Universität Siegen
Fachbereich 12 - Elektrotechnik und Informatik

Diplomarbeit

Einsatz multidimensionaler Suchstrukturen zur Optimierung der Bestimmung von Dokumentdifferenzen

Christoph Treude

Erstprüfer: Prof. Dr. Udo Kelter, Universität Siegen
Zweitprüfer: Prof. Dr. Wolfgang Merzenich, Universität Siegen

Siegen, im März 2007

For the wise man looks into space and he knows there is no limited dimensions.

– Lao-tse

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abstract

Due to the trend to model-driven engineering, difference tools for model documents have recently gained relevance. During the development of the underlying models, especially differences between various versions of the same document and differences between documents which are edited by multiple developers are important to the development process.

Calculating the differences between model documents is much more complex than to determine differences between text documents. Usually, model documents are treated as typed graphs with each model element being represented by a node. During the actual calculation, all elements from the respective documents are compared pairwise, grouped by their types. This implicates quadratic runtime behavior.

In order to reduce the runtime, this thesis presents an in-memory high-dimensional balanced data structure called S^3V tree for the management of model elements. The tree arranges similar elements next to each other. Thus, not all elements have to be considered for finding elements that are similar to a given one.

Since the S^3V tree is a data structure for numeric vectors, model elements have to be mapped onto these vectors. This mapping is accomplished by both metrical and lexical indices. Metrical indices are based on the concept of software metrics and represent certain characteristics of elements as numbers. The intention of lexical indices is to detect similarities between names; this is achieved by using names and name parts as indices for the vectors.

Furthermore, this thesis presents several ways in which S^3V trees can be integrated into the calculation of differences. The concluding evaluation of the approach especially with large UML class diagrams shows that the tree-based algorithm is able to reduce the runtime of the difference calculation by a factor of up to 50 for large documents.

Zusammenfassung

Durch den Trend zu modellbasierter Entwicklung hat die Bedeutung von Differenzwerkzeugen für Modelldokumente stark zugenommen. Differenzen sind besonders zwischen unterschiedlichen Versionen eines Dokuments und bei Beteiligung mehrerer Personen an der Entwicklung eines Dokuments von Interesse.

Die Berechnung von Differenzen zwischen nicht-textuellen Dokumenten gestaltet sich deutlich komplexer als zwischen reinen Textdokumenten. Dabei werden Modelldokumente meist als getypte Graphen interpretiert, in denen die einzelnen Elemente der Dokumente durch Knoten repräsentiert werden. Im Zuge der Differenzberechnung werden alle Elemente der zu vergleichenden Dokumente paarweise und gruppiert nach ihren Typen verglichen.

Um die daraus resultierende quadratische Laufzeit, die vor allem für größere Dokumente inakzeptabel ist, zu verringern, wird in dieser Arbeit mit dem S^3V Baum eine hauptspeicherbasierte, hochdimensionale und balancierte Datenstruktur zur Verwaltung von Dokumentelementen vorgestellt, die ähnliche Elemente benachbart anordnet. Somit müssen nicht alle Elemente betrachtet werden, um zu einem gegebenen Element die Menge der ähnlichen Elemente in anderen Dokumenten zu finden.

Da der S^3V Baum numerische Vektoren verwaltet, werden weiterhin mit lexikalischen und metrischen Indizes zwei Mechanismen dargestellt, mit denen Dokumentelemente auf Vektoren abgebildet werden können. Metrische Indizes basieren auf dem Konzept von Software-Metriken und bilden Eigenschaften der Elemente auf Zahlen ab. Das Ziel lexikalischer Indizes ist die Erfassung von Namensähnlichkeiten durch den Einsatz von Namen bzw. Namensteilen als Vektorindizes.

Für die Integration der Datenstruktur in die Differenzberechnung ergeben sich mehrere Alternativen, die in dieser Arbeit aufgezeigt werden. Die abschließende Evaluierung des Ansatzes vor allem mit UML-Klassendiagrammen hat ergeben, dass die Laufzeit für große Dokumente teilweise um einen Faktor von bis zu 50 reduziert werden kann.

1 Einleitung und Motivation

Technische Dokumente sind heute für viele Geschäftsprozesse von essentieller Bedeutung. Eine angemessene Dokumentation und Abstraktion, wie sie in technischen Dokumenten aller Art stattfinden kann, ermöglichen die Zusammenarbeit mehrerer Entwickler sowie die Präzisierung von Anforderungen in einem Umfeld, das von stets wachsender Komplexität gekennzeichnet ist. Als Beispiele für technische Dokumente können in diesem Zusammenhang die Ergebnisse rechnerunterstützter Konstruktion (CAD) in Architektur oder Anlagenbau, UML-Spezifikationen in der Software-Entwicklung oder Simulink-Diagramme für Systemmodellierungen genannt werden.

Ein charakteristisches Merkmal technischer Dokumente ist deren Versionierung. Oft erstreckt sich die Entwicklung über einen längeren Zeitraum, so dass die zugehörige Dokumentation fortwährend angepasst wird und unterschiedliche Versionen eines Dokuments in einer eindeutigen zeitlichen Beziehung zueinander stehen. Darüber hinaus können durch das parallele Bearbeiten durch unterschiedliche Entwickler mehrere Versionen in parallelen Entwicklungszweigen entstehen.

In beiden Fällen ist es während der Entwicklung entscheidend, sich über die Unterschiede zwischen unterschiedlichen Versionen eines Dokuments im Klaren zu sein. Während für textuelle Dokumente wie Textdateien oder Quelltexte eine Vielzahl von Differenzwerkzeugen offeriert wird, ist das Angebot auf dem Markt für die Differenzberechnung nicht-textueller Dokumente deutlich kleiner.

Im Gegensatz zu textuellen Dokumenten weisen nicht-textuelle Dokumente, die oft graphisch dargestellt werden, einen hohen Grad an Strukturiertheit auf, der die Differenzberechnung stark beeinflusst. Während reine Textdokumente lediglich aus einer Liste von Zeilen bestehen, die wiederum Zeichen enthalten, setzen sich nicht-textuelle Dokumente aus einer Vielzahl von Elementen mit unterschiedlichem Typ zusammen, die zueinander in Beziehung stehen können. Konkrete Beispiele für diese Elemente sind Schaltzeichen in Schaltplänen, Punkte und Linien in zweidimensionalen CAD-Dokumenten oder Klassen und Operationen in UML-Klassendiagrammen.

Um die Differenz zwischen nicht-textuellen Dokumenten zu berechnen, muss zunächst untersucht werden, welche Elemente aus verschiedenen Dokumenten miteinander korrespondieren. So definiert [Kel06] die Differenz zwischen zwei Dokumenten, deren Dokumentstruktur eine Multimenge ist[1], als eine Menge von Korrespondenzen zwischen Elementen. Zusätzliche Anforderungen ergeben sich durch die nichttriviale Struktur der Dokumente.

Die Vielzahl von Elementen, die zu einem Dokument gehört, wirft die Frage auf,

[1] Eine Multimenge unterscheidet sich von einer Menge darin, dass ein Element in einer Multimenge mehrfach vorkommen kann.

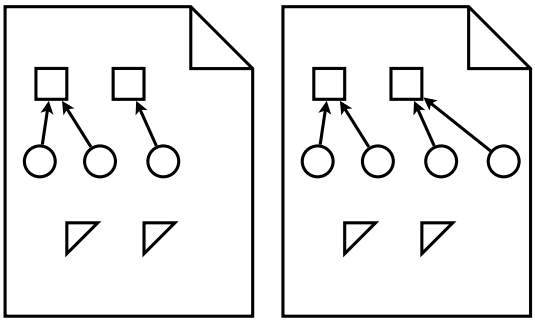

Abbildung 1.1: Symbolische Darstellung zweier Dokumente mit Elementen

wie diese Identifizierung korrespondierender Elemente umgesetzt wird.

Ein erster Ansatz besteht darin, im Zuge der Differenzbestimmung zweier Dokumente jedes Element im ersten Dokument mit jedem Element im zweiten Dokument zu vergleichen und einen Ähnlichkeitswert zu berechnen. Selbstverständlich werden nur Elemente miteinander verglichen, wenn sie vom gleichen Typ sind. Zusätzlich ist zu beachten, dass zwischen den Elementen in einem Dokument Beziehungen bestehen. Von besonderem Interesse sind hier die Teil-von Beziehungen; eine Operation in einem UML-Klassendiagramm ist beispielsweise Teil einer Klasse in eben diesem Diagramm. Teil-von Beziehungen haben Auswirkungen auf die Reihenfolge, in der Elemente im Zuge der Differenzbestimmung miteinander verglichen werden können. Um eine Korrespondenz zwischen je einem Element aus jedem Dokument zu erkennen, müssen zuerst etwaige Elementteile auf Korrespondenzen hin untersucht werden.

Die resultierende Situation ist beispielhaft in Abbildung 1.1 dargestellt. Da hier die kreisförmigen Elemente einen Teil der quadratischen Elemente bilden, müssen die kreisförmigen Elemente aus beiden Dokumenten miteinander verglichen werden, bevor Aussagen über Korrespondenzen zwischen den quadratischen Elementen gemacht werden können. Wann die dreieckigen Elemente verglichen werden, spielt keine Rolle, da sie keinen Teil einer Teil-von Beziehung darstellen.

Problemstellung

Aufgrund der großen Anzahl von zu betrachtenden Elementen in nicht-textuellen Dokumenten stellt sich die Frage, inwieweit ein paarweiser Vergleich aller Elemente nach Elementtypen gruppiert effizient ist. Legt man beispielsweise zwei UML-Klassendiagramme mit je 10 Klassen zugrunde, so ergibt sich durch die Betrachtung aller Elementtypen (Klassen, Operationen, Attribute, Parameter, Datentypen, Stereotypen, etc.) leicht eine Elementanzahl im dreistelligen Bereich je Dokument. Wenn – gruppiert nach Elementtypen – jedes Element im ersten Dokument mit jedem Element im zweiten Diagramm verglichen wird, wird eine vierstellige Zahl an direkten Vergleichen nötig. Selbst bei dem sehr kleinen Beispiel in Abbildung 1.1 erreicht man bei sieben bzw. acht Elementen pro Dokument 20 notwendige Direkt-

vergleiche[2].

Für den direkten Vergleich zweier Elemente gleichen Elementtyps werden unterschiedliche Kriterien herangezogen, so dass der Vergleich oft mehrere Instruktionen umfasst. Kriterien, die meist eine Rolle spielen, sind der Name des Elements, sein Inhalt oder seine Position im Dokument.

Die vorliegende Diplomarbeit zielt darauf ab, dieses Vorgehen in der Vergleichsphase zu optimieren. Der Grundgedanke besteht darin, eine Datenstruktur zu entwickeln, in der die Elemente so angeordnet werden, dass ähnliche Elemente dicht beieinander stehen. Wäre eine derartige Struktur vorhanden, so müssten in der Vergleichsphase für ein Element im ersten Dokument statt aller Elemente gleichen Typs im zweiten Dokument nur die benachbarten Elemente betrachtet werden.

Die eigentliche Schwierigkeit dieses Ansatzes liegt darin, Kriterien für die Ähnlichkeit von Elementen zu definieren sowie eine Suchstruktur zu entwickeln, die anhand dieser Kriterien ähnliche Elemente benachbart anordnet. Im Rahmen dieser Arbeit soll nun untersucht werden, inwieweit mehrdimensionale Suchstrukturen eingesetzt werden können, um die Laufzeit der Differenzberechnung zu verbessern.

Es soll eine Datenstruktur entwickelt und implementiert werden, die an den in [Hen90] beschriebenen LSD-Baum angelehnt ist. Da diese Baumstruktur Vektoren reeller Zahlen verwaltet, muss weiterhin eine Möglichkeit gefunden werden, Dokumentelemente auf diese Vektoren abzubilden.

Gliederung der Arbeit

In Kapitel 2 werden zunächst unterschiedliche Datenstrukturen vorgestellt, die zur Lösung des Problems in Betracht gezogen wurden. Dies beinhaltet eine kurze Beschreibung der jeweiligen Struktur sowie eine Einschätzung der Verwendungsmöglichkeiten für die beschriebene Problemstellung.

Darauf folgt in Kapitel 3 eine allgemeine Erläuterung der implementierten mehrdimensionalen Suchbaumstruktur sowie in Kapitel 4 die Darstellung der gewählten Abbildung von Elementen auf die linearen Strukturen der Baumstruktur. Letzteres wird sowohl auf abstrakter Ebene als auch anhand der konkreten Umsetzung für den Dokumenttyp UML-Klassendiagramm betrachtet.

Nachdem die Organisation der Dokumentelemente in einer mehrdimensionalen Suchbaumstruktur erfolgt ist, muss untersucht werden, wie dieses Zwischenergebnis für die Differenzberechnung verwendet werden kann. Dazu werden in Kapitel 5 mehrere Alternativen aufgezeigt.

Eine Evaluierung der Ergebnisse findet in Kapitel 6 statt. Dabei wird zum einen das Verhalten der implementierten Baumstruktur an konkreten Beispiel-Dokumenten und zum anderen das Laufzeitverhalten in der Praxis untersucht. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick.

[2] $2 \cdot 2$ Quadrate + $3 \cdot 4$ Kreise + $2 \cdot 2$ Dreiecke

2 Auswahl geeigneter Datenstrukturen

Für die Verwaltung von Dokumentelementen sind viele Ansätze denkbar, von denen hier einige vorgestellt und kritisch betrachtet werden sollen. An die jeweiligen Strukturen werden zwei Anforderungen gestellt:

- Es muss ein Ähnlichkeitsmaß definiert werden, das es ermöglicht, die Ähnlichkeit zweier Elemente zueinander zu bestimmen. Das Ähnlichkeitsmaß soll so variabel sein, dass die Semantik unterschiedlicher Dokumenttypen berücksichtigt werden kann. Um dies zu erreichen, ist eine Abbildung der Dokumentelemente auf Hilfsstrukturen denkbar, sofern die charakteristischen Eigenschaften der Elemente erhalten bleiben.
- Die zum Auffinden ähnlicher Elemente korrespondierende Laufzeitklasse muss kleiner als $O(n)$ sein, wenn n die Anzahl der Elemente bezeichnet. Ist diese Anforderung nicht erfüllt, so wird keine Verbesserung gegenüber einem paarweisen Vergleich der Elemente erreicht.

Im Folgenden werden beginnend mit Strukturen aus dem Information Retrieval in Abschnitt 2.1 Ansätze aus verschiedenen Bereichen der Informatik beschrieben und auf ihre Eignung zur Lösung der Problemstellung dieser Arbeit hin untersucht.

2.1 Strukturen aus dem Bereich Information Retrieval

Unter Information Retrieval (IR) wird meist die inhaltliche Suche in Texten verstanden, das Hauptanwendungsgebiet von IR stellen somit Literaturdatenbanken dar. Diese Definition deckt jedoch nur einen Teilbereich dessen ab, was sich aus dem Begriff Information Retrieval ableiten lässt. Es geht grundsätzlich um das Auffinden von Informationen.

IR-Systeme erweitern herkömmliche Datenbanksysteme um die Möglichkeit von vagen Abfragen. Vage Abfragen werden verwendet, wenn ein Benutzer nicht genau spezifizieren kann, wonach er sucht, und statt einem exakten Ergebnis erwartet, dass er vom IR-System möglichst gute Treffer, evtl. sortiert nach einer vom System berechneten Relevanz, geliefert bekommt.

Da die vage Suche darauf basiert, dass ausgehend von einer Suchanfrage relevante Elemente identifiziert werden können, soll im Folgenden untersucht werden, inwieweit sich unterschiedliche Techniken des Information Retrieval dazu verwenden

lassen, die Suche nach ähnlichen Dokumentelementen im Rahmen der Differenzbestimmung auf eine vage Suche nach dem Ausgangselement abzubilden.

Freitextsuche

Die Freitextsuche ist die am häufigsten verwendete Technik innerhalb des Information Retrieval. Nach [Fuh96] werden zwei Ansätze unterschieden:

informatischer Ansatz. Der informatische Ansatz erweitert die Suche nach Zeichenketten um Truncation und Kontextoperatoren. Truncation ermöglicht die Suche nach Wortteilen, Kontextoperatoren dienen zur Suche nach Ausdrücken, die mehrere Wörter umfassen.

computerlinguistischer Ansatz. Die unterschiedlichen Verfahren des computerlinguistischen Ansatzes gehen auf die Besonderheiten der jeweiligen Sprache ein. Graphematische und lexikalische Verfahren bieten eine Reduktion von Wörtern auf Wortstämme, wobei graphematische Verfahren regelbasiert vorgehen und lexikalische Verfahren auf Wörterbüchern basieren. Lexikalische Verfahren sind somit für Sprachen geeignet sind, die regelbasiert nicht oder nur schwer erfasst werden können. Weiterhin werden zur Bestimmung von Wortklassen syntaktische Verfahren verwendet.

Diese Verfahren gehen alle von der Annahme aus, dass vage Anfragen beantwortet werden können, indem Regeln definiert werden, mit denen einige Informationen (z.B. Wortendungen) ignoriert werden, um Übereinstimmungen der relevanten Teile (z.B. Wortklassen) ermitteln zu können. Da diese Regeln fest definiert werden, eignet sich der Ansatz nur sehr bedingt, um die Bestimmung von Dokumentdifferenzen zu optimieren. Für Dokumente mit textuellen Bestandteilen (z.B. Namen von Dokumentelementen) können die Ideen jedoch in Betracht gezogen werden.

Dokumentationssprachen

Dokumentationssprachen stellen einen ersten Schritt dar, um Elemente in einem abstrakten Kontext zu betrachten (vgl. [Bur91]). Im Rahmen der Freitextsuche können beispielsweise Thesauri benutzt werden, um von irrelevanten Unterscheidungen bei der Wortwahl zu abstrahieren und stattdessen die Gesamtaussage zu repräsentieren. Einen ähnlichen Ansatz stellen Klassifikationssysteme dar. Sie bieten ein fest vorgegebenes, oft hierarchisches Klassifikationsschema, in dem sich alle Elemente eindeutig einordnen lassen.

Beide Ansätze bieten keine Variabilität, sondern gehen von einer im Voraus bekannten Elementmenge aus. Diese Annahme trifft im Rahmen der Bestimmung von Dokumentdifferenzen nicht zu, so dass Dokumentationssprachen hier nicht effizient eingesetzt werden können.

Boolesches Retrieval

Boolesches Retrieval setzt voraus, dass jedes Element im IR-System anhand von Indextermen indiziert worden ist. Bei Literaturdatenbanken bieten sich wichtige Begriffe als Indexterme an. Suchanfragen können daraufhin unter Verwendung der booleschen Operatoren `and`, `or` sowie `not` gestellt werden.

Während bei Booleschem Retrieval davon ausgegangen wird, dass ein Indexterm in einem Element entweder vorkommt oder nicht und somit als Wert für eine Kombination aus Element und Indexterm nur 0 oder 1 gewählt werden kann, erweitert das Fuzzy-Retrieval diesen Wertebereich auf Werte zwischen 0 und 1. Damit wird es möglich, alle Elemente im IR-System in Bezug auf eine Suchanfrage nach ihrer Relevanz zu ordnen.

Die Grundidee der Indizierung von Elementen aufgrund von Indextermen lässt sich für die Bestimmung von Dokumentdifferenzen nutzen, wenn eine Möglichkeit gefunden wird, Indexterme so zu definieren, dass im Sinne der Differenzberechnung ähnliche Elemente ähnliche Werte aufweisen. Nicht nutzbar sind hingegen die Suchanfragen, da beim Fuzzy-Retrieval nur dieselbe Menge an Fragen wie beim einfachen Booleschen Retrieval möglich ist. Es ist also nicht möglich, nach den genauen Indexwerten eines Elements im IR-System zu suchen, sondern lediglich nach 0 bzw. 1 für jeden Indexwert.

Vektorraummodell

Im Vektorraummodell nach [SWY75] werden sowohl Elemente im IR-System als auch Suchanfragen als Punkte in einem mehrdimensionalen, metrischen Vektorraum betrachtet. Das Ergebnis einer Suchanfrage ist die Menge aller Elemente, geordnet nach der Distanz zum Vektor der Suchanfrage. Als Maß für die Distanz wird der euklidische Abstand der Vektoren verwendet.

Die entscheidende Erweiterung im Vergleich zum Fuzzy-Retrieval ist somit die Tatsache, dass die gleiche Struktur – nämlich ein Vektor – für Elemente im IR-System und für Suchanfragen verwendet wird. Damit ist es im Vektorraummodell möglich, nach einem bestimmten Element zu suchen und alle anderen Elemente nach der Distanz zum gesuchten Element geordnet auszugeben. Diese Idee ist auf die Bestimmung von Dokumentdifferenzen übertragbar, wenn eine Möglichkeit gefunden wird, Elemente so auf metrische Vektoren abzubilden, dass genügend Information für die Ähnlichkeitsaussagen erhalten bleibt.

Darüber hinaus muss das Auffinden ähnlicher Elemente effizienter gestaltet werden. Die Fragestellung im Rahmen des Vektorraummodells ist lediglich eine Sortierung aller Elemente im System nach ihrer Distanz zum Suchvektor. Bei der Optimierung der Bestimmung von Dokumentdifferenzen wird diese Fragestellung dahingehend erweitert, dass nur die Elemente zurück geliefert werden, die eine hinreichende Ähnlichkeit aufweisen, und dass zusätzlich die Bestimmung dieser Menge in einer kleineren Laufzeitklasse als $O(n)$ realisiert werden soll, wobei n die Anzahl von Elementen im System bezeichnet.

Das Vektorraummodell lässt sich also nicht ohne Anpassungen auf die Problemstellung anwenden. Es fehlt in erster Linie an einer Datenstruktur, die das definierte Ähnlichkeitsmaß als Grundlage einer benachbarten Anordnung ähnlicher Elemente verwendet.

Resource Description Framework

Das im Zusammenhang mit dem *Semantic Web* entstandene und in [LS98] beschriebene Resource Description Framework (RDF) stellt eine formale Sprache zur Bereitstellung von Metadaten im Internet dar. Hier soll untersucht werden, inwieweit die Ideen des RDF auf die Bestimmung von Dokumentdifferenzen übertragbar sind.

Das Datenmodell des RDF besteht aus drei Objekttypen:

Resources. Unter Ressourcen versteht man im Zusammenhang mit RDF beliebige Objekte im WWW, z.B. Webseiten oder einzelne HTML-Elemente. Ressourcen werden durch URIs identifiziert.

Properties. Properties bezeichnen gerichtete Beziehungen zwischen Ressourcen, denen eine bestimmte Bedeutung zugewiesen wird.

Statements. Statements bestehen aus drei Teilen: einer Ressource, einer benannten Property und einer weiteren Ressource. Die Teile stehen im Subjekt – Prädikat – Objekt Verhältnis zueinander.

Das Resource Description Framework eignet sich für die Bestimmung von Dokumentdifferenzen insofern, dass es eine standardisierte Beschreibungssprache definiert, die für die interne Behandlung von Dokumenten verwendet werden kann. Es gibt jedoch bisher keine vage Abfragesprache, mit deren Hilfe man Ähnlichkeiten zwischen Ressourcen definieren und erkennen kann. Weiterhin existiert bei der Abbildung von Dokumentelementen auf RDF ein großer Spielraum, der – ähnlich wie bei XML-Dateien – zu Kompatibilitätsproblemen und aufwändigen Transformationen führen kann[1].

Probabilistisches Datalog

Probabilistische Modelle beschreiben im Information Retrieval mehrstufige Prozesse, die auf Feedback basieren. Dabei wird die Ähnlichkeit zwischen Elementen über Wahrscheinlichkeiten definiert, die aufgrund von Interaktionen mit dem Benutzer des IR-Systems angepasst werden. Dieser Ansatz ist nicht auf die Bestimmung von Dokumentdifferenzen übertragbar, da keine Interaktion mit dem Benutzer möglich ist.

Das Probabilistische Datalog nach [Fuh00] macht dagegen nicht von Feedback Gebrauch und soll somit auf seine Verwendbarkeit bei der Differenzbestimmung hin untersucht werden. Datalog ist eine Sprache für Regeln und Fakten, die eine

[1] Zur Abbildung von RDF auf XML existiert eine Empfehlung des W3C (siehe [Bec04]).

Untermenge der funktionsfreien, auf Hornklauselprädikatslogik aufbauenden Programmiersprache Prolog darstellt. Im Zusammenhang mit der Differenzberechnung bietet sich Datalog für die Abbildung von Dokumentelementen und ihren Beziehungen an. So könnte beispielsweise über den Fakt `class_has_attribute (ClassA, AttributeB)` die Tatsache modelliert werden, dass in einem UML-Klassendiagramm die Klasse *ClassA* über das Attribut *AttributeB* verfügt. Als Beispiel für eine Regel im selben Kontext dient `package_has_attribute (X,Y) :- package_has_class (X,Z), class_has_attribute (Z,Y)`. Diese Regel sagt aus, dass ein Attribut Y genau dann Teil eines Packages X ist, wenn es eine Klasse Z gibt, die in Package X enthalten ist und ihrerseits das Attribut Y besitzt.

Wie diese Beispiele zeigen, ist es durch angemessene Konventionen möglich, die in einem Dokument enthaltenen Informationen mit Datalog abzubilden. Eine geeignete Modellierung ermöglicht es insbesondere, Zyklen in Dokumenten problemlos zu erfassen. Probabilistisches Datalog erweitert das reine Datalog um Wahrscheinlichkeiten, die sowohl Fakten als auch Regeln zugewiesen werden. Jeder Fakt und jede Regel wird um eine Wahrscheinlichkeit ergänzt; die oben genannten Beispiele könnten demnach in `0.8 class_has_attribute (ClassA, AttributeB)` bzw. `0.7 package_has_attribute (X,Y) :- package_has_class (X,Z), class_has_attribute (Z,Y)` überführt werden.

Die Erweiterung der Struktur impliziert eine Erweiterung der Abfragemöglichkeiten. Während bei klassischem Datalog lediglich Anfragen, die mit `yes` oder `no` zu beantworten sind, und Anfragen mit Variablen, bei denen das System alle zutreffenden Variablenbelegungen zurück liefert, gestellt werden können, ist es mit Probabilistischem Datalog möglich, Ergebnisse nach ihrer Wahrscheinlichkeit geordnet zurück geben zu lassen.

Auf die Bestimmung von Dokumentdifferenzen kann diese Idee übertragen werden, wenn die Wahrscheinlichkeiten der Regeln und Fakten nicht als Wahrscheinlichkeiten sondern als Gewichte für die Ähnlichkeitsbestimmung verstanden werden. Durch eine geeignete Modellierung wäre es damit möglich, Anfragen wie `similar (ClassA, X)` vom System beantworten zu lassen. Damit könnten alle Elemente ermittelt werden, die dem Element *ClassA* hinreichend ähnlich sind. Die Ähnlichkeitskriterien würden über Regeln und ihre Gewichte ausgedrückt.

[Chi01] beschreibt mit dem FERMI multimedia model eine Möglichkeit zur Abbildung von Multimedia-Dokumenten, die vor allem die logische Struktur der Dokumente berücksichtigt. Die Abbildung erfolgt ebenfalls in Fakten und Regeln, so dass dieser Ansatz herangezogen werden kann, um Anregungen für die Abbildung der Dokumente im Rahmen der Differenzbestimmung zu erhalten.

Da zum Probabilistischen Datalog jedoch bisher nur wenige Prototypen existieren und vor allem die Laufzeit noch ungeklärt ist, soll das Thema in dieser Arbeit nicht weiter vertieft werden. Festzuhalten bleibt, dass Probabilistisches Datalog einen Ansatz zur Abbildung von Dokumenten bietet, der sich von anderen Vorgehensweisen klar abgrenzt. Empirische Resultate liegen auf diesem Gebiet jedoch noch nicht vor.

2.2 Clustering

Die zentrale Aufgabe des Clustering besteht darin, Ähnlichkeiten von Elementen zu nutzen, um von einem relevanten Element zu weiteren relevanten Elementen zu gelangen. Diese Problemstellung lässt sich nahtlos auf die Bestimmung von Dokumentdifferenzen übertragen, bei der zum Auffinden von Korrespondenzen zu einem Element in einem Dokument alle dazu ähnlichen Elemente in einem anderen Dokument gefunden werden sollen[2]. Daher werden im Folgenden in Anlehnung an [JMF99] wichtige Clustering-Techniken vorgestellt und auf ihre Eignung zur Lösung der Problemstellung hin eingeschätzt.

Agglomeratives Clustering

Agglomeratives Clustering basiert auf einer Ähnlichkeitsmatrix, die für jede Kombination zweier Elemente deren Ähnlichkeit zueinander angibt. Die Zuordnung von Elementen zu Clustern kann auf verschiedene Arten erfolgen:

single-link. Alle Elementpaare, deren Ähnlichkeitswert eine festgelegte Grenze überschreitet, werden miteinander verbunden. Dabei entsteht ein potentiell unzusammenhängender Graph, der alle Elemente als Knoten enthält und dessen zusammenhängende Teilgraphen Cluster bilden.

complete-link. Während bei single-link Clustering Elemente in ein Cluster aufgenommen werden, wenn sie zu mindestens einem der bereits im Cluster vorhandenen Elemente einen hinreichenden Ähnlichkeitswert aufweisen, setzt complete-link voraus, dass der Grenzwert für die Ähnlichkeit zu allen Cluster-Elementen erreicht wird.

Laufzeittechnisch ist beim agglomerativen Clustering die Berechnung der Ähnlichkeitsmatrix kritisch. Da hier jedes Element in Verbindung zu jedem anderen Element betrachtet werden muss, ergibt sich eine Laufzeit aus $O(n^2)$, wenn n die Zahl der Elemente bezeichnet. Das ist aber gerade die Laufzeitklasse, die im Rahmen der hier angestrebten Optimierung unterschritten werden soll.

Partitionierendes Clustering

Beim partitionierenden Clustering wird eine Laufzeit aus $O(n)$ erreicht, indem die Anzahl der Cluster vorgegeben wird. Zusätzlich wird für jedes Cluster ein so genanntes Seed-Element festgelegt. Jedes weitere Element wird dann dem Cluster zugeordnet, zu dessen Seed-Element es die größte Ähnlichkeit hat. Dabei muss jedes Element mit jedem Seed-Element verglichen werden, was einem Aufwand von $k \cdot n$

[2] Dabei wird unterstellt, dass sich die Differenzberechnung auf lediglich zwei Dokumente bezieht. Die Idee ist jedoch ohne Weiteres auf die Bestimmung von Korrespondenzen eines Elementes in mehreren anderen Dokumenten übertragbar.

entspricht, wenn k die Zahl der Cluster angibt. Da k konstant ist, ist die Laufzeit linear in n.

Kritisch ist die Auswahl der Seed-Elemente, die im Zuge der Bestimmung von Dokumentdifferenzen aufgrund unterschiedlicher Dokumenttypen und unterschiedlicher Dokumente nicht fest vorgegeben werden können. Stattdessen muss ein Verfahren realisiert werden, das zur Laufzeit aus der vorhandenen Elementmenge geeignete Seed-Elemente herausfiltert. Wenn man suboptimale Seed-Elemente zwecks Laufzeitoptimierung akzeptiert, so kann partitionierendes Clustering für die gegebene Problemstellung eingesetzt werden.

Hierarchisches Clustering

Das Ergebnis von hierarchischem Clustering ist eine baumartige Hierarchie von Clustern. Auf oberster Ebene gehören alle Elemente zu demselben Cluster, auf unterster Ebene enthält jedes Cluster nur ein Element. Auch dabei zeichnen sich Cluster dadurch aus, dass sie Elemente enthalten, deren Ähnlichkeit zueinander hoch ist. Dazu sind zwei Vorgehensweisen denkbar:

bottom-up. Im Rahmen der bottom-up Vorgehensweise wird der Baum an den Blättern beginnend aufgebaut. Anfangs wird jedes Element als eigenes Cluster betrachtet, dann werden rekursiv immer die beiden Cluster zu einem neuen Cluster verbunden, die die größte Ähnlichkeit zueinander aufweisen, bis nur noch ein alle Elemente enthaltendes Cluster vorhanden ist.

top-down. Die top-down Variante baut den Baum an der Wurzel beginnend auf. Dazu müssen existierende Cluster jeweils in zwei neue Cluster geteilt werden. Für das genaue Vorgehen bei dieser Aufteilung sind diverse Kriterien denkbar.

Während die Laufzeit der bottom-up Variante quadratisch ist, da die Elemente paarweise verglichen werden müssen um die besten Paare zu identifizieren, ist die Laufzeit der top-down Variante von der Vorgehensweise bei der Clusteraufteilung abhängig. Eine Aufteilung, die denselben Baum ergibt wie die bottom-up Variante, benötigt jedoch auch hier mindestens quadratische Laufzeit.

Fuzzy-Clustering

Beim Fuzzy-Clustering können Elemente zu mehreren Clustern gleichzeitig gehören, wobei der Grad der Zugehörigkeit durch einen Wert zwischen 0 und 1 ausgedrückt wird. Da die Zuordnung zu den Clustern iterativ erfolgt und schrittweise verbessert wird, bis ein vorgegebenes Qualitätskriterium erreicht ist bzw. bis die Cluster stabil sind, lässt ich keine Aussage über die Laufzeit des Fuzzy-Clustering machen. Daraus ergibt sich die Nichtanwendbarkeit für die Optimierung der Bestimmung von Dokumentdifferenzen.

Scatter-Gather-Clustering

Als letztes Clustering Verfahren wird das Scatter-Gather-Clustering nach [CKPT92] betrachtet. Cluster werden durch die folgenden Schritte interaktiv berechnet:

1. Elemente werden grob in vorläufige Cluster mit bestimmten Themen unterteilt *(scatter)*.

2. Dem Benutzer werden die Inhalte der vorläufigen Cluster anhand von Stichworten angezeigt.

3. Der Benutzer wählt einige der Cluster aus *(gather)*.

4. Die Elemente in den ausgewählten Clustern stellen die Ausgangsmenge für die nächste Iteration dar, die wieder mit Schritt 1 beginnt.

Durch die Interaktivität kommt Scatter-Gather-Clustering für die Optimierung der Bestimmung von Dokumentdifferenzen nicht in Frage.

2.3 Support Vector Machines

Support Vector Machines (SVM) nach [CST00] stellen einen Ansatz aus dem Bereich Machine Learning zur Klassifizierung von Elementen dar. Es wird davon ausgegangen, dass die Elemente eine Repräsentation in einem mindestens zweidimensionalen, metrischen Vektorraum haben. Das zur Klassifizierung benötigte Modell lernen Support Vector Machines anhand von Trainingsdaten, also Elementen, die bereits außerhalb der SVM klassifiziert worden sind. Basierend auf den Trainingsdaten mit n Dimensionen wird eine $(n-1)$-dimensionale Hyperebene konstruiert, die die Trainingsbeispiele mit maximaler Trennspanne separiert.

Da die Trennung bei SVMs nur in jeweils zwei Klassen erfolgt, muss der Ansatz rekursiv genutzt werden, um für die Ähnlichkeitssuche bei Dokumentelementen eingesetzt werden zu können. Für ein Element im ersten Dokument werden dann nur die Elemente gleichen Typs im zweiten Dokument als Kandidaten für Korrespondenzen betrachtet, die von der SVM in dieselbe Klasse eingeordnet wurden.

Implementierungen von Support Vector Machines sind vorhanden, so dass das eigentliche Problem bei der Anwendung von SVMs auf die hier betrachtete Problemstellung in der Abbildung der Vergleichsphase auf Machine Learning Techniken besteht. Dabei spricht vor allem die Tatsache, dass die Klassifizierungen sowie einige Trainingselemente für die entsprechenden Klassen vorgegeben werden müssen, gegen einen Einsatz von SVMs. Das Vorgeben von Klassen setzt Wissen voraus, das entweder zur Laufzeit nicht vorhanden ist oder nur laufzeitaufwändig ermittelt werden könnte.

2.4 Ansätze aus dem Bereich Data Mining

[HK00, S. 5] definiert die Aufgabe von Data Mining als *„extracting knowledge from large amounts of data“*. Data Mining ist also ein Prozess, der Wissen aus einer großen Menge von Daten extrahiert. Dies geschieht meist dadurch, dass die vorliegende Datenmenge auf bestimmte Muster, statistische Auffälligkeiten und andere Regelmäßigkeiten hin untersucht wird.

Im Folgenden soll dargestellt werden, wie sich bestimmte Techniken aus dem Bereich Data Mining für die Bestimmung von Dokumentdifferenzen anwenden lassen. Ausgangspunkt ist jeweils die Idee, alle vorhandenen Elemente als Datenmenge zu betrachten und Elemente anhand von Mustern und statistischen Auffälligkeiten zu gruppieren. Die benötigte Ähnlichkeitsbestimmung von Elementen wird also auf Data Mining Ansätze abgebildet.

Hauptkomponentenanalyse

Die Hauptkomponentenanalyse wurde erstmal in [Pea01] vorgestellt und stellt eine multivariate Analysemethode dar. Das bedeutet, dass Variablen nicht isoliert betrachtet werden, sondern vielmehr das Zusammenwirken mehrerer Variablen sowie die zugehörigen Abhängigkeiten. Die Zielsetzung besteht darin, aus Elementen mit vielen Eigenschaften die bestimmenden Eigenschaften zu extrahieren.

Im Zusammenhang mit der Bestimmung von Dokumentdifferenzen kann die Hauptkomponentenanalyse eingesetzt werden, um Kriterien für die Ähnlichkeit von Elementen zu bestimmen. Dazu müssen zunächst die Elementeigenschaften auf Variablen abgebildet werden. Daraufhin wird die Hauptkomponentenanalyse benutzt, um basierend auf der Elementmenge die Variablen zu extrahieren, die die bestimmenden Eigenschaften der Elemente abbilden.

Da die Berechnung der Hauptkomponentenanalyse rechenintensiv ist, eignet sich diese Methode nicht, um bestimmende Eigenschaften zur Laufzeit zu ermitteln. Es ist jedoch möglich, anhand einer typischen Elementverteilung eine derartige Analyse durchzuführen, deren Ergebnisse für alle weiteren Bestimmungen von Dokumentdifferenzen verwendet werden. Vorausgesetzt wird hier, wie bei den meisten bisher erläuterten Verfahren, dass die Eigenschaften der Elemente auf Zahlwerte abgebildet werden können. Möglichkeiten dazu werden ausführlich in Kapitel 4 betrachtet.

Faktorenanalyse

Die Intention der Faktorenanalyse unterscheidet sich im Wesentlichen nicht von der Zielsetzung der Hauptkomponentenanalyse. Im Rahmen der Faktorenanalyse wird versucht, die Zahl der Variablen zu reduzieren, indem solche Variablen gebündelt werden, deren Aussagen sich nicht oder nur wenig unterscheiden.

Auch für die Faktorenanalyse gilt, dass sie zu aufwändig ist, um zur Laufzeit jedes Mal neu ausgeführt zu werden. Eine Vorabanalyse ist jedoch sinnvoll, wenn anhand von Testdaten Erkenntnisse gewonnen werden können, die sich auf alle weiteren

Ausführungen übertragen lassen. Diese Testläufe sind offensichtlich dokumenttypspezifisch durchzuführen.

Attributsgewichtung

Die auch als Sensitivitätsanalyse bekannte Attributsgewichtung bezeichnet im Zusammenhang mit Data Mining die Ermittlung des Einflusses bestimmter Attribute auf das Ergebnis. Was konkret als Ergebnis angesehen wird, ist vom Einsatzgebiet abhängig.

Bei der Bestimmung von Dokumentdifferenzen bietet es sich an, die Menge der Elemente, die vom System als ähnlich zu einem gegebenen Element eingeschätzt werden, als Ergebnis anzusehen. Daraufhin kann untersucht werden, wie stark sich bestimmte Elementeigenschaften auf den Inhalt dieser Ergebnismenge auswirken.

Damit ergibt sich zusätzlich die Möglichkeit, aufgrund des Ergebnisses von Sensitivitätsanalysen die Qualität der Abbildung von Elementeigenschaften einzuschätzen. Wirken sich irrelevante Änderungen stark auf die Ähnlichkeitsaussagen aus, so muss die Abbildung entsprechend angepasst werden. Dies sei an einem Beispiel aus Simulink-Diagrammen verdeutlicht. Verändert sich bei zwei Blöcken, die als korrespondierend angesehen werden sollen, die Ähnlichkeitsaussage, wenn die Anzahl der Input-Parameter verändert wird, so ist zu untersuchen, ob dieser Effekt erwünscht ist bzw. ob die Anzahl der Input-Parameter tatsächlich Einfluss auf die Ähnlichkeit zweier Blöcke haben sollte. Diese Einschätzung muss unter semantischen Gesichtspunkten stattfinden und kann nicht automatisiert werden.

Regressionsanalyse

Regressionsanalysen können ebenfalls eingesetzt werden, um Erkenntnisse über den Einfluss bestimmter Elementeigenschaften auf die Definition von Ähnlichkeiten zwischen Elementen zu gewinnen. Bei Regressionsanalysen handelt es sich um eine statistische Analysemethode zur Aufdeckung von Abhängigkeiten zwischen Attributen. Durch Abhängigkeiten zwischen Elementeigenschaften kann es vorkommen, dass die gleiche Information mehrfach erfasst wird.

Künstliche neuronale Netze

Künstliche neuronale Netze stellen einen Teilbereich des Fachgebiets Künstliche Intelligenz dar. Nach dem Aufbau der Netztopologie erfolgt ähnlich wie bei den in Abschnitt 2.3 vorgestellten Support Vector Machines eine Trainingsphase, in der das Netz durch das Verändern bzw. das Verbinden von Neuronen lernt.

Die Verwendung von neuronalen Netzen hat sich vor allem dann als sinnvoll erwiesen, wenn wenig systematisches Wissen über den betrachteten Problembereich vorhanden ist. Diese Voraussetzung trifft auf die Bestimmung von Dokumentdifferenzen nicht zu, da hier dokumenttypspezifisch sehr viel systematisches Wissen über die Elemente vorliegt und es lediglich darum geht, dieses Wissen in einer Weise zu erfassen, die die Laufzeit der Vergleichsphase optimiert.

2.5 Linearisierung

Eine erste Möglichkeit, die Laufzeit der paarweisen Vergleichsphase von Elementen bei der Bestimmung von Dokumentdifferenzen zu optimieren, ist der Einsatz von Hashwerten für Elemente. Dazu werden Hashwerte für alle Elemente in beiden zu vergleichenden Dokumenten berechnet, bevor die paarweise Vergleichsphase beginnt. Daraufhin wird für jedes Element im ersten Dokument festgestellt, ob es ein Element mit übereinstimmendem Hashwert im zweiten Dokument gibt. Da die Verwaltung in Hash-Tabellen es ermöglicht, in einer Laufzeit aus $O(1)$ auf einzelne Einträge zuzugreifen, kann diese Phase in $O(n)$ durchgeführt werden, wenn n die Anzahl der Elemente angibt.

In der anschließenden Vergleichsphase, die Korrespondenzen zwischen Elementen erkennen soll, die nicht exakt übereinstimmen sondern nur eine hinreichende Ähnlichkeit aufweisen, müssen dann alle Elemente, die bereits im Rahmen der Hashphase einem korrespondierenden Element zugeordnet werden konnten, nicht mehr betrachtet zu werden.

Die Grundidee der Linearisierung besteht darin, die berechneten Hashwerte neben der Erkennung von exakten Übereinstimmungen auch für Erkennung von Ähnlichkeiten zu nutzen. Wäre eine Sortierung der Hashwerte möglich, so dass ähnliche Elemente nebeneinander stehen, so müssten lediglich die umliegenden Elemente für die Ähnlichkeitsbestimmung betrachtet werden. Um diesen Zweck zu erfüllen, müssen die berechneten Hashwerte aussagekräftig sein.

Das Problem dieses Ansatzes liegt in seiner Eindimensionalität. Während man beispielsweise Blöcke aus Simulink-Diagrammen nach der Anzahl ihrer *in-ports* sortieren kann, lassen sie sich nicht gleichzeitig anhand der Anzahl der *out-ports* sortieren. Es müsste also eine Sortierung gefunden werden, die sowohl Elemente mit ähnlicher Anzahl von *in-ports* als auch Elemente mit einer ähnlichen Anzahl an *out-ports* benachbart anordnet[3]. Bereits dieses kleine Beispiel mit nur zwei betrachteten Kriterien pro Element macht deutlich, dass eine lineare Sortierung bei einer größeren Zahl von Kriterien für die Ähnlichkeit nicht möglich ist.

Aus diesem Grund werden im Weiteren ausschließlich mehrdimensionale Ansätze betrachtet, die eine benachbarte Anordnung von Elementen nach mehreren Kriterien gleichzeitig ermöglichen.

2.6 Baumstrukturen

Als Datenstruktur für mehrdimensionale Daten bieten sich bestimmte Baumstrukturen an, die eine benachbarte Anordnung nach mehr als einer Dimension zulassen. Im Folgenden werden zunächst der R-Baum nach [Gut84] und der Quadtree nach [FB74] vorgestellt. Mit dem LSD-Baum nach [Hen90] und dem im Rahmen dieser

[3]Hier wird unterstellt, dass die Anzahl von *in-ports* bzw. *out-ports* ein Kriterium für die Ähnlichkeit von Blöcken in Simulink-Diagrammen darstellt.

Arbeit entwickelten S^{3}V Baum finden anschließend in Kapitel 3 zwei weitere Baumstrukturen für mehrdimensionale Daten ausführliche Betrachtung.

R-Baum

Beim R-Baum handelt es sich um eine an den B*-Baum angelehnte, mehrdimensionale und balancierte Datenstruktur, die häufig in Datenbanksystemen eingesetzt wird. Die Blattknoten des Baumes enthalten die zu indizierenden Daten, bei denen es sich sowohl um Punkte als auch um hochdimensionale Elemente handeln kann. Die inneren Knoten des Baumes enthalten Verweise auf weitere innere Knoten bzw. auf Blattknoten. Zusätzlich sind alle inneren Knoten außer der Wurzel mindestens zur Hälfte gefüllt.

Die inneren Knoten verwalten minimale achsparallele Objekte[4], die alle im darunter liegenden Teilbaum vorhandenen Elemente umfassen. Das Einfügen neuer Elemente in einen R-Baum erfolgt in vier Schritten:

1. An der Wurzel beginnend wird für jeden inneren Knoten geprüft, welcher Teilbaum das neue Element aufnehmen soll. Dazu wird immer der Teilbaum ausgewählt, bei dem die resultierende Erweiterung des zugehörigen Datenraumes am kleinsten ist.

2. Wenn der Blattknoten gefunden wurde, der das neue Element aufnehmen soll, wird das neue Element eingefügt. Ist der Blattknoten daraufhin überfüllt, so wird Schritt 3 ausgeführt, ansonsten kann mit Schritt 4 fortgefahren werden.

3. Im Falle eines überfüllten Blattknotens wird ein Split durchgeführt. Die vorhandenen Elemente werden so in zwei Mengen aufgeteilt, dass die beiden neuen Datenräume möglichst klein sind. Dazu schlägt [Gut84] mehrere Algorithmen vor, die sich in ihrer Laufzeit unterscheiden. Für jeden Split die optimale Aufteilung zu finden ist zu aufwändig, so dass im Allgemeinen mit heuristischen Verfahren gearbeitet wird.

4. Die Änderungen an der Baumstruktur müssen nach dem Einfügen vom Blattknoten zur Wurzel propagiert werden. Da sich der Datenraum des Blattknotens geändert hat, haben sich potentiell auch alle anderen Datenräume auf dem entsprechenden Pfad verändert.

Charakteristisch für den R-Baum ist die Möglichkeit der Überlappung von Datenräumen, die durch die Minimierung der Datenräume während der Splits entsteht. Abbildung 2.1 zeigt anschaulich für einen zweidimensionalen Fall, warum ein Split trotz Überlappung optimal im Hinblick auf die Minimierung der Datenräume sein kann. Die mit einem X gekennzeichneten Punkte stellen die Elemente des

[4] Im anschaulichen zweidimensionalen Fall handelt es sich bei diesen Objekten um achsparallele Rechtecke.

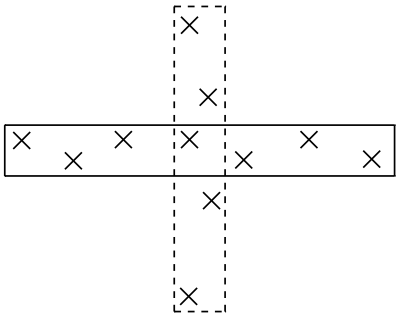

Abbildung 2.1: Beispiel für die Überlappung von Datenräumen nach einem Split im R-Baum

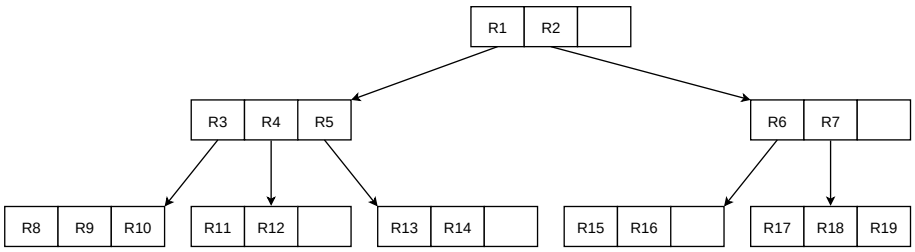

Abbildung 2.2: Beispiel für einen R-Baum nach [Gut84, S. 3]

übergelaufenen Blattknotens dar, die beiden Rechtecke markieren die neuen Datenräume, die durch den Split entstehen.

Die entstehende Hierarchie von Datenräumen wird anhand der Abbildungen 2.2 und 2.3 verdeutlicht. Abbildung 2.2 zeigt den am B*-Baum orientierten Aufbau des R-Baumes an einem Beispiel. In den Knoten stehen jeweils die Identifizierer der *regions*, die die Elemente im zugehörigen Unterbaum umspannen und die in Abbildung 2.3 dargestellt sind.

Aufgrund der Überlappungen kann der R-Baum im *worst case* für Such- und Bereichsanfragen nur eine lineare Laufzeit garantieren, die sich durch das Betrachten aller Elemente ergibt. So müssten beispielsweise für das Auffinden eines Elementes im oberen rechten Teil von *R16* an der Wurzel sowohl *R1* als auch *R2* verfolgt werden, da beide das Element enthalten. Auf der nächsten Ebene ergeben sich mit *R4* und *R6* wiederum zwei Bereiche, die parallel verfolgt werden müssen.

Andererseits zeigen die von [Gut84] durchgeführten Studien, dass die Laufzeit von Such- und Bereichsanfragen im *average case* bei zwei Dimensionen deutlich geringer als linear ist. Wäre also im Bezug auf die Bestimmung von Dokumentdifferenzen eine Abbildung von Dokumentelementen auf zweidimensionale Strukturen gegeben, so ließe sich der R-Baum für die Optimierung der paarweisen Vergleichsphase einsetzen.

Bisherige Arbeiten auf dem Gebiet der Bestimmung von Dokumentdifferenzen haben jedoch gezeigt, dass oft deutlich mehr Kriterien für den Vergleich von Elementen herangezogen werden müssen (vgl. z.B. [KWN05]) und somit zwei Dimensionen

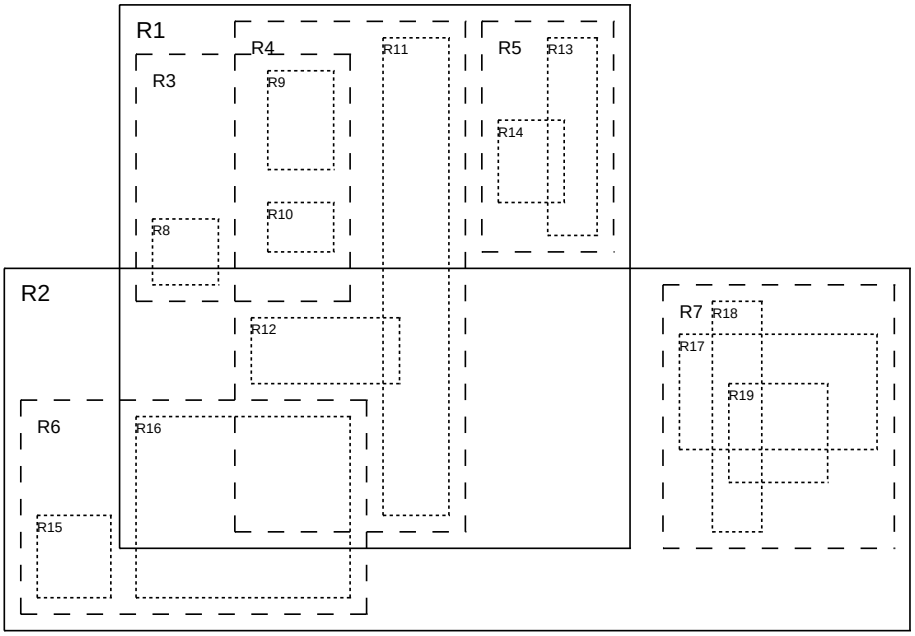

Abbildung 2.3: Beispiel-Aufteilung des Datenraumes durch den R-Baum nach [Gut84, S. 3]

beim R-Baum nicht hinreichend sind. Studien, die im Rahmen der Entwicklung des X-Tree in [BKK96] veröffentlicht wurden, haben ergeben, dass der R-Baum mit zunehmender Dimensionsanzahl an Effizienz verliert, da die Anzahl der Überlappungen zunimmt. Deshalb werden in Kapitel 3 Baumstrukturen vorgestellt, die speziell auf hochdimensionale Elemente ausgelegt sind.

Quadtree

Der in [FB74] vorgestellte Quadtree ist eine mehrdimensionale Datenstruktur, die vor allem in der Computer Grafik Bedeutung erlangt hat. Die Idee wird zunächst im zweidimensionalen Raum vorgestellt, der Ansatz kann analog auf höhere Dimensionen erweitert werden[5].

Der Quadtree ist das Ergebnis einer Kombination der Ideen von binären Suchbäumen und Rastergrafiken. In seiner Basisvariante wird der Quadtree eingesetzt, um auf die einzelnen Rasterpunkte einer Grafik in logarithmischer Laufzeit zugreifen zu können. Die Wurzel des Quadtrees repräsentiert eine quadratische Fläche, die rekursiv solange in jeweils vier gleich große Teile zerlegt wird, bis die gewünschte Auflösung erreicht ist. Jeder innere Knoten des Baumes hat somit vier Kinder und jeder Blattknoten repräsentiert einen Rasterpunkt. Da jeder Rasterpunkt abgebildet wird und gleichzeitig alle Blattknoten auf derselben Ebene im Baum stehen, ist der Baum balanciert und vollständig.

[5] Im dreidimensionalen Raum hat sich die Bezeichnung *Octree* für die entsprechende Datenstruktur etabliert.

Darüber hinaus sind mehrere Varianten des Quadtree bekannt:

- Die einfachste Variante besteht darin, den Baum nur soweit aufzuspannen wie nötig. Wird für einen bestimmten Quadranten keine weitere Unterteilung gebraucht, obwohl die Auflösung noch nicht erreicht ist, so muss für diesen Quadranten kein Teilbaum aufgebaut werden. Diese Situation kann im Anwendungsgebiet Computer Grafik beispielsweise eintreten, wenn alle Rasterpunkte in einem Teilbaum dieselbe Farbe aufweisen. In diesem Fall kann die entsprechende Farbe an einem höher im Baum gelegenen Knoten vermerkt werden.

- Anstatt Quadranten an den inneren Knoten immer in vier gleich große Teil-Quadranten zu unterteilen, ist es auch möglich, bereits mit dieser Aufteilung auf die Eigenschaften der tatsächlich im Baum vorhandenen Elemente zu reagieren. Anschaulich werden im zweidimensionalen Fall an jedem inneren Knoten zwei Trennlinien berechnet, eine waagerechte und eine senkrechte, so dass vier Teile entstehen. Beide Trennlinien können achsparallel verschoben werden. Dies bietet sich vor allem an, um den Baum bei ungleichen Verteilungen der Elemente zu balancieren.

Wie der R-Baum ist auch der Quadtree für hochdimensionale Elemente nur bedingt geeignet. Dies ist in erster Linie darauf zurückzuführen, dass die Anzahl der Kindknoten, die ein innerer Knoten hat, quadratisch mit der Anzahl der Dimensionen wächst. Damit wird die Struktur für Such- oder Bereichsanfragen ineffizient, da an jedem inneren Knoten die Teilbäume identifiziert werden müssen, die weiter zu verfolgen sind.

3 Der S^{3}V Baum

Der S^{3}V Baum orientiert sich im prinzipiellen Aufbau an der in [HSW89] als LSD-Baum beschriebenen mehrdimensionalen Zugriffsstruktur. Aufgrund zusätzlicher Anforderungen und wegfallender Rahmenbedingungen sowie dank des vorgegebenen Einsatzbereiches ergeben sich notwendige Erweiterungen und mögliche Optimierungen, die den S^{3}V Baum vom LSD-Baum abgrenzen und vor allem den Aufbau der Suchstruktur verändern.

Im Folgenden wird zunächst der LSD-Baum nach [Hen90] beschrieben, bevor in Abschnitt 3.2 die Schritte vom LSD-Baum zum S^{3}V Baum dargestellt werden.

3.1 Der LSD-Baum

Beim LSD-Baum handelt es sich um eine Zugriffsstruktur für mehrdimensionale Elemente. Die Zahl der Dimensionen ist nicht begrenzt, der Baum ist insbesondere in der Lage, hochdimensionale Elemente zu verarbeiten. Die einzelnen Elemente werden in Buckets fester Größe verwaltet, jedes Bucket stellt einen Teilraum des gesamten mehrdimensionalen Datenraumes dar. Teilräume unterschiedlicher Buckets sind disjunkt[1]. Die zugrunde liegende Aufteilung wird in einem so genannten Directory verwaltet, dazu wird ein verallgemeinerter k-d-Baum verwendet.

Der k-d-Baum nach [Ben75] stellt ebenfalls eine Zugriffsstruktur für mehrdimensionale Elemente dar. Dabei werden in einem ersten Schritt alle zu verwaltenden Elemente anhand einer Splitposition in der ersten Dimension in zwei disjunkte Mengen aufgeteilt. Als Splitposition bietet sich der Median aller Werte erster Dimension an, so dass die resultierenden Mengen gleich groß sind. Im zweiten Schritt werden die beiden entstandenen Mengen jeweils anhand einer Splitposition in der zweiten Dimension geteilt, so dass schließlich vier disjunkte Mengen vorliegen. Dies wird so lange fortgeführt, bis die Mengen klein genug sind, so dass sie in ein Bucket passen und nicht weiter geteilt werden müssen. Die Größe eines Buckets ist variabel und kann z.B. an die Größe von Speicherblöcken angepasst werden.

Der entscheidende Unterschied vom k-d-Baum zum LSD-Baum ist die Tatsache, dass beim k-d-Baum auf erster Ebene nur anhand der ersten Dimension geteilt wird, auf zweiter Ebene nur anhand der zweiten Dimension, etc. Ist einmal über alle Dimensionen iteriert worden, so wird im nächsten Schritt wieder bei der ersten Dimension begonnen.

Beim LSD-Baum hingegen sind die Splits nicht derart vorgegeben. Die Abkürzung LSD steht für *local split decision*, die unterschiedlichen Aufteilungen in disjunkte

[1] Damit steht der LSD-Baum im Gegensatz zum in [Gut84] vorgestellten R-Baum.

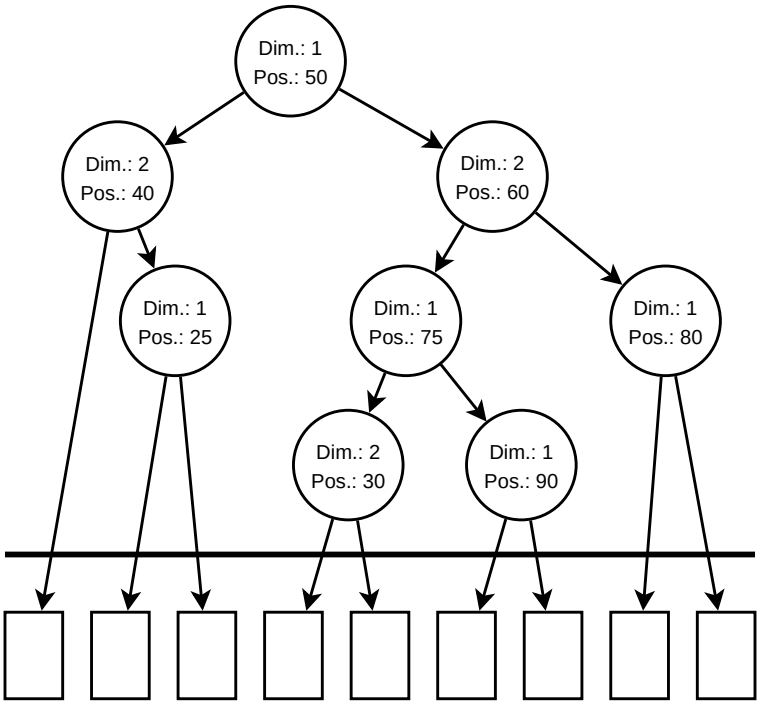

Abbildung 3.1: Beispiel eines LSD-Baumes nach [Hen90, S. 61]

Mengen sind weder in ihrer Dimension noch in ihrer Splitposition vorgegeben. Vielmehr kann jeder Split aufgrund der aktuell vorliegenden Elementmenge optimiert werden und muss sich nicht an anderen Splits orientieren. Abbildung 3.1 zeigt dazu ein Beispiel für zwei Dimensionen. Es fällt auf, dass auf der dritten Ebene unter der Wurzel sowohl ein Split in erster Dimension (Splitposition 90) als auch ein Split in zweiter Dimension (Splitposition 30) stattfindet, damit werden die Kriterien für einen k-d-Baum verletzt.

Durch seine höhere Flexibilität kann sich der LSD-Baum besser als der k-d-Baum an die zu verwaltenden Daten anpassen und für Splits dynamisch Splitdimensionen und Splitpositionen auswählen, die die vorhandenen Elemente möglichst gut verteilen. Anhand welcher Kriterien eine gute Verteilung definiert werden kann, liegt u.a. am beabsichtigten Einsatzgebiet des LSD-Baumes. In den meisten Fällen wird die Optimierung im Hinblick auf Bereichsanfragen erfolgen. Zusätzlich wird beim LSD-Baum von dynamischem Einfügen ausgegangen, d.h. zu dem Zeitpunkt, an dem ein neues Element in die Datenstruktur eingefügt wird, ist nicht bekannt, welche Elemente im Weiteren eingefügt werden sollen.

Dynamisches Einfügen

[Hen90, S. 63ff.] beschreibt dazu einen Einfügealgorithmus, der an der Wurzel beginnend durch das Directory navigiert, bis das Bucket gefunden wurde, das das neue Element aufnehmen müsste. Ist die Bucketkapazität bereits erreicht, so wird ein

Bucket-Split durchgeführt. Dazu werden aufgrund des Inhalts dieses Buckets und des neuen Elements eine Splitdimension und eine Splitposition ermittelt und anstatt des übergelaufenen Buckets wird an dieser Stelle ein neuer Directory-Knoten mit Splitdimension und -position eingefügt, der wiederum auf zwei neue Buckets verweist, die sich die Elemente des übergelaufenen Buckets teilen.

Durch das dynamische Einfügen ergibt sich zusätzlich die Anforderung, die Splitdimensionen und -positionen so zu wählen, dass der LSD-Baum bei künftigen Einfügungen nicht degeneriert. Die einzelnen Splits sind also so durchzuführen, dass die betrachtete Elementmenge möglichst in zwei gleich große Menge geteilt wird – nur so kann die Balancierung gewährleistet werden. Dabei stellt sich das Problem, dass zum Zeitpunkt des Splits noch nicht alle Elemente bekannt sind, die von dieser Splitentscheidung betroffen sein werden, weil sie erst nach dem Split in den LSD-Baum eingefügt werden.

[HSW89] unterscheidet zwei Strategien, um eine solche Splitentscheidung zu treffen:

datenabhängige Splitstrategien. Bei datenabhängigen Splitstrategien wird die Splitentscheidung auf Basis der aktuell im übergelaufenen Bucket befindlichen Elemente getroffen. Nachteil dieses Ansatzes ist das mögliche Degenerieren des Baumes, falls die nachfolgend in den Baum eingefügten Elemente andere Eigenschaften aufweisen als die, die zum Zeitpunkt der Splitentscheidung im fraglichen Bucket waren. Andererseits stellt der Bezug der Splitentscheidung auf tatsächlich vorhandene Elemente einen Vorteil dar, da sich der Baum auf diese Weise an die Elemente anpassen kann.

verteilungsabhängige Splitstrategien. Verteilungsabhängige Strategien treffen die Splitentscheidung unabhängig von den tatsächlich im Baum befindlichen Elementen. Stattdessen wird eine Hypothese über die Verteilung der Elemente herangezogen, die bereits vor dem ersten Einfügen eines Elementes in den Baum getroffen wird. Dies hat offensichtlich den Nachteil, dass die Struktur des Baumes nicht auf tatsächlich eingefügte Elemente reagiert; dem gegenüber steht jedoch der Vorteil, dass für die angestrebte Balancierung des Baumes nicht nur die bereits eingefügten Elemente betrachtet werden, die ein unscharfes Bild von der Gesamtverteilung geben.

Ob beim dynamischen Einfügen in den LSD-Baum von einer verteilungsabhängigen oder von einer datenabhängigen Strategie Gebrauch gemacht wird, hängt somit von dem konkreten Einsatzbereich ab. Generell bietet sich für große Datenmengen eine verteilungsabhängige Strategie an, da große Mengen eher aufgestellte Hypothesen über die Verteilung erfüllen als kleine Mengen. Bei kleinen Datenmengen hingegen kann die Tatsache ausgenutzt werden, dass mit einer datenabhängigen Strategie die Baumstruktur an die konkret vorhandenen Elemente angepasst werden kann.

Suche nach gegebenen Elementen

Die Suche nach einem gegebenen Element stellt im LSD-Baum offensichtlich kein Problem dar. Es wird an der Wurzel beginnend durch das Directory navigiert, wobei an jedem Knoten im Directory gemäß der korrespondierenden Splitentscheidung weiterverzweigt wird. Ist man bei dem fraglichen Bucket angelangt, so muss nur noch untersucht werden, ob das gesuchte Element in dem Bucket enthalten ist oder nicht. Dieser Prozess kann bei einer großen Bucketkapazität optimiert werden, indem innerhalb der Buckets keine lineare, sondern ebenfalls eine baumartige Struktur zur Verwaltung benutzt wird. Im Rahmen der Bestimmung von Dokumentdifferenzen wird jedoch ausschließlich mit kleinen Buckets gearbeitet, so dass dieses Thema hier keine weitere Vertiefung finden soll.

Bereichsanfragen

Bereichsanfragen stellen oft den eigentlichen Verwendungszweck einer mehrdimensionalen Struktur dar. Wäre nur die Suche nach gegebenen Elementen gefordert, so könnten anstelle einer Baumstruktur einfach Hashwerte über alle Dimensionswerte gebildet werden und Übereinstimmungen anhand dieser Hashwerte bestimmt werden. Im Gegensatz dazu besteht das Ziel mehrdimensionaler Suchstrukturen darin, Optimierungen für Bereichsanfragen anzubieten, so dass nicht linear durch alle Elemente navigiert werden muss, um die Antwortmenge zu Bereichsanfragen zu bestimmen.

Mehrdimensionale Suchstrukturen können in diesem Zusammenhang auch als Sekundärindizes in relationalen Datenbanksystemen verwendet werden, um die Bearbeitungszeit von Anfragen wie

```
SELECT * FROM table
WHERE (columnA > 10 AND columnA < 50) AND
      (columnB > 0 AND columnB < 10)
```

zu optimieren. Darüber hinaus bieten solche Bereichsanfragen natürlich die Möglichkeit, alle Elemente zu finden, die in der Nachbarschaft eines gegebenen Elements liegen. Der Begriff der Nachbarschaft kann unterschiedlich definiert werden. Es bietet sich z.B. an, die Distanz zwischen unterschiedlichen Elementen in der Suchstruktur mittels des euklidischen Abstands zu definieren und zwei Elemente als benachbart zu definieren, wenn die Distanz zwischen ihnen einen vorgegebenen Grenzwert nicht überschreitet.

Die Bereichsanfrage geht im LSD-Baum so vor, dass für jeden Knoten im Directory und für jedes Bucket ein Datenraum berechnet wird, der alle Elemente beinhaltet, die sich in dem entsprechenden Teilbaum befinden. Dazu wird für den Wurzelknoten der Datenraum durch den minimalen und maximalen Wert in jeder Dimension über alle im Baum enthaltenen Elemente vorgegeben. Der Datenraum für einen tiefer liegenden Directory-Knoten oder ein Bucket ergibt sich dann durch den Datenraum

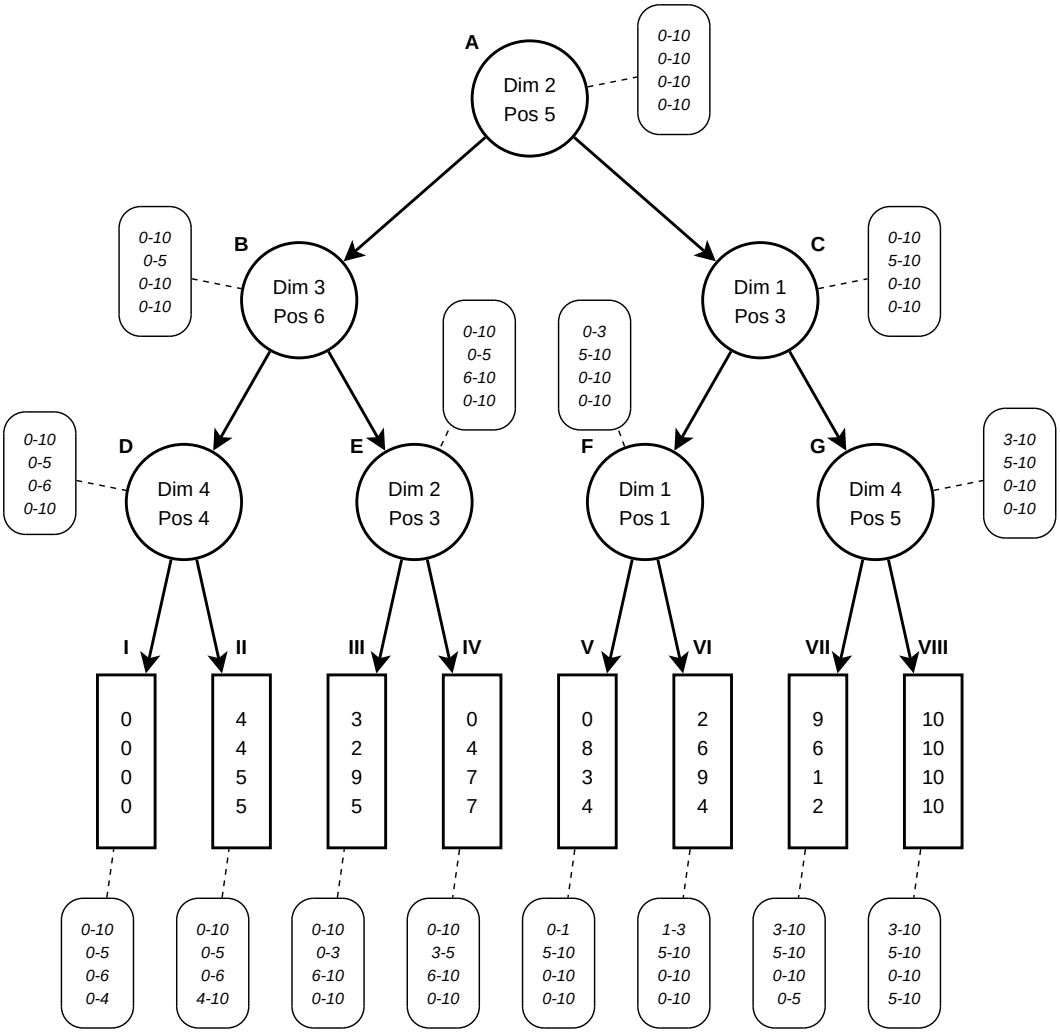

Abbildung 3.2: Beispielsituation für Bereichsanfragen im LSD-Baum

seines Vater-Directory-Knotens, wobei er an der Splitdimension gemäß der Splitposition reduziert wird.

Abbildung 3.2 zeigt dazu ein Beispiel, bei dem vierdimensionale Elemente in einem LSD-Baum mit Bucketgröße 1 abgespeichert wurden[2]. In den Directory-Knoten steht jeweils die Splitdimension und die Splitposition. Außerdem ist zusätzlich an jedem Knoten der Datenbereich vermerkt, der ihm während der Bereichsanfrage im LSD-Baum zugewiesen wird. Diese Angaben sind an der Wurzel beginnend zu lesen. Am mit A bezeichneten Wurzelknoten ist lediglich der gesamte Datenraum (in diesem Fall der Bereich von 0 bis 10 für jede der vier Dimensionen) bekannt; dazu müssen im Zuge des Einfügens von Elementen in den LSD-Baum die Minimal- und Maximalwerte für jede Dimension für den gesamten Baum aktuell gehalten werden. Auf der nächsten Ebene wird der gesamte Datenraum in zwei disjunkte Teilräume aufgeteilt[3].

Diese disjunkten Teilräume ergeben sich durch den Datenraum des Vater-Knotens im Directory, wobei dieser an der Splitdimension gemäß der Splitposition verkleinert wird. So ist im Beispiel der Datenraum von Directory-Knoten B identisch mit dem von Wurzel-Knoten A, bis auf die Tatsache, dass an zweiter Dimension nur Werte von 0 bis 5 in den Datenraum von Knoten B fallen. Entsprechend schränkt Directory-Knoten C den Datenraum an zweiter Dimension auf Werte zwischen 5 und 10 ein.

Jede Bereichsanfrage beginnt an der Wurzel. Teilbäume werden im Directory genau dann weiter verfolgt, wenn der Suchbereich und der Datenraum des Teilbaumes nicht disjunkt sind. Da die Teilräume von Subknoten im Directory oder Buckets automatisch disjunkt vom Suchbereich sind, wenn der Teilraum eines Vaterknotens disjunkt ist, werden selbstverständlich nur die Teilräume berechnet, die tatsächlich benötigt werden.

Welche Form der Suchbereich hat und wie er zu spezifizieren ist, wird nicht vorgeschrieben. Bei kreisförmigen Suchbereichen bietet es sich an, das zentrale Element des Bereichs festzulegen und die Maximaldistanz anzugeben, die zwischen Elementen des Suchbereiches und dem zentralen Element bestehen darf. Anschaulich kreisförmig ist dies natürlich nur im zweidimensionalen Fall, die Idee lässt sich aber nahtlos auf höhere Dimensionen übertragen.

Anhand eines kreisförmigen Suchbereiches und des Beispiels in Abbildung 3.2 wird noch einmal die Vorgehensweise der Bereichsanfrage erläutert. Es wird für jeden Directory-Knoten und jedes Bucket berechnet, wie groß die Minimaldistanz zwischen dem korrespondierenden Datenraum und dem zentralen Element im Suchbereich ist. Für ein zentrales Element Z mit den Dimensionswerten (4,5; 4,5; 4,5; 4,5) und einen Suchbereichsradius von 1 gemäß euklidischem Abstand würden sich demnach für die Knoten die in Tabelle 3.1 genannten Werte ergeben (alle Werte wurden auf 2 Nachkommastellen gerundet). Zusätzlich ist zu jedem Knoten vermerkt, ob er im

[2]Das Beispiel wird in Abschnitt 3.2 noch einmal verwendet, um die verbesserte Bereichsanfrage im S³V Baum zu beschreiben.

[3]Für dieses Beispiel wird angenommen, dass Elemente, deren Wert an der Splitdimension exakt der Splitposition entspricht, jeweils in den rechten Teilbaum einsortiert werden. Auf die genaue Behandlung dieses Falles wird bei der Erläuterung des S³V Baumes eingegangen.

Knoten im LSD-Baum	Distanz zu Z	besuchen	berechnen
Directory-Knoten A	0,00	ja	ja
Directory-Knoten B	0,00	ja	ja
Directory-Knoten C	0,50	ja	ja
Directory-Knoten D	0,00	ja	ja
Directory-Knoten E	1,50	nein	ja
Directory-Knoten F	1,58	nein	ja
Directory-Knoten G	0,50	ja	ja
Bucket I	0,50	ja	ja
Bucket II	0,00	ja	ja
Bucket III	2,12	nein	nein
Bucket IV	1,50	nein	nein
Bucket V	3,54	nein	nein
Bucket VI	1,58	nein	nein
Bucket VII	0,50	ja	ja
Bucket VIII	0,71	ja	ja

Tabelle 3.1: Minimaldistanzen zwischen Datenräumen und dem zentralen Element (4,5; 4,5; 4,5; 4,5) im Suchbereich im LSD-Baum

Zuge der Bereichsanfrage besucht wird. Dies ist immer dann der Fall, wenn die minimale Distanz zwischen dem korrespondierenden Teilraum und dem zentralen Element kleiner als der Radius des Suchbereiches ist. Berechnet werden muss darüber hinaus der Datenraum immer dann, wenn der direkte Vaterknoten des Knotens besucht worden ist, um vom Vaterknoten aus die Entscheidung zu treffen, welche der Kindknoten weiter verfolgt werden müssen.

Alle Elemente in den Buckets, die im Rahmen der Bereichsanfrage besucht werden, müssen anschließend noch darauf geprüft werden, ob sie im Suchbereich liegen. Im Beispiel trifft das lediglich auf das Element in Bucket II zu, seine Distanz zum zentralen Element des Suchbereiches beträgt genau 1. Alle anderen Elemente werden von der Bereichsanfrage verworfen.

Nachbarsuche

Ziel der Nachbarsuche ist es, von einem Zielelement ausgehend alle anderen Elemente im LSD-Baum nach der Distanz zum Zielelement geordnet auszugeben. Mittels einer entsprechenden Abbruchbedingung kann der zugrunde liegende Algorithmus offensichtlich so geändert werden, dass nur die nächsten n Nachbarn zurückgegeben werden, wobei n eine Zahl zwischen 1 und der Gesamtanzahl an Elementen im LSD-Baum ist.

Der Algorithmus zur Nachbarsuche orientiert sich eng am Algorithmus zur Bereichsabfrage und soll hier nur kurz dargestellt werden. Für eine ausführliche Be-

trachtung wird auf [Hen90, S. 81ff.] verwiesen.

Die Bereichsanfrage wird um zwei Datenstrukturen erweitert, eine zum Speichern von Directory-Knoten und Buckets, die andere zum Speichern von Elementen. Beide Datenstrukturen sortieren neue Einträge automatisch nach der Distanz zum Zielelement ein. Auf dem Weg von der Wurzel zu dem Bucket, das das Zielelement enthalten müsste[4], werden alle nicht besuchten Directory-Knoten in der ersten Datenstruktur abgelegt und anschließend geordnet nach der Distanz zum Zielelement abgearbeitet. Elemente aus bereits besuchten Buckets werden in die zweite Datenstruktur eingefügt und von dort in die geordnete Antwortmenge übernommen, falls ihre Distanz zum Zielelement kleiner ist als der kleinste Abstand zwischen dem Zielelement und dem nächsten Directory-Knoten oder Bucket in der ersten Datenstruktur.

Elemente bereits besuchter Buckets können nicht sofort in die Antwortmenge übernommen werden, da die Tatsache, dass ein Element im selben Bucket ist wie das Zielelement, noch keine Aussage über die Distanz zwischen diesen beiden Elementen zulässt. Sie wurden lediglich anhand bestimmter Splitdimensionen identisch gruppiert, könnten jedoch an anderen Dimensionen völlig unterschiedliche Werte aufweisen. Festzuhalten bleibt, dass durch das Ordnen der Elemente im Vergleich zur Bereichsanfrage zusätzlicher Aufwand entsteht.

3.2 Anpassungen im S³V Baum

Der S³V Baum wurde als Datenstruktur entwickelt, um die bisherige Vergleichsphase von Elementen mit quadratischer Laufzeit während der Bestimmung von Dokumentdifferenzen zu optimieren.

Der Grundgedanke dazu ist beim Vergleich zweier Dokumente A und B, dass für jeden bei der Differenzbestimmung relevanten Elementtyp ein S³V Baum angelegt wird, der alle Elemente diesen Typs aus Dokument A enthält. In der Vergleichsphase werden dann für jedes Element des entsprechenden Typs aus Dokument B nur noch die Elemente aus Dokument A betrachtet, die diesem Element ähnlich sind. Die Ähnlichkeit wird über die Distanz zwischen Elementen definiert, so dass für die Bestimmung der Menge ähnlicher Elemente die Bereichsanfrage benutzt werden kann. Wie Dokumentelemente auf Elemente mit Dimensionen abgebildet werden, wird in Kapitel 4 dargestellt.

Bei der Entwicklung des S³V Baumes wurde der LSD-Baum als Ausgangspunkt genommen und untersucht, inwieweit sich durch den vorgegebenen Einsatzbereich des Baumes mögliche Optimierungen der Struktur ergeben. Der hier beschriebene S³V Baum stellt das Ergebnis dieser Untersuchungen dar.

Der Name S³V geht auf die Abkürzung SSSV zurück, die für *similarity search sparse vector* steht und damit zwei der Haupteigenschaften der Struktur aufzeigt. Zum einen handelt es sich um eine Datenstruktur, die für Ähnlichkeitssuche und somit für Bereichsanfragen optimiert ist. Zum anderen ist über die im Baum zu

[4] Ob das Zielelement tatsächlich im Baum vorhanden ist, spielt für den Algorithmus keine Rolle.

verwaltenden Elemente bekannt, dass sie zwar mehrdimensional zu indizieren sind, jedoch oft an vielen Dimensionen den Wert 0 aufweisen[5].

Im Folgenden wird ausgehend vom LSD-Baum beschrieben, aus welchen Gründen und auf welche Weise sich der S³V Baum vom LSD-Baum unterscheidet.

Hauptspeicherorientierung

Beim LSD-Baum wird davon ausgegangen, dass nicht die gesamte Baumstruktur im Hauptspeicher gehalten werden kann. Daraus ergeben sich zwei Implikationen:

- Die entscheidende Anforderung im Zusammenhang mit jeder Laufzeitoptimierung ist, die Zahl der Festplattenzugriffe zu reduzieren. Um das zu erreichen, stellt [Hen90] das Konzept der externen Balancierung vor, das auf der partiellen Paginierung von Binärbäumen beruht. Ein Baum gilt genau dann als extern balanciert, wenn sich die Anzahl der externen Seiten auf zwei beliebigen Pfaden in dem Baum von der Wurzel zu einem der Blätter maximal um 1 unterscheidet.

- Die Größe der Buckets im LSD-Baum sollte sich an der Größe von Speicherblöcken orientieren, so dass zum Laden eines Buckets in den Hauptspeicher nicht mehrere Speicherblöcke übertragen werden müssen.

Aufgrund heutiger Hauptspeichergrößen und durch relativ kleine zu verwaltende Elementmengen kann der S³V Baum komplett im Hauptspeicher gehalten werden. Damit fallen sowohl die Einschränkungen im Bezug auf die Bucketgröße als auch die externe Balancierung weg.

Als Bucketgröße bietet sich im hauptspeicherorientierten Fall 1 an, da der Baum durch möglichst weitgehende Verzweigungen am besten strukturiert werden kann und somit die Vorteile einer mehrdimensionalen Datenstruktur am ehesten zum Tragen kommen. Gerade bei Bereichsanfragen müssen in Frage kommende Buckets linear durchsucht werden, dies kann durch eine Bucketgröße von 1 verhindert werden. Nachteil einer sehr kleinen Bucketgröße ist jedoch, dass die Anzahl notwendiger Bucket-Splits beim Einfügen in den Baum entsprechend zunimmt. Wie sich unterschiedliche Bucketgrößen auf die Laufzeit der Bestimmung von Dokumentdifferenzen auswirken, wird in Kapitel 6 untersucht.

Es bleibt die Frage, wie der S³V Baum trotz fehlender externer Balancierung ausgeglichen werden kann.

Statisches Einfügen

Aus dem Einsatzbereich der Bestimmung von Dokumentdifferenzen heraus ergibt sich, dass dynamisches Einfügen im S³V Baum nicht notwendig ist. Alle Elemente, die der Baum verwalten soll, sind bereits vor dem ersten Einfügen eines Elements

[5] Der Aufbau der Vektoren wird in Kapitel 4 erläutert.

bekannt – nämlich alle Elemente eines Elementtyps in einem der zu vergleichenden Dokumente – und die Bereichsanfragen zur Suche nach ähnlichen Elementen starten erst, nachdem alle Elemente eingefügt worden sind.

Weiterhin müssen im S³V Baum keine Löschungen vorgenommen werden. Wurden einmal alle Elemente eines Elementtyps für Bereichsanfragen eingefügt, so besteht kein Anlass, sie wieder zu löschen.

Aufgrund dieser Zusatzinformationen ergibt sich nun die Möglichkeit, den S³V Baum zu balancieren, indem der Baum nicht durch schrittweises Einfügen der Elemente sondern systematisch aufgrund aller einzufügenden Elemente aufgebaut wird. Dazu werden in einem ersten Schritt alle Elemente betrachtet, die verwaltet werden sollen, und gemäß der Splitstrategie in zwei disjunkte Mengen unterteilt. Die entstandenen Mengen werden im zweiten Schritt wiederum gemäß der Splitstrategie in zwei Mengen unterteilt, so dass sich der S³V Baum aufspannt.

An die Splitstrategie werden in diesem Zusammenhang zwei Anforderungen gestellt:

- Um die Balancierung des S³V Baumes zu gewährleisten, sollte jeder Split die Eingangsmenge in zwei gleich große Ausgangsmengen unterteilen.
- Die Splits sollten so durchgeführt werden, dass bereits beim Aufbau der Baumstruktur Optimierungen bezüglich der Bereichsanfragen vorgenommen werden.

Das statische Einfügen von Elementen in den S³V Baum wird anhand von Algorithmus 1 verdeutlicht. Es erfolgt rekursiv durch den Aufruf der Operation *insertRecursively* mit allen einzufügenden Elementen als Parameter. Diese Operation liefert als Ergebnis entweder ein Bucket oder einen neuen Directory-Knoten mit Splitdimension und Splitposition zurück. Übersteigt die Anzahl der Elemente, die eingefügt werden soll, die Bucketkapazität, so wird eine so genannte *split line* berechnet, die aus der Splitdimension und der Splitposition besteht. Die entsprechende Operation ist generisch und wird über den Parameter *splitter* gesteuert, in dem die Splitstrategie abgebildet wird. Aufgrund der *split line* wird dann der neue Directory-Knoten gebildet und die Elementmenge wird gemäß der Splitentscheidung in zwei Mengen geteilt. Elemente, deren Wert an der Splitdimension exakt der Splitposition entspricht, können in beide Teilbäume verschoben werden. Bei der Verteilung der entsprechenden Elementmenge *esEqual* (vgl. Zeile 13 im Algorithmus) wird darauf geachtet, dass der Baum möglichst ausgeglichen wird. Dazu werden die Elemente aus *esEqual* in den Teilbaum verschoben, der bisher weniger Elemente enthält. Die Berechnung der Teilbäume erfolgt rekursiv durch Aufrufe von *insertRecursively* für beide Teilbäume.

Splitstrategie und Optimierung der Bereichsanfragen

Da der S³V Baum nach dem einmaligen Einfügen der Elemente ausschließlich für Bereichsanfragen eingesetzt wird, kann die Optimierung der Struktur ausschließlich im Hinblick auf diese Anfragen geschehen.

Algorithmus 1 Statisches Einfügen in den S^3V Baum

```
 1: function insertRecursively(Elements es) : Node
 2:     Node node
 3:     if es.size <= bucketSize then
 4:         node = new Bucket ()
 5:         node.addElements (es)
 6:     else
 7:         (sPos, sDim) = computeSplitLine (es, splitter)
 8:         left = new Node ()
 9:         right = new Node ()
10:         node = new DictionaryNode (sDim, sPos, left, right)
11:         Elements esLeft = es.getSmaller (sDim, sPos)
12:         Elements esRight = es.getGreater (sDim, sPos)
13:         Elements esEqual = es.getEqual (sDim, sPos)
14:         for all element in esEqual do
15:             if esLeft.size < esRight.size then
16:                 esLeft.addElement (element)
17:             else
18:                 esRight.addElement (element)
19:             end if
20:         end for
21:         left = insertRecursively (esLeft)
22:         right = insertRecursively (esRight)
23:     end if
24:     return node
25: end function
```

Wie bereits in Abschnitt 3.1 beschrieben, werden während der Bereichsanfrage für jeden Knoten im Directory und für jedes Bucket Teilräume berechnet, die alle Elemente im entsprechenden Teilbaum enthalten. Teilbäume werden genau dann weiterverfolgt, wenn der Suchbereich und der entsprechende Teilraum nicht disjunkt sind. Daraus folgt, dass die Laufzeit von Bereichsanfragen durch kleinere Teilräume optimiert werden kann. Wie das Beispiel in Abbildung 3.2 deutlich macht, sind die Teilräume, die während der Abarbeitung von Bereichsanfragen für die Knoten berechnet werden, deutlich größer als sie sein müssten, um alle Elemente im Teilbaum zu enthalten. Die Tatsache, dass kein dynamisches Einfügen im S^3V Baum stattfindet, kann somit ausgenutzt werden, um hier eine Optimierung vorzunehmen.

Dazu werden Directory-Knoten im S^3V Baum um eine zusätzliche Splitposition erweitert. Ein Directory-Knoten hat damit neben einer Split-Dimension den Maximalwert an dieser Dimension im linken Teilbaum und den Minimalwert an dieser Dimension im rechten Teilbaum. Stimmen diese beiden Werte überein, liegt ein klassischer LSD-Baum vor. Unterscheiden sich die Werte jedoch, so kann im Vergleich zum LSD-Baum Laufzeit gewonnen werden, da die berechneten Teilräume kleiner werden.

Abbildung 3.3 zeigt dazu den S^3V Baum, der entsteht, wenn das Beispiel aus Abbildung 3.2 um eine zweite Splitposition erweitert wird. An der Wurzel sind beispielsweise jetzt die Splitpositionen 4 und 6 eingetragen, da 4 der größte tatsächlich vorkommende Wert zweiter Dimension im linken Teilbaum und 6 der kleinste tatsächlich vorkommende Wert zweiter Dimension im rechten Teilbaum ist. Gegenüber der einen Splitposition 5 im korrespondierenden LSD-Baum kann somit zusätzliche Information gewonnen werden und die Teilräume der Directory-Knoten B und C (und aller darunter liegenden Knoten!) können entsprechend verkleinert werden.

Diese strukturelle Erweiterung des Baumes kann nun bei der Wahl der Splitstrategie einbezogen werden. Optimal wäre somit beim statischen Einfügen ein Split, der

- die Elementmenge in zwei gleich große Mengen unterteilt (zwecks Balancierung) und gleichzeitig
- eine möglichst große Differenz zwischen den beiden Splitpositionen aufweist (zwecks Verkleinerung der von den Directory-Knoten repräsentierten Teilräumen).

Diese beiden Ziele können sich selbstverständlich im Einzelfall widersprechen und es liegt an der konkreten Elementmenge, wie genau vorgegangen wird. Eine Möglichkeit besteht darin, die Elementmenge nacheinander nach jeder Dimension zu sortieren und diejenige Dimension als Splitdimension zu verwenden, bei der die Differenz zwischen dem $(\frac{n}{2})$. und dem $(\frac{n}{2}+1)$. Dimensionswert am größten ist, wenn n der Elementanzahl entspricht. Damit würden unter der Nebenbedingung Balancierung die Teilräume möglichst klein gestaltet. Dabei gilt, dass dieser Algorithmus am besten funktioniert, wenn viele Dimensionen vorliegen, da die Chance, eine für den Split günstige Dimension zu finden, dann größer ist.

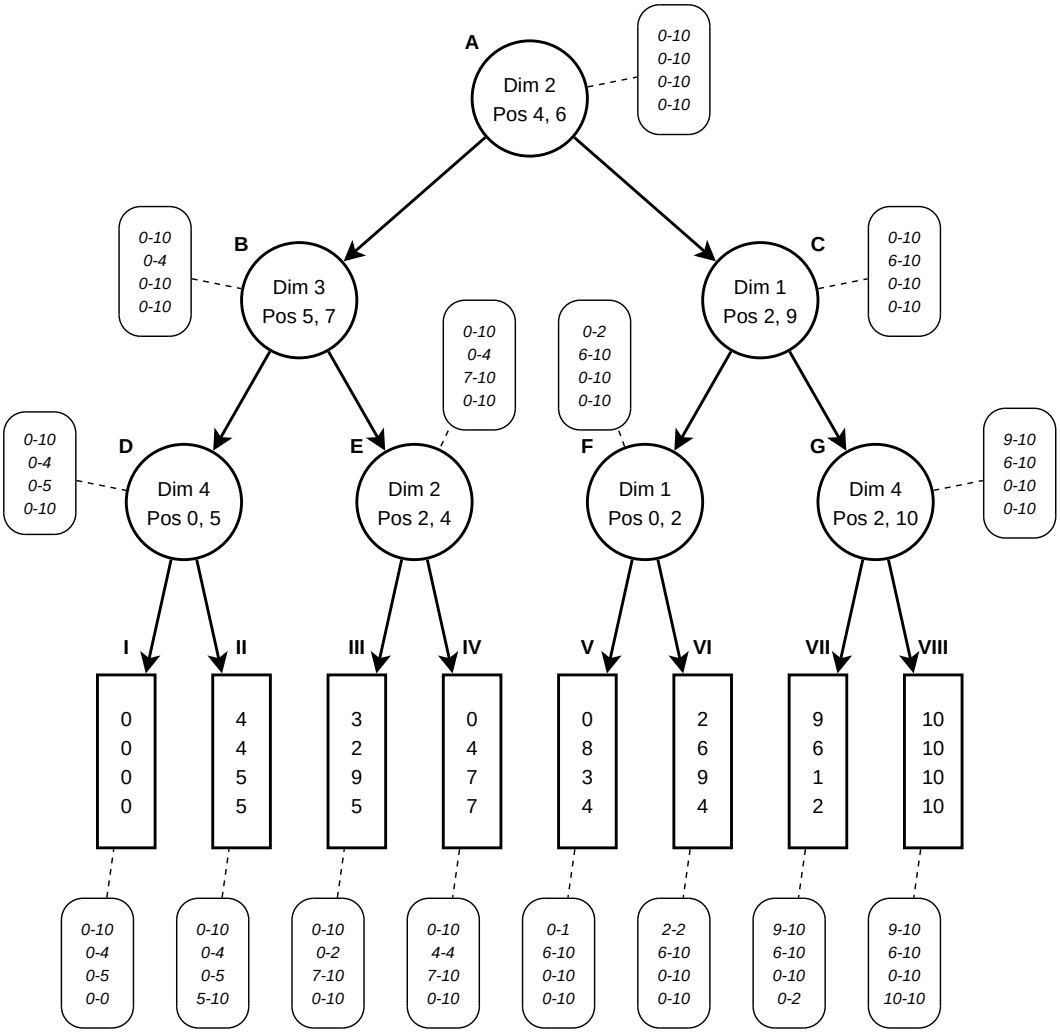

Abbildung 3.3: Beispielsituation für Bereichsanfragen im S^3V Baum

Knoten im S^3V Baum	Distanz zu Z	besuchen	berechnen
Directory-Knoten A	0,00	ja	ja
Directory-Knoten B	0,00	ja	ja
Directory-Knoten C	1,50	nein	ja
Directory-Knoten D	0,50	ja	ja
Directory-Knoten E	2,55	nein	ja
Directory-Knoten F	2,92	nein	nein
Directory-Knoten G	4,74	nein	nein
Bucket I	4,53	nein	ja
Bucket II	0,71	ja	ja
Bucket III	3,54	nein	nein
Bucket IV	2,55	nein	nein
Bucket V	4,74	nein	nein
Bucket VI	2,92	nein	nein
Bucket VII	5,36	nein	nein
Bucket VIII	7,26	nein	nein

Tabelle 3.2: Minimaldistanzen zwischen Datenräumen und dem zentralen Element (4,5; 4,5; 4,5; 4,5) im Suchbereich im S^3V Baum

Inwieweit diese Annahmen auf die Bestimmung von Dokumentdifferenzen zutreffen, wird in Kapitel 6 beschrieben. Der Vollständigkeit halber wird an dieser Stelle noch dargestellt, welche Einsparungen sich bei der Bereichsanfrage im Beispiel aus Abbildung 3.3 im Vergleich zum LSD-Baum ergeben. Dazu sind in Tabelle 3.2 die neu berechneten Minimaldistanzen zwischen den Datenräumen und dem zentralen Element im Suchbereich dargestellt. Es fällt auf, dass sich die Anzahl der besuchten Knoten im Vergleich zu Tabelle 3.1 von 9 auf 4 verringert, während die Zahl der Knoten, für die der korrespondierende Teilraum berechnet werden muss, von 11 auf 7 zurück geht.

Theoretisch könnten die Teilräume noch weiter verkleinert werden, indem für jeden Knoten im Directory und für jedes Bucket der minimale Teilraum berechnet würde, der alle Elemente im entsprechenden Teilbaum enthält. Dies stellt jedoch einen enormen Aufwand dar, da für jeden Knoten und für jede Dimension der maximale und der minimale Wert im entsprechenden Teilbaum bestimmt werden müssten. Vor allem im hochdimensionalen Fall steht der zusätzliche Aufwand in keinem Verhältnis zur möglichen Laufzeiteinsparung.

Effiziente Berechnung der Teilräume

Im Rahmen der Bereichsanfrage mit kreisförmigen Suchbereichen wird an jedem besuchten inneren Knoten des S^3V Baumes die Distanz zwischen dem korrespondierenden Teilraum und dem zentralen Element des Suchbereiches bestimmt. Diese

Berechnung ist aufwändig, weil für die Ermittlung der Distanz jede der Dimensionen betrachtet werden muss.

Da sich jeder Teilraum eines inneren Knotens im Vergleich zum Teilraum seines Vaterknotens jedoch nur an einer Dimension unterscheidet – nämlich an der Splitdimension des Vaterknotens – kann die Berechnung optimiert werden. Damit ergibt sich die Distanz d an einem Knoten aus der Distanz d_f an seinem Vaterknoten durch

$$d = \sqrt{(d_f)^2 - (d_{sDim\_f})^2 + (d_{sDim})^2}. \tag{3.1}$$

d_{sDim_f} gibt die Distanz an, die an der Splitdimension $sDim$ zwischen dem zentralen Element des Suchbereiches und dem Teilraum des Vaterknotens vorliegt. Die Distanz an der Splitdimension $sDim$ am aktuellen Knoten wird entsprechend mit d_{sDim} bezeichnet.

Die Laufzeit der Berechnung kann weiter verbessert werden, wenn anstelle von einfachen Werten konsequent quadrierte Werte verwendet werden. Dazu wird der Radius des Suchbereiches vor den Bereichsanfragen quadriert und auf das Ziehen der Wurzel bei den Distanzberechnungen verzichtet. Die veränderte Formel lautet

$$d' = d'_f - (d_{sDim\_f})^2 + (d_{sDim})^2, \tag{3.2}$$

wobei d' und d'_f die quadrierten Werte von d bzw. d_f bezeichnen. Die Berechnung der Distanz über alle Dimensionen muss lediglich für den Wurzel-Knoten stattfinden. Dies kann ebenfalls entfallen, wenn bekannt ist, dass das zentrale Element des Suchbereiches selbst im Baum enthalten ist. Im dem Fall wird die Distanz mit dem Wert 0 initialisiert, da das zentrale Element Teil des Datenraumes der Wurzel sein muss.

Dünnbesetzte Vektoren

Im S^3V Baum werden Optimierungen für dünnbesetzte Vektoren vorgenommen. Der Vektor für ein Element ist durch sämtliche Paare aus Dimension und entsprechendem Dimensionswert für dieses Element definiert. Es hat sich gezeigt, dass im Zuge der Bestimmung von Dokumentdifferenzen der Dimensionswert 0 relativ häufig vorkommt. Um nicht alle Dimension/Wert-Paare speichern zu müssen, werden im S^3V Baum nur die Paare gespeichert, bei denen der Dimensionswert ungleich 0 ist. Die interne Realisierung erfolgt über Java *HashMaps*, in denen eine Zuordnung von einem Vektorindex zu einem Zahlwert stattfindet.

Durch diese Erweiterung kann die Berechnung von Distanzen zwischen Elementen optimiert werden. Gemäß euklidischem Abstand ist die Distanz zwischen zwei Vektoren $\vec{x}$ und $\vec{y}$ definiert als

$$d(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}. \tag{3.3}$$

Daraus ergibt sich, dass während der Distanzbestimmung nur die Vektorindizes betrachtet werden müssen, die bei mindestens einem der betrachteten Punkte einen

Wert ungleich 0 aufweisen. Es muss nicht über alle vorhandenen Vektorindizes iteriert werden. Zusätzlich kann durch die Implementierung dünnbesetzter Vektoren natürlich Speicherplatz eingespart werden.

Möglichkeiten zur Balancierung bei dynamischem Einfügen

In diesem Absatz wird kurz dargestellt, warum eine Balancierung des S^3V Baumes[6] mit dynamischem Einfügen problematisch und laufzeitaufwändig ist.

Zum Balancieren von Binärbäumen mit dynamischem Einfügen und Löschen von Elementen existieren viele Ansätze. Exemplarisch werden hier der AVL-Baum nach [AVL62] und der Rot-Schwarz-Baum nach [Bay72] betrachtet.

Der AVL-Baum garantiert eine maximale Höhe des Baumes von $1,4404 \cdot \log n$, wobei n die Anzahl der Knoten im Baum ist. Der Grundgedanke dabei ist, sicherzustellen, dass sich an jedem Knoten im Baum die Höhe des linken Teilbaumes von der Höhe des rechten Teilbaumes um maximal 1 unterscheidet. Wird diese Bedingung durch eine Einfüge- oder Löschoperation verletzt, so findet eine so genannte Rotation statt. Bei Rotationen sind verschiedene Fälle zu unterscheiden, im Prinzip werden aber jeweils Knoten vertauscht, indem – bezogen auf den nicht balancierten Teilbaum – der Wurzelknoten aus dem größeren Unterbaum als neue Wurzel des Teilbaumes genommen wird und die bisherige Wurzel in den anderen Unterbaum verschoben wird.

Dieses Vorgehen kann nicht nahtlos auf den mehrdimensionalen Fall übertragen werden. Wenn die zu vertauschenden Directory-Knoten unterschiedliche Splitdimensionen aufweisen, so ist nicht gewährleistet, dass der Baum nach dem Vertauschen noch korrekt aufgebaut ist. Dazu müssten die gesamten Unterbäume betrachtet werden und die Rotationen würden sich fortpflanzen. Im *worst case* müssen die Unterbäume komplett neu aufgebaut werden.

Beim Rot-Schwarz-Baum gilt eine ähnliche Argumentation. Der Baum garantiert, dass der längste Weg von der Wurzel zu einem der Blätter höchstens doppelt so lang ist wie der kürzeste Weg. Da dies aber ebenfalls durch ein Vertauschen von Knoten erreicht wird, muss auch hier im mehrdimensionalen Fall überprüft werden, inwiefern die Unterbäume noch korrekt aufgebaut sind. Verbesserungen ließen sich im mehrdimensionalen Fall erreichen, indem die Variabilität der Splitdimensionen eingeschränkt wird und somit Nebenbedingungen für die Rotationen bekannt sind. Damit würde aber gerade der Vorteil der Flexibilität des S^3V Baumes verloren gehen.

Einen komplett anderen Ansatz wählt der UB-Baum nach [BM98]. Durch bitweise Verschränkung der Dimensionswerte berechnet man für jedes Element einen so genannten Z-Wert, der dann für die Sortierung der Elemente genutzt wird. Die Verwendung dieses Ansatzes hätte im Zusammenhang mit der Bestimmung von Dokumentdifferenzen zwei Nachteile: Zum einen ist der UB-Baum nicht für hochdimensionale Elemente ausgelegt, zum anderen wird für die Berechnung der Z-Werte ein Gitter zugrunde gelegt, das den gesamten Datenraum in gleich große Teilräume

[6] Für den LSD-Baum gilt dieselbe Argumentation.

aufteilt. Damit kann die Struktur – im Gegensatz zum S^3V Baum – nicht auf ungleichmäßige Verteilungen reagieren.

Laufzeitaussagen

Abschließend wird noch die Laufzeit von Einfügeoperationen und Bereichsanfragen im S^3V Baum betrachtet.

Statisches Einfügen. Die kritische Operation beim statischen Einfügen von Elementen in den S^3V Baum ist das Aufteilen der Elementmengen in zwei Ausgangsmengen gemäß der Splitstrategie. Diese Operation muss einmal für jeden Directory-Knoten aufgerufen werden. Wird eine Splitstrategie vorausgesetzt, die die Balancierung des Baumes gewährleistet, so beträgt die Zahl der Directory-Knoten bei n Elementen im Baum $\frac{n}{b} - 1$, wenn b die Bucketkapazität bezeichnet. Bei einer Bucketgröße von 1 entspricht die Anzahl der notwendigen Splits also $n - 1$ und ist somit linear von der Zahl der Elemente abhängig.

Dabei gilt zu beachten, dass nur beim ersten Split alle Elemente betrachtet werden müssen. Bei den Splits auf tieferen Ebenen im Baum werden nur noch entsprechend kleine Teilmengen geteilt, so dass die Splits weniger aufwändig sind. Bei einem vollständigen Baum und einer Bucketgröße von 1 teilt über die Hälfte der Splits lediglich je zwei Elemente.

Bereichsanfragen. Die Laufzeit von Bereichsanfragen ist im *worst case* ebenfalls aus $O(n)$, nämlich dann, wenn alle im Baum gespeicherten Elemente in den Suchbereich fallen. In diesem Fall müssen sowohl alle Buckets als auch alle Directory-Knoten betrachtet werden und es müssen alle Teilräume berechnet werden. Eine Bucketgröße von 1 vorausgesetzt, besteht ein S^3V Baum mit n Elementen aus ebenso vielen Buckets und $n - 1$ Directory-Knoten, es müssten also $2n - 1$ Teilräume berechnet werden.

Diese Laufzeit lässt sich im *worst case* nicht verbessern – liegen alle Elemente im Suchbereich, dann müssen auch alle Elemente betrachtet und zurückgegeben werden. Dieser Fall tritt in der Praxis jedoch höchst selten auf, so dass die Bereichsanfrage im *average case* eine deutlich bessere Laufzeit aufweist. Durch die oben beschriebene Splitstrategie mit zwei Splitpositionen pro Directory-Knoten kann die Wahrscheinlichkeit, dass ein Knoten im S^3V Baum während einer Bereichsanfrage betrachtet werden muss, weiter gesenkt werden. Außerdem ist durch die Balancierung beim statischen Einfügen die maximale Höhe des Baumes durch $O(\log n)$ beschränkt, der Baum kann nicht degenerieren.

In Kapitel 6 wird untersucht, wie sich der S^3V Baum im vorgegebenen Anwendungsfall der Bestimmung von Dokumentdifferenzen verhält. Dabei wird auch empirisch dargestellt, wie viele Knoten für die Bereichsanfrage tatsächlich betrachtet werden müssen.

4 Abbildung der Elemente

Wie die Ergebnisse der vorhergehenden Kapitel zeigen, basieren alle Datenstrukturen, die als geeignet für die Optimierung der Bestimmung von Dokumentdifferenzen eingeordnet werden, darauf, dass die zu vergleichenden Elemente als numerische Vektoren vorliegen oder zumindest auf numerische Vektoren abgebildet werden können. Dieses Kapitel beschreibt einen möglichst generischen Weg zur Abbildung von Dokumentelementen auf Vektoren.

Die entscheidende Anforderung an die Abbildung besteht darin, dass die Kriterien für die Ähnlichkeit von Elementen erhalten bleiben, damit die Vektoren eingesetzt werden können, um Ähnlichkeitsaussagen zu treffen. Dazu werden in Abschnitt 4.1 zunächst unabhängig von den Vektoren Kriterien für die Ähnlichkeit von Elementen vorgestellt, die sich in der Praxis bewährt haben. Die Betrachtung beschränkt sich zunächst auf den Dokumenttyp UML-Klassendiagramm. Inwieweit sich die Erkenntnisse zu diesem Dokumenttyp auf andere Dokumenttypen übertragen lassen, wird in Abschnitt 4.6 betrachtet.

Mit metrischen und lexikalischen Indizes werden in den Abschnitten 4.2 und 4.3 die beiden Mechanismen vorgestellt, die für die Abbildung benutzt werden. Außerdem widmen sich die Abschnitte 4.4 und 4.5 der Frage, wie eine sinnvolle Konfiguration der Abbildung erfolgen kann.

4.1 Ähnlichkeitskriterien für Dokumentelemente

[Weh04] beschreibt unter dem Titel „Ein XMI-basiertes Differenzwerkzeug für UML-Diagramme“ einen Algorithmus zum Vergleich zweier UML-Dokumente. Dabei liegt der Fokus auf den Dokumenttypen Klassendiagramm und Zustandsdiagramm. Der Hauptteil des Algorithmus unterteilt sich, wie in Abschnitt 2.5 beschrieben, in eine Hashing-Phase und eine Matching-Phase.

In der Matching-Phase wird elementtypspezifisch jedes Element im ersten Dokument mit jedem Element im zweiten Dokument verglichen, sofern die Elemente nicht bereits in der Hashing-Phase einem korrespondierenden Element zugeordnet wurden. [Weh04, S. 60f.] listet für UML-Klassendiagramme auf, welche Kriterien für die Bestimmung der Ähnlichkeit der einzelnen Elementtypen herangezogen werden.

Tabelle 4.1 zeigt exemplarisch die Vergleichskriterien für Klassen aus UML-Klassendiagrammen. Für jedes Elementpaar wird ein Wert zwischen 0 und 1 berechnet, wobei 0 für keine Ähnlichkeit und 1 für Übereinstimmung steht. Jedes der Vergleichskriterien wird gewichtet, die Summe der Gewichte muss 1 ergeben. Die entsprechende Angabe findet sich in Spalte w. Die mit t überschriebene Spalte gibt

Elementtyp	t	ComparedItem	Parameter	w
Class	0,4	CISimilarNames	-	0,4
		CIUnorderedCompartment	Operation	0,2
		CIUnorderedCompartment	Attribute	0,2
		CIUnorderedCompartment	Generalization	0,1
		CIMatchingParents	-	0,1

Tabelle 4.1: Vergleichskriterien für UML-Klassen nach [Weh04, S. 61]

den Grenzwert an, den die Ähnlichkeit eines Elementpaares überschreiten muss, damit die Elemente als ähnlich zueinander angesehen werden. Die Ähnlichkeit zweier Elemente wird als Summe der gewichteten Ähnlichkeiten, die sich durch die einzelnen Vergleichskriterien ergeben, definiert.

Für die Ähnlichkeit zweier Klassen sind demnach zu 40% die Ähnlichkeit der Klassennamen[1] sowie zu je 20% die Ähnlichkeiten ihrer Attribute und Operationen zuständig. Die verbleibenden 20% werden gemäß der Vererbungshierarchie und der betreffenden Packages ermittelt.

Die grundsätzliche Beobachtung, wonach zum einen der Name der Elemente und zum anderen die Eigenschaften der Subelemente für die Ähnlichkeit ausschlaggebend sind, lassen sich auf die anderen Elementtypen aus UML-Klassendiagrammen übertragen. Damit muss die Abbildung der Elemente auf Vektoren so erfolgen, dass sowohl Namensähnlichkeiten als auch Ähnlichkeiten der Subelemente erhalten bleiben.

4.2 Metrische Indizes

Software-Metriken stellen einen ersten Ansatz dar, um Elemente aus UML-Klassendiagrammen auf Zahlen abzubilden. Metriken sind Funktionen zur Ermittlung von Kennzahlen und werden oft im Kontext Qualitätsmanagement verwendet. Für UML-Klassen, Operationen und Attribute wurden viele Metriken vorgeschlagen. Eine Übersicht bietet [Lan99, S. 11f.]. Dabei fällt auf, dass die meisten der Metriken auf einfache Zähloperationen zurückzuführen sind.

Um dies zu verdeutlichen, wird in Abbildung 4.1 ein internes Datenmodell für Dokumente dargestellt, deren Differenz berechnet werden soll. Das Modell stellt nur eine von vielen Möglichkeiten dar, wie Dokumente modelliert werden können; das grundsätzliche Prinzip ist jedoch wegen seines generischen Ansatzes allgemeingültig. Dokumente werden als Graphen aufgefasst, die aus getypten *Nodes* und *Edges* bestehen. Alle Elemente eines Dokuments werden auf *Nodes* abgebildet, alle Beziehungen zwischen Elementen werden über *Edges* erfasst. Über den Typ herausgehende

[1] Die Ähnlichkeit von Namen wird in [Weh04] nach dem *longest common subsequence*-Verfahren (LCS) ermittelt.

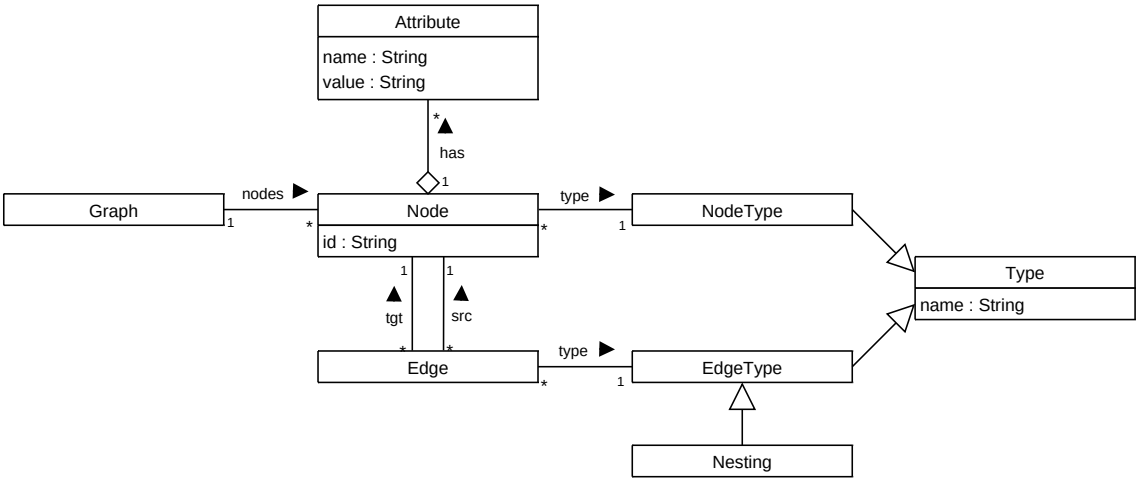

Abbildung 4.1: Internes Datenmodell für die Differenzberechnung nach [WK06]

Eigenschaften der *Nodes* werden über *Attributes* erfasst, die Schlüssel/Wert-Paare enthalten.

Im Anwendungsfall UML-Klassendiagramme werden demnach Klassen, Operationen, Attribute, etc. jeweils auf *Nodes* abgebildet, während die Tatsache, dass eine bestimmte Operation Teil einer bestimmten Klasse ist, über eine *Edge* zwischen den entsprechenden *Nodes* modelliert wird.

Ist eine derartige Abbildung gegeben, so lassen sich die Software-Metriken in mehrere Gruppen einteilen:

- Metriken, die sich durch Zählen der Nachbar-*Nodes* von einem bestimmten Typ berechnen lassen.

  Beispiel: Anzahl der Operationen für eine Klasse (in [Lan99] mit NOM – *number of methods* – abgekürzt[2]). Dazu müssen ausgehend von dem *Node*, der die Klasse repräsentiert, alle benachbarten *Nodes* gezählt werden, deren Typ angibt, dass es sich um Operationen handelt.

- Metriken, die sich durch Zählen der Nachbar-*Nodes* von einem bestimmten Typ und mit einem bestimmten Wert für ein Attribut berechnen lassen.

  Beispiel: Anzahl der Klassenvariablen für eine Klasse (in [Lan99] mit NCV – *number of class variables* – abgekürzt). Der Wert ergibt sich durch Zählen aller Nachbar-*Nodes*, deren Typ Attribute repräsentiert und die

[2]In der objektorientierten Begriffswelt werden die Begriffe *Operation* und *Methode* oft synonym verwendet (vgl. dazu [Kel03, S. 11]). Im Rahmen dieser Arbeit wird gemäß der aktuellen UML-Spezifikation (siehe [UML07, S. 105-109]) der Begriff Operation benutzt. Unter einer Methode wird die Implementierung einer Operation – also der Operationsrumpf – verstanden, der nicht Teil von UML-Klassendiagrammen ist. Metriken lassen sich jedoch sowohl auf Basis der Operationsdeklaration als auch auf Basis des Operationenrumpfes definieren. Daher hat sich für die englischen Bezeichnungen der Metriken z.B. nach [Lan99] der Begriff *method* durchgesetzt. Diese Bezeichnungen werden in der vorliegenden Arbeit beibehalten.

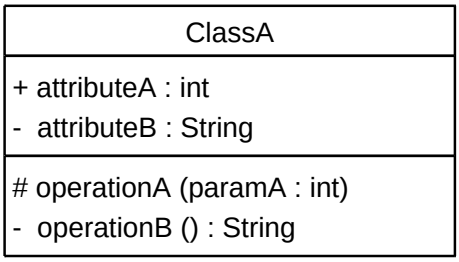

Abbildung 4.2: Beispiel UML-Klasse

zusätzlich über ein *Attribute* angeben, dass es sich nicht um Instanzvariablen handelt.

- Spezielle Metriken, die sich nicht verallgemeinern lassen. Dazu zählen bei UML-Klassendiagrammen alle Metriken, die auf dem Prinzip der Vererbung basieren.

  Beispiel: Anzahl geerbter Operationen für eine Klasse (in [Lan99] mit NMI – *number of methods inherited* – abgekürzt). Diese Metrik gibt an, wie viele Operationen eine Klasse von ihren Superklassen erbt, ohne sie neu zu definieren. Die Berechnung setzt voraus, dass die Vererbungshierarchie vorverarbeitet wurde, damit ausgehend von einer Klasse auf alle Operationen aller Superklassen zugegriffen werden kann.

Darüber hinaus werden oft Metriken vorgeschlagen, die nur berechnet werden können, wenn der entsprechende Quelltext vorliegt (z.B. Anzahl der Codezeilen für eine Operation). Diese sind für die Betrachtung von UML-Klassendiagrammen offensichtlich irrelevant.

Das Zählen benachbarter *Nodes* von einem bestimmten Typ bzw. mit weiteren speziellen Eigenschaften lässt sich auf alle Elementtypen aus UML-Klassendiagrammen übertragen und kann generell zur Berechnung von Metriken eingesetzt werden. Für Packages kann beispielsweise die Anzahl der Klassen oder Unter-Packages ermittelt werden und für Operationen lässt sich die Anzahl von Parametern von einem bestimmten Datentyp ermitteln. Der Ansatz lässt sich ebenfalls auf andere Dokumenttypen übertragen. In Simulink-Diagrammen kann zum Beispiel die Anzahl der *in-ports* für einen Block bestimmt werden, indem für den Block-*Node* alle benachbarten *Nodes* gezählt werden, die *in-ports* repräsentieren.

Da durch Software-Metriken mit Hilfe von Zahlenwerten die Eigenschaften von Elementen aus UML-Klassendiagrammen beschrieben werden, stellen sie eine erste Möglichkeit dar, um Dokumentelemente auf Vektoren abzubilden. Indizes, die aus Metriken hervorgehen, werden hier als metrische Indizes bezeichnet. An metrische Indizes wird nicht die Anforderung gestellt, dass sie Qualitätsmerkmale repräsentieren sollen. Sie werden jedoch auf die gleiche Weise wie klassische Software-Qualitätsmetriken berechnet und bilden Elementeigenschaften ab, die als Ähnlichkeitskriterien für Elemente verwendet werden können.

Abbildung 4.2 zeigt ein Beispiel einer Klasse aus einem UML-Klassendiagramm. Tabelle 4.2 zeigt dazu einen Vektor bestehend aus 26 metrischen Indizes. Der erste Teil dieser Indizes lässt sich durch einfache Zähloperationen realisieren, für den zweiten Teil muss eine Vorverarbeitung der Klassenhierarchie stattfinden.

Metrische Indizes eignen sich vor allem, um Eigenschaften von Subelementen abzubilden. Es lassen sich sowohl ihre Anzahlen gruppiert nach Typen als auch die speziellen Eigenschaften, die im internen Datenmodell über *Attributes* modelliert werden, erfassen.

4.3 Lexikalische Indizes

Mit metrischen Indizes können keine Namensähnlichkeiten erfasst werden[3]. Da für die Ähnlichkeitsbestimmung von Dokumentelementen die Ähnlichkeit von Namen aber von großer Bedeutung ist, muss eine Möglichkeit gefunden werden, die Eigenschaften von Elementnamen ebenfalls in den Vektoren zu erfassen. Diese Erfassung muss so erfolgen, dass nicht nur exakte Übereinstimmungen sondern auch Ähnlichkeiten von Namen erkannt werden können.

Das Problem des Auffindens ähnlicher Zeichenfolgen existiert auch in anderen Anwendungsgebieten. Rechtschreibprüfungen, die bei unbekannten Zeichenfolgen dem Benutzer eine Auswahl ähnlicher Wörter anzeigen, arbeiten beispielsweise meist auf Basis der in [Fre60] vorgestellten Datenstruktur Trie. Dazu wird ein Baum konstruiert, der in jedem Knoten einen Buchstaben verwaltet, so dass jeder Pfad eine Buchstabenfolge ergibt. Alle der Rechtschreibprüfung bekannten Wörter werden als an der Wurzel beginnende Pfade in diesem Baum abgelegt. Um Wörter mit demselben Präfix zu finden, muss somit nur der letzte Teil der Zeichenkette abgetrennt werden und geprüft werden, welche anderen Wörter mit diesem Präfix existieren. Sollen auch effizient Wörter mit demselben Suffix gefunden werden, so kann ein zusätzlicher Trie aufgebaut werden, der die Wörter in umgekehrter Reihenfolge enthält.

Dieser Ansatz kann für die Abbildung von Dokumentelementen auf Vektoren nicht verwendet werden, da die Anordnung lediglich alphabetisch erfolgt. Zwei Dokumentelemente, deren Namen sich nur in ihren ersten Zeichen unterscheiden, würden somit keine Ähnlichkeit aufweisen.

Die einzige Möglichkeit, die Eigenschaften von Elementnamen auf Vektoren abzubilden, besteht darin, die Namen selbst als Indizes zu verwenden. Dazu muss zunächst die Menge aller Elementnamen bestimmt werden, die in den zu vergleichenden Dokumenten vorkommen. Dann werden sämtliche Namen als lexikalische Indizes verwendet. Für jedes Element wird eine 1 als Wert an dem Index in den Vektor eingetragen, der den Namen des Elements angibt. Alle anderen Werte im lexikalischen Teil des Vektors sind 0.

Mit diesem Verfahren lassen sich bisher nur exakte Übereinstimmungen von Namen erkennen. Der lexikalische Teil der Vektoren stimmt genau dann überein, wenn

[3]Die in Abschnitt 4.2 definierte Metrik LON, die die Länge von Namen in Buchstaben angibt, ist offensichtlich nicht hinreichend.

Index	Bedeutung	Wert
LON *(length of name)*	Länge des Klassennamens	6
NAG *(# getters)*	Anz. get-Operationen	0
NAM *(# abstract methods)*	Anz. abstrakter Operationen	0
NAPAC *(# package-visible attributes)*	Anz. Attribute, keine spezifizierte Sichtbark.	0
NAPRI *(# private attributes)*	Anz. Attribute, Sichtbark. private	1
NAPRO *(# protected attributes)*	Anz. Attribute, Sichtbark. protected	0
NAPUB *(# public attributes)*	Anz. Attribute, Sichtbark. public	1
NAS *(# setters)*	Anz. set-Operationen	0
NCV *(# class variables)*	Anz. Klassenvariablen	0
NIV *(# instance variables)*	Anz. Instanzvariablen	2
NMPAC *(# package-visible methods)*	Anz. Operationen, keine spezifizierte Sichtbark.	0
NMPRI *(# private methods)*	Anz. Operationen, Sichtbark. private	1
NMPRO *(# protected methods)*	Anz. Operationen, Sichtbark. protected	1
NMPUB *(# public methods)*	Anz. Operationen, Sichtbark. public	0
NOA *(# attributes)*	Anz. Attribute	2
NOC *(# children in class hierarchy)*	Anz. direkter Subklassen	0
NOCM *(# constructors)*	Anz. Konstruktoren	0
NOM *(# methods)*	Anz. Operationen	2
SUP *(# superclasses)*	Anz. direkter Superklassen	0
NIA *(# attributes inherited)*	Anz. geerbter Attribute	0
NMA *(# methods added)*	Anz. hinzugefügter Operationen gemäß Vererbungshierarchie	2
NMI *(# methods inherited)*	Anz. geerbter Operationen	0
NMO *(# methods overridden)*	Anz. überschriebener Operationen	0
NMOI *(# methods overloaded, inherited methods are considered)*	Anz. überladener Operationen unter Berücksichtigung geerbter Operationen	0
NMOO *(# methods overloaded, without considering inheritance)*	Anz. überladener Operationen ohne Berücksichtigung geerbter Operationen	0
WNOC *(whole # descendants in class hierarchy)*	Anz. aller Subklassen	0

Tabelle 4.2: Vektor für Beispiel-Klasse mit metrischen Indizes

Indexart	Pfad	*Attribute*	Teilstrings bestimmen
Klassenname	this	name	nein
Teil eines Klassennamens	this	name	ja
Operationsname	this → operation	name	nein
Teil eines Operationsnamens	this → operation	name	ja

Tabelle 4.3: Definition lexikalischer Indizes

an der gleichen Stelle im Vektor eine 1 steht. Stimmen die Namen nicht exakt überein, so ergibt sich die Distanz zwischen den lexikalischen Vektoren $\vec{x}$ und $\vec{y}$ als

$$d(\vec{x}, \vec{y}) = \sqrt{2}, \tag{4.1}$$

da sich die Vektoren nur an 2 Stellen unterscheiden, an denen jeweils in einem Vektor eine 1 und im anderen Vektor eine 0 steht.

Um auch Namensähnlichkeiten über den euklidischen Abstand erfassen zu können, müssen zusätzlich zu den kompletten Namen Namensteile als lexikalische Indizes verwendet werden. Dazu sind mehrere Ansätze denkbar:

- Für die Ermittlung von Namensteilen können gängige Konventionen in Betracht gezogen werden. In UML-Klassendiagrammen ist es beispielsweise üblich, bei Namen, die aus mehreren Teilen bestehen, die einzelnen Teile jeweils mit einem Großbuchstaben beginnen zu lassen (z.B.: `getAttributeValue()`). Jeder so ermittelte Namensteil kann als lexikalischer Index eingesetzt werden.

- Unabhängig von Konventionen können über Algorithmen wie LCS (vgl. [Gus97]) gemeinsame Teile aller in den zu vergleichenden Dokumenten vorhandenen Elementnamen bestimmt werden. Wird LCS paarweise angewandt, so ergibt sich jedoch eine quadratische Laufzeit.

Zusätzlich zur Abbildung der Elementnamen können lexikalische Indizes verwendet werden, um die Namen von Subelementen abzubilden. Beispielsweise können in dem Vektor zu einer Klasse aus einem UML-Klassendiagramm außer dem Namen der Klasse auch die Namen der Operationen dieser Klasse abgebildet werden.

Lexikalische Indizes können grundsätzlich für alle Dokumentelemente verwendet werden, die über Eigenschaften verfügen, die als Zeichenkette repräsentiert werden. Wenn ein internes Datenmodell wie in Abbildung 4.1 verwendet wird, trifft das auf alle Elemente zu, die über mindestens ein *Attribute* verfügen. Um lexikalische Indizes zu definieren, genügt die Angabe dieses *Attribute* sowie eines Pfades von dem *Node*, dessen Vektor berechnet wird, zu den *Nodes*, die über das *Attribute* verfügen. Tabelle 4.3 zeigt die Definition von vier lexikalischen Indexarten für den UML-Elementtyp Klasse.

Index	Indexart	Wert
ClassA	Klassenname	1
ClassB	Klassenname	0
⋮	⋮	⋮
ClassN	Klassenname	0
Class	Teil des Klassennamens	1
A	Teil des Klassennamens	1
B	Teil des Klassennamens	0
⋮	⋮	⋮
N	Teil des Klassennamens	0
operationA	Operationsname	1
operationB	Operationsname	1
operationC	Operationsname	0
⋮	⋮	⋮
operationN	Operationsname	0
operation	Teil des Operationsnamens	1
A	Teil des Operationsnamens	1
B	Teil des Operationsnamens	1
C	Teil des Operationsnamens	0
⋮	⋮	⋮
N	Teil des Operationsnamens	0

Tabelle 4.4: Vektor für Beispiel-Klasse mit lexikalischen Indizes

Der Pfad gibt jeweils an, wie zu den Knoten, deren *Attribute* für lexikalische Indizes verwendet werden soll, navigiert werden kann. Um die Operationsnamen im Vektor einer Klasse zu berücksichtigen, muss beispielsweise zu den Knoten navigiert werden, die die Operationen der Klasse repräsentieren. Wenn keine Navigation stattfinden muss, weil der Ausgangs-*Node* über das gesuchte *Attribute* verfügt, ist an der entsprechenden Stelle *this* eingetragen. Die mit *Attribute* bezeichnete Spalte gibt an, wie das Attribut heißt, dessen Werte betrachtet werden sollen. In der letzten Spalte wird angegeben, ob komplette Attributwerte oder Teile der Zeichenketten als Indizes verwendet werden sollen.

Wenn Attributwerte von Subelementen in Betracht gezogen werden, erweitert sich zusätzlich der Wertebereich der lexikalischen Indizes. Da derselbe Attributwert bei mehreren Subelementen vorkommen kann, können sich Werte ergeben, die größer als 1 sind. Im Beispiel aus Tabelle 4.3 tritt dies ein, wenn ein Operationsname in einer Klasse überladen ist. In dem Fall würden über den angegebenen Pfad mehrere *Nodes* mit identischem Operationsnamen gefunden.

Tabelle 4.4 zeigt den lexikalischen Teil des Vektors, der sich für das Beispiel aus Abbildung 4.2 mit den in Tabelle 4.3 definierten Indexarten ergibt. Im Gegensatz zu

metrischen Indizes beschränkt sich die Ermittlung der lexikalischen Werte nicht lokal auf das Element, für das der Vektor erzeugt werden soll. Vielmehr muss die Menge aller Elementnamen in allen zu vergleichenden Dokumenten bestimmt werden, um die Vektorindizes zu definieren. Aus diesem Grund ist das Beispiel in Tabelle 4.4 unvollständig und geht davon aus, dass weitere Elemente in den zu vergleichenden Dokumenten nach demselben Prinzip benannt sind wie in der Beispiel-Klasse.

Neben Elementnamen können beschreibende Eigenschaften wie zum Beispiel die Sichtbarkeiten von Attributen oder Operationen in UML-Klassendiagrammen mit lexikalischen Indizes erfasst werden. Anstatt die Sichtbarkeiten *private*, *protected* und *public* bei der Definition von metrischen Indizes fest vorzugeben, kann eine lexikalische Indexart so definiert werden, dass ausgehend von einer Operation das *Attribute* betrachtet wird, das die Sichtbarkeit angibt. Im Rahmen der Bestimmung der Menge möglicher Werte in den zu vergleichenden Dokumenten werden dann die konkret vorhandenen Sichtbarkeiten ermittelt.

Mit metrischen und lexikalischen Indizes existieren somit zwei Mechanismen, mit denen für ein gegebenes Element sowohl sein Name als auch seine Eigenschaften und die seiner Subelemente abgebildet werden können. Durch eine Konkatenation der metrischen und lexikalischen Teile erhält man eine erste Version der Vektoren, die in S^3V Bäumen verwaltet werden können um ähnliche Dokumentelemente benachbart anzuordnen. Als Ähnlichkeitsmaß wird der euklidische Abstand zwischen den Vektoren verwendet.

4.4 Normierung und Skalierung

Um als sinnvolle Ausgangsbasis für die Ähnlichkeitsbestimmung eingesetzt werden zu können, müssen die Vektoren normiert und skaliert werden. Die Normierung schafft zunächst eine einheitliche Ausgangsposition, bevor die Skalierung eine Konfiguration der Vektoren ermöglicht.

Normierung

Wie das Beispiel aus Tabelle 4.2 zeigt, weisen unterschiedliche metrische Indizes unterschiedliche Wertebereiche auf. Damit wirken sich verschiedenartige Änderungen unterschiedlich stark auf die Ähnlichkeit zwischen den Vektoren nach euklidischem Abstand aus. Beispielsweise führt das Hinzufügen einer Superklasse zu einer Klasse in einem UML-Klassendiagramm zu einem neuen Summanden von 1 unter der Wurzel bei der Berechnung des euklidischen Abstands, während das Hinzufügen von 3 Attributen einen Summanden von mindestens 3 erzeugt[4].

[4]Der Summand wird größer, falls das Hinzufügen eines Attributs mehrfach erfasst wird. Das geschieht, wenn neben einem metrischen Index, der die Anzahl der Attribute repräsentiert, zusätzlich Indizes vorhanden sind, die die Anzahl von Attributen mit bestimmten Sichtbarkeiten bzw. die Anzahl von Klassen- und Objektvariablen erfassen.

Da sich diese Unterschiede zufällig aus der Definition der metrischen Indizes ergeben und nicht aufgrund einer Aussage über die Bedeutung unterschiedlicher Änderungen für die Ähnlichkeit von Elementen entstanden sind, müssen die Wertebereiche normiert werden. Dabei bietet es sich an, den Wertebereich von 0 bis 1 als Norm zu wählen. Für jeden Wert v_i in jedem Vektor $\vec{v}$ ergibt sich der neue Wert v_{i_norm} dann als

$$v_{i\_norm} = \frac{v_i - \min\left(\{x_i | \vec{x} \in V\}\right)}{\max\left(\{x_i | \vec{x} \in V\}\right) - \min\left(\{x_i | \vec{x} \in V\}\right)}. \tag{4.2}$$

V gibt die Menge aller Vektoren an. Mit dieser Formel wird der maximale Wert für einen bestimmten Index auf 1 abgebildet, während der minimale Wert auf 0 abgebildet wird. Alle Zwischenwerte werden entsprechend skaliert. Wenn für einen Index der maximale und der minimale Wert übereinstimmen, kann die Formel nicht angewendet werden, da sonst eine Division durch 0 stattfindet. Eine Normierung ist aber in dem Fall sowieso nicht erforderlich, da dieser Index keine Auswirkung auf Ähnlichkeitsaussagen haben wird.

Eine Normierung muss ebenfalls für lexikalische Indizes durchgeführt werden, wenn lexikalische Indexarten existieren, die Eigenschaften der Subelemente abbilden. Werden nur die Namen der Elemente selbst abgebildet, so sind die Wertebereiche bereits normiert, da jedes Element nur einen Namen hat.

Für lexikalische Indizes existiert außerdem das Problem, dass die Anzahl der Indizes sich erst zur Laufzeit ergibt und proportional von der Gesamtzahl an Elementen in den zu vergleichenden Dokumenten abhängt. Je größer die Dokumente sind, desto stärker fallen die lexikalischen Indizes gegenüber den metrischen Indizes ins Gewicht. Zur Lösung dieser Situation sind zwei Ansätze denkbar:

- Um sicherzustellen, dass der maximale euklidische Abstand zwischen dem lexikalischen und dem metrischen Teil der Vektoren unabhängig von der Anzahl der Indizes immer identisch ist, können alle Werte v_i im lexikalischen Teil des Vektors weiter angepasst werden:

  $$v_{i\_norm} = \frac{v_i \cdot \sqrt{m}}{\sqrt{l}} \tag{4.3}$$

  Dabei bezeichnet m die Länge des metrischen Teilvektors und l die Länge des lexikalischen Teilvektors. Durch die Normierung aller Wertebereiche auf 0 bis 1 beträgt die maximale Distanz zweier Vektoren mit n Elementen $\sqrt{n \cdot 1}$ und kann somit durch Division mit $\sqrt{n}$ auf 1 normiert werden.

- Nach dem zweiten Ansatz wird keine weitere Normierung vorgenommen. Wie aus Tabelle 4.4 ersichtlich, zeichnet sich der lexikalische Teil der Vektoren dadurch aus, dass er sehr viele Nullwerte enthält und somit die Distanz zwischen zwei lexikalischen Teilvektoren relativ klein ist. Im einfachsten Fall mit nur einer lexikalischen Indexart, die die Elementnamen abbildet, beträgt die maximale Distanz $\sqrt{2}$ – unabhängig von der Länge der Vektoren.

Ob eine zusätzliche Normierung hinsichtlich der Länge der lexikalischen Teilvektoren vorgenommen wird, liegt somit in erster Linie daran, inwieweit Teile von Zeichenfolgen und Eigenschaften der Subelemente als lexikalische Indizes berücksichtigt werden und wie groß somit die Distanz zwischen zwei lexikalischen Teilvektoren werden kann.

Skalierung

Mit der Normierung der Vektoren wird eine einheitliche Ausgangslage geschaffen, auf deren Basis die einzelnen Werte in den Vektoren skaliert werden können, um Gewichtungen für die Ähnlichkeitsbestimmung vorzunehmen. Der skalierte Wert v_{i_scale} ergibt sich für jeden Wert v_i durch Multiplikation mit einem Skalierungsfaktor f_i:

$$v_{i\_scale} = v_i \cdot f_i \tag{4.4}$$

Die Skalierungsfaktoren f_i werden für jeden metrischen Index sowie jede lexikalische Indexart festgelegt. Unterschiedliche lexikalische Indizes derselben Indexart müssen gleich gewichtet werden, um eine Gleichbehandlung aller Elemente zu gewährleisten.

4.5 Ähnlichkeitskriterien und Elementaränderungen

Ziel der Skalierung ist es, die Kriterien für die Ähnlichkeit von Elementen aus dem paarweisen Vergleich möglichst genau nachzubilden, so dass Elemente, die aufgrund einer hohen Ähnlichkeit ihrer Vektoren benachbart angeordnet werden, auch nach dem paarweisen Vergleich eine hohe Ähnlichkeit aufweisen. Anhand von drei Beispielen zu UML-Klassen wird in diesem Abschnitt untersucht, wie sich Änderungen unterschiedlichen Ausmaßes auf den euklidischen Abstand der Vektoren auswirken. Dazu werden zunächst die in den Abschnitten 4.2 und 4.3 vorgestellten Beispiel-Indizes skaliert.

Beispiel-Skalierung

Für die vor der Skalierung notwendige Normierung müssen die zu vergleichenden Dokumente vollständig bekannt sein, da die Wertebereiche aufgrund der Minima und Maxima der einzelnen Indizes angepasst werden. Diese Tatsache zeigt einen weiteren Unterschied zwischen dem paarweisen Elementvergleich und dem Einsatz mehrdimensionaler Strukturen für das Auffinden ähnlicher Dokumentelemente. Die Vektoren passen sich dynamisch an die zu vergleichenden Dokumente an. Es ist allerdings auch möglich, diese Anpassung zu verhindern, um standardisierte Ergebnisse zu erreichen. Dazu müssen feste Minima und Maxima vorgegeben werden.

Zwecks Überschaubarkeit der Beispiele werden hier nicht die gesamten Dokumente gezeigt, sondern folgende Annahmen getroffen:

metrischer Index	Skalierungsfaktor	metrischer Index	Skalierungsfaktor
LON	1	NMPUB	1
NAG	0,5	NOA	0
NAM	0,5	NOC	1
NAPAC	1	NOCM	0,5
NMPRI	1	NOM	0
NAPRO	1	SUP	1
NAPUB	1	NIA	0,5
NAS	0,5	NMA	0,5
NCV	0,5	NMI	0,5
NIV	0,5	NMO	0,5
NMPAC	1	NMOI	0,5
NMPRI	1	NMOO	0,5
NMPRO	1	WNOC	0,5

Tabelle 4.5: Beispiel-Skalierung der metrischen Indizes

- Das Minimum beträgt für jeden Vektorindex 0.
- Die maximale Länge von Klassennamen beträgt 10.
- Die maximale Anzahl von Operationen in einer Klasse sowie von Operationen mit einer bestimmten Eigenschaft (Sichtbarkeit, Konstruktoren, etc.) beträgt 2.
- Die maximale Anzahl von Attributen in einer Klasse sowie von Attributen mit einer bestimmten Eigenschaft (Sichtbarkeit, Gültigkeitsbereich, etc.) beträgt 2.
- Teile eines Klassennamens kommen höchstens einmal pro Namen vor.
- Operationsnamen und Teile von Operationsnamen treten maximal zweimal pro Klasse auf.

Damit kann die Normierung stattfinden, indem für alle metrischen Indizes und alle lexikalischen Indexarten, die Operationsnamen behandeln, die Werte halbiert werden und für die Metrik LON (*length of name*) die Werte durch 10 dividiert werden. Mit dem Ergebnis der Normierung kann anschließend die Skalierung stattfinden, die die Indizes im Hinblick auf ihre Bedeutung für die Ähnlichkeit von Elementen gewichtet.

Die Tabellen 4.5 und 4.6 zeigen eine Beispiel-Skalierung. Die mehrfache Berücksichtigung von Änderungen bezüglich der Operationen und Attribute einer Klasse wird minimiert, indem die Gesamtanzahlen von Operationen und Attributen (Indizes NOA und NOM) gar nicht in die Ähnlichkeitsbestimmung eingehen. Stattdessen werden sie gruppiert nach ihren Sichtbarkeiten erfasst (Indizes NAPAC,

lexikalische Indexart	Skalierungsfaktor
Klassenname	1
Teil eines Klassennamens	0,5
Operationsname	0,5
Teil eines Operationsnamens	0,25

Tabelle 4.6: Beispiel-Skalierung der lexikalischen Indizes

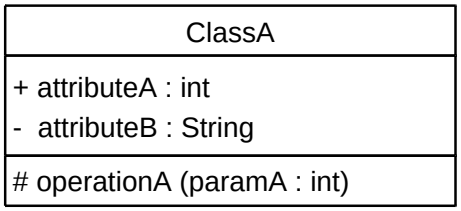

Abbildung 4.3: Beispiel UML-Klasse mit Elementaränderung

NAPRI, NAPRO, NAPUB, bzw. NMPAC, NMPRI, NMPRO und NMPUB). Alle weiteren Metriken, die Eigenschaften der Operationen und Attribute angeben, werden mit 0,5 skaliert, so dass sie weniger Einfluss auf die Ähnlichkeitsaussagen haben.

Bei den lexikalischen Indizes werden die Indizes, die Klassennamen entsprechen, stärker gewichtet als die Indizes, die Operationsnamen repräsentieren. Namensteile werden jeweils schwächer gewichtet als die gesamten Namen. Mit dieser Skalierung ist es nun möglich, konkrete Beispiele zu betrachten.

Beispiel einer Elementaränderung

Als erstes wird untersucht, wie sich das Löschen einer Operation auf den euklidischen Abstand zwischen den zugehörigen Vektoren auswirkt. Abbildung 4.3 zeigt dazu die Klasse, die aus dem Beispiel aus Abbildung 4.2 hervorgeht, wenn die Operation *operationB* gelöscht wird.

Die skalierten Vektoren der beiden Klassen unterscheiden sich in den Werten an 5 Indizes:

- Metrischer Index NMPRI. Die Anzahl an Operationen mit Sichtbarkeit *private* hat sich von 2 auf 1 verringert. Der entsprechende Indexwert fällt aufgrund der Normierung von 1 auf 0,5, so dass die Distanz an diesem Index 0,5 beträgt.

- Metrischer Index NMA. Die Anzahl an Operationen, die im Vergleich zu den Superklassen hinzugefügt wurden, verringert sich ebenfalls von 2 auf 1. Durch die Skalierung mit Skalierungsfaktor 0,5 beträgt die Distanz an diesem Index jedoch nur 0,25.

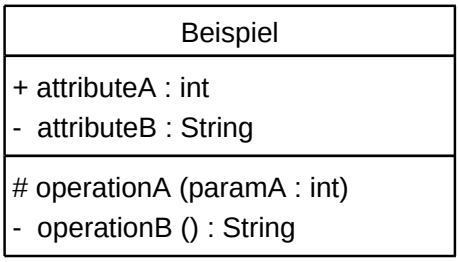

Abbildung 4.4: Beispiel UML-Klasse mit Namensänderung

- Lexikalischer Index für den Operationsnamen *operationB*. Da dieser Operationsname nur in einer der Klassen auftritt, ergibt sich nach Skalierung und Normierung eine Distanz von 0,25.
- Lexikalischer Index für den Teil *operation* eines Operationsnamens. Nach Normierung und Skalierung beträgt die Distanz an diesem Index 0,125.
- Lexikalischer Index für den Teil *B* eines Operationsnamens. Nach Normierung und Skalierung beträgt die Distanz an diesem Index 0,125.

Wie dieses Beispiel zeigt, wirkt sich das Löschen der Operation nur an einem Index entscheidend auf den euklidischen Abstand aus[5]. Alle anderen Änderungen geben lediglich Zusatzinformationen. Die Berechnung der Länge des Distanzvektors zwischen den beiden Klassen ergibt einen Wert von 0,64.

Beispiel einer Namensänderung

Abbildung 4.4 zeigt das Ergebnis einer Änderung des Klassennamens von *ClassA* in *Beispiel*. Diese Änderung wirkt sich vor allem auf den lexikalischen Teil der Vektoren aus. An den Indizes, die die beiden betroffenen Klassennamen repräsentieren, entsteht jeweils eine Distanz von 1 und an den Indizes, die die beiden betroffenen Namensteile *Class* und *A* repräsentieren, beträgt der Abstand durch die Skalierung jeweils 0,5. Zusätzlich entsteht an dem metrischen Index, der die Länge des Klassennamens beschreibt, eine Distanz von 0,2 nach Normierung und Skalierung.

Der berechnete euklidische Abstand beträgt mit 1,59 deutlich mehr als im ersten Beispiel. Änderungen des Klassennamens haben somit einen größeren Einfluss auf die Ähnlichkeit zweier Klassen als das Löschen einer Operation. Dieser Effekt erfüllt die in Abschnitt 4.1 gestellten Anforderungen.

Beispiel nicht-korrespondierender Elemente

Abbildung 4.5 zeigt abschließend ein Beispiel einer Klasse, die nur wenige Gemeinsamkeiten mit der Ausgangsklasse hat. Die Länge des Distanzvektors zwischen den

[5]Durch das Quadrieren der Summanden beim Berechnen des euklidischen Abstands werden die Unterschiede der Auswirkungen noch deutlicher.

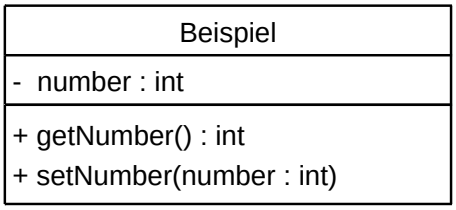

Abbildung 4.5: Beispiel einer nicht-korrespondierenden UML-Klasse

Klassen beträgt 2,17 und übertrifft somit die Werte aus den vorhergehenden Beispielen.

Die hier aufgeführten Beispiele zeigen exemplarisch, dass die Abbildung der Dokumentelemente auf Vektoren die Eigenschaften erhält, die für die Ähnlichkeitsbestimmung von Bedeutung sind. Eine detaillierte Evaluierung findet in Kapitel 6 statt.

4.6 Verallgemeinerung

Das Konzept metrischer und lexikalischer Indizes lässt sich auf alle Dokumenttypen verallgemeinern. Wenn eine Abbildung der Dokumente auf eine interne Datenstruktur wie in Abbildung 4.1 gezeigt vorliegt, können folgende Indizes erzeugt werden:

Metrische Indizes. Metrische Indizes können ausgehend von einem *Node* für alle benachbarten *Nodes* gruppiert nach Typen sowie nach weiteren, über die *Attributes* abgebildeten, Eigenschaften erzeugt werden. Zusätzlich lassen sich dokumenttypspezifisch weitere Metriken definieren (z.B. basierend auf der Vererbungshierarchie in UML-Klassendiagrammen).

Lexikalische Indizes. Lexikalische Indizes können basierend auf jedem *Attribute* eines *Nodes* und seiner benachbarten *Nodes* erzeugt werden.

Damit liegt eine sehr umfangreiche Menge möglicher Vektorindizes vor, die auf die Indizes reduziert werden muss, die tatsächlich eine Aussage über die Ähnlichkeit von Elementen erlauben. Diese Reduktion kann nicht automatisiert werden. Es können jedoch die in Abschnitt 2.4 beschriebenen Techniken des Data Mining eingesetzt werden, um Hinweise auf besonders aussagekräftige Indizes zu erhalten.

Darüber hinaus besteht durch die Skalierung die Möglichkeit, Indizes zu gewichten und so ihre Bedeutung für die Ähnlichkeit von Dokumentelementen zu beeinflussen.

5 Integration in die Differenzberechnung

Für die Integration der bisherigen Ergebnisse in die Berechnung von Dokumentdifferenzen gibt es mehrere Möglichkeiten, die in diesem Kapitel betrachtet werden. Dazu wird zunächst in Abschnitt 5.1 detailliert der paarweise Vergleich von Dokumentelementen vorgestellt, dessen Optimierung Ziel dieser Arbeit ist. Anschließend wird untersucht, wie der S^3V Baum genutzt werden kann, um zu einem gegebenen Dokumentelement ähnliche Elemente in anderen Dokumenten zu finden. Die Abschnitte 5.3 und 5.4 stellen schließlich mit der Top-Down und der Bottom-Up Variante die beiden wichtigsten Alternativen für die Integration vor.

5.1 Paarweiser Elementvergleich

Der paarweise Vergleich von Dokumentelementen kann nicht in beliebiger Reihenfolge stattfinden. Vielmehr geben Abhängigkeiten zwischen den Elementen die Vergleichsreihenfolge vor. Abhängigkeiten existieren vor allem dann, wenn zwischen zwei Elementen eine Teil-von Beziehung besteht. In diesem Fall muss die Ähnlichkeit der untergeordneten Elemente geklärt sein, bevor Ähnlichkeitsaussagen bezüglich des übergeordneten Elements getroffen werden können. So muss beispielsweise für den Vergleich zweier Klassen in UML-Klassendiagrammen bekannt sein, ob sich in beiden Klassen korrespondierende Operationen befinden.

Da Teil-von Beziehungen grundsätzlich gerichtet sind und keine Zyklen durch Teil-von Beziehungen entstehen können, impliziert diese Argumentation eine Reihenfolge der Vergleiche, die mit den kleinsten Teilen beginnt und mit solchen Elementen endet, die selbst nicht Teil eines anderen Dokumentelements sind. Wann Elemente verglichen werden, die nicht Teil einer Teil-von Beziehung sind, spielt keine Rolle.

Zusätzlich besteht das Problem, dass für untergeordnete Elemente oft anhand des paarweisen Vergleichs keine eindeutige Ähnlichkeitsaussage getroffen werden kann. Untergeordnete Elemente zeichnen sich meist dadurch aus, dass sie in großer Zahl in den zu vergleichenden Dokumenten vorliegen und nur wenige charakteristische Eigenschaften haben. Sie lassen sich oft erst dann korrespondierenden Elementen zuordnen, wenn bekannt ist, dass eine Korrespondenz zwischen den übergeordneten Elementen besteht.

Abbildung 5.1 zeigt schematisch, wie dieser Sachverhalt in [Weh04] gelöst wird. Das Beispiel basiert auf UML-Klassendiagrammen und der darin enthaltenen Kette von Teil-von Beziehungen. Parameter sind Teil von Operationen, die wiederum Teil

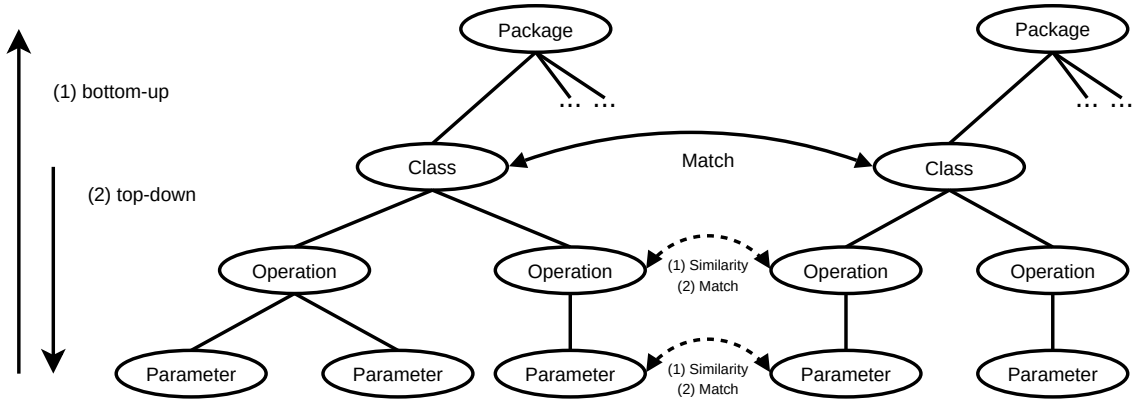

Abbildung 5.1: Schematischer Ablauf des paarweisen Vergleichs nach [Weh04, S. 42]

von Klassen und Packages sind[1].

In der ersten, mit *bottom-up* bezeichneten, Phase werden beginnend mit den Parametern alle Elemente gruppiert nach Typen paarweise verglichen und genau dann einander zugeordnet, wenn diese Zuordnung eindeutig ist und der geforderte Grenzwert für die Ähnlichkeit nicht auch noch mit anderen Elementen überschritten wird. Sobald eine Korrespondenz entdeckt wird (in der Abbildung als *Match* bezeichnet), wechselt der Algorithmus in die *top-down* Phase. In dieser Phase wird die neu gefundene Korrespondenz an die zugehörigen untergeordneten Elemente propagiert. Mit der Zusatzinformation über korrespondierende übergeordnete Elemente wird ein Element in der *top-down* Phase einem anderen Element zugeordnet, wenn die Ähnlichkeit den geforderten Grenzwert überschreitet. Ist dies mehr als einmal der Fall, so wird die Korrespondenz mit dem höchsten Ähnlichkeitswert gebildet. Nach Abschluss der *top-down* Phase wird die *bottom-up* Phase an der unterbrochenen Stelle fortgesetzt.

Der nach Elementtypen gruppierte, paarweise Vergleich aller Elemente wirkt sich vor allem bei untergeordneten Elementen, die selbst weitere Teil-Elemente besitzen, negativ auf die Gesamtlaufzeit aus, da von ihnen zum einen sehr viele in einem Dokument existieren und zum anderen jede neu gefundene Korrespondenz an alle untergeordneten Elemente propagiert werden muss. Ein Beispiel für solche kritischen Elemente stellen Operationen in UML-Klassendiagrammen dar.

Die Reihenfolge der Vergleiche muss bei der Integration von S^{3}V Bäumen in die Differenzberechnung beibehalten werden. Die grundsätzliche Idee der Integration besteht somit darin, die Menge der zu vergleichenden Elemente beim paarweisen Vergleich deutlich zu verringern, indem nur die aufgrund des S^{3}V Baumes ähnlichen Elemente tatsächlich verglichen werden und für alle anderen Element-Paare eine Ähnlichkeit von 0 angenommen wird.

[1] Weitere Teil-von Beziehungen in UML-Klassendiagrammen sind aus Gründen der Übersichtlichkeit nicht dargestellt.

5.2 Identifikation ähnlicher Elemente

In Kapitel 3 wurden mit der Nachbarsuche und der Bereichsanfrage zwei Mechanismen vorgestellt, mit denen zu einem gegebenen Element ähnliche Elemente im S^3V Baum gefunden werden können. Als Ähnlichkeitsmaß dient der euklidische Abstand zwischen den entsprechenden Vektoren. Die Bereichsanfrage gibt einen Grenzwert für die Ähnlichkeit vor, während für die Nachbarsuche eine bestimmte Anzahl von Nachbarn spezifiziert wird. Damit findet die Nachbarsuche immer die gleiche Anzahl an Elementen, während die Anzahl bei der Bereichsanfrage variiert[2].

Für die Verwendung der Bereichsanfrage bei der Identifikation ähnlicher Elemente im Zuge der Bestimmung von Dokumentdifferenzen sprechen zwei Argumente:

- Bei der Nachbarsuche wird keine Angabe darüber gemacht, wie groß die Ähnlichkeit der gefundenen Elemente zu dem gesuchten Element ist. Gibt es zu einem Element keine ähnlichen Elemente in einem anderen Dokument, so gibt die Nachbarsuche trotzdem die spezifizierte Anzahl an Nachbarn zurück. Andererseits werden nicht alle ähnlichen Elemente zurückgeliefert, falls die vorgegebene Anzahl an Nachbarn kleiner als die Zahl der tatsächlich vorhandenen ähnlichen Elemente ist. Das Hauptproblem liegt somit in einer sinnvollen Spezifikation der Anzahl an Nachbarn.

- Wie in Kapitel 3 beschrieben, ist die Bereichsanfrage effizienter als die Nachbarsuche, da keine Sortierung der Elemente vorgenommen werden muss und an jedem inneren Knoten des S^3V Baumes eindeutig entschieden werden kann, welche der darunter liegenden Teilbäume weiter zu verfolgen sind.

Im Folgenden werden 3 Möglichkeiten vorgestellt, die für die Konfiguration und die Optimierung der Bereichsanfrage für die Identifikation ähnlicher Dokumentelemente im Rahmen der Differenzberechnung herangezogen werden können.

Skalierung der Grenzwerte

Bei der Bereichsanfrage stellt sich die Frage, mit welchem Parameter für die Größe des Suchbereichs sie aufgerufen wird. Bei einem kreisförmigen Suchbereich um das Element, zu dem ähnliche Elemente gefunden werden sollen, muss somit der Radius des Bereichs bzw. der maximale euklidische Abstand, der den Grenzwert der Ähnlichkeit definiert, angegeben werden.

Wie in Kapitel 4 gezeigt, variieren die im S^3V Baum verwalteten Vektoren sowohl in ihrer Länge als auch – durch die Skalierung – in den Wertebereichen ihrer Indizes. Für die Beantwortung der Frage, wie dementsprechend die Grenzwerte für die Bereichsanfrage skaliert werden müssen, um ihre Vergleichbarkeit zu gewährleisten, muss zwischen lexikalischen und metrischen Indizes unterschieden werden:

[2] Die Nachbarsuche kann selbstverständlich maximal nur so viele Elemente zurückgeben, wie im S^3V Baum verwaltet werden. Wird nach mehr Nachbarn gesucht als Elemente vorhanden sind, so variiert auch die Anzahl an Elementen, die von der Nachbarsuche gefunden werden.

lexikalische Indizes. Während die Anzahl lexikalischer Indizes zwar von Dokumentvergleich zu Dokumentvergleich aufgrund der unterschiedlichen Anzahl an Elementnamen variiert, bleibt die Zahl der Indizes, an denen ein anderer Wert als 0 eingetragen ist, konstant. Die Höhe der entsprechenden Werte ergibt sich durch die Skalierung der Vektoren.

metrische Indizes. Die Anzahl metrischer Indizes ist für unterschiedliche Dokumente eines Dokumenttyps konstant. Auch hier ergibt sich die Höhe der Werte anhand der Skalierung.

Mit diesen Informationen kann der maximale euklidische Abstand d_{max} zwischen zwei Vektoren wie folgt bestimmt werden:

$$d_{max} = \sqrt{\sum_{i=1}^{m}(fm_i)^2 + 2\sum_{i=1}^{l}(fl_i)^2}. \tag{5.1}$$

Dabei bezeichnet m die Anzahl metrischer Indizes und fm_i gibt den Skalierungsfaktor für den i-ten metrischen Index an. Entsprechend ist l die Anzahl lexikalischer Indexarten, deren Skalierungsfaktoren mit fl_i identifiziert werden.

Die Formel ergibt sich unter der Annahme, dass einer der Vektoren an allen metrischen Indizes den minimalen Wert – nämlich 0 – aufweist, während der andere jeweils den maximalen Wert annimmt. Außerdem wird davon ausgegangen, dass sich alle Namen und Namensteile der entsprechenden Elemente unterscheiden. Damit ergibt sich für jede lexikalische Indexart ein Abstand, der doppelt so groß ist wie der entsprechende Skalierungsfaktor, da sich die Unterschiede jeweils bei den Werten aus beiden Dokumenten auswirken.

Die Formel zur Berechnung des maximalen euklidischen Abstands zeigt, dass dieser Wert nicht dokumentabhängig ist und insbesondere nicht von der Anzahl an Elementen in den zu vergleichenden Dokumenten beeinflusst wird. Daraus folgt, dass die Grenzwerte für die Bereichsanfrage nicht dokumentabhängig sind und für unterschiedliche Dokumente eines Dokumenttyps keine Skalierung der Grenzwerte stattfinden muss. Eine Skalierung der Grenzwerte muss lediglich dann vorgenommen werden, wenn S^{3}V Bäume für unterschiedliche Elementtypen aufgebaut werden und somit andere Skalierungsfaktoren gelten.

Gestaffelte Grenzwerte

Der Grenzwert für die Ähnlichkeit von Elementen muss so definiert werden, dass trotz der verringerten Zahl an paarweisen Vergleichen immer noch die Vergleiche durchgeführt werden, die notwendig sind, um alle Korrespondenzen zu erkennen. Andererseits liegt oft der Fall vor, dass bereits mit einem relativ kleinen Grenzwert alle Elemente gefunden werden, die zu betrachten sind. In anderen Fällen würde ein derart kleiner Grenzwert jedoch bei der Bereichsabfrage keine Elemente zurückliefern.

Daraus ergibt sich die Idee, zwei gestaffelte Grenzwerte für Bereichsanfragen zu definieren. Im ersten Durchlauf wird die Bereichsanfrage mit dem kleineren der Grenzwerte durchgeführt. Falls damit bereits Elemente gefunden werden, ist die Anfrage beendet. Nur wenn die erste Anfrage keine Elemente findet, wird die Bereichsanfrage mit dem zweiten, größeren Grenzwert wiederholt.

Gestaffelte Grenzwerte stellen eine Möglichkeit dar, die Zahl der notwendigen paarweisen Vergleiche weiter zu verringern. Andererseits bietet der S^3V Baum keine effiziente Möglichkeit, gestaffelte Grenzwerte bei den Bereichsanfragen zu berücksichtigen. Werden beim ersten Durchlauf keine Elemente gefunden, so muss eine komplett neue Bereichsanfrage durchgeführt werden. Dies führt zu einer höheren Laufzeit.

Ob bei der Verwendung von gestaffelten Grenzwerten die Laufzeiteinsparung durch weniger paarweise Vergleiche oder die zusätzliche Laufzeit durch weitere Bereichsanfragen überwiegt, wird in Kapitel 6 an konkreten Beispielen untersucht.

Paarweiser Vektorvergleich

Während bei Bereichsanfragen im S^3V Baum effizient zu den Buckets navigiert werden kann, die Kandidaten für Elemente aus dem Suchbereich enthalten, ist der Test, ob die in den Buckets enthaltenen Elemente tatsächlich innerhalb des Suchbereichs liegen, relativ aufwändig, da jeweils über alle Dimensionen die Distanz berechnet werden muss. Durch die Optimierung bezüglich dünnbesetzter Vektoren kann der Aufwand dieser Berechnung zwar reduziert werden, er ist aber trotzdem noch deutlich höher als der Aufwand zum Auffinden der Buckets.

Der entsprechende Test ist nichts anderes als ein Vergleich jeden Vektors in den fraglichen Buckets mit dem zentralen Element des Suchbereiches. Da alle von der Bereichsanfrage zurück gelieferten Elemente aber anschließend noch anhand der in Abschnitt 5.1 beschriebenen Abhängigkeiten mit dem gesuchten Element verglichen werden, ergibt sich die Überlegung, die Bereichsanfrage so zu konfigurieren, dass sie alle Elemente in den fraglichen Buckets zurück liefert und auf den paarweisen Vektorvergleich verzichtet. Damit wird die Laufzeit der Bereichsanfragen signifikant verringert, es sind jedoch mehr paarweise Vergleiche der Elemente nötig. Diese Variante wird ebenfalls in Kapitel 6 evaluiert.

Der gleichzeitige Einsatz von gestaffelten Grenzwerten und ein Verzicht auf den paarweisen Vektorvergleich schließen sich aus. Da ohne paarweisen Vektorvergleich auch Elemente miteinander verglichen werden, die sich nur anhand einiger Dimensionen ähneln, in anderen Dimensionen aber komplett unterscheiden können, darf die Bereichsabfrage nicht bereits mit einem sehr kleinen Grenzwert abgebrochen werden, wenn mindestens ein Element gefunden wird. Ohne paarweisen Vektorvergleich kann über die Ähnlichkeit zwischen den Elementen in der Ergebnismenge der Bereichsabfrage und dem zentralen Element des Suchbereichs keine Aussage gemacht werden.

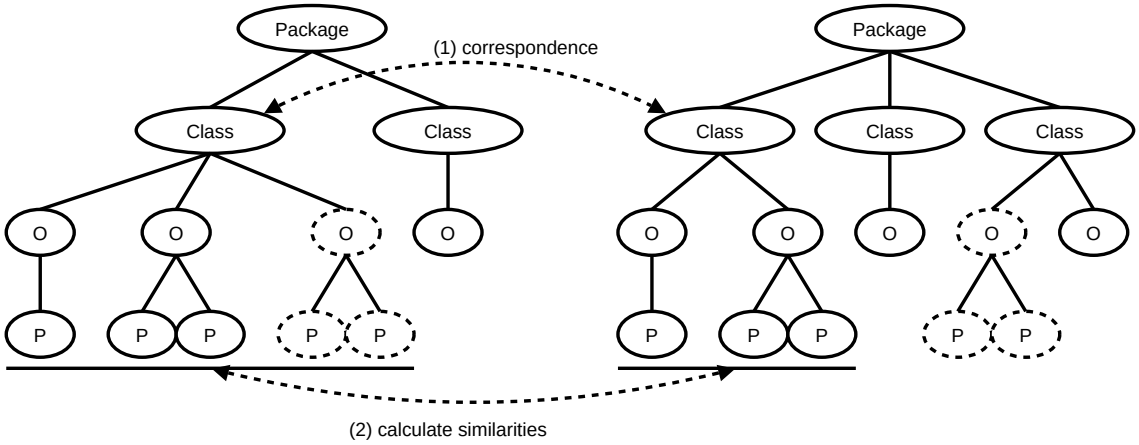

Abbildung 5.2: Schematischer Ablauf des Vergleichs in der Top-Down Variante

5.3 Integration Top-Down

Die Top-Down Variante der Integration setzt voraus, dass in den zu vergleichenden Dokumenten Teil-von Beziehungen existieren und dass es einen Elementtyp gibt, der direkt oder indirekt Elternelementtyp mehrerer anderer Elementtypen ist und von dem außerdem mehrere Elemente in jedem Dokument existieren. Bei UML-Klassendiagrammen trifft dies auf Klassen zu, bei Simulink-Diagrammen auf Blöcke.

Beim paarweisen, nach Elementtypen gruppierten Vergleich der Elemente fallen vor allem untergeordnete Elementtypen ins Gewicht, da von ihnen typischerweise sehr viele Elemente in den Dokumenten vorliegen. Ziel der Top-Down Integration ist es, nur solche untergeordneten Elemente zu vergleichen, die direkt oder indirekt Teil desselben Elternelements sind, indem im Vorfeld des paarweisen Vergleichs bereits eine vorläufige Zuordnung der übergeordneten Elemente stattfindet.

Abbildung 5.2 zeigt den Ablauf schematisch anhand des Dokumenttyps UML-Klassendiagramm und den Elementtypen *Package*, *Class*, *Operation* und *Parameter*. In einer ersten Phase werden anhand eines S^3V Baumes Klassen einander zugeordnet. Dabei ist es möglich, einer Klasse aus dem ersten Dokument mehr als eine Klasse aus dem zweiten Dokument zuzuordnen – dies ist in der Abbildung aus Gründen der Übersichtlichkeit nicht dargestellt.

In der zweiten Phase werden alle Elemente paarweise verglichen, wobei solche Elemente, die Teile in der Teil-von Hierarchie sind, nur mit Elementen verglichen werden, die über ein korrespondierendes übergeordnetes Element verfügen. Auf das Beispiel UML-Klassendiagramme übertragen bedeutet dies, dass eine vorläufige Zuordnung der Klassen stattfindet und alle untergeordneten Elemente wie Operationen, Attribute oder Parameter anschließend nur noch mit den entsprechenden Elementen korrespondierender Klassen verglichen werden. Die endgültige Zuordnung der übergeordneten Elemente findet statt, wenn die Vergleiche der untergeordneten Elemente abgeschlossen sind.

Ein Nachteil der Top-Down Variante besteht darin, dass Verschiebungen von un-

tergeordneten Elementen nicht erkannt werden können, da untergeordnete Elemente mit unterschiedlichen übergeordneten Elementen nicht zwangsläufig miteinander verglichen werden. In Abbildung 5.2 ist dieser Sachverhalt durch die gestrichelte Markierung einer der Operationen mit ihren Parametern verdeutlicht. Eine Verschiebung dieser Operation würde mit der Top-Down Variante nicht erkannt, da kein direkter Vergleich der beiden entsprechenden Elemente stattfindet[3].

Algorithmus 2 Korrespondenzen bestimmen mit paarweisem Elementvergleich

```
1: function FINDMATCHINGS(List elements1, List elements2) : Void
2:     calculateSimilarities (elements1, elements2)
3:     newMatchings = calculateMