Universität des Saarlandes
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung Informatik
Bachelor-Studiengang Informatik

Bachelorarbeit

Formale Begriffsanalyse in Java

Entwurf und Implementierung effizienter Algorithmen

vorgelegt von

Daniel Norbert Götzmann

am 13. März 2007

angefertigt unter der Leitung von

Prof. Dr. Andreas Zeller

betreut von

Dr. Christian Lindig

begutachtet von

Prof. Dr. Andreas Zeller
Prof. Dr. Raimund Seidel

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und alle verwendeten Quellen angegeben habe.

Saarbrücken, den 13. März 2007

Einverständniserklärung

Hiermit erkläre ich mich damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek der Fachrichtung Informatik aufgenommen wird.

Saarbrücken, den 13. März 2007

Inhaltsverzeichnis

Kapitel 1

Einleitung

Formale Begriffsanalyse ist eine mathematische Theorie aus dem Bereich der Algebra (Ganter and Wille, 1999). Den Ausgangspunkt für Begriffsanalyse bildet dabei eine binäre Relation $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$. Jede binäre Relation besitzt eine nichtleere Menge von formalen Begriffen, die eine Hierarchie bilden. Effiziente Algorithmen zur Berechnung dieser Begriffshierachie sind der Gegenstand dieser Arbeit.

Veranschaulicht man eine Relation wie in Abbildung 1.1 als eine Kreuztabelle, dann entspricht ein formaler Begriff einem maximalen Block aus Kreuzen in dieser Tabelle. Genauer gesagt ist ein formaler Begriff ein Paar aus zwei Mengen (O, A). Die Menge O enthält dabei die Elemente einer Kante eines maximalen Blocks, die Menge A enthält die Elemente der anderen Kante.

Formale Begriffsanalyse ist, vereinfacht ausgedrückt, die mathematische Theorie über formale Begriffe. Insbesondere existiert bei formalen Begriffen eine Ordnung, welche Ober- und Unterbegriffe unterscheidet. Alle Begriffe einer Relation bilden einen Begriffsverband. In diesem Verband existiert zu jeder Teilmenge von Begriffen ein größter gemeinsamer Unterbegriff und ein kleinster gemeinsamer Oberbegriff.

Diese Arbeit stellt die Implementierung einer Java-Bibliothek für Begriffsanalyse vor. Die Hauptaufgaben dieser Implementierung (Colibri/Java) bestehen in der Berechnung der Menge aller Begriffe und ihrer Verbandsstruktur für eine Ausgangsrelation. Eine Schwierigkeit ist, dass ein Begriffsverband exponentiell mit der Größe der Ausgangsrelation wachsen kann, und deshalb sowohl Platz- als auch Zeiteffizienz von praktischem Interesse sind.

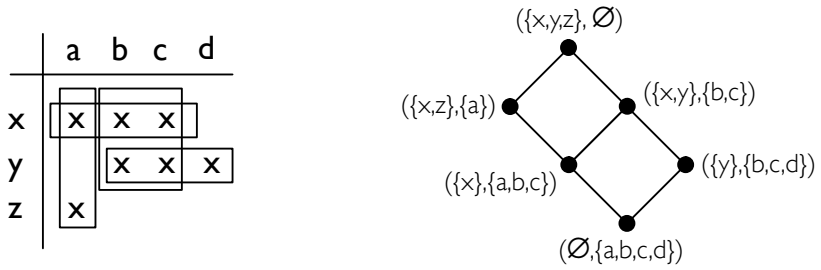


Abbildung 1.1: Eine binäre Relation kann als Kreuztabelle dargestellt werden. Ein Begriff entspricht einem maximalen Block in dieser Tabelle. Die Menge aller Begriffe bildet einen Verband, dessen Struktur als Hasse-Diagramm dargestellt werden kann.

```
// Ein Lattice-Objekt wird fuer eine Relation rel angelegt.
Lattice lat = new HybridLattice (rel);

// Die Methode conceptIterator liefert einen
// Iterator ueber die Begriffe von rel.
Iterator<Concept> it = lattice.conceptIterator(Traversal.BOTTOM_ATTRSIZE);

// Dieser Iterator kann z.B. verwendet werden, um alle Begriffe auszugeben.
while (it.hasNext()) {
    Concept c = it.next();
    System.out.println(c.toString());
}
```

Abbildung 1.2: Dieses Codefragment zeigt, wie ein Anwender über alle Begriffe des Verbandes iterieren kann. Zunächst wird ein Lattice-Objekt für eine Relation `rel` angelegt. Der Aufruf von `conceptIterator` liefert dann einen Iterator, der über alle Begriffe von `rel` iteriert.

Colibri/Java bietet für die Traversierung des Verbandes Iteratoren an, die nur lokale Ausschnitte des Verbandes berechnen und so unnötige zeit- und platzaufwendige Berechnungen vermeiden. Gleichzeitig ist der Programmierer durch die Iteratoren von diesen Optimierungen abgeschirmt. Der Code in Abbildung 1.2 demonstriert, wie einfach eine Traversierung des Verbandes aus Anwendersicht sein kann. Für Anwendungen, die mehr Kontrolle verlangen bietet Colibri/Java auch primitive Operationen an, auf denen die Implementierung dieser Iteratoren basiert.

In der praktischen Anwendung spielen hauptsächlich dünn besetzte Relationen eine Rolle. Ein wesentliches Designziel bestand daher darin, die Implementierung für dünn besetzte Relationen zu optimieren. Dazu wurden verschiedene Datenstrukturen zur Repräsentation von Relationen untersucht. Neben der relativen Performance verschiedener Datenstrukturen wurde Colibri/Java mit zwei existierenden Implementierungen zur Begriffsanalyse verglichen: Concepts (in C implementiert) und Colibri/ML (in Objective Caml implementiert). Für große Relationen ist die neue Implementierung deutlich effizienter: Für die Berechnung aller Begriffe einer Relation $\mathcal{R} = \mathcal{O} \times \mathcal{A}$ mit $|\mathcal{O}| = |\mathcal{A}| = 1000$ und $|\mathcal{R}| = 20000$ benötigen Concepts und Colibri/ML 205 Sekunden bzw. 134 Sekunden, während Colibri/Java nur 79 Sekunden benötigt.

Der Rest dieser Arbeit ist wie folgt gegliedert: Kapitel 2 gibt eine Einführung in die formale Begriffsanalyse. Im Anschluss daran wird ein Algorithmus dargestellt, der für eine gegebene Relation alle Begriffe sowie die Verbandsstruktur berechnet. In Kapitel 3 werden die aus Anwendersicht wichtigen Schnittstellen von Colibri/Java und ihre Methoden kurz vorgestellt. Kapitel 4 liefert einen Einblick in die Details der Implementierung. Insbesondere wird dort auch auf die verwendeten Datenstrukturen eingegangen. Darüber hinaus wird auch die Implementierung einiger wichtiger Algorithmen dargestellt. Kapitel 5 stellt die durchgeführten Korrektheitstests und Performance-Messungen vor. Bei den Performance-Messungen wurden dabei insbesondere auch Vergleiche mit Concepts und Colibri/ML durchgeführt. Kapitel 6 fasst schließlich die wichtigsten Ergebnisse dieser Arbeit zusammen.

Kapitel 2

Formale Begriffsanalyse

Bei der formalen Begriffsanalyse handelt es sich um eine mathematische Theorie aus dem Bereich der Algebra für binäre Relationen (Ganter and Wille, 1999).

2.1 Kontext und Begriff

Definition 2.1 (Kontext) *Sei $\mathcal{O}$ eine Menge von Objekten, $\mathcal{A}$ eine Menge von Attributen und $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$ eine Relation zwischen diesen beiden Mengen. Das Tripel $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ heißt formaler Kontext.*

Ein Kontext ist also eine binäre Relation, die angibt, welche Beziehungen zwischen bestimmten Objekten und Attributen bestehen. Ein Kontext lässt sich als eine Kontexttabelle darstellen, wobei die Zeilen mit den Objekten aus $\mathcal{O}$ und die Spalten mit den Attributen aus $\mathcal{A}$ beschriftet sind. Das zu o und a gehörende Feld in der Tabelle ist dabei genau dann markiert, wenn das Paar (o, a) in der Relation $\mathcal{R}$ enthalten ist.

Ein Beispiel für eine Kontexttabelle ist in Tabelle 2.1 dargestellt. Die Objektmenge $\mathcal{O}$ enthält einige Tierarten, die Attributmenge $\mathcal{A}$ enthält Eigenschaften wie *kann fliegen*. Das Kreuz zwischen *Ente* und *kann fliegen* steht für das Element *(Ente, kann fliegen)* $\in \mathcal{R}$, wobei $\mathcal{R}$ die Relation des formalen Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ist.

Ausgehend von einem Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ interessiert man sich häufig für die Attribute, die allen Objekten einer Objektmenge $O \subseteq \mathcal{O}$ gemeinsam sind. Formal sind die gemeinsamen Attribute O' einer Objektmenge O und die gemeinsamen Objekte A' einer Attributmenge A wie folgt definiert:

Definition 2.2 (Gemeinsame Objekte/Attribute) *Sei $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein formaler Kontext. Für $O \subseteq \mathcal{O}$ bezeichnet*

$$O' = \{a \in \mathcal{A} | \forall o \in O : (o, a) \in \mathcal{R}\}$$

die Menge der gemeinsamen Attribute von O. Analog dazu ist für $A \subseteq \mathcal{A}$ die Menge der gemeinsamen Objekte A' definiert als

$$A' = \{o \in \mathcal{O} | \forall a \in A : (o, a) \in \mathcal{R}\}$$

Einer leeren Objektmenge sind alle Attribute gemeinsam, das heißt für $O = \emptyset$ gilt: $O' = \emptyset' = \mathcal{A}$. Entsprechend sind einer leeren Attributmenge alle Objekte gemeinsam.

	Wirbeltier	Säugetier	Vogel	kann fliegen	2 Beine	4 Beine	6 Beine
Ente	×		×	×	×		
Fledermaus	×	×		×	×		
Hund	×	×				×	
Katze	×	×				×	
Krokodil	×					×	
Libelle				×			×
Schildkröte	×					×	
Schwan	×		×	×	×		
Wespe				×			×

Tabelle 2.1: Beispiel einer Kontexttabelle

In dem Beispiel aus Tabelle 2.1 sind die gemeinsamen Attribute von $O = \{$*Ente, Fledermaus*$\}$ gegeben durch $O' = \{$*Wirbeltier, kann fliegen, 2 Beine*$\}$.

Das folgende Theorem fasst einige wichtige Eigenschaften des Operators $'$ zusammen (Ganter and Wille, 1999).

Theorem 2.1 *Sei $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein Kontext und seien O, O_1, $O_2 \subseteq \mathcal{O}$ Mengen von Objekten und A, A_1, $A_2 \subseteq \mathcal{A}$ Mengen von Attributen. Es gelten die folgenden Eigenschaften:*

a) $O_1 \subseteq O_2 \Rightarrow O_2' \subseteq O_1'$ und $A_1 \subseteq A_2 \Rightarrow A_2' \subseteq A_1'$

b) $O \subseteq O''$ und $A \subseteq A''$

c) $O' = O'''$ und $A' = A'''$

d) $O \subseteq A' \Leftrightarrow A \subseteq O' \Leftrightarrow O \times A \subseteq \mathcal{R}$

Die folgende Eigenschaft des $'$-Operators erlaubt es, die Gemeinsamkeiten einer Vereinigung von Mengen auf die Gemeinsamkeiten der einzelnen Mengen zurückzuführen (Ganter and Wille, 1999):

Theorem 2.2 *Ist T eine Indexmenge und gilt für alle $t \in T$, dass $O_t \subseteq \mathcal{O}$ eine Menge von Objekten ist, dann gilt:*

$$(\bigcup_{t \in T} O_t)' = \bigcap_{t \in T} O_t'$$

Entsprechendes gilt für Attributmengen.

Mit Hilfe des $'$-Operators lässt sich die zentrale Definition der formalen Begriffsanalyse, die Definition des formalen Begriffs, folgendermaßen formulieren:

Definition 2.3 (Begriff) *Sei $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein formaler Kontext. Ein formaler Begriff ist ein Paar $(O, A) \in \mathfrak{P}(\mathcal{O}) \times \mathfrak{P}(\mathcal{A})$, so dass gilt: $O' = A$ und $A' = O$.*
$B(\mathcal{O}, \mathcal{A}, \mathcal{R}) = \{(O, A) \in \mathfrak{P}(\mathcal{O}) \times \mathfrak{P}(\mathcal{A}) | O' = A \wedge A' = O\}$ bezeichnet die Menge aller Begriffe des Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$.

Ein formaler Begriff ist also ein Paar, das aus einer Objektmenge $O \subseteq \mathcal{O}$ und einer Attributmenge $A \subseteq \mathcal{A}$ besteht, so dass für alle $o \in \mathcal{O}$, $a \in \mathcal{A}$ gilt: $(o, a) \in \mathcal{R}$. Außerdem gilt, dass es kein Objekt $o \in \mathcal{O} \setminus O$ gibt, so dass für alle $a \in A$ gilt: $(o, a) \in \mathcal{R}$. Umgekehrt gilt auch, dass es kein Attribut $a \in \mathcal{A} \setminus A$ gibt, so dass für alle $o \in O$ gilt: $(o, a) \in \mathcal{R}$. In der Kontexttabelle hat ein Begriff daher die Form eines maximalen Blocks.

Für den Kontext aus Abbildung 2.1 ist beispielsweise das Paar ({*Ente, Fledermaus, Schwan*}, {*Wirbeltier, kann fliegen, 2 Beine*}) ein Begriff, denn es gilt: {*Ente, Fledermaus, Schwan*}$'$ = {*Wirbeltier, kann fliegen, 2 Beine*} und {*Wirbeltier, kann fliegen, 2 Beine*}$'$ ={*Ente, Fledermaus, Schwan*}.

Insbesondere gilt auch, dass jedes Paar (O'', O') mit $O \subseteq \mathcal{O}$ ein Begriff ist, denn nach Theorem 2.1 gilt: $(O'')' = O'$. Entsprechend ist auch jedes Paar (A', A'') mit $A \subseteq \mathcal{A}$ ein Begriff.

Die Begriffe eines Kontextes sind partiell geordnet.

Definition 2.4 (Unter- und Oberbegriffe) *Sind $c_1 = (O_1, A_1)$ und $c_2 = (O_2, A_2)$ Begriffe des Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ mit $O_1 \subseteq O_2$, dann heißt c_1 Unterbegriff von c_2 und c_2 Oberbegriff von c_1. Die Notation $c_1 \leq c_2$ drückt aus, dass c_1 Unterbegriff von c_2 ist. Gilt zusätzlich $c_1 \neq c_2$, so heißt c_1 echter Unterbegriff von c_2 (Notation: $c_1 < c_2$). Ist $c_1 < c_2$ und gibt es keinen Begriff c_3 mit $c_1 < c_3 < c_2$, so heißt c_1 direkter Unterbegriff oder unterer Nachbar von c_2 (c_2 wird entsprechend als direkter Oberbegriff oder oberer Nachbar von c_1 bezeichnet).*

Ein Oberbegriff besitzt also mehr Objekte als sein echter Unterbegriff. Daraus folgt mit Theorem 2.1, dass ein Oberbegriff weniger Attribute als sein echter Unterbegriff hat. Denn aus $(O_1, A_1) \leq (O_2, A_2)$ folgt $O_1 \subset O_2$ und mit Theorem 2.1 ergibt sich daraus: $A_1 = O_1' \supset O_2' = A_2$.

Für manche Zwecke ist es nützlich, eine totale Ordnung auf Begriffen zu definieren. Da die partielle Ordnung aus Definition 2.4 auf der Teilmengenrelation der zugrundeliegenden Objektmengen beruht, ist es naheliegend, eine totale Ordnung von Begriffen mit Hilfe einer totalen Ordnung auf Teilmengen von $\mathcal{O}$ zu definieren.

Ganter definiert ausgehend von einer totalen Ordnung auf Objekten eine totale lektische Ordnung $\prec$ auf Objektmengen:

Definition 2.5 *Seien $O_1, O_2 \subseteq \mathcal{O} = \{o_1, o_2, \ldots, o_{|\mathcal{O}|}\}$ und sei $\prec$ eine totale Ordnung, so dass $o_1 \prec o_2 \prec \ldots \prec o_{|\mathcal{O}|}$ gilt. O_1 heißt lektisch kleiner als O_2 ($O_1 \prec O_2$), wenn das bezüglich $\prec$ kleinste Element, in dem sich O_1 und O_2 unterscheiden, in O_2 enthalten ist. Es gilt:*

$$O_1 \prec O_2 \Leftrightarrow \exists o_i \in O_2 \setminus O_1 : O_1 \cap \{o_1, \ldots, o_{i-1}\} = O_2 \cap \{o_1, \ldots, o_{i-1}\}$$

Ausgehend von Definition 2.5 lässt sich eine totale Ordnung auf Begriffen wie folgt definieren:

Definition 2.6 *Seien $c_1 = (O_1, A_1), c_2 = (O_2, A_2) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ Begriffe. c_1 heißt lektisch kleiner als c_2 ($c_1 \prec c_2$) genau dann wenn $O_1 \prec O_2$ gilt.*

Theorem 2.3 beschreibt eine wichtige Eigenschaft der lektischen Ordnung. Es besagt, dass jeder Unterbegriff bezüglich $\prec$ lektisch kleiner als sein Oberbegriff ist. Allerdings gilt die Umkehrung nicht, das heißt der lektisch kleinere von zwei Begriffen ist nicht unbedingt Unterbegriff des anderen.

Theorem 2.3 *Seien $c_1 = (O_1, A_1), c_2 = (O_2, A_2) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ zwei verschiedene Begriffe. Es gilt: $c_1 \leq c_2 \Rightarrow c_1 \prec c_2$*

Beweis in Lindig (1999b), S. 17

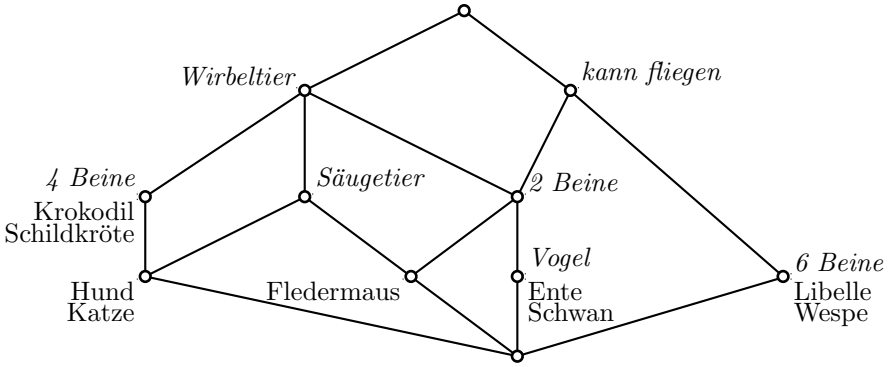


Abbildung 2.1: Hasse-Diagramm der Verbandsstruktur des Beispiels.

Theorem 2.4 *Sei $C \subseteq B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ eine Menge von Begriffen und sei $c \in C$ das kleinste Element in C bezüglich der totalen Ordnung $\prec$. Dann existiert in C kein Unterbegriff c_0 von c mit $c_0 < c$.*

Beweis: Ist c der bzgl. $\prec$ kleinste Begriff in C, dann gibt es keinen Begriff $c_0 \in C$ mit $c_0 \prec c$. Mit der Kontraposition von Theorem 2.3 folgt, dass es kein $c_0 \in C$ mit $c_0 < c$ gibt, also ist c minimal.

2.2 Begriffsverband

Definition 2.7 *Eine geordnete Menge $V = (\mathcal{V}, \leq)$ heißt vollständiger Verband, falls zu jeder Teilmenge $W \subseteq \mathcal{V}$ das Supremum $\bigwedge W = \max\{v \in \mathcal{V} | \forall w \in W : v \leq w\}$ und das Infimum $\bigvee W = \min\{v \in \mathcal{V} | \forall w \in W : w \leq v\}$ existiert.*

Zu jedem formalen Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ gehört eine Menge $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$, die alle Begriffe von $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ enthält. $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ bildet einen vollständigen Verband, den sogenannten Begriffsverband. Das heißt, dass für jede Menge $C \subseteq B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein eindeutiger größter Unterbegriff $\bigwedge C$ und ein eindeutiger kleinster Oberbegriff $\bigvee C$ existiert.

Theorem 2.5 (Hauptsatz, (Ganter and Wille, 1996)) *Sei $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein Kontext und sei $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ die Menge aller Begriffe dieses Kontextes. $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ist ein vollständiger Verband, wobei für Supremum $\bigvee C$ und Infimum $\bigwedge C$ gilt:*

$$\bigvee_{t \in T} (O_t, A_t) = ((\bigcup_{t \in T} (O_t))'', \bigcap_{t \in T} A_t)$$

$$\bigwedge_{t \in T} (O_t, A_t) = (\bigcap_{t \in T} (O_t), (\bigcup_{t \in T} A_t)'')$$

Da $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein vollständiger Verband ist, existieren der bezüglich $\leq$ kleinste Begriff $\perp$ und der bezüglich $\leq$ größte Begriff $\top$. Dabei ist $\perp$ der Begriff, der alle Attribute aus $\mathcal{A}$ enthält und $\top$ der Begriff, der alle Objekte aus $\mathcal{O}$ enthält, also $\perp = (\mathcal{A}', \mathcal{A}) = (\emptyset'', \emptyset')$ und $\top = (\mathcal{O}, \mathcal{O}') = (\emptyset', \emptyset'')$.

Abbildung 2.1 zeigt das Hasse-Diagramm des Begriffsverbands des Kontextes aus dem Beispiel in Tabelle 2.1. Jeder Knoten steht für einen Begriff des Verbands. Zwei Knoten sind genau dann

durch eine Kante verbunden, wenn die dazugehörigen Begriffe Nachbarn sind, wenn also der eine Begriff bezüglich $\leq$ ein direkter Oberbegriff des anderen ist.

Alle Begriffe in $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ deren Attributmenge ein fest gewähltes Attribut $a \in \mathcal{A}$ enthält, besitzen einen kleinsten gemeinsamen Oberbegriff $c_a = (\{a\}', \{a\}'')$. Entsprechend besitzen alle Begriffe, deren Objektmenge ein fest gewähltes Objekt $o \in \mathcal{O}$ enthält, einen grössten gemeinsamen Unterbegriff $c_o = (\{o\}'', \{o\}')$. Dabei ist a in der Attributmenge von c_a und o in der Objektmenge von c_o enthalten. Für eine übersichtliche Darstellung wird daher im Hasse-Diagramm nur der Begriff c_a mit a markiert. Entsprechend wird auch nur der Begriff c_o mit o markiert. Da alle Oberbegriffe von c_o das Objekt o enthalten und alle Unterbegriffe von c_a das Attribut a enthalten, lassen sich aus diesen Beschriftungen die Objekt- und Attributmengen der einzelnen Begriffe bestimmen.

2.3 Algorithmus

Für viele Anwendungen der Begriffsanalyse ist die Verbandsstruktur interessant. Daher wird ein Algorithmus benötigt, der sowohl die Menge aller Begriffe als auch die Verbandsstruktur effizient berechnet. Für einen endlichen Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ können $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ und die dazugehörige Verbandsstruktur berechnet werden, indem man beim untersten Begriff beginnt und rekursiv die oberen Nachbarn berechnet. Ausgangspunkt für die Berechnung der oberen Nachbarn ist dabei das folgende Theorem.

Theorem 2.6 *Sei $(O, A) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ und $(O \neq \mathcal{O})$. Die Objektmengen der direkten Oberbegriffe von (O, A) sind die (bzgl. $\subseteq$) minimalen Mengen der Form*

$$(O \cup \{o\})'', \quad o \notin O.$$

Zur Berechnung der oberen Nachbarn eines Begriffs (O, A) werden zunächst alle Mengen $(O \cup \{o\})''$ mit $o \in \mathcal{O} \setminus O$ betrachtet. Allerdings gehört nicht jede dieser Mengen zu einem direkten Oberbegriff von (O, A). Eine Menge $O_1 = (O \cup \{o_1\})''$ ist genau dann Teil eines direkten Oberbegriffs von (O, A), wenn sie bezüglich der $\subseteq$-Relation minimal unter allen anderen ist, wenn es also kein $o \in \mathcal{O} \setminus (O \cup \{o_1\})$ gibt, so dass $O_1 \supset (O \cup \{o\})''$ gilt.

Für eine effiziente algorithmische Bestimmung dieser minimalen Mengen ist das folgende Theorem hilfreich.

Theorem 2.7 *Für einen Begriff $(O, A) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$, $O \neq \mathcal{O}$ und $o \notin O$ ist $(O \cup \{o\})''$ genau dann Objektmenge eines oberen Nachbarn von (O, A), wenn für alle $x \in ((O \cup \{o\})'' \setminus O)$ gilt: $(O \cup \{x\})'' = (O \cup \{o\})''$.*

Beweis in Lindig (1999b) S. 35

Theorem 2.7 ist die Grundlage für den Algorithmus UpperNeighbors in Abbildung 2.2, der die oberen Nachbarn eines Begriffes (O, A) im zum Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ gehörenden Verband berechnet.

Zunächst enthält die Menge min alle Objekte aus $\mathcal{O} \setminus O$. Dann wird für alle $o \in \mathcal{O} \setminus O$ die Menge $(O \cup \{o\})''$ berechnet, wobei die Reihenfolge unerheblich ist. Falls diese Menge außer o noch ein weiteres Element x enthält, so dass $x \in$ min ist, wird x aus min entfernt. Falls es dagegen kein $x \neq o$ mit $x \in$ min gibt, ist nach Theorem 2.7 sichergestellt, dass $(O_1 = O \cup \{o\})''$ zu einem direkten Oberbegriff von (O, A) gehört. In diesem Fall wird $((O_1, A_1) = (O_1, O_1') = (O_1, (O \cup \{o\})''') = (O_1, (O \cup \{o\})')$ in die Menge der direkten Oberbegriffe von (O, A) eingefügt.

```
UpperNeighbors ((O, A), (𝒪, 𝒜, ℛ))
    min := 𝒪 \ O
    neighbors := ∅
    for each o ∈ 𝒪 \ O do
        A_1 := (O ∪ {o})'
        O_1 := A_1'
        if ((min ∩ (O_1 \ (O ∪ {o}))) = ∅) then
            neighbors := neighbors ∪ {(O_1, A_1)}
        else
            min := min \{o}
    return neighbors
```

Abbildung 2.2: Algorithmus zur Berechnung der oberen Nachbarn eines Begriffs

Außerdem wird o nicht aus min entfernt, so dass min am Ende des Algorithmus eine minimale Menge von Objekten ist, die alle oberen Nachbarn von (O, A) erzeugen.

Auf diese Weise wird jeder direkte Oberbegriff genau einmal in die Menge der oberen Nachbarn eingefügt. Gleichzeitig ist jedoch sichergestellt, dass indirekte Oberbegriffe nicht in die Menge der oberen Nachbarn eingefügt werden. Da jeder indirekte Oberbegriff von (O, A) Objekte enthält, die auch in einem direkten Oberbegriff von (O, A) enthalten sind, enthält jeder indirekte Oberbegriff von (O, A) mindestens ein Objekt, das in min enthalten ist.

Tabelle 2.2 zeigt, wie die Berechnung der oberen Nachbarn des Begriffs $(O, A) = (\{\text{Hund, Katze}\}, \{\text{Wirbeltier, Säugetier, 4 Beine}\}$ im Kontext aus Tabelle 2.1 abläuft. Jede Zeile entspricht dabei einem Durchlauf der For-Schleife. In der ersten Spalte steht das aktuell betrachtete Objekt o. Die zweite Spalte zeigt die in min enthaltenen Objekte zu Beginn dieses Schritts. Die Spalten 3 und 4 geben die in diesem Schritt berechnete Attributmenge A_1 bzw. die berechnete Objektmenge O_1 an. In der fünften Spalte steht, ob der Begriff (O_1, A_1) als direkter Oberbegriff von (O, A) akzeptiert wird.

Im ersten Schritt wird das Objekt *Ente* betrachtet. Zunächst werden die gemeinsamen Attribute von {*Hund, Katze, Ente*} berechnet, in diesem Fall also $A_1 = \{$*Hund, Katze, Ente*$\}' = \{$*Wirbeltier*$\}$. Dann wird für A_1 die Menge der gemeinsamen Objekte $O_1 = A_1'$ berechnet. In diesem Fall enthält O_1 außer *Hund*, *Katze* und *Ente* noch weitere Objekte, die in min enthalten sind, z.B. *Fledermaus*. Folglich wird der Begriff (O_1, A_1) nicht als direkter Oberbegriff von (O, A) akzeptiert und das in diesem Schleifendurchlauf betrachtete Objekt *Ente* wird aus min gelöscht.

Im zweiten Schritt wird das Objekt *Fledermaus* betrachtet. Die resultierende Objektmenge $O_1 = \{$*Hund, Katze, Fledermaus*$\}$ hat mit min nur das Objekt *Fledermaus* gemeinsam, also nur das in diesem Schleifendurchlauf betrachtete Objekt. Daher wird der Begriff (O_1, A_1) als direkter Oberbegriff von (O, A) akzeptiert. Das Objekt *Fledermaus* wird nicht aus min entfernt. So ist sichergestellt, dass Oberbegriffe von (O_1, A_1) (die nach Definition 2.4 das Objekt *Fledermaus* enthalten) in späteren Schleifendurchläufen verworfen werden.

Im dritten Schritt wird schließlich das Objekt *Krokodil* betrachtet. Der resultierende Begriff $(O_1, A_1) = (\{$*Hund, Katze, Krokodil, Schildkröte*$\}, \{\text{Wirbeltier, 4 Beine}\})$ ist tatsächlich ein direkter Oberbegriff von (O, A). Trotzdem wird er in diesem Schritt nicht akzeptiert, da mit *Schildkröte* ein zusätzliches Objekt in O_1 enthalten ist, das auch in min enthalten ist. Allerdings wird *Krokodil* aus min entfernt. Im fünften Schritt, wenn *Schildkröte* das betrachtete Objekt

Objekt o	min	A_1	O_1	akz?
Ente	{Ente, Fledermaus, Krokodil, Libelle, Schildkröte, Schwan, Wespe}	{Wirbeltier}	{Ente, Fledermaus, Hund, Katze, Krokodil, Schildkröte, Schwan}	nein
Fledermaus	{Fledermaus, Krokodil, Libelle, Schildkröte, Schwan, Wespe}	{Wirbeltier, Säugetier}	{Fledermaus, Hund, Katze}	ja
Krokodil	{Fledermaus, Krokodil, Libelle, Schildkröte, Schwan, Wespe}	{Wirbeltier, 4 Beine}	{Hund, Katze, Krokodil, Schildkröte}	nein
Libelle	{Fledermaus, Libelle, Schildkröte, Schwan, Wespe}	$\mathcal{A}$	$\emptyset$	nein
Schildkröte	{Fledermaus, Schildkröte, Schwan, Wespe}	{Wirbeltier, 4 Beine}	{Hund, Katze, Krokodil, Schildkröte}	ja
Schwan	{Fledermaus, Schildkröte, Schwan, Wespe}	{Wirbeltier}	{Ente, Fledermaus, Hund, Katze, Krokodil, Schildkröte, Schwan}	nein
Wespe	{Fledermaus, Schildkröte, Wespe}	$\mathcal{A}$	$\emptyset$	nein

Tabelle 2.2: Berechnung der oberen Nachbarn des Begriffs (O, A) = ({Hund, Katze}, {Wirbeltier, Säugetier, 4 Beine}).

ist, wird der gleiche Begriff noch einmal berechnet und akzeptiert. So ist sichergestellt, dass der gleiche Begriff nicht mehrfach akzeptiert wird.

Der in Abbildung 2.3 dargestellte Algorithmus LATTICE berechnet alle Begriffe eines Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ und die dazugehörige Verbandsstruktur, indem er mit Hilfe des Algorithmus UPPERNEIGHBORS rekursiv direkte Oberbegriffe berechnet.

Die Menge agenda enthält alle bereits berechneten Begriffe, für die noch kein Aufruf von UPPERNEIGHBORS stattgefunden hat. Zunächst wird der kleinste Begriff $(\emptyset'', \emptyset')$ berechnet und in den Verband lattice sowie die Menge agenda eingefügt. Anschließend werden in einer Schleife für jeden Begriff des Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ dessen direkte Oberbegriffe berechnet.

Innerhalb dieser Schleife wird nun zunächst durch den Aufruf von next(agenda) ein Begriff c ermittelt, der bezüglich $\leq$ minimal unter allen Begriffen in agenda ist. Dieser Begriff wird anschließend aus der Agenda entfernt. Dann werden die oberen Nachbarn von c berechnet und für jeden dieser oberen Nachbarn werden die Verbandsbeziehungen ergänzt.

Dazu wird für jeden oberen Nachbarn x von c überprüft, ob x bereits in lattice enthalten ist. Ist das nicht der Fall, wird x zu lattice hinzugefügt. Da x ein oberer Nachbar von c ist, wird x zu der Menge der oberen Nachbarn von c und c zu der Menge der unteren Nachbarn von x hinzugefügt. Schließlich wird x in die Agenda eingetragen, damit in einem zukünftigen Schleifendurchlauf auch die oberen Nachbarn von x berechnet werden.[1]

Der Algorithmus nutzt die partielle Ordnung der Begriffe aus. Indem durch next(agenda) ein Begriff c zurückgegeben wird, der bezüglich $\leq$ minimal unter allen Begriffen in agenda ist, ist sichergestellt, dass kein Begriff in agenda eingefügt wird, nachdem UPPERNEIGHBORS für

[1] Da es sich bei agenda um eine Menge handelt, ist x höchstens einmal in agenda enthalten.

```
LATTICE (𝒪, 𝒜, ℛ)
   c := (∅″, ∅′)
   insert(c, lattice)
   agenda := {c}
   while (agenda ≠ ∅) do
      c := next(agenda)
      agenda := agenda \{c}
      for each x ∈ UPPERNEIGBHBORS(c, (𝒪, 𝒜, ℛ)) do
         try x̄ := lookup(x, lattice)
         with NotFound
            insert(x, lattice)
            x̄ := lookup(x, lattice)
         x̄.lower := x̄.lower ∪ {c}
         c̄ := lookup(c, lattice)
         c̄.upper := c̄.upper ∪ {x}
         agenda := agenda ∪ {x}
   return lattice
```

Abbildung 2.3: Algorithmus zur Berechnung des Begriffsverbands

diesen Begriff bereits ausgeführt wurde. Das folgt daraus, dass innerhalb der Schleife nur direkte Oberbegriffe berechnet werden, die bezüglich $\leq$ echt größer als der ursprüngliche Begriff sind. Wegen der Transitivität von $\leq$ und der Minimalität von c wird daher zur Agenda kein Begriff x mit $x \leq c$ hinzugefügt, nachdem c aus der Agenda entfernt wurde.

Da der bezüglich $\prec$ kleinste Begriff in der Agenda nach Theorem 2.4 minimal bezüglich $\leq$ ist, kann die in Definition 2.6 definierte totale Ordnung von Begriffen hier ausgenutzt werden, um einen minimalen Begriff in agenda zu bestimmen.[2]

Durch kleine Änderungen kann der Algorithmus so modifiziert werden, dass er den Anforderungen verschiedener Anwendungen gerecht wird. So ist es möglich, anstatt beim untersten Begriff bei einem beliebigen anderen Begriff zu beginnen, was dazu führt, dass von diesem Begriff aus ein Teilverband des gesamten Verbandes berechnet wird. Für Anwendungen, welche die Informationen über die Verbandsstruktur nicht benötigen, kann auf die Speicherung der jeweiligen oberen und unteren Nachbarn der Begriffe in LATTICE und die damit verbundenen Aufrufe von lookup verzichtet werden, wobei die Funktionsweise des Algorithmus nicht beeinträchtigt wird. Das heißt alle Begriffe werden weiterhin korrekt berechnet.

In diesem Fall ergeben sich weitere Möglichkeiten der Modifikation, da die Notwendigkeit wegfällt, Begriffe zu speichern, für die UPPERNEIGHBORS bereits ausgeführt wurde. Ein Verzicht auf die Speicherung dieser Begriffe ist sinnvoll bei Anwendungen, die nur an Begriffen mit bestimmten Eigenschaften interessiert sind, da der Speicherplatzbedarf reduziert wird. Für Anwendungen, die Kanten[3] mit bestimmten Eigenschaften suchen, kann der Algorithmus so verändert werden, dass nur die Kanten gespeichert werden, die diese Eigenschaften erfüllen, so dass auch hier, im Vergleich zur Speicherung der kompletten Verbandsstruktur, weniger Speicherplatz benötigt wird.

[2] Die Verwendung einer anderen totalen Ordnung auf Begriffen, die Theorem 2.3 erfüllt, hätte den gleichen Effekt.

[3] Mit *Kante* ist hier ein Paar von Begriffen gemeint, so dass der eine Begriff oberer Nachbar des anderen ist. Im Hasse-Diagramm sind solche Paare durch eine Kante verbunden.

Kapitel 3

Begriffsanalyse als Java-Bibliothek

Die zentralen Elemente der formalen Begriffsanalyse sind Kontext, Begriff und Begriffsverband. Daher bilden Klassen und Schnittstellen für diese drei Elemente die Grundlage der Implementierung. Darüber hinaus werden zahlreiche weitere Klassen und Schnittstellen benötigt, welche die nötige Infrastruktur bereitstellen. Abbildung 3.1 gibt einen Überblick über alle Klassen und Schnittstellen.

Da in der praktischen Anwendung hauptsächlich dünn besetzte Kontexte vorkommen, ist es erforderlich, dass die Performance der Implementierung auf dünn besetzten Kontexten gut ist. Um dies zu erreichen, wurden verschiedene Datenstrukturen verwendet und die Performance der verschiedenen Implementierungen verglichen.

Dieses Kapitel bietet einen Überblick über die aus Anwendersicht wichtigen Schnittstellen. In Kapitel 4 werden die verschiedenen Datenstrukturen vorgestellt und Einzelheiten der Implementierung wichtiger Methoden dargestellt.

3.1 Die Schnittstelle Relation

Ein Kontext ist ein Tripel bestehend aus einer Objektmenge $\mathcal{O}$, einer Attributmenge $\mathcal{A}$ und einer binären Relation $\mathcal{R}$ zwischen diesen beiden Mengen. In der Implementierung wird ein Kontext durch die Schnittstelle `Relation` repräsentiert. Eine Schnittstelle ist hier sinnvoll, um verschiedene Datenstrukturen vergleichen zu können. Dadurch ist es möglich, die Implementierung des Algorithmus von der Implementierung dieser Datenstrukturen zu trennen.

In der Schnittstelle `Relation` sind Methoden definiert, die es dem Anwender ermöglichen, einen Kontext durch das Hinzufügen von Objekten, Attributen und Objekt-Attribut-Paaren zu erstellen. Darüber hinaus enthält `Relation` Methoden, die für die Begriffsanalyse wichtig sind. Abbildung 3.2 zeigt alle in der Schnittstelle `Relation` definierten Methoden.

Da es für die Implementierung des Algorithmus vorteilhaft ist, wenn auf Objekten und Attributen eine totale Ordnung definiert ist, lassen sich zu `Relation` nur `Comparable`-Objekte hinzufügen. Für ein korrektes Funktionieren der Implementierung ist es notwendig, dass die `compareTo`-Methode dieser Objekte genau dann 0 zurückgibt, wenn die `equals`-Methode `true` zurückgibt.

Die Methode `add` dient zum Hinzufügen von Elementen zu der Relation. Das Paar `(o,a)` wird der Relation `rel` durch den Aufruf von `rel.add(o,a)` hinzugefügt. Dabei ist zu beachten, dass

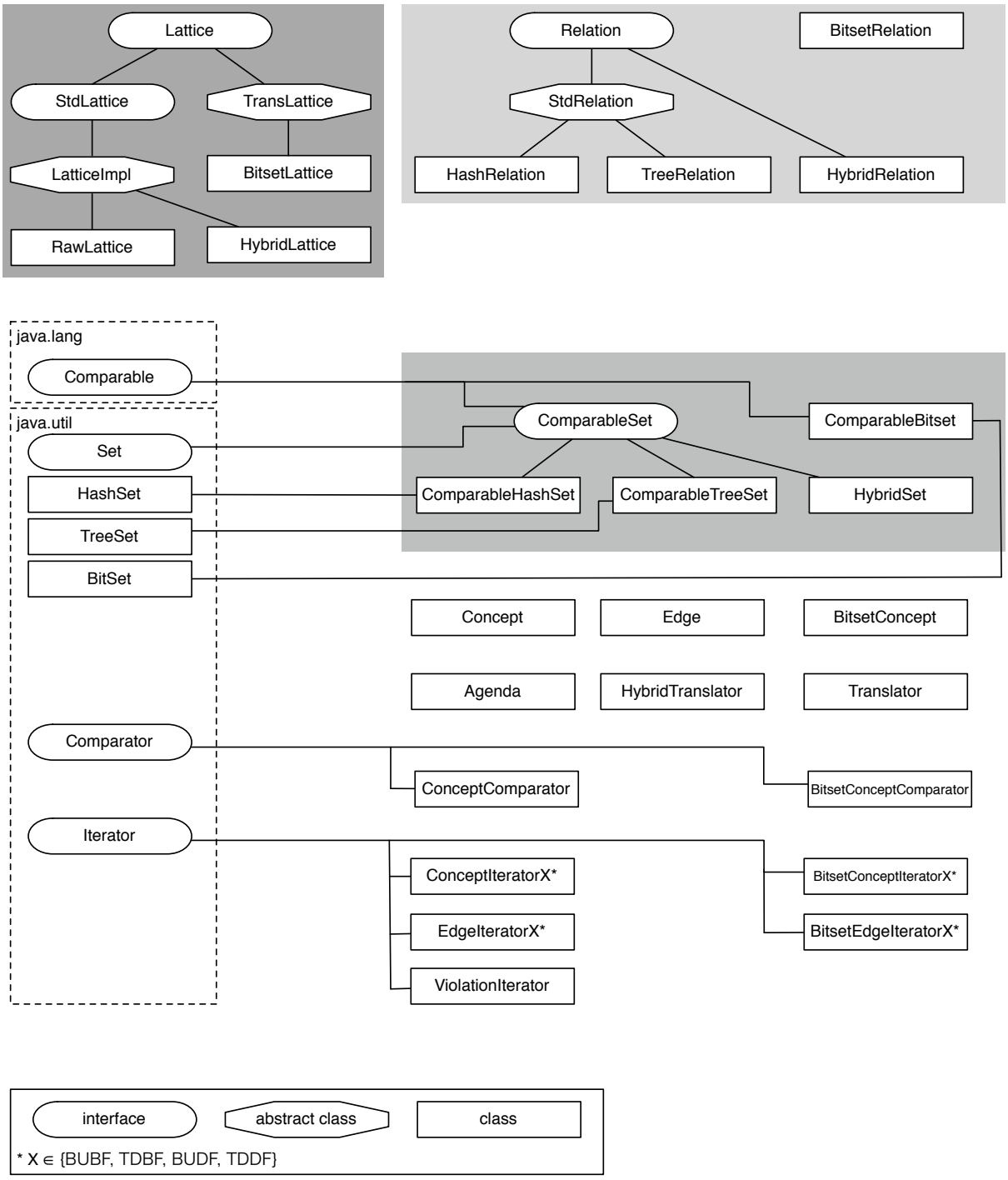


Abbildung 3.1: Übersicht über die Schnittstellen und Klassen der Implementierung

```
public interface Relation {
  void add(Comparable o, Comparable a);
  void remove(Comparable o, Comparable a);
  boolean contains(Comparable o, Comparable a);
  ComparableSet getAllObjects();
  ComparableSet getAllAttributes();
  ComparableSet getSizeObjects();
  ComparableSet getSizeAttributes();
  Iterator getObjects(Comparable a);
  Iterator getAttributes(Comparable o);
  ComparableSet getObjectSet(Comparable a);
  ComparableSet getAttributeSet(Comparable o);
  ComparableSet commonObjects(Collection c);
  ComparableSet commonAttributes(Collection c);
  boolean disallowChanges()
}
```

Abbildung 3.2: Methoden der Schnittstelle `Relation`

alle Objekte untereinander durch `compareTo` vergleichbar sein müssen. Das gleiche gilt für die Attribute. Da eine Relation eine Menge von Paaren darstellt, existiert jedes Paar maximal einmal. Der Aufruf `rel.add(o,null)` entspricht einer Erweiterung der Objektmenge $\mathcal{O}$ im Kontext um das Objekt `o`. Analog entspricht der Aufruf `rel.add(null,a)` dem Hinzufügen eines Attributs zur Attributmenge $\mathcal{A}$.

Dabei ist es jedoch nicht erforderlich, dass der Anwender die Objektmenge und die Attributmenge zu Beginn mittels `rel.add(o,null)` und `rel.add(null,a)` spezifiziert. Klassen, die `Relation` implementieren, sollen dafür sorgen, dass die Methode `add` die Objekt- und Attributmengen gegebenenfalls entsprechend anpasst.

Mit der Methode `remove` können Elemente aus der Relation entfernt werden. Das Paar `(o,a)` wird durch `rel.remove(o,a)` aus der Relation entfernt. Der Aufruf `rel.remove(o,null)` entspricht dem Löschen eines Objekts aus der Objektmenge $\mathcal{O}$, wobei zu `o` gehörende Paare natürlich ebenfalls gelöscht werden. Entsprechendes gilt für den Aufruf `rel.remove(null,a)`.

Die anderen Methoden der Schnittstelle `Relation` dienen dazu, die Informationen über den Kontext auszulesen.

Der Aufruf `rel.contains(o,a)` liefert genau dann `true`, wenn das Paar `(o,a)` in `rel` enthalten ist. Der Aufruf `rel.contains(o,null)` gibt genau dann `true` zurück, wenn das Objekt `o` in der zu `rel` gehörenden Objektmenge enthalten ist. Auch hier gilt entsprechendes für `rel.contains(null,a)`.

Die Methoden `getAllObjects` und `getAllAttributes` liefern ein `ComparableSet`[1], das die Menge aller Objekte bzw. Attribute repräsentiert. Diese Methoden sind nötig, da die Berechnung der oberen Nachbarn im Algorithmus UpperNeighbors (Abbildung 2.2) die Menge aller Objekte benötigt, um die oberen Nachbarn eines Begriffs zu berechnen.

Die Methoden `getSizeObjects` und `getSizeAttributes` liefern die Anzahl der Objekte bzw. Attribute.

[1] siehe Abschnitt 4.1

```
public interface Lattice {
  public Concept conceptFromObjects (Collection<Comparable> objects)
  public Concept conceptFromAttributes (Collection<Comparable> attributes)
  public Concept top ()
  public Concept bottom ()
  public Iterator<Concept> lowerNeighbors (Concept concept)
  public Iterator<Concept> upperNeighbors (Concept concept)
  public Iterator<Concept> conceptIterator (Traversal trav)
  public Iterator<Edge> edgeIterator (Traversal trav)
  public Iterator<Edge> violationIterator (int supp, float conf, int diff)
  public Concept join (Collection<Concept> concepts)
  public Concept meet (Collection<Concept> concepts)
}
```

Abbildung 3.3: Methoden der Schnittstelle `Lattice`

Die Methoden `getAttributes` und `getAttributeSet` liefern die Attribute, die das als Argument übergebenen Objekt hat. Dabei liefert `getAttributes` einen Iterator, `getAttributeSet` ein `ComparableSet`. Entsprechendes gilt für `getObjects` und `getObjectSet`.

Da die Berechnung gemeinsamer Objekte und Attribute für die Begriffsanalyse wichtig ist, definiert die Schnittstelle `Relation` auch die Methoden `commonObjects` und `commonAttributes`.

Für zusätzliche Sicherheit definiert `Relation` die Methode `disallowChanges`. Nachdem `disallowChanges` aufgerufen wurde, soll die Relation keine Aufrufe der Methoden `add` und `remove` mehr akzeptieren. Eine Klasse, die `Relation` implementiert, muss `disallowChanges` nicht zwingend implementieren[2]. Falls die Methode jedoch nicht in diesem Sinn implementiert wird, kann die Relation auch noch verändert werden, nachdem sie an ein Objekt vom Typ `Lattice`[3] übergeben wurde, wodurch die Berechnung des Verbandes gestört wird.

3.2 Die Klasse Concept

Die Klasse `Concept` repräsentiert Begriffe. Da ein Begriff aus einer Objektmenge und einer Attributmenge besteht, werden diese beiden Mengen dem Konstruktor von `Concept` übergeben. Mit Hilfe der Methoden `getObjects` und `getAttributes` können diese beiden Mengen abgefragt werden. Setter für Objektmenge und Attributmenge existieren nicht, da ein Objekt der Klasse `Concept`, in Übereinstimmung mit einem wirklichen Begriff, nicht veränderbar sein soll.

3.3 Die Schnittstelle Lattice

Die Schnittstelle `Lattice` und die Klassen, die sie implementieren, bilden den Kern der Implementierung von Begriffsanalyse. `Lattice` definiert die wichtigsten Methoden für Operationen auf Begriffsverbänden. Abbildung 3.3 zeigt eine Übersicht über alle Methoden der Schnittstelle `Lattice`.

[2] Falls `disallowChanges` nicht in diesem Sinn implementiert wird, soll `false` zurückgegeben werden.
[3] siehe Abschnitt 3.3

Da ein Begriffsverband nur relativ zu einem Kontext definiert ist, ist es naheliegend, dass den Konstruktoren der Klassen, die `Lattice` implementieren, eine `Relation` als Argument übergeben wird.

Die Methoden `conceptFromObjects` und `conceptFromAttributes` dienen dazu, aus einer Menge von Objekten bzw. Attributen einen Begriff zu berechnen. `top` berechnet den größten Begriff $\top$, `bottom` den kleinsten Begriff $\bot$.

Die Methode `upperNeighbors` berechnet die direkten Oberbegriffe eines Begriffs und gibt einen Iterator über diese Begriffe zurück. Analog werden durch `lowerNeighbors` die direkten Unterbegriffe eines Begriffs berechnet.

Mit Hilfe dieser Methoden kann ein Anwender ausgehend von einem beliebigen Begriff durch den Begriffsverband wandern. Ein derartiges Durchwandern des Begriffsverbands ist aber insbesondere dann mühsam, wenn der gesamte Verband berechnet werden soll. Zur Berechnung des gesamten Begriffsverbands existieren daher die zusätzlichen Methoden `conceptIterator` und `edgeIterator`. Diese Methoden geben Iteratoren zurück, die über alle Begriffe bzw. über alle Kanten des Begriffsverbands iterieren. Insbesondere wird dabei jeder Begriff bzw. jede Kante genau einmal von der jeweiligen `next`-Methode des Iterator zurückgegeben. Dabei kann der Anwender beim Aufruf der Methoden `conceptIterator` und `edgeIterator` wählen, in welcher Reihenfolge die Begriffe bzw. Kanten von dem Iterator zurückgegeben werden sollen.

Die Methode `violationIterator` liefert einen speziellen Iterator, der über die Kanten des Begriffsverbandes iteriert, deren Begriffe sehr ähnlich sind. Die Details sind in Abschnitt 4.9 beschrieben.

Kapitel 4

Implementierung

Es gibt drei konkrete Klassen, die die Schnittstelle `Lattice` implementieren: `RawLattice`, `HybridLattice` und `BitsetLattice`. Die Klasse `RawLattice` verwendet für die Berechnungen die `Relation`, die ihr im Konstruktor übergeben wird. Die Klassen `BitsetLattice` und `HybridLattice` übersetzen die ihnen im Konstruktor übergebene `Relation` dagegen zuerst in eine andere Datenstruktur.

Die Klasse `HybridLattice` verwendet dabei eine auf Bitsets basierende Datenstruktur, `HybridRelation`. Da `HybridRelation` die Schnittstelle `Relation` implementiert, können `RawLattice` und `HybridLattice` die gleiche Implementierung des Algorithmus verwenden. Daher ist der eigentliche Algorithmus in der abstrakten Klasse `LatticeImpl` implementiert (siehe Abschnitt 4.3). Die Klassen `RawLattice` und `HybridLattice` erweitern `LatticeImpl` lediglich um einen Konstruktor.

Die Klasse `BitsetLattice` verwendet intern ebenfalls eine auf Bitsets basierende Datenstruktur, `BitsetRelation`. Der wesentliche Unterschied liegt darin, dass die Klasse `BitsetRelation` die Schnittstelle `Relation` nicht implementiert. Dadurch ist es möglich, dass bei der eigentlichen Berechnung keine Übersetzungen zwischen den verschiedenen Datenstrukturen mehr nötig sind. Aus objektorientierter Sicht liegt ein großer Nachteil jedoch darin, dass der eigentliche Algorithmus neu implementiert werden muss.

Die Abschnitte 4.10 und 4.11 enthalten detailliertere Beschreibungen der von `HybridLattice` bzw. `BitsetLattice` intern verwendeten Datenstrukturen.

4.1 Die Schnittstelle ComparableSet

Die Schnittstelle `ComparableSet` erweitert `Set<Comparable>` um einige zusätzliche Methoden, die für die effiziente Umsetzung des Algorithmus aus Abschnitt 2.3 nützlich sind. Insbesondere erweitert `ComparableSet` die Schnittstelle `Comparable`, so dass eine `compareTo`-Methode zur Verfügung steht. Diese Methode dient dazu, zwei Instanzen von `ComparableSet` bezüglich der lexikalischen Ordnung $\prec$ zu vergleichen. Das ist sinnvoll, weil sich die `agenda` aus dem Algorithmus Lattice aus Abbildung 2.3 damit leichter als `SortedSet` implementieren lässt, so dass der bezüglich $\prec$ kleinste Begriff immer durch den Aufruf von `first` zurückgeliefert wird.

Außerdem definiert `ComparableSet` die überladene Methode `containsNone`. Der Aufruf `a.containsNone(b)` liefert `true` genau dann, wenn `a` und `b` disjunkt sind. Der Aufruf `a.contains-`

`None(b, c)` liefert `true` genau dann, wenn `a` und `b` disjunkt sind oder `c` das einzige gemeinsame Element von `a` und `b` ist. Die Notwendigkeit einer effizienten Implementierung von `containsNone` ergibt sich aus dem Algorithmus zur Berechnung der oberen Nachbarn in Abbildung 2.2. Dort wird ein Begriff nur dann akzeptiert, wenn der Durchschnitt von min und $(O_1 \setminus (O \cup \{o\}))$ leer ist. Das ist genau dann der Fall, wenn min und $O_1 \setminus \{o\}$ disjunkt sind, da min keine Objekte aus O enthält.

`ComparableHashSet` und `ComparableTreeSet` sind zwei Klassen, die die Schnittstelle `ComparableSet` implementieren. Die Klasse `ComparableHashSet` ist dabei eine Unterklasse der Klasse `java.util.HashSet`, die Klasse `ComparableTreeSet` ist eine Unterklasse von `java.util.TreeSet`. Bei der Implementierung der `compareTo`-Methode von `ComparableTreeSet` wurde ausgenutzt, dass der Iterator von `ComparableTreeSet` die Elemente der Menge nach wachsender Größe zurückgibt. Über die zu vergleichenden Mengen wird parallel iteriert, bis sich die zurückgegebenen Werte unterscheiden. Dadurch kann die Berechnung beschleunigt werden, was bei `ComparableHashSet` nicht möglich ist. Dort müssen aufgrund der nicht vorhandenen Sortierung alle Elemente betrachtet werden.

4.2 Die Klassen HashRelation und TreeRelation

Die abstrakte Klasse `StdRelation` implementiert `Relation`. Objektmenge und Attributmenge werden in ComparableSets gespeichert. Mit Hilfe von `Map`-Objekten werden einzelne Objekte auf die Menge ihrer Attribute abgebildet. Außerdem existiert eine `Map`, die einzelne Attribute auf die Menge ihrer Objekte abbildet. Ohne diese redundante Speicherung der Information wäre ein effizienter Zugriff auf die Daten nicht möglich.

`TreeRelation` und `HashRelation` sind konkrete Unterklassen von `StdRelation`, wobei die interne Repräsentation der Mengen bei `TreeRelation` durch `ComparableTreeSet` und bei `HashRelation` durch `ComparableHashSet` erfolgt. Dabei werden die Methoden, die von beiden Unterklassen verwendet werden können, bereits in `StdRelation` implementiert, so dass nur noch die Methoden, bei denen es aufgrund der verschiedenen verwendeten Datenstrukturen zu Unterschieden kommt, in den beiden konkreten Klassen implementiert werden.

4.3 Die abstrakte Klasse LatticeImpl

Die abstrakte Klasse `LatticeImpl` implementiert alle in der Schnittstelle `Lattice` definierten Methoden.

Die Implementierung der Methode `upperNeighbors` basiert auf dem in Abbildung 2.2 angegebenen Algorithmus. Aus Effizienzgründen wurden in der Implementierung jedoch weitere Eigenschaften ausgenutzt, die nicht sofort aus dem Pseudocode in Abbildung 2.2 ersichtlich sind.

Ausgehend von dem Begriff (O, A) wird in dem Algorithmus $A_1 = (O \cup \{o\})'$ berechnet, wobei $o \notin O$ ein weiteres Objekt ist. Die Berechnung der gemeinsamen Attribute in der Methode `commonAttributes` ist in Bezug auf die Laufzeit sehr teuer, da $|O|$ Schnittoperationen durchgeführt werden müssen. Deswegen nutzt die Implementierung hier Theorem 2.2 aus. Das heißt, dass die gemeinsamen Attribute von $(O \cup \{o\})$ nicht durch einen Aufruf von `commonAttributes` berechnet werden. Stattdessen wird direkt $A \cap \{o\}'$ berechnet. Es wird also nur eine Schnittoperation durchgeführt.

```
public Concept next() {
    if (!hasNext())
        throw new NoSuchElementException();
    Concept concept = agenda.pop();
    Iterator<Concept> iterator = lattice.upperNeighbors(concept);
    while (iterator.hasNext())
        agenda.add(iterator.next());
    return concept;
}
```

Abbildung 4.1: Java-Implementierung der `next`-Methode des *bottom-up-breadth-first*-Begriffsiterators.

Die Methoden `conceptIterator` und `edgeIterator` liefern Iteratoren, die über alle Begriffe bzw. über alle Kanten des Verbands iterieren. Da der Endbenutzer zwischen mehreren verschiedenen Traversierungen wählen können soll, wurden mehrere verschiedene Klassen für Iteratoren implementiert.

Da `LatticeImpl` den Verband aus Speicherplatzgründen nicht komplett vorberechnet, findet ein großer Teil der Berechnung innerhalb der Iteratorklassen statt. Die Implementierung der Begriffsiteratoren wird in den Abschnitten 4.4 und 4.7 erklärt. Die Implementierung der Kanteniteratoren wird in den Abschnitten 4.5 und 4.8 erläutert.

4.4 Breadth-first-Begriffsiteratoren

Die Grundlage für die *breadth-first*-Iteratoren bildet der Algorithmus LATTICE aus Abbildung 2.3. Da jeder Begriff nur einmal ausgegeben werden soll und die Struktur des Begriffsverbandes nicht berechnet werden soll, wird auf die Speicherung der Begriffe und der Verbandsstruktur verzichtet. Für den *bottom-up-breadth-first*-Iterator wird im Konstruktor der kleinste Begriff $\perp$ der Agenda hinzugefügt. Bei jedem Aufruf von `next` wird zuerst durch einen Aufruf der `pop`-Methode der Agenda der erste Begriff aus der Agenda entfernt. Für diesen Begriff werden dann die oberen Nachbarn berechnet und in die Agenda eingefügt. Schließlich wird dieser Begriff zurückgegeben. Die Implementierung der `next`-Methode ist in Abbildung 4.1 dargestellt.

Damit die Begriffe in der Agenda richtig sortiert werden können, muss der Agenda im Konstruktor ein `Comparator` für Begriffe übergeben werden. Für die Korrektheit ist es wichtig, dass bezüglich der `compare`-Methode Oberbegriffe immer größer als ihre Unterbegriffe sind. Das ist beispielsweise dann der Fall, wenn die `compare`-Methode die lektische Ordnung für Begriffe (Definition 2.6) implementiert, Begriffe also nach der lektischen Ordnung ihrer Objektmenge (Definition 2.5) vergleicht. Das ist leicht möglich, da Objektmengen in der Klasse `Concept` als `ComparableSet` gespeichert sind und die `compareTo`-Methode von `ComparableSet` die lektische Ordnung aus Definition 2.5 implementiert. Außerdem ist es möglich, Begriffe nach der lektischen Ordnung ihrer Attributmenge zu ordnen.

Beide Ordnungen garantieren, dass die Iteratoren funktionieren und dass Oberbegriffe immer später als ihre Unterbegriffe zurückgegeben werden. Allerdings wirkt die Traversierung in beiden Fällen sehr chaotisch. Daher bietet es sich an, dass Begriffe nach der Größe ihrer Objekt- oder Attributmenge sortiert werden. Nur in dem Fall, in dem die Objekt- bzw. Attributmengen zweier Begriffe die gleiche Anzahl an Elementen haben, werden diese Begriffe anhand der

lektischen Ordnung der Objekt- bzw. Attributmenge verglichen. Da die Objektmenge eines Unterbegriffs immer eine echte Teilmenge der Objektmenge eines Oberbegriffs ist und folglich weniger Elemente enthält, kann auch solch eine Ordnung verwendet werden, ohne die Korrektheit der Implementierung zu gefährden. Analog gilt, dass die Attributmenge eines Unterbegriffs immer eine echte Obermenge der Attributmenge eines Oberbegriffs ist und damit mehr Elemente enthält. Also ist auch eine Ordnung über die Größe der Attributmenge problemlos möglich. Die Verwendung dieser Iteratoren bietet sich insbesondere dann an, wenn die Iteration beim Erreichen einer bestimmten Anzahl von Objekten bzw. Attributen abgebrochen werden soll.

Der *top-down-breadth-first*-Begriffsiterator basiert auf dem selben Prinzip. Die Unterschiede bestehen darin, dass zu Beginn anstatt des kleinsten Begriffs $\bot$ der größte Begriff $\top$ zur Agenda hinzugefügt wird und dass anstatt der oberen Nachbarn die unteren Nachbarn berechnet werden. Außerdem wird ein `Comparator` verwendet, bezüglich dessen `compare`-Methode Oberbegriffe immer kleiner als ihre Unterbegriffe sind, damit das erste Element in der Agenda bezüglich $\leq$ (Definition 2.4) maximal ist.

4.5 Breadth-first-Kanteniteratoren

Bei näherer Betrachtung der *breadth-first*-Begriffsiteratoren fällt auf, dass die oberen Nachbarn (bei den *bottom-up*-Iteratoren) bzw. die unteren Nachbarn (bei den *top-down*-Iteratoren) für jeden Begriff genau einmal berechnet werden und zwar immer dann, wenn als nächstes der dazugehörige Begriff ausgegeben wird. Da eine Kante im Verband nichts anderes als ein Begriff und einer seiner Nachbarn ist, bedeutet das, dass jede Kante genau einmal berechnet wird. Daher ist es naheliegend, die Methoden der Begriffsiteratoren so zu modifizieren, dass anstatt der Begriffe die Kanten ausgegeben werden.

Die *breadth-first*-Kanteniteratoren nutzen wie die *breadth-first*-Begriffsiteratoren eine `Agenda`, in der die noch nicht abgearbeiteten Begriffe enthalten sind. Dabei nutzen die Kanteniteratoren die gleichen Comparatoren wie die Begriffsiteratoren, so dass auch hier sichergestellt ist, dass ein bereits abgearbeiteter Begriff nicht noch einmal in die Agenda eingetragen wird, dass also für jeden Begriff genau einmal die Methode `upperNeighbors` bzw. `lowerNeighbors` aufgerufen wird.

Da ein Begriff mehrere obere bzw. untere Nachbarn haben kann, `next` jedoch bei jedem Aufruf genau eine Kante zurückgeben soll, sind einige Änderungen gegenüber den Begriffsiteratoren nötig.

Beim *bottom-up-breadth-first*-Kanteniterator wird der aktuelle Begriff im privaten Feld `current` gespeichert, der Iterator über seine oberen Nachbarn wird im privaten Feld `localIterator` gespeichert, so dass bei jedem Aufruf von `next` die Kante zwischen `current` und dem nächsten Begriff aus `localIterator` zurückgegeben werden kann.

Die `hasNext`-Methode soll genau dann `true` zurückgeben, wenn es eine Kante gibt, die zuvor noch nicht durch `next` zurückgegeben wurde. `hasNext` testet also, ob `localIterator.hasNext() == true` gilt. Wenn das der Fall ist, gibt `hasNext` `true` zurück, denn es gibt für `current` einen oberen Nachbarn, so dass die Kante zwischen diesen beiden Begriffen noch nicht zurückgegeben wurde. Im anderen Fall wird der nächste Begriff aus der Agenda ermittelt, sofern die Agenda nicht leer ist. Falls die Agenda leer ist, ist klar, dass alle Kanten des Verbandes ausgegeben wurden und `hasNext` gibt `false` aus. Sonst wird der nächste Begriff in der Agenda in `current` gespeichert und dessen obere Nachbarn werden in `localIterator` gespeichert. Falls der Begriff

keine oberen Nachbarn hat[1], gibt `hasNext false` zurück, sonst `true`.

Die `next`-Methode ruft `hasNext` auf und wirft eine Ausnahme, wenn `hasNext false` liefert. Liefert `hasNext true`, dann ist sichergestellt, dass `current` einen Begriff und `localIterator` einen nichtleeren Iterator enthält, so dass die Kante zwischen dem Begriff in `current` und dem nächsten Begriff in `localIterator` zuvor noch nicht zurückgegeben wurde. `next` gibt daher diese Kante zurück und fügt der Agenda den Begriff hinzu, der von `localIterator.next()` zurückgegeben wurde.

4.6 Die Klasse Agenda

Im Algorithmus LATTICE aus Abbildung 2.3 werden Begriffe in eine Agenda eingefügt. Später wird dort jeweils ein bezüglich $\leq$ minimaler Begriff aus dieser Agenda entfernt. Es ist also naheliegend, dass in der Implementierung eine Klasse `Agenda` existiert, die sich gleich verhält wie die Agenda aus Abbildung 2.3. Die Klasse `Agenda` enthält die folgenden Methoden:

```
class Agenda<T> {
  Agenda (Comparator<T> c)
  T pop ()
  void add (T item)
  boolean isEmpty()
}
```

Im Konstruktor muss ein `Comparator` übergeben werden, damit die Elemente, die später zur Agenda hinzugefügt werden, untereinander verglichen werden können.

Durch Aufruf der `add`-Methode wird das übergebene Element der Agenda hinzugefügt. Da die Agenda zur Speicherung der Elemente intern ein `SortedSet` verwendet, ist sichergestellt, dass kein Element zweimal in der Agenda enthalten ist.

Die `pop`-Methode liefert das bezüglich der `compare`-Methode, die im Konstruktor übergeben wurde, kleinste Element zurück und entfernt es aus der Agenda. Dieses Element kann durch den Aufruf der `first`-Methode auf dem `SortedSet` effizient ermittelt werden.

4.7 Depth-first-Begriffsiteratoren

Die Implementierung der *depth-first*-Iteratoren ist komplizierter als die Implementierung der *breadth-first*-Iteratoren, da nicht sicher ist, dass die für einen Begriff berechneten oberen oder unteren Nachbarn zuvor noch nicht ausgegeben wurden. Man könnte alle bereits entdeckten Begriffe speichern und so später überprüfen, ob ein Begriff zuvor schon einmal ausgegeben wurde. Insbesondere für große Verbände wäre solch ein Verfahren jedoch in Bezug auf Zeit und Speicherplatz ineffizient. Daher nutzt die Implementierung des *bottom-up-depth-first*-Iterators die Eigenschaft des Begriffsverbandes aus, dass ein Begriff $c_1 = (O_1, A_1)$ genau dann Oberbegriff eines Begriffs $c = (O, A)$ ist, wenn $O_1 \supseteq O$ gilt.

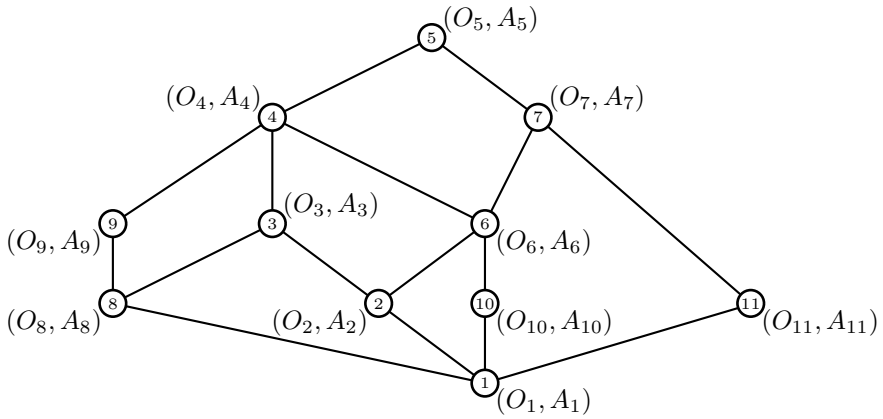
Für den *bottom-up-depth-first*-Iterator bedeutet das: Nachdem alle Oberbegriffe eines Begriffs c abgearbeitet wurden, muss keine Information über diese Oberbegriffe mehr gespeichert werden. Es genügt, stattdessen nur die Objektmenge von c zu speichern. Wird danach ein Begriff

[1] Das ist nur beim größten Begriff *top* der Fall, in diesem Fall ist also auch die Agenda leer.

```
private StdLattice lattice;
private LinkedList<Set<ComparableSet>> past;
private LinkedList<Concept> current;
private LinkedList<Iterator<Concept>> future;
private Concept nextConcept;

private boolean computeNext() {
    boolean accept = false;
    Concept concept = null;
    while (!future.isEmpty() && !accept && nextConcept == null) {
        Iterator<Concept> futureLevel = future.getLast();
        concept = futureLevel.next();
        if (!seenBefore(concept)) {
            accept = true;
            Iterator<Concept> upperNeighbors = lattice.upperNeighbors(concept);
            future.addLast(upperNeighbors);
            past.addLast(new TreeSet<ComparableSet>());
            futureLevel = future.getLast();
            current.addLast(concept);
        }
        while(!future.isEmpty()
              && (futureLevel == null || !futureLevel.hasNext())) {
            future.removeLast();
            past.removeLast();
            if (!future.isEmpty()) {
                futureLevel = future.getLast();
            }
            if (!current.isEmpty()) {
                ComparableSet objects = current.getLast().getObjects();
                past.getLast().add(objects);
                current.removeLast();
            }
        }
    }
    if (accept) {
        nextConcept = concept;
    } else {
        nextConcept = null;
    }
    return accept;
}
```

Abbildung 4.2: Die Methode `computeNext` des *bottom-up-breadth-first*-Begriffsiterators



`i`	`past[i]`	`current[i]`	`future[i]`
Vor dem ersten Schleifendurchlauf:			
0	$\emptyset$	(O_1, A_1)	$[(O_8, A_8), (O_{10}, A_{10}), (O_{11}, A_{11})]$
1	$\{(O_3)\}$	(O_2, A_2)	[ ]
2	$\emptyset$	(O_6, A_6)	[ ]
3	$\emptyset$	(O_7, A_7)	$[(O_5, A_5)]$
Nach dem ersten Schleifendurchlauf:			
0	$\{(O_2)\}$	(O_1, A_1)	$[(O_8, A_8), (O_{10}, A_{10}), (O_{11}, A_{11})]$
Nach dem zweiten Schleifendurchlauf:			
0	$\{(O_2)\}$	(O_1, A_1)	$[(O_{10}, A_{10}), (O_{11}, A_{11})]$
1	$\emptyset$	(O_8, A_8)	$[(O_3, A_3),(O_9, A_9)]$

Abbildung 4.3: Beispiel einer *bottom-up-depth-first*-Iteration.

„entdeckt“, dessen Objektmenge eine Obermenge der Objektmenge von c ist, so ist sicher, dass dieser Begriff und alle seine Oberbegriffe bereits zuvor ausgegeben wurden. Solch ein Begriff muss dann verworfen werden.

Damit die `hasNext`-Methode korrekt funktioniert, muss bekannt sein, ob noch ein Begriff existiert, der noch nicht von der `next`-Methode zurückgegeben wurde. Dazu muss jedoch versucht werden, den nächsten Begriff zu berechnen. `hasNext` ruft daher die private Methode `computeNext` auf, die den nächsten Begriff berechnet und diesen in dem privaten Feld `nextConcept` ablegt. `computeNext` gibt genau dann `false` zurück, wenn kein weiterer Begriff existiert. Falls `hasNext true` zurückgibt, gibt `next` den in `nextConcept` gespeicherten Begriff zurück und setzt `nextConcept` auf `null`. Die Implementierung von `computeNext` ist in Abbildung 4.2 dargestellt.

Die Implementierung des *bottom-up-depth-first*-Begriffsiterators enthält drei Listen: `past`, `current` und `future`. Jedes Mal, wenn im Verband ein Schritt nach oben gegangen wird, wird am Ende jeder Liste ein Element angefügt. Wenn im Verband ein Schritt zurück zu einem bereits besuchten Begriff gegangen wird, wird bei jeder Liste das letzte Element entfernt. In der `current`-Liste ist der aktuelle Pfad von dem kleinsten Begriff $\perp$ zum aktuellen Begriff gespeichert. Die `future`-Liste enthält Iteratoren über die oberen Nachbarn der Begriffe in der `current`-Liste. Die Elemente der `past`-Liste sind Mengen, die die Objektmengen der Begriffe enthalten, von denen alle Oberbegriffe bereits ausgegeben wurden. Genauer gesagt ist das i-te Element der `past`-Liste eine Menge, welche die Objektmengen der vollständig abgearbeiteten oberen Nachbarn des Begriffs enthält, der in der `current`-Liste an der i-ten Stelle steht.

In jedem Durchlauf der äußeren While-Schleife wird ein Begriff ermittelt, wobei die Schleife verlassen wird, sobald ein Begriff gefunden wurde, der zuvor noch nicht ausgegeben wurde. Dazu wird zuerst ein Begriff ermittelt, indem auf dem Begriffsiterator, der das letzte Element der `future`-Liste ist, die `next`-Methode aufgerufen wird. Anschließend wird überprüft, ob dieser Begriff zuvor bereits ausgegeben wurde. Dafür wird jedes Element der `past`-Liste darauf überprüft, ob darin eine Menge enthalten ist, die Teilmenge der Objektmenge des aktuellen Begriffs ist. Wenn das nicht der Fall ist, wurde der aktuelle Begriffe zuvor noch nicht ausgegeben. In diesem Fall wird bei allen drei Listen am Ende ein weiteres Element eingefügt. Am Ende der `current`-Liste wird der in diesem Durchlauf ermittelte Begriff hinzugefügt. Am Ende der `future`-Liste wird der Iterator über die oberen Begriffe dieses Begriffs hinzugefügt. Am Ende der `past`-Liste wird eine leere Menge hinzugefügt.

Die innere While-Schleife dient dazu, gegebenenfalls im Verband wieder zurück zu laufen. Das ist immer dann nötig, wenn es sich bei dem letzten Element in der `future`-Liste um einen Iterator handelt, der kein Element enthält. In diesem Fall ist sicher, dass alle Oberbegriffe des Begriffs, der am Ende der `current`-Liste steht, abgearbeitet wurden. Daher wird die Objektmenge dieses Begriffs zu der Menge hinzugefügt, die in der `past`-Liste an vorletzter Stelle steht. Anschließend wird bei jeder der drei Listen das letzte Element entfernt. An den Elementen der `current`-Liste und der `future`-Liste werden sonst keine Änderungen vorgenommen. Auch die tiefer liegenden Elemente der `past`-Liste werden in diesem Schleifendurchlauf nicht verändert. Die innere While-Schleife wird verlassen, wenn das letzte Element der `future`-Liste ein Iterator ist, der ein weiteres Element enthält, oder wenn die `future`-Liste leer ist.

Dadurch ist insbesondere sichergestellt, dass das letzte Element der `future`-Liste zu Beginn jedes Schleifendurchlaufs der äußeren While-Schleife ein nicht-leerer Iterator ist.

In Abbildung 4.3 sind die Knoten in der Reihenfolge markiert, in der sie vom *bottom-up-depth-first*-Iterator zurückgegeben werden. Abbildung 4.3 zeigt, wie die Berechnung des nächsten

Begriffs aussieht, wenn zuvor der Begriff (O_7, A_7) ausgegeben wurde.

Vor dem ersten Durchlauf der äußeren While-Schleife enthält die `current`-Liste den Pfad vom kleinsten Begriff $\bot = (O_1, A_1)$ über (O_2, A_2) und (O_6, A_6) zu (O_7, A_7). In der `future`-Liste ist für jeden dieser Begriffe ein Iterator über dessen obere Nachbarn abgelegt. Die Menge, die in der `past`-Liste an Position 1 steht, enthält die Objektmenge O_3, da (O_3, A_3) und alle seine Oberbegriffe bereits zuvor ausgegeben wurden.

Nach dem Eintritt in die While-Schleife wird durch den Aufruf von `next` auf dem Iterator an Position 3 der `future`-Liste ein Oberbegriff des Begriffs (O_7, A_7) bestimmt. In diesem Fall liefert `next` den Begriff (O_5, A_5). Nun wird überprüft, ob (O_5, A_5) zuvor bereits ausgegeben wurde. Da O_3 in der `past`-Liste enthalten ist und $O_5 \supseteq O_3$ gilt, ist sicher, dass (O_5, A_5) zuvor bereits ausgegeben wurde. Der Begriff darf also kein zweites Mal ausgegeben werden.

Da der Iterator an Position 3 jetzt kein weiteres Element mehr enthält, wird die innere While-Schleife ausgeführt. Da auch die Iteratoren an den Positionen 2 und 1 keine weiteren Elemente mehr enthalten, wird die innere While-Schleife dreimal durchlaufen, so dass jede Liste am Ende nur noch die Länge eins hat. Dabei wurde an Position 0 der `past`-Liste die Objektmenge O_2 eingefügt, da alle Oberbegriffe von (O_2, A_2) bereits ausgegeben wurden. Dies ergibt sich daraus, dass (O_2, A_2) in der `current`-Liste an Position 1 stand und die `future`-Liste an Position 1 leer war.

Da im ersten Durchlauf der äußeren While-Schleife kein neuer Begriff bestimmt werden konnte, wird der Schleifenrumpf noch einmal ausgeführt. Dazu wird wieder ein Begriff bestimmt, indem `next` auf dem Iterator aufgerufen wird, der an Position 0 der `future`-Liste gespeichert ist. Dieser Aufruf liefert den Begriff (O_8, A_8). Da in der `past`-Liste keine Objektmenge eines Unterbegriffs von O_8 ist, ist sicher, dass (O_8, A_8) zuvor noch nicht ausgegeben wurde.

Daher wird bei jeder Liste ein Element angefügt. In die `current`-Liste wird der soeben bestimmte Begriff (O_8, A_8) eingetragen, in die `future`-Liste der Iterator über dessen obere Nachbarn, der durch einen Aufruf von `upperNeighbors` auf dem zugrundeliegenden `Lattice`-Objekt bestimmt wird.

Da mit (O_8, A_8) ein Begriff gefunden wurde, der zuvor noch nicht ausgegeben wurde, wird die While-Schleife verlassen und dieser Begriff wird in dem privaten Feld `nextConcept` abgelegt.

Die Implementierung des *top-down-depth-first*-Begriffsiterators ist sehr ähnlich. Die einzigen Unterschiede bestehen darin, dass anstatt der oberen Nachbarn die unteren Nachbarn berechnet werden und dass anstatt der Objektmengen die Attributmengen in der `past`-Liste gespeichert werden.

4.8 Depth-first-Kanteniteratoren

Betrachtet man die `computeNext`-Methode aus Abbildung 4.2 genauer, fällt auf, dass bei jedem Durchlauf der äußeren While-Schleife genau eine Kante des Begriffsverbandes betrachtet wird. Da außerdem gilt, dass keine Kante zweimal betrachtet wird, lässt sich diese Methode leicht in eine Methode umwandeln, die die nächste Kante berechnet. Um die nächste Kante zu berechnen, muss der Rumpf der While-Schleife genau einmal ausgeführt werden.

4.9 Die Klasse ViolationIterator

Die Klasse `ViolationIterator` implementiert einen speziellen Iterator. Er liefert Paare von benachbarten Begriffen, die einander sehr ähnlich sind.

`ViolationIterator` bekommt im Konstruktor drei Zahlen übergeben: `supp`, `conf` und `diff`. Er iteriert nun über alle Begriffspaare (c_1, c_2) mit $c_1 = (O_1, A_1)$, $c_2 = (O_2, A_2)$, welche die folgenden Eigenschaften erfüllen:

- c_1 ist direkter Unterbegriff von c_2.
- $|O_1| \geq$ `supp`
- $|O_1|/|O_2| \geq$ `conf`
- $|A_1| - |A_2| \leq$ `diff`

Da nur über Paare aus benachbarten Begriffen iteriert wird und die Kanteniteratoren alle Paare benachbarter Begriffe liefern, ist es naheliegend, den `ViolationIterator` mit Hilfe eines Kanteniterators zu realisieren. Da die Objektmenge der Begriffe außerdem eine bestimmte Größe überschreiten muss, ist es sinnvoll, einen *top-down-breadth-first*-Kanteniterator zu verwenden.

Der `ViolationIterator` verwendet daher einen *top-down-breadth-first*-Kanteniterator, der die Kanten nach absteigender Größe der Objektmenge der jeweiligen Oberbegriffe traversiert. Der `ViolationIterator` ermittelt den nächsten auszugebenden Begriff, indem er so lange die `next`-Methode von dem Kanteniterator aufruft, bis diese ein Begriffspaar zurückgibt, das die geforderten Eigenschaften hat.

Sobald jedoch von der `next`-Methode ein Begriffspaar zurückgeliefert wird, in welchem die Objektmenge des Oberbegriffs weniger als `supp` Elemente enthält, ist sicher, dass vom Kanteniterator später kein Begriffspaar zurückgegeben wird, welches die geforderten Eigenschaften erfüllt. Daher muss für die Berechnung des `ViolationIterator` nicht der gesamte Verband berechnet werden.

4.10 Die Klasse HybridLattice

Die Klasse `HybridLattice` verwendet als interne Datenstruktur nicht die ursprüngliche `Relation`, die ihr im Konstruktor übergeben wird. Stattdessen wird aus der ursprünglichen `Relation` eine `HybridRelation` erstellt. In der Klasse `HybridRelation` werden die Informationen über den Kontext in Objekten der Klasse `HybridSet` gespeichert. `HybridSet` erweitert die Schnittstelle `ComparableSet` und verwendet als interne Datenstruktur Objekte der Klasse `java.util.BitSet`.

Die Verwendung von Bitsets als interne Datenstruktur bietet sich an, weil der Algorithmus zur Bestimmung gemeinsamer Objekte bzw. Attribute sehr viele Schnittoperationen auf Mengen ausführen muss. Diese Schnittoperationen können auf Bitsets effizient durchgeführt werden.

Für die Repräsentation der Mengen durch Bitsets ist es erforderlich, dass die Größe der Objekt- und Attributmenge des Kontextes bekannt ist. Der Anwender erstellt daher keine `HybridRelation`, sondern eine `HashRelation` oder eine `TreeRelation` und übergibt diese dem Konstruktor von `HybridLattice`. Dieser sorgt dann dafür, dass diese `Relation` in eine `HybridRelation` übersetzt wird.

Dafür wird eine `HybridRelation` erzeugt, der im Konstruktor die ursprüngliche `Relation` übergeben wird. In diesem Konstruktor findet dann die Übersetzung der ursprünglichen `ComparableSet`-Objekte in `HybridSet`-Objekte statt.

Für die Übersetzung der ursprünglichen durch `Comparable`-Objekte repräsentierten Objekte und Attribute werden zwei bijektive Abbildungen benötigt. Diese bilden die ursprünglichen `Comparable`-Objekte auf $\{0, \ldots, |\mathcal{O}| - 1\}$ bzw. $\{0, \ldots, |\mathcal{A}| - 1\}$ ab. Diese bijektiven Abbildungen werden jeweils in einem `HybridTranslator`-Objekt erstellt und gespeichert.

Die `HybridSet`-Objekte verhalten sich nach außen hin so, wie die ursprünglichen `ComparableSet`-Objekte. Intern ist die Information jedoch in einem Bitset gespeichert. Eingaben und Ausgaben werden mit Hilfe des `HybridTranslators` übersetzt.

Der große Vorteil besteht nun darin, dass Schnittoperationen sehr effizient ausgeführt werden können. Zwei `HybridSet`-Objekte, denen der gleiche `HybridTranslator` zugrundeliegt, können geschnitten werden, indem man die `and`-Methode auf ihren Bitsets anwendet.

4.11 Die Klasse BitsetLattice

Die Klasse `BitsetLattice` verwendet für Berechnungen intern eine auf Bitsets basierende Datenstruktur. Im Vergleich mit `HybridLattice` führt `BitsetLattice` jedoch weniger Übersetzungen zwischen interner und externer Datenstruktur durch.

Die Klasse `BitsetLattice` verwendet intern für Berechnungen Objekte der Klassen `BitsetRelation`, `BitsetConcept` und `ComparableBitset`. In der Interaktion mit dem Endbenutzer werden jedoch Objekte von `Relation`, `Concept` und `ComparableSet` verwendet. Die Klasse `Translator` dient dazu, zwischen diesen Darstellungen zu übersetzen.

`BitsetLattice` verwendet für die eigentliche Berechnung ausschließlich die interne Datenstruktur. Nur unmittelbar nachdem eine Methode von außen aufgerufen wurde und bevor ein Objekt nach außen gegeben wird, findet eine Übersetzung von der einen in die andere Datenstruktur statt. Bei `HybridLattice` wird dagegen teilweise während der Ausführung des eigentlichen Algorithmus zwischen interner und externer Datenstruktur übersetzt.

Durch dieses Verhalten von `BitsetLattice` lassen sich zwar unnötige Übersetzungen zwischen interner und externer Datenstruktur vermeiden. Da `BitsetLattice` im Gegensatz zu `HybridLattice` jedoch selbst jede Ein- und Ausgabe übersetzen muss, ist es nicht möglich, dass `BitsetLattice` die in der abstrakten Klasse `LatticeImpl` implementierten Methoden verwendet.

Kapitel 5

Evaluierung

Damit die Implementierung ihren Zweck erfüllt, ist es wichtig, dass die Berechnungen korrekt sind. Außerdem ist es wünschenswert, dass die Performance gut ist. Daher wurden zahlreiche Tests und Messungen durchgeführt, mit denen Korrektheit und Performance getestet wurden.

Für die Korrektheitstests ist es naheliegend, die Berechnungen von Colibri/Java mit den Berechnungen eines anderen Begriffsanalyse-Programmes zu vergleichen. Um die Performance richtig einschätzen zu können, ist ein Vergleich mit anderen Implementierungen ebenfalls sinnvoll.

Als Referenzimplementierungen dienten die Programme Colibri/ML (Lindig, 2007) und Concepts (Lindig, 1999a). Das Programm Concepts ist ein in der Programmiersprache C geschriebenes Programm für formale Begriffsanalyse. Für die Berechnung des Begriffsverbandes verwendet es den Algorithmus von Ganter (Ganter and Wille, 1999). Colibri/ML ist ein in Objective Caml geschriebenes Programm. Es berechnet den Begriffsverband mit dem in Abschnitt 2.3 vorgestellten Algorithmus.

5.1 Grundlagen

Für Tests und Messungen werden Kontexte als Eingabe benötigt. Um die Testläufe automatisieren zu können, wurde eine Java-Applikation erstellt, die Kontexte generiert. Dieser Applikation (`RandomRelationWriter`) werden die gewünschten Anzahlen von Objekten und Attributen sowie die gewünschte Dichte der Relation übergeben. Die Applikation erstellt dann Objekt- und Attributmengen in der angegebenen gewünschten Größe und wählt zufällig Paare, die zur Relation hinzugefügt werden. Der erzeugte Kontext wird schließlich in einer Datei gespeichert.

Im Übrigen war es nötig, eine Java-Applikation zu implementieren, welche einen Kontext aus einer Datei einliest, die zu überprüfenden Berechnungen von der Begriffsanalyse-Bibliothek Colibri/Java ausführen lässt und die Ergebnisse in angemessener Form ausgibt. Diese Applikation (`Analyzer`) wird mit Parametern aufgerufen, die angeben, welche Berechnung ausgeführt werden soll und welche Implementierungen von `Lattice` und `Relation` dabei verwendet werden sollen.

5.2 Korrektheit

Um sicherzustellen, dass die Implementierung die Berechnungen korrekt ausführt, sollte man die Korrektheit im Idealfall beweisen. Da der Aufwand dafür jedoch zu groß wäre, wurde eine Validierung durchgeführt. Diese erfolgte durch manuell erstellte JUnit-Tests und durch Vergleiche mit den Referenzimplementierungen.

Vergleiche mit Referenzimplementierungen wurden für die Berechnungen der Iteratorklassen durchgeführt, um auch mögliche Fehlerquellen zu testen, die von internen Tests nicht abgedeckt werden können. Mit internen Tests könnte zwar getestet werden, dass jeder berechnete Begriff tatsächlich ein Begriff ist und dass jedes Begriffspaar, das von einem Kanteniterator zurückgegeben wird, ein Paar aus Unter- und Oberbegriff ist. Allerdings wären die Fälle, dass nicht alle Begriffe oder Kanten berechnet werden, von solchen internen Tests nicht abgedeckt. Ebenso könnten Fälle nicht aufgedeckt werden, in denen vom Kanteniterator ein Begriffspaar zurückgegeben wird, in dem der eine Begriff nur indirekter Oberbegriff des anderen ist.

Um die Korrektheit der Kanteniteratoren zu testen, wurden die Berechnungen der Java-Implementierung mit den Berechnungen des Programms Concepts verglichen. Die Tests erfolgten automatisiert mit Hilfe eines Python-Skripts. Zunächst wurden mit Hilfe des `RandomRelationWriter` zufällige Kontexte verschiedener Größe und Dichte erstellt. Die Größe der einzelnen Relationen wurde dabei auf 5000 beschränkt, da die Begriffsverbände für größere Kontexte zu groß sind, um die Gleichheit zweier Berechnungen effizient zu testen.

Für jeden erstellten Kontext wurde die Verbandsstruktur von den Kanteniteratoren der verschiedenen `Lattice`-Implementierungen berechnet. Das Ergebnis wurde jeweils in einer Datei gespeichert. Das Programm Concepts berechnete ebenfalls die Verbandsstruktur für diesen Kontext.

Um die Korrektheit zu testen, musste die Berechnung von Colibri/Java mit der Berechnung des Programms Concepts verglichen werden. Da Traversierung und Ausgabeformate jedoch unterschiedlich sind, genügte es nicht, die Dateien auf Gleichheit zu testen. Daher wurde eine weitere Java-Applikation erstellt, der jeweils eine mit Concepts und eine mit der Java-Implementierung erstellte Datei übergeben wurde. Diese beiden Dateien wurden eingelesen und die Menge der enthaltenen Kanten wurde auf Gleichheit getestet. Um zu testen, dass die Kanteniteratoren keine Kante doppelt zurückgeben, wurde während des Einlesens überprüft, dass keine Kante doppelt in der Datei gespeichert war.

Durch diese Tests wurde auch ein Fehler in der Implementierung der Methode `commonAttributes` von `BitsetRelation` aufgedeckt, der bei *top-down*-Iteration in einigen Fällen eine fehlerhafte Berechnung des Begriffsverbandes verursachte.

Die Korrektheit der Begriffsiteratoren wurde durch ein ähnliches Verfahren getestet.

Um die Korrektheit der Klasse `ViolationIterator` zu testen, wurden die Berechnungen mit den Berechnungen des Programms Colibri/ML verglichen. Dazu wurden Aufrufrelationen statt zufällig erzeugter Kontexte verwendet. Der Grund dafür liegt darin, dass die Begriffsverbände zufällig erzeugter Kontexte nur sehr wenige benachbarte Begriffe enthalten, welche die Kriterien erfüllen, um vom `ViolationIterator` zurückgegeben zu werden.

Auch dieser Test wurde mit einem Python-Skript durchgeführt. Es startete `Analyzer` und Colibri/ML und verglich anschließend die beiden Berechnungen miteinander.

5.3 Performance

Für die Performance-Messungen wurden mit dem `RandomRelationWriter` erstellte Kontexte verwendet. Die Größe des Begriffsverbands des Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$, und damit auch die Laufzeit, hängt dabei im Wesentlichen von der Größe ($|\mathcal{R}|$) und von der Dichte ($|\mathcal{R}|/|\mathcal{O} \times \mathcal{A}|$) ab. Dabei gilt, dass ein größerer Kontext typischerweise mehr Begriffe besitzt als ein kleinerer, wenn beide die gleiche Dichte haben. Eine Versuchsreihe von Lindig (1999b, S. 128) hat ergeben, dass von zwei Kontexten gleicher Größe typischerweise der Kontext mit der größeren Dichte die größere Anzahl von Begriffen besitzt.

Da in der praktischen Anwendung normalerweise dünn besetzte Kontexte eine Rolle spielen, wurden Testreihen mit Relationen der Dichte 1 Prozent, 2 Prozent und 4 Prozent durchgeführt. Die Größe der Relationen wurde dabei variiert. Damit die Daten untereinander vergleichbar sind, wurden jedoch nur Kontexte verwendet, deren Objekt- und Attribumengen jeweils die gleiche Größe haben.

Die Performance-Messungen wurden mit Hilfe eines Python-Skripts automatisiert durchgeführt. Für jeden von `RandomRelationWriter` erzeugten Kontext wurden Laufzeitmessungen durchgeführt. Dabei wurden sowohl die Laufzeiten der Kanteniteratoren der verschiedenen `Lattice`-Implementierungen, als auch die Laufzeiten von Concepts und Colibri/ML gemessen. Die gemessenen Laufzeiten wurden in Dateien gespeichert, um sie später auswerten zu können. Die Abbildungen 5.1, 5.2 und 5.3 zeigen die Ergebnisse dieser Laufzeitmessungen.

Es fällt auf, dass `BitsetLattice` und `HybridLattice` im Vergleich zu `RawLattice` deutlich weniger Zeit für die Berechnung des Begriffsverbandes benötigen. Bei Kontexten der Dichte 1 Prozent ist die Laufzeit von `RawLattice` in der Regel ab einer Relationsgröße von 16000 mindestens dreimal so lang, wie die Laufzeit von `BitsetLattice` oder `HybridLattice`. Der Grund dafür dürfte darin liegen, dass Schnittoperationen auf Hashsets und Treesets nicht so effizient durchgeführt werden können, wie auf Bitsets. Schnittoperationen werden jedoch häufig zur Berechnung der gemeinsamen Objekte bzw. Attribute benötigt.

Bei dem Vergleich zwischen `RawLattice` mit zugrundeliegender `TreeRelation` und `RawLattice` mit zugrundeliegender `HashRelation` ergibt sich kein einheitliches Bild. Auf Kontexten der Dichte 1 Prozent ist die Verwendung von `TreeRelation` etwas effizienter als die Verwendung von `HashRelation` (Abbildung 5.1). Auf Kontexten der Dichte 4 Prozent ist jedoch kein Unterschied zwischen `RawLattice` mit `HashRelation` und `RawLattice` mit `TreeRelation` erkennbar (Abbildung 5.3). Da `RawLattice` im Vergleich mit den anderen Implementierungen sehr ineffizient ist, wurde jedoch auf eine genauere Untersuchung verzichtet.

Da Concepts für die Berechnung des Begriffsverbands einen anderen Algorithmus verwendet, ist auch ein Vergleich zwischen Concepts und den anderen Implementierungen interessant. Aus Abbildung 5.4 geht hervor, dass Concepts den Begriffsverband für kleine, dünn besetzte Kontexte am schnellsten von allen Implementierungen berechnet. Für die Berechnung aller Begriffe eines Kontextes der Dichte ein Prozent mit 10000 Paaren benötigt Concepts beispielsweise nur 5 Sekunden. `BitsetLattice` benötigt dagegen mit 13 Sekunden für die gleiche Berechnung mehr als doppelt so lang.

Auch für Kontexte mit einer Dichte von einem Prozent (Abbildung 5.1) und 30000 Paaren ist die Laufzeit von Concepts (175 Sekunden) noch vergleichbar mit den Laufzeiten von Colibri/ML (269 Sekunden), `BitsetLattice` (91 Sekunden) und `HybridLattice` (104 Sekunden). Bei großen, dichter besetzten Kontexten benötigt Concepts jedoch für die Berechnung deutlich länger als alle anderen Implementierungen (Abbildung 5.3). Beispielsweise benötigt Concepts

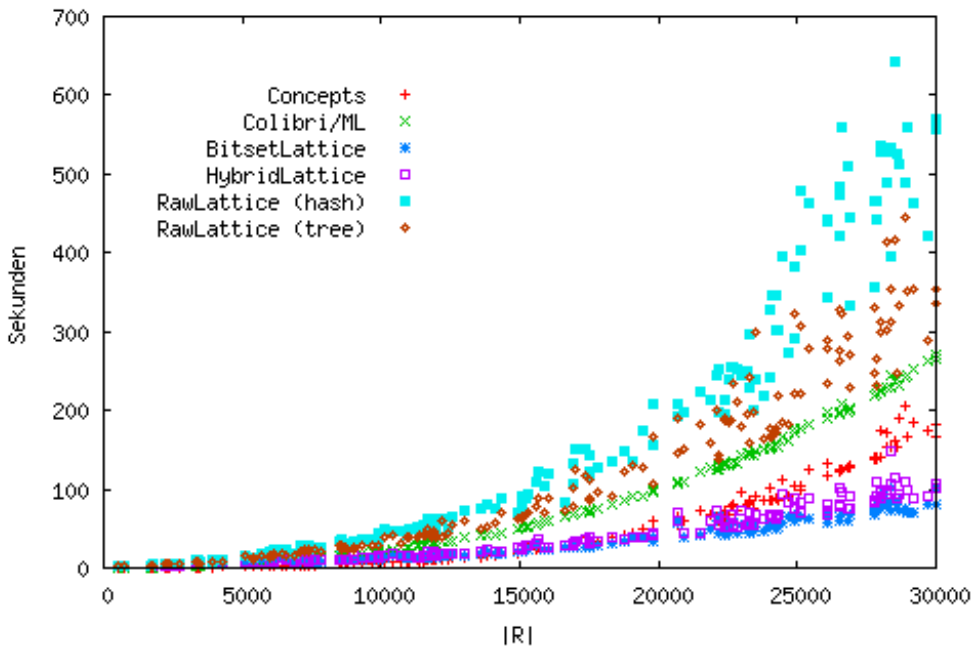


Abbildung 5.1: Laufzeiten der verschiedenen Implementierungen für zufällige Kontexte der Dichte 1 Prozent.

für die Berechnung aller Begriffe eines Kontextes der Dichte 4 Prozent mit 20000 Paaren 1378 Sekunden und damit über zwanzig mal so lang wie `HybridLattice`, welcher nur 65 Sekunden für diese Berechnung benötigt.

Abbildung 5.5 zeigt einen Vergleich zwischen Colibri/ML, `BitsetLattice` und `HybridLattice`. Ein direkter Vergleich dieser Implementierungen bietet sich an, da diese Implementierungen alle auf dem gleichen Algorithmus basieren und als interne Datenstruktur Bitsets verwenden. Der Vergleich zeigt, dass `BitsetLattice` und `HybridLattice` bei großen Kontexten weniger Zeit brauchen als Colibri/ML. Bei Kontexten der Dichte 1 Prozent benötigt `HybridLattice` ab einer Relationsgröße von 16000 in der Regel höchsten halb so lang wie Colibri/ML. Dabei ist die Laufzeit von `HybridLattice` in der Regel etwas länger als die Laufzeit von `BitsetLattice`.

Bei kleineren Kontexten ist die Laufzeit von Colibri/ML jedoch geringer als die Laufzeit der Java-Implementierungen (Abbildung 5.4). Dies dürfte auf die Startup-Zeiten von Java zurückzuführen sein. Die Programme Concepts und Colibri/ML liegen als Maschinencode vor, so dass bei ihnen keine Startup-Zeiten anfallen.

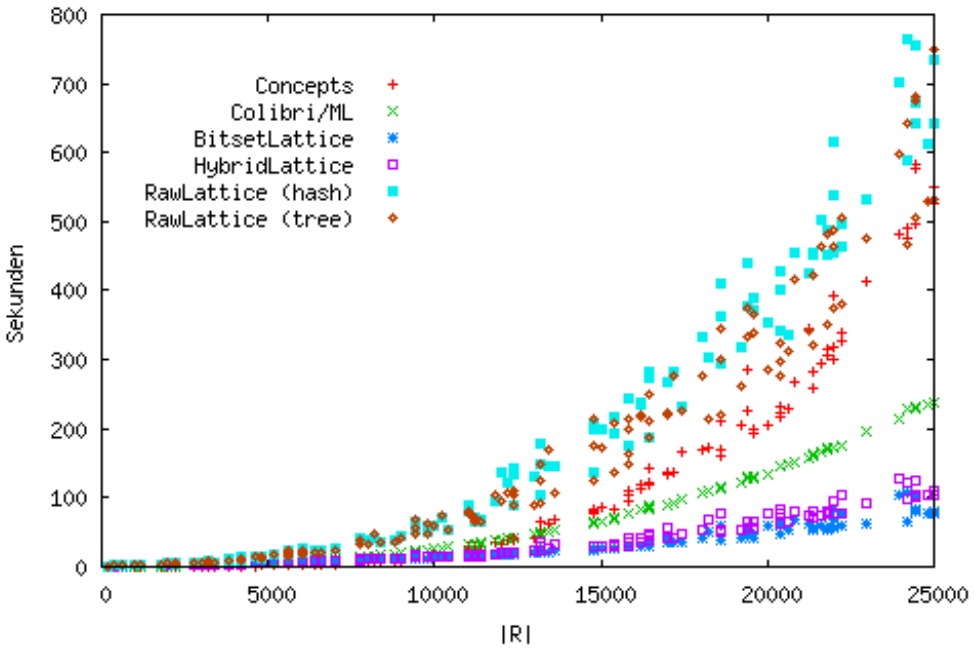


Abbildung 5.2: Laufzeiten der verschiedenen Implementierungen für zufällige Kontexte der Dichte 2 Prozent.

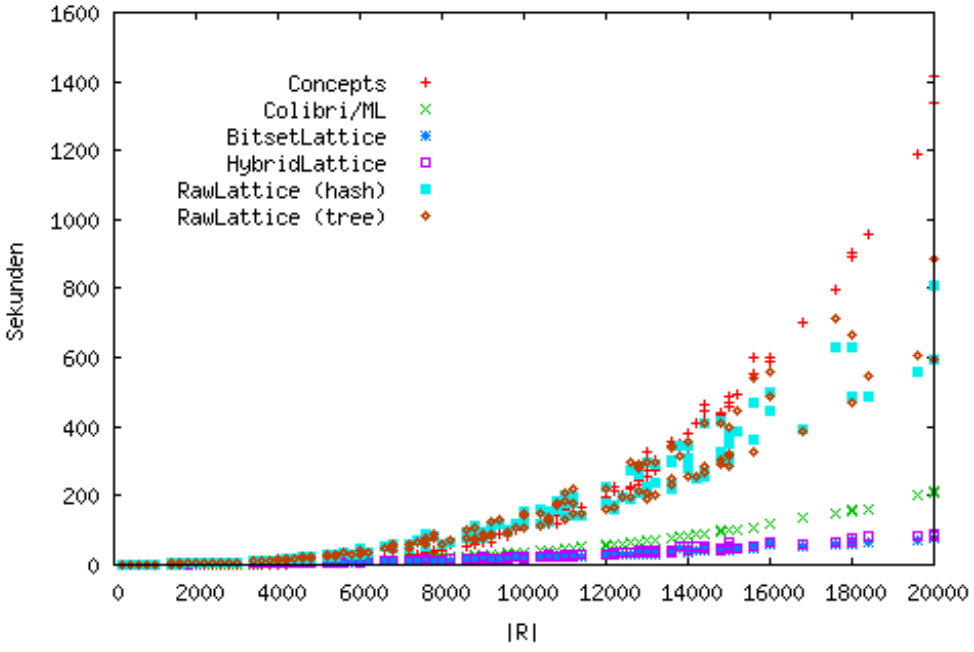


Abbildung 5.3: Laufzeiten der verschiedenen Implementierungen für zufällige Kontexte der Dichte 4 Prozent.

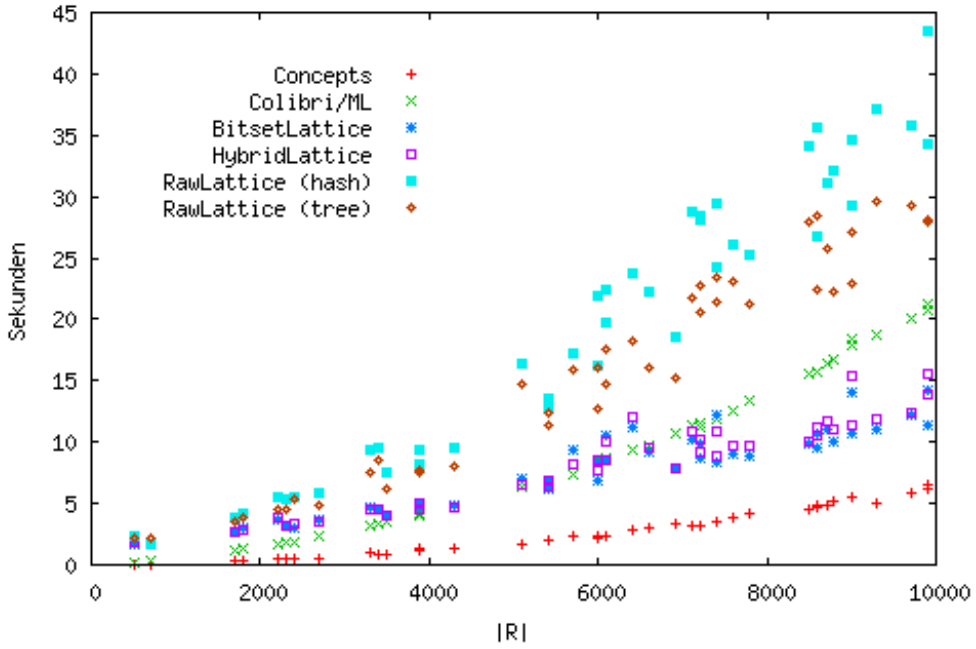


Abbildung 5.4: Laufzeiten der verschiedenen Implementierungen für zufällige Kontexte der Dichte 1 Prozent.

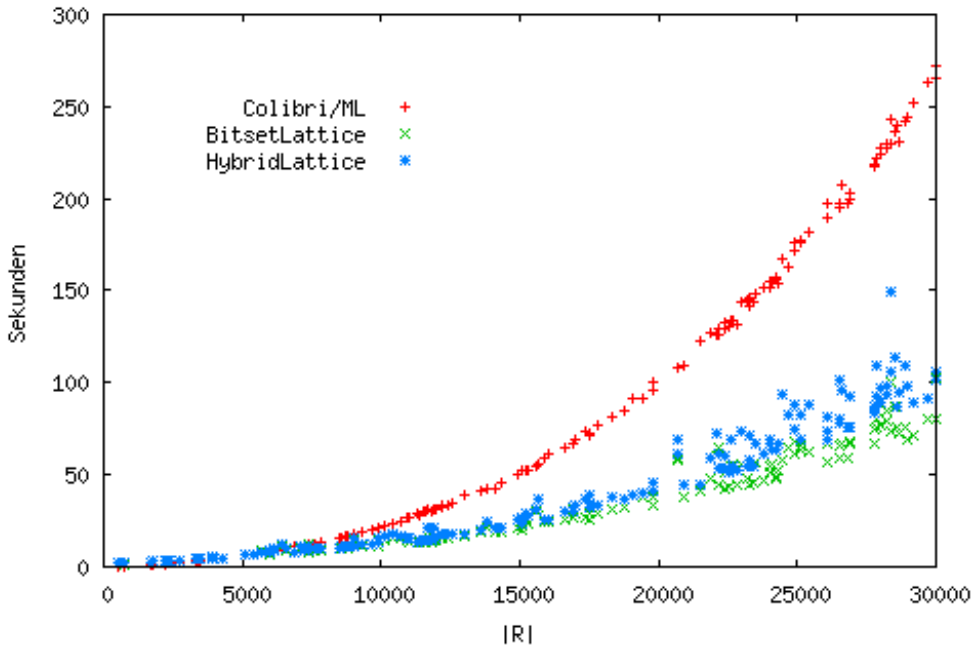


Abbildung 5.5: Laufzeiten der verschiedenen Implementierungen für zufällige Kontexte der Dichte 1 Prozent.

Kapitel 6

Ergebnisse

Die Schnittstelle `Lattice` definiert die für die Begriffsanalyse wichtigen Methoden. Insgesamt existieren drei konkrete Klassen, welche die Schnittstelle `Lattice` implementieren: `RawLattice`, `HybridLattice` und `BitsetLattice`. Während `RawLattice` zur Berechnung des Verbands die ihm im Konstruktor übergebene `Relation` direkt benutzt, verwenden `BitsetLattice` und `HybridLattice` intern eine auf Bitsets basierende Datenstruktur.

Für die Berechnung aller Begriffe wurden Begriffsiteratoren implementiert. Die sogenannten Kanteniteratoren dienen zur Berechnung der Verbandsstruktur. Sie iterieren über alle Paare von Begriffen, die zueinander in einer direkten Ober-/Unterbegriffsbeziehung stehen, also genau über die Paare von Begriffen, die im zum Verband gehörenden Hasse-Diagramm durch eine Kante verbunden sind. Darüber hinaus hat der Anwender die Möglichkeit, den Begriffsverband manuell mit den Methoden `upperNeighbors` und `lowerNeighbors` zu durchwandern.

Durch die Verwendung von Iteratoren muss der Verband nicht komplett vorberechnet werden. Das ist sinnvoll, da Begriffsverbände sehr groß werden können. Wenn der Endbenutzer nur an einem Teil des Verbands interessiert ist, kann er die Iteration vorzeitig abbrechen. Da der Verband nicht vorberechnet wird, lassen sich so unnötige Berechnungen vermeiden.

6.1 Vergleich der Implementierungen

Ein Vorteil von Colibri/Java gegenüber den anderen Programmen liegt darin, dass die Java-Bibliothek direkt in Java-Programmen eingesetzt werden kann. Dies ist für den Anwender bequemer, da er kein externes Tool zur Berechnung des Begriffsverbandes aufrufen muss. Colibri/Java erlaubt es dem Anwender auch, die Berechnung stärker an die Bedürfnisse der jeweiligen Anwendung anzupassen. Beispielsweise ist es möglich, die Iteration an einer bestimmten Stelle abzubrechen, so dass nicht der gesamte Verband berechnet werden muss. Außerdem kann der Begriffsverband mit den Methoden `upperNeighbors` und `lowerNeighbors` manuell durchwandert werden.

Auch im Hinblick auf die Laufzeit scheint Colibri/Java für praktische Anwendungen geeignet zu sein. Hier ist jedoch ein deutlicher Unterschied zwischen den verschiedenen Implementierungen von `Lattice` zu erkennen. Während die Performance von `BitsetLattice` und `HybridLattice` auf großen Kontexten besser als die Performance von Colibri/ML ist, benötigt `RawLattice` für die Berechnungen deutlich länger. Bei Kontexten der Dichte 1 Prozent gilt, dass

die Laufzeit von `RawLattice` in der Regel ab einer Relationsgröße von 16000 mindestens dreimal so lang ist, wie die Laufzeit von `BitsetLattice` oder `HybridLattice`. Dies dürfte darauf zurückzuführen sein, dass die häufig ausgeführten Schnittoperationen auf Bitsets effizienter als auf Hashsets oder Treesets durchgeführt werden können. Für praktische Anwendungen bietet es sich also an, `BitsetLattice` oder `HybridLattice` anstatt `RawLattice` zu verwenden.

`BitsetLattice` schneidet im Hinblick auf die Laufzeit etwas besser ab als `HybridLattice`. Dies dürfte daran liegen, dass bei `HybridLattice` einige Übersetzungen zwischen interner und externer Datenstruktur vorgenommen werden, die bei `BitsetLattice` nicht erforderlich sind.

Auf großen Kontexten sind `HybridLattice` und `BitsetLattice` deutlich effizienter als Concepts und Colibri/ML. In der Regel gilt für Kontexte der Dichte 1 Prozent, die mindestens 16000 Paare enthalten, dass die Laufzeit von `HybridLattice` höchstens halb so lang wie die Laufzeit von Colibri/ML ist. Für Kontexte der Dichte 4 Prozent, die mindestens 10000 Paare enthalten, benötigt `HybridLattice` für die Berechnung in der Regel weniger als ein Fünftel der Zeit, die das Programm Concepts benötigt.

Beispielsweise benötigt `HybridLattice` für die Berechnung aller Begriffe einer zufällig erzeugten Relation der Dichte 2 Prozent mit 25000 Paaren 108 Sekunden. Für die gleiche Berechnung benötigt `BitsetLattice` 79 Sekunden. Die Referenzimplementierungen benötigen dafür 538 Sekunden (Concepts) bzw. 237 Sekunden (Colibri/ML).

Bei kleinen, dünn besetzten Kontexten ist die Laufzeit von Concepts und Colibri/ML dagegen kürzer als die Laufzeit von Colibri/Java. Insbesondere wenn die Begriffsverbände für viele kleine, dünn besetzte Kontexte berechnet werden sollen, bieten diese beiden Programme also Vorteile gegenüber den Java-Implementierungen.

6.2 Ausblick

Obwohl Colibri/Java die Hauptfunktionalität für formale Begriffsanalyse bereits bereitstellt, gibt es eine Reihe möglicher Erweiterungen.

Es ist bereits möglich, nur einen Teil des Verbandes berechnen zu lassen, indem eine Iteration über Begriffe oder Kanten vorzeitig abgebrochen wird. Dadurch lassen sich für die jeweilige Anwendung unnötige Berechnungen vermeiden. Der Beginn der Iteration lässt sich jedoch noch nicht beeinflussen. Jede Iteration beginnt entweder beim kleinsten Begriff $\bot$ (bei *bottom-up*-Iteration) oder beim größten Begriff $\top$ (bei *top-down*-Iteration). Für bestimmte Anwendungen könnte es jedoch wünschenswert sein, von einem anderen Begriff über einen Teilverband zu iterieren, wenn die anderen Teile des Verbandes für diese Anwendung nicht von Interesse sind. Diese Funktionalität ließe sich leicht einbauen. Dazu müsste den Iteratorklassen lediglich ein weiterer Konstruktor hinzugefügt werden, der zusätzlich den Begriff übergeben bekommt, von dem aus die Iteration beginnen soll.

Weitere spezielle Iteratoren sind vorstellbar. Beispielsweise wäre es denkbar, einen veränderten *bottom-up-breadth-first*-Begriffsiterator zu implementieren, der im Konstruktor zusätzlich eine Menge von Attributen übergeben bekommt. Der so konstruierte Iterator könnte für jeden entdeckten Begriff überprüfen, ob dieser Begriff mindestens eines dieser Attribute enthält. Nur falls das der Fall ist, würde dieser Begriff zur Agenda hinzugefügt und zurückgegeben. Solch ein Iterator wäre sinnvoll, wenn eine Anwendung sich nur für Begriffe interessiert, die mindestens eines dieser Attribute enthalten.

Dies ließe sich noch verallgemeinern. Man könnte eine Schnittstelle definieren, die eine Methode definiert, welche ein `Concept`-Objekt übergeben bekommt und `boolean` als Rückgabetyp hat. Ähnlich wie `java.util.Comparator` eine Schnittstelle ist, die dazu dient, einem `SortedSet` die gewünschte Sortierung mitzuteilen, könnte diese Schnittstelle dazu dienen, einem Iterator mitzuteilen, welche Begriffe zurückgegeben und in die Agenda eingefügt werden sollen und welche nicht. Dadurch ließen sich in manchen Fällen unnötige Berechnungen vermeiden.

Eine weitere mögliche Erweiterung wäre eine zusätzliche Klasse, welche die Schnittstelle `Lattice` implementiert und die Verbandsstruktur intern speichert. Die Speicherung der Verbandsstruktur wurde zwar bewusst vermieden, da Begriffsverbände sehr groß werden können. Allerdings könnte es für bestimmte Anwendungen, die vorwiegend mit kleinen Verbänden arbeiten, sinnvoll sein, die Verbandsstruktur zu speichern, um Berechnungen nicht mehrfach ausführen zu müssen.

Literaturverzeichnis

Silvia Breu, Thomas Zimmermann, and Christian Lindig. Mining eclipse for cross-cutting concerns. In Stephan Diehl, Harald Gall, and Ahmed E. Hassan, editors, *MSR*, pages 94–97. ACM, 2006. ISBN 1-59593-397-2. URL `http://doi.acm.org/10.1145/1137983.1138006`.

Bernhard Ganter and Rudolf Wille. *Formale Begriffsanalyse—Mathematische Grundlagen.* Springer, Berlin/Heidelberg, 1996.

Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer, Berlin – Heidelberg – New York, 1999.

Christian Lindig. Fast concept analysis. In Gerhard Stumme, editor, *Working with Conceptual Structures - Contributions to ICCS 2000*, pages 152–161, Aachen, Germany, August 2000. Shaker Verlag. ISBN ISBN 3-8265-7669-1.

Christian Lindig. Colibri, 2007. URL `http://code.google.com/p/colibri-ml/`. Open-source tool for concept analysis, implements algorithm from Lindig (2000).

Christian Lindig. Concepts, 1999a. URL `http://code.google.com/p/colibri-concepts/`. Command line application that computes for a binary relation a so-called concept lattice.

Christian Lindig. *Algorithmen zur Begriffsanalyse und ihre Anwendung bei Softwarebibliotheken.* PhD thesis, Technische Universitt Braunschweig, D–38106 Braunschweig, Germany, November 1999b.