Vorwärtskante oder Querkante
- *EdgeClass.NOT_TREE* = Vorwärtskante, Rückwärtskante oder Querkante

8.3 Graph Sail-Komponente

8.3.1 Sesame Sail API

Zum Zugriff auf RDF verwendet Sesame zwei Abstraktionsschichten (siehe [10]). Die Repository API ist für den High-Level Zugriff auf das Repository zuständig und der zentrale Zugriffspunkt des Entwicklers. Über sie können zum Beispiel SPARQL Abfragen an das Repository gestellt werden. Die zweite Schicht ist der Storage and

Inference Layer (Sail). Die Sail API abstrahiert die Details zum Zugriff auf das verwendete Speichermedium, so dass unterschiedliche Persistenz-Strategien zum Einsatz kommen können (siehe [13]). Sesame bietet über das Interface *StackableSail* die Möglichkeit *Sail* Objekte zu stapeln, wodurch ein sogenannter Sail-Stack entsteht. Methodenaufrufe werden im Sail-Stack von unten nach oben und von oben nach unten weitergereicht. Das *SailRepository* realisiert die Repository API und verwendet dafür das oberste *Sail* des Sail-Stack. Durch die Implementierungen eines *StackableSail* besteht hierdurch die Möglichkeit, das Verhalten der Repository API durch eine transparente Zwischenschicht vielfältig anzupassen.

Für die Realisierung des Frameworks wurde ein *StackableSail* durch die Klassen *GraphSail* implementiert. Sie stellt das Hauptverbindungsglied zwischen Sesame und dem Framework dar. Die Abbildung 8.6 zeigt die Eingliederung des Frameworks in eine auf Sesame basierende Applikation.

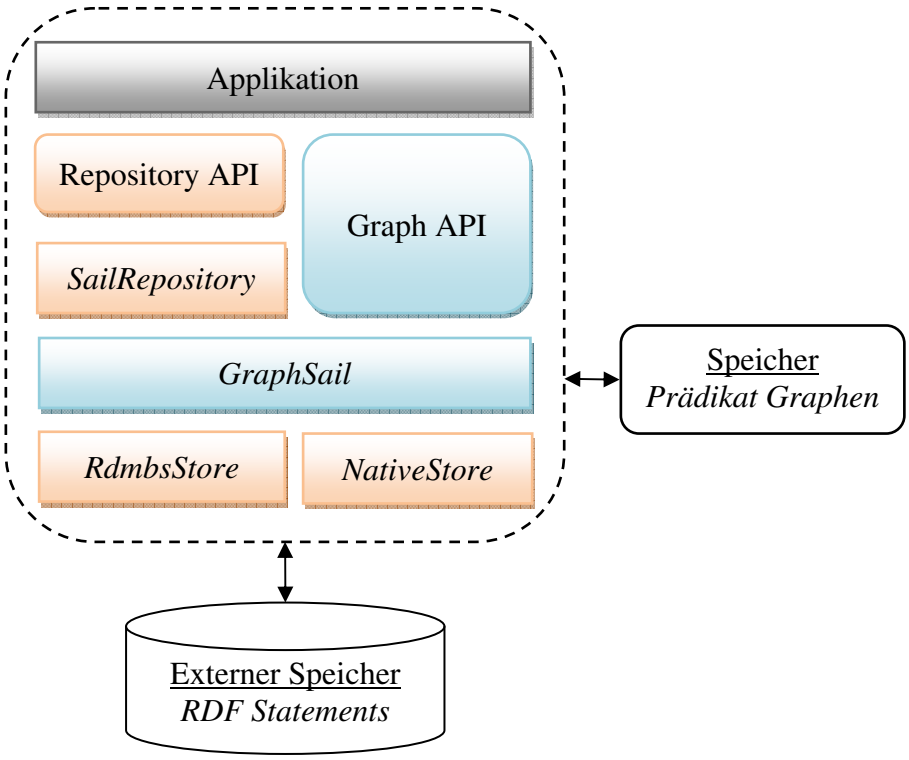

Abbildung 8.6: Eingliederung des Frameworks in eine auf Sesame basierende Applikation

8.3.2 RDF-Graphen

Um RDF-Graphen zu repräsentieren, verwendet die Graph Sail-Komponente die Graph Klassen der Graph-Komponente. Das Interface aller RDF-Graphen ist *RDFGraph.* Dieses ist vom *Graph* Interface der Graph-Komponente mit den beiden Typparametern *Value* und *Statement* für Knoten und Kanten abgeleitet. Da das *Value* Interface von den Sesame Klassen *URI, BNode* und *Literal* implementiert wird, können alle diese Objekte Knoten der Graphen darstellen. Ein Prädikat-Graph wird durch das *PredicateGraph* realisiert. Das *PredicateGraph* Interface ist von den Interfaces *RDFGraph* und *SingleGraph* abgeleitet. Um einen RDF-Graphen mit unterschiedlichen Properties erstellen zu können, existiert die Klasse *MultiPredicateGraph.* Ihr können mehrere *PredicateGraph* Instanzen übergeben werden. Abbildung 8.7 zeigt die Zusammenhänge zwischen den Graph Klassen und Interfaces der beiden Komponenten.

Das für einen Prädikat-Graphen eindeutige Property wird durch das Label des *SingleGraph* definiert. Die *PropertyDescription* beinhaltet neben der URI des Property zusätzlich die Information, ob das Property symmetrisch ist und ob ein inverses Property existiert. Eine *PropertyDescription* kann sowohl manuell als auch durch Einlesen des Schemas aus dem Repository erstellt werden.

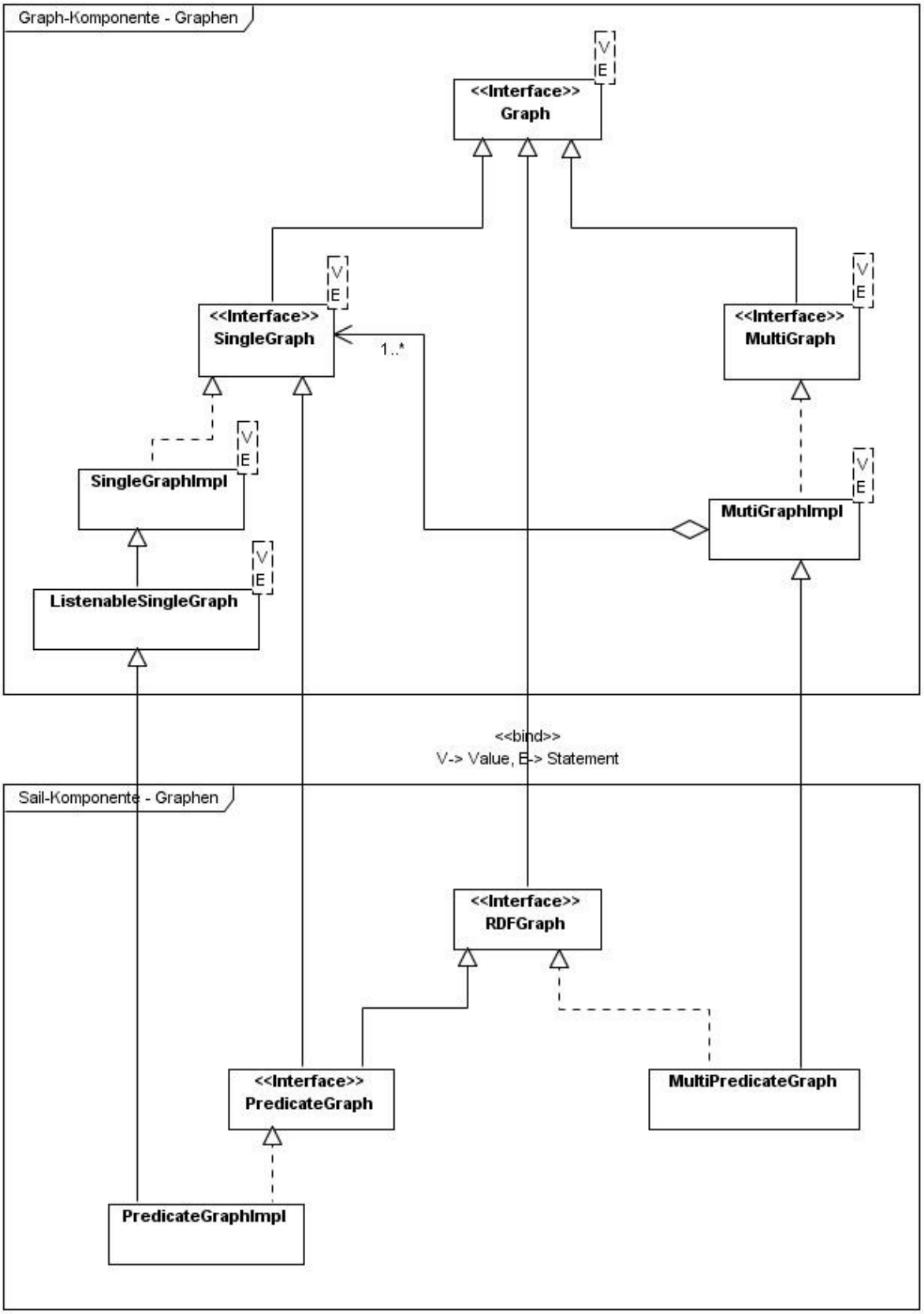

Abbildung 8.7: Graph Klassen der beiden Framework Komponenten

8.3.3 Laden von Prädikat-Graphen

Das Laden von Prädikat-Graphen geschieht über die *PropertyDescription,* die als Parameter der *GraphSail* Methode *loadPredicateGraph()* übergeben wird. Daraufhin wird zunächst die *GraphStructure* des Graphen erstellt. Der konkrete Typ der *GraphStructure* beziehungsweise des Graphen wird durch die Eigenschaften der *PropertyDescription* wie im Kapitel 7.1.3 beschrieben festgelegt.

Gesondert erwähnt werden muss hier das Vorgehen bei inversen Properties. Wird ein Prädikat-Graph ohne den Graphen seiner inversen Property geladen, so wird eine *DirectedGraphStructure* genutzt. Ist dagegen ein inverser Graph bereits geladen, wird für beide Graphen eine *BidirectionalGraphStructure* genutzt. Hier wird die interne Funktionsweise der *BidirectionalGraphStructure* verwendet um einerseits beide Graphen kompakt zu repräsentieren und andererseits die Konsistenz zu gewährleisten. Dies geschieht dadurch, dass die beiden Graphen der inversen Properties jeweils die inverse *BidirectionalGraphStructure* nutzen. Prädikat-Graphen aus symmetrischen Properties sind, wie im Kapitel 7.1.3 beschrieben, ungerichtete Graphen, für die eine *UndirectedGraphStructure* verwendet wird.

Das *GraphSail* nutzt intern zwei Verfahren, um die *GraphStructure* für einen *PredicateGraph* zu erstellen. Der erste Weg ist, sie über die Sail API aus Statements mit dem entsprechenden Prädikat zu erstellen. Hierbei werden von der Graph-Komponente angebotene *BufferedStructureBuilder* verwendet. Diese erlauben einen wesentlich schnelleren Aufbau der Graphstruktur als die über die *add()* Methoden der *GraphStructure.* Der zweite Weg ist, sie aus einem persistenten Cache zu laden. Dieser wird durch die Klasse *GraphStructureRepository* realisiert. Über sie können *GraphStructure* Objekte auf einen externen Speicher gespeichert und geladen werden. Das Speichern geschieht automatisch beim Entladen eines Prädikat-Graphen. Die Ladezeit für Prädikat-Graphen aus dem *GraphStructureRepository* ist um ein Vielfaches geringer als das Laden aus Sesame. Voraussetzung für die Nutzung einer persistenten *GraphStructure* ist jedoch, dass sie synchron mit dem Sesame Repository ist. Die persistente *GraphStructure* eines Prädikat-Graph wird inkonsistent, wenn Statements mit dem Property des Prädikat-Graphen als Prädikat, gelöscht oder hinzugefügt werden. Das Sesame Repository wird deshalb auf Änderungen, die einen persistenten Prädikat-Graphen betreffen, überwacht. Findet hier eine Änderung statt, wird die entsprechende *GraphStructure* aus dem *GraphStructureRepository* entfernt.

Wurde die Struktur eines Prädikat-Graphen erstellt, wird eine entsprechende *PredicateGraph* Instanz mit der *GraphStructure* und dem *RDFGraphObjectMapping* erzeugt.

8.3.4 Synchronisation

Wird ein Prädikat-Graph durch ein *GraphSail* verwaltet, übernimmt dieses die Bidirectionale Synchronisation von Prädikat-Graphen und Sesame Repository. Hierfür ist je Prädikat-Graph eine Instanz der Klasse *PredicateGraphSynchronizer* zuständig. Diese werden durch die vom *GraphSail* an die Repository API vergebenen Connections des Typs *PredicateChangeNotifyingConnection* über Änderungen des Repository informiert. Treten Änderungen auf, werden diese in den entsprechenden Prädikat-Graphen übernommen. Um die Synchronisation von Änderungen am Prädikat-Graphen zu ermöglichen, ist die Implementierung des *PredicateGraph* Interface von *ListenableSingleGraph* abgeleitet. Der *PredicateGraphSynchronizer* kann sich so als *SingleGraphListener* beim Prädikat-Graphen registrieren und so Änderungen an Sesame weitergeben.

Falls eine automatische Synchronisation explizit nicht gewünscht ist, kann der Graph über das *GraphSail* entladen werden. Hierdurch wird die Synchronisation beendet, der Graph kann jedoch weiter verwendet werden.

8.3.5 Value ID

Bei der im Kapitel 7.2.1 erläuterten Trennung von Struktur und Daten wird eine eindeutige ID für jedes RDF Value Objekt aus Sesame benötigt. Das einheitliche Interface für das Mapping ist *RDFGraphObjectMapping*. Die konkrete Implementierung wird bei der Initialisierung der *GraphSail* Klasse instanziiert. Da es wegen des Speicherverbrauchs der Objekte nicht möglich ist, sie direkt im Speicher zu halten (zum Beispiel einer in Map), wurde ein anderer eingesetzt.

Als einzige Implementierung steht zurzeit die Klasse *NativeGraphObjectMapping* zur Verfügung. Sie nutzt einen Mechanismus, der von der Sesame Sail Implementierung *NativeStore* eingesetzt wird. Der *NativeStore* verwendet zum Zugriff auf die *Value* Objekte die Klasse *ValueStore*. Sie ist für die persistente Speicherung der Values wie Literale oder URIs zuständig. Intern wird für jedes Value ein Hash generiert. Der Hash

wird, falls noch nicht vorhanden, mit einer fortlaufenden Integer ID in eine Datei (*HashFile*) gespeichert. Der eigentliche "Wert" des Value wird in einer eigenen Datei (*DataFile*) gespeichert. Um von ID zu Value und von Value zu ID mappen zu können, wird die ID des Values zusammen mit dem Offset des Speicherorts des Valuewerts in eine dritte Datei (*IDFile*) geschrieben. Für das Framework ist hier wichtig, dass sich die gespeicherten IDs nicht ändern, solange das Repository nicht vollständig geleert wird. Hierdurch ist die ID immer eindeutig für ein Value. Die Klasse *NativeGraphObjectMapping* nutzt den Sesame *ValueStore* um die Value Objekte in IDs und IDs in Value Objekte zu mappen.

Über das Interface *RDFGraphObjectMapping* besteht zudem die Möglichkeit auch ein Mapping für andere Persistenz-Strategien zu realisieren. Der Sesame *RdbmsStore* verwendet intern ebenfalls IDs für Value Objekte, wobei hier unter Umständen auf Änderungen Rücksicht genommen werden muss. Durch das *GraphSail* ist es auch möglich ein generisches Mapping unabhängig von der Persistenzschicht zu realisieren. Hierzu kann ein Wrapper für die *ValueFactory* implementiert werden, der das Mapping durchführt und dann die Aufrufe weiterleitet.

9 Test-Messungen

Um zu beurteilen, ob die Anforderungen hinsichtlich des Speicherverbrauchs und der Performance des Frameworks erfüllt wurden, wurden Testmessungen durchgeführt. Die Verarbeitung der Graphen im Framework ähnelt in vielerlei Hinsicht der eines üblichen Graph-Frameworks. Daher werden die Ergebnisse der Speicher- und Performance-Messungen mit denen der Graph-Frameworks Jung und JGraphT verglichen.

Dabei wurden die im Kapitel 6.2 beschriebenen Graphen des Typs *Graph<Integer, IDEdge>* zur Repräsentation verwendet. Da sich Jung und JGraphT jedoch in einigen Punkten von dem entwickelten Framework funktional unterscheiden, sind die Messungen nicht als qualitativer Vergleich, sondern als Referenzwert zur Einschätzung des Frameworks gedacht.

Als Testumgebung für die Speicher- und Performance-Messungen kam ein Intel(R) Core(TM) i5 CPU 2.67GHz (4 Kerne) mit 8 GB Arbeitsspeicher und Microsoft Windows 7 - 64 Bit Betriebssystem zum Einsatz. Für die Messungen der Ladezeiten wurde eine aktuelle Festplatte mit einer Umdrehungsgeschwindigkeit von 7200U/min einem Cache von 32MB und einer mittlere Zugriffszeit von 8,9 ms verwendet. Für jede Messung wurden jeweils separate Tests implementiert. Bei jeder Messung wurden zunächst warm-up Durchläufe durchgeführt. Das Endergebnis wurde über den Mittelwert mehrerer Messdurchläufe berechnet.

9.1 Speicherverbrauch

Zum Vergleich des Speichverbrauchs mit den Frameworks Jung und JGraphT wurde über das *GraphSail* der *PredicateGraph* für *skos:narrower* des WikiNet geladen. Da die *skos:narrower* Statements auch bei den Messungen im Kapitel 6.2 verwendet wurden, ergibt sich derselbe Graph mit etwa 3,2 Mio. Kanten und etwa 1 Mio. Knoten. Der einzelne *skos:narrower* Graph verwendet intern eine *DirectedGraphStructure,* welche im Vergleich zu den Frameworks funktional eingeschränkt ist. Aus diesem Grund wurde in einer weiteren Messung zusätzlich zu dem *skos:narrower* der *skos:broader PredicateGraph* geladen, wodurch intern beide Graphen eine *BidirectionalGraphStructure* nutzen. Bei den Messungen wurde darauf geachtet, nur den Speicherverbrauch des jeweiligen Graphen zu bestimmen. Dabei wurden die Methoden der Java *Runtime* Klasse verwendet. Da das Framework sowohl für 32-Bit als

auch 64-Bit Systeme eingesetzt werden soll, wurden die Speichermessungen mit dem Sun JDK 1.6 32-Bit und 64-Bit durchgeführt. Abbildung Abbildung 9.1 zeigt die Ergebnisse der Messungen.

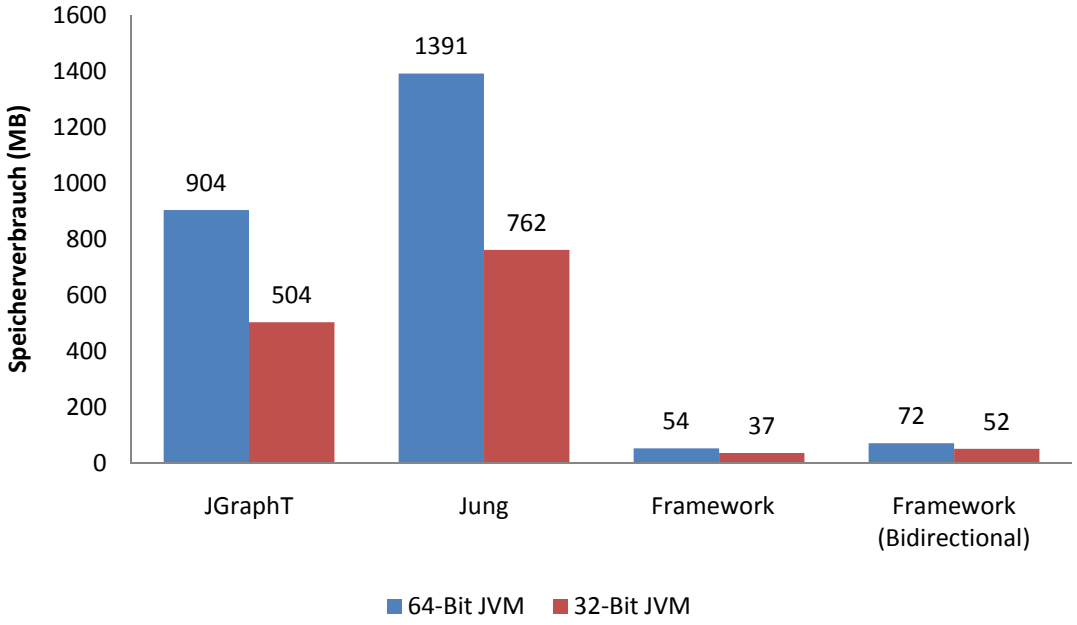

Abbildung 9.1: Speicherverbrauch WikiNet Narrower Graph

In einer weiteren Messung wurde auch der Speicherverbrauch für den vollständigen RDF-Graph des WikiNet gemessen. Es wurden die Prädikat-Graphen aller vorkommenden Properties geladen und über einen *MultiGraph* zu einem Graphen vereinigt. Für den vollständigen RDF-Graph mit ca. 8 Mio. Knoten und ca. 16 Mio. Kanten wurde ein Speicherverbrauch von 328 MB für die 64-Bit JVM und 196 MB für die 32-Bit JVM ermittelt.

9.2 Performance

Um die Performance des Frameworks beurteilen zu können, wurden verschiedene Testroutinen implementiert und die Laufzeiten gemessen. Ausgeführt wurden die Tests mit dem Sun JDK 1.6 64-Bit, wobei über den JVM Parameter *-Xmx* die maximale Heap-Größe 4000 MB festgelegt wurde.

Für die Beurteilung der Zugriffsgeschwindigkeit auf die Prädikat-Graphen wurde über das *GraphSail* der *skos:narrower PredicateGraph* geladen und mit den verschiedenen *SingleSourceGraphIterator* Implementierungen von einem fixen Startknoten aus der Graph traversiert. Zur Beurteilung der Zugriffsgeschwindigkeit wurde die Traversierung gewählt, da sie der klassische Anwendungsfall des Frameworks ist. Das Jung Framework ermöglicht keine direkt vergleichbare Traversierung. Aus diesem Grund wurde nur der Vergleichswert für JGraphT gemessen.

Der Teilgraph, über den ausgehend von dem Startknoten durch Verfolgen der gerichteten Kanten traversiert wird, enthält ca. 1 Mio. Knoten und ca. 2,8 Mio. Kanten. Die Ergebnisse der Laufzeitmessungen sind in Abbildung 9.2 dargestellt.

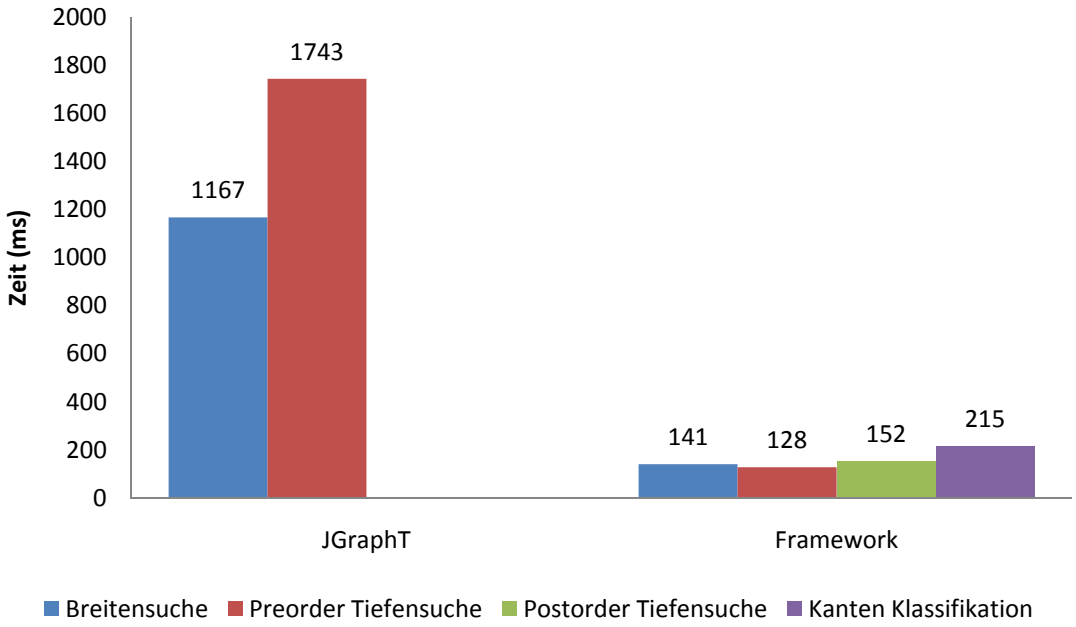

Abbildung 9.2: Laufzeit der WikiNet Narrower Graph Traversierung

Neben der Zugriffsgeschwindigkeit ist die Performance von Änderungen an den Graphen von Interesse. Hierfür wird wie bei der Traversierung wieder die *skos:narrower* Statements verwendet, wobei zunächst sämtliche Kanten erstellt wurden. Abbildung 9.3 zeigt die Zeit, die benötigt wird, um die Kanten dem Graphen hinzuzufügen beziehungsweise sie zu löschen. Zum Sinnvollen Vergleich wurde ein Graph mit *BidirectionalGraphStructure* verwendet.

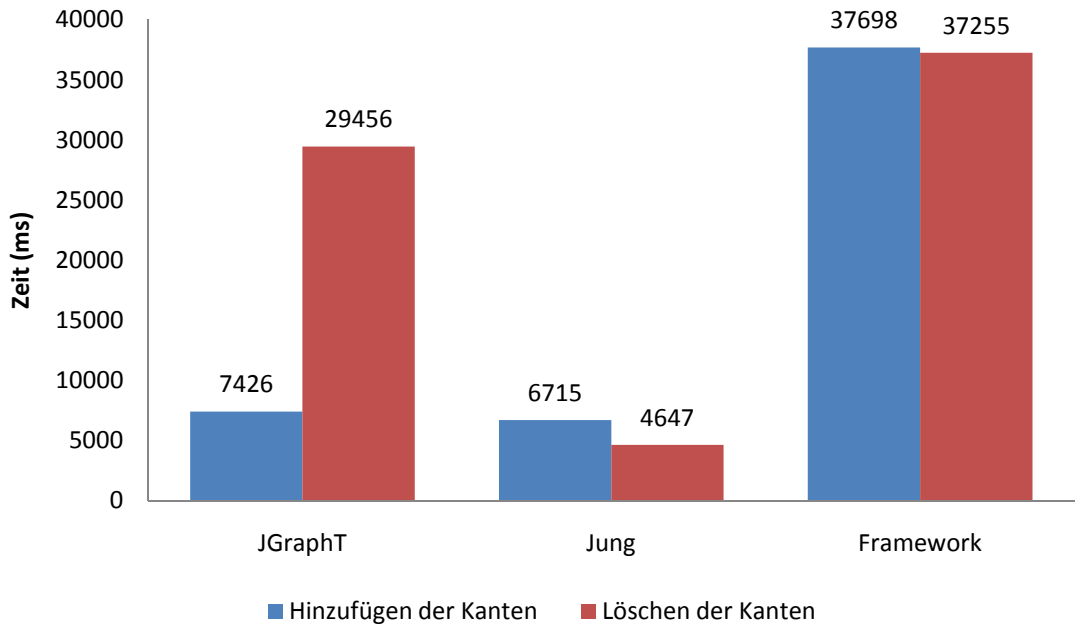

Abbildung 9.3: Laufzeit für das iterative Erstellen und Löschen des WikiNet Narrower Graph

Eine der beschriebenen Problematiken war das zum Laden von RDF-Graphen über die Sesame API aufgrund vieler I/O Operationen auf einen externen Speicher lange Ladezeiten aufgetreten sind. Durch den durch das Framework realisierte Persistente Caching können diese Ladezeiten vermieden werden. Finden keine Änderungen an einem gecachten Graphen statt muss dieser nur einmal aus Sesame geladen werden. Danach muss beim laden und entladen nur noch aus dem Cache geladen beziehungsweise gespeichert werden. Diese Zeiten wurden gemessen wobei wiederrum der WikiNet Narrower Graph zum Einsatz kam. Für das Laden des Graphen aus dem Persistenten Cache wurde eine Laufzeit von 269 ms gemessen. Das Speichern des Graphen benötigte 811 ms. Zum Vergleich benötigt das laden aller *skos:narrower* Statements über die in Kapitel 4 beschriebene *getStatements()* der Repository API 138555ms.

10 Ergebnisse

In diesem Kapitel werden die Ergebnisse, welche durch das entwickelte Framework erzielt werden konnten, thematisch zusammengefasst.

10.1 Graph Repräsentation

Ausgehend von dem allgemeinen Verständnis des RDF-Graphen als gerichteter Graph wurde der RDF-Graph über die Prädikate der Statements in Prädikat-Graphen unterteilt. Das Prädikat eines Prädikat-Graphen wird dabei als Label aller Kanten des Graphen verstanden. Hierdurch wird es möglich, mehrere Prädikat-Graphen zu einem Graphen zu vereinen, um somit auch Graphen aus Statements mit unterschiedlichen Prädikaten verarbeiten zu können. Dadurch, dass Prädikat-Graphen von inversen Properties synchronisiert und Prädikat-Graphen von symmetrischen Properties durch ungerichtete Graphen dargestellt werden, wird die spezielle Semantik dieser Properties berücksichtigt.

10.2 Management der RDF-Graphen

Die Konzepte, die für die Repräsentation der Statements als Graph umgesetzt wurden, sind Teil der Graph Sail-Komponente. Da sich das als zentraler Zugriffspunkt auf die Graphen implementierte *GraphSail* als transparente Schicht in den Sesame Sail Stack einbinden lässt, können bestehende Anwendungen ohne großen Aufwand um die Funktionalität des Frameworks erweitert werden.

Durch den realisierten persistenten Cache können die Prädikat-Graphen sehr schnell in den Speicher geladen werden. Dies ist jedoch nur in den Anwendungsfällen möglich, in denen keine Änderungen stattfinden während der Prädikat-Graph persistent gecached ist. Kann ein Prädikat-Graph nicht aus dem Cache geladen werden, ergeben sich weiterhin lange Ladezeiten. Somit konnte diese Problematik nur teilweise gelöst werden. Vorteil des Caches ist, dass er es ermöglicht, die Prädikat-Graphen zur Laufzeit dynamisch zu laden und zu entladen. Hierdurch müssen sie nur, wenn sie wirklich benötigt werden, im Speicher gehalten werden. Durch die automatische Bidirektionale-Synchronisation wird die Konsistenz von Graph und Sesame Repository bewahrt.

10.3 Algorithmische Verarbeitung

Die API, die über die Graph-Komponente zur Verarbeitung der Graphen zur Verfügung steht, orientiert sich analog zu anderen Graph Frameworks an graphentheoretischen Konzepten. Hierdurch wird die Umsetzung von Graph-Algorithmen, wie sie im Kapitel 3.4 beschrieben werden, unterstützt. Die implementierten Iteratoren liefern über die *EdgeIdentifier* gleichzeitig eine Sequenz von Kanten und Knoten. Zusätzlich stehen bei der Iteration die aktuelle Tiefe und der aktuelle Pfad zur Verfügung. In Verbindung mit der durch die *EdgeSelection* möglichen Steuerung ermöglichen die Iteratoren so ein hohes Maß an Flexibilität. Neben den Iteratoren stehen auch Methoden zur Verfügung, die Collections, wie zum Beispiel die Nachbarn eines Knoten, liefern.

Die Graph-Komponente bietet keine direkte Unterstützung für gewichtete Graphen. Im konkreten Anwendungsfall können gewichtete Kanten oder Knoten jedoch sehr effizient durch die typspezifischen Collections des *fastutil* Frameworks implementiert werden. Hierdurch kann zum Beispiel der im Kapitel 3.3.2 beschriebene Algorithmus zur Kategorisierung mit geringem Aufwand implementiert werden.

Da die Graph-Komponente eine den meisten Graph Frameworks ähnliche API besitzt, lassen sich mit relativ geringem Umfang Adapter zu anderen Frameworks realisieren. Hierdurch können Funktionen der Frameworks wie zum Beispiel eine Visualisierung genutzt werden.

Ein wichtiger Punkt bei der Verarbeitung ist, dass der Zugriff auf den Prädikat-Graphen nicht thread-safe ist. Da die Graphen und das Sesame Repository synchronisiert sind, muss dies auch bei der Benutzung der Repository API berücksichtigt werden.

10.4 Performance

Wie die Messungen aus Kapitel 9.1 zeigen, konnte der Speicherverbrauch der zur Repräsentation sehr großer semantischer Netze im Speicher benötigt wird, im Vergleich zu den bisherigen Lösungen und den als Referenz verwendeten Graphen-Frameworks deutlich reduziert werden. Dies konnte einerseits durch die Trennung von Struktur und Daten der RDF-Graphen sowie durch die kompakten *GraphStructure* Implementierungen erreicht werden. Dass die implementierte *GraphStructure* Klassen neben einer sehr kompakten Repräsentation der Graphen auch einen sehr performanten Zugriff auf den Graphen ermöglichen, zeigen die Laufzeitmessungen der Traversierung

aus Kapitel 9.2. Hier hat sich besonders das Iterator Konzept als effektiv herausgestellt. Die Iteratoren ermöglichen den Zugriff auf die Graphstruktur, wobei die intern verwendeten primitiven *int* Arrays weitgehend vor der API isolieren. Die internen *int* Arrays wiederum ermöglichen eine sehr effektive Implementierung und Steuerung der Iteratoren.

Im Gegensatz zum lesenden Zugriff sind Änderungen an der Graphstruktur deutlich langsamer als bei den zum Vergleich genutzten Frameworks Jung und JGraphT. Dies daran, das für Änderungen häufig neue *int* Arrays erzeugt und kopiert werden müssen. Um einen schnelleren Aufbau der Graphen zu ermöglichen wurde aus diesem Grund verschiedene *BufferedStructureBuilder* implementiert.

Die Graph-Komponente ist sehr konsequent auf Performance optimiert. So werden zum Beispiel bei einigen Iteratoren die gelieferten Objekte bei der nächsten Iteration wieder verwendet. Diese Form der Optimierung hat jedoch auch ihre Schattenseiten. Da ein solches Verhalten nicht direkt ersichtlich ist, muss der Benutzer des Frameworks ein gutes Verständnis der internen Funktionsweise besitzen. Vor diesem Hintergrund gilt es auch die Performance-Messungen zu beurteilen. Die Frameworks Jung und JGraphT sind mehr auf einfache und sichere Handhabung sowie konsequentes objektorientiertes Design ausgerichtet.

Das zur Trennung von Struktur und Daten des RDF-Graphen verwendete Mapping bietet einige Vorteile wird aber in Anwendungsfällen, in denen häufig auf die *Value* und *Statement* Objekte zugegriffen werden muss, wird das Mapping zum Bottleneck. Die entsprechenden Objekte müssen aus dem *ValueStore* geladen werden, was durch die nötigen I/O Operationen vergleichsweise sehr zeitaufwendig ist. Zudem setzt die verwendete Mapping-Implementierung voraus das der Sesame *NativeStore* verwendete wird.

11 Fazit und Ausblick

Ziel dieser Arbeit war es, ein auf Sesame aufsetzendes Framework zu entwickeln, das die Verarbeitung großer semantischer Netze ermöglicht. Das Framework soll die Probleme lösen, die sich bei der xdot GmbH mit den bisher eingesetzten Technologien ergeben haben. Hierbei bestand eines der Hauptprobleme darin, den bei der Repräsentation der semantischen Netze benötigten Speicherverbrauch auf ein für den praktischen Einsatz verträgliches Maß zu reduzieren. Die im Kapitel 9.1 dargestellten Ergebnisse der Speichermessungen zeigen, dass dies durch die implementierten Datenstrukturen des Frameworks realisiert werden konnte.

Die API der Graph-Komponente orientiert sich im Vergleich zur Sesame API deutlich mehr an graphentheoretischen Konzepten. Hierdurch lassen sich Anwendungsfälle, bei denen semantische Netze über Graph Algorithmen verarbeitet werden, erheblich leichter umsetzen. Insbesondere die implementierten Möglichkeiten zur Traversierung stellen ein flexibles Werkzeug hierfür bereit.

Für den Zugriff auf den Graphen konnte eine sehr gute Performance erreicht werden. So wurde zum Beispiel für die Traversierung und gleichzeitige Kanten-Klassifikation eines Graphen mit 1 Mio. Knoten und ca. 2,8 Mio. Kanten eine Laufzeit von nur 215 Millisekunden gemessen (siehe Abbildung 9.2). Diese sehr gute Performance ermöglicht es, Verfahren einzusetzen, die bisher zu langsam für den produktiven Einsatz waren.

Bereits bei der Implementierung der Unit-Tests und den Testmessungen zeigte sich wie groß die Vorteile sind, die sich durch das Framework bei Durchführung der Beispielanwendungen ergeben. Zuvor war jeder Testlauf bei der Entwicklung von Algorithmen mit minutenlangen Ladezeiten verbunden. Mit dem Framework treten diese langen Wartezeiten im Normalfall nicht mehr auf. Dies spart vor allem Zeit, Nerven und auch Kosten,

Zusammenfassend können die gestellten Anforderungen als erfüllt betrachtet werden.

Interessant für die Zukunft wäre eine Erweiterung der Graph Sail-Komponente, sodass auch für SPARQL Abfragen intern die Graphen genutzt werden. Dies würde eine weitere enorme Performance-Verbesserung bedeuten.

Ein Punkt, der sich in Zukunft als vorteilhaft beziehungsweise notwendig herausstellen könnte, ist die Synchronisation der Graphen in Bezug auf Threadsicherheit. Hier ist noch nicht klar ob, und in welcher Ebene diese notwendig werden könnte.

Eine weitere interessante Erweiterung stellt die Unterstützung von mehreren benannten Graphen dar. Sesame realisiert dies über den Context eines Statements. Um dieses Konzept im Framework zu berücksichtigen, müsste „nur“ jeweils eine *PropertyDescription* pro Property/Context Paar genutzt werden.

Die Erkenntnisse die hinsichtlich der Eigenschaften der Verarbeitung semantischer Netze durch Graph Algorithmen gewonnen wurden, können in vielen Fällen auch auf andere Ontologie-Typen übertragen werden. Damit besitzt das Framework das Potential in vielen weiteren Anwendungsfällen Verwendung zu finden.

12 Abbildungsverzeichnis

13 Tabellenverzeichnis

14 Listingverzeichnis

15 Literaturverzeichnis

[1] Fellbaum, Christiane D.: *WordNet: An Electronic Lexical Database.* MIT Press (1998).

[2] Gruber, Thomas R.: *A Translation Approach to Portable Ontology Specifications.* Knowledge Acquisition, (1993), Bd. 5, S. 199-220.

[3] Berners-Lee, Tim, Hendler, James und Lassila, Ora.: *The Semantic Web.* Scientific American, (2001), Bd. 284, S. 34-43.

[4] Klyne, Graham und Carroll, Jeremy J: Resource Description Framework (RDF): Concepts and Abstract Syntax. URL: http://www.w3.org/TR/rdf-concepts/ (zuletzt abgerufen am 30. Juli 2010)

[5] Krumke, Sven Oliver und Noltemeier, Hartmut: *Graphentheoretische Konzepte und Algorithmen.* Vieweg+Teubner Verlag / GWV Fachverlage GmbH, Wiesbaden, (2009).

[6] Kochut, Krys und Janik, Maciej: *SPARQLeR: Extended Sparql for Semantic Association Discovery.* Springer (2007). Bd. 4519, S. 145-159.

[7] Angles, Renzo, Gutierrez, Claudio und Hayes, Jonathan: *RDF Query Languages Need Support for Graph Properties,* (2004).

[8] Mihalcea, Rada, Tarau, Paul und Figa, Elizabeth: *PageRank on semantic networks, with application to word sense disambiguation.* Association for Computational Linguistics (2004), S. 1126.

[9] Page, Lawrence; Brin, Sergey; Motwani, Rajeev & Winograd, Terry: *The PageRank Citation Ranking: Bringing Order to the Web.* Stanford University (1999).

[10] Aduna B.V.: *User Guide for Sesame 2.3*.
URL: http://www.openrdf.org/doc/sesame2/users/ (zuletzt abgerufen am 30. Juli 2010)

[11] Hayes, Jonathan und Gutierrez, Claudio**:** *Bipartite Graphs as Intermediate Model for RDF.* Springer-Verlag (2004). Bd. 3298, S. 47-61.

[12] Miles, Alistair und Bechhofer, Sean: *SKOS Simple Knowledge Organization System Reference*. URL: http://www.w3.org/TR/2009/REC-skos-reference-20090818/ (zuletzt abgerufen am 30. Juli 2010)

[13] Broekstra, Jeen, Kampman, Arjohn und van. Harmelen, Frank: *Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema.* (2002). S. 54-68.

[14] Budanitsky, Alexander und Hirst, Graeme: *Semantic distance in WordNet: An experimental, application-oriented evaluation of five measures.* Pittsburgh, (2001). S. 29-24.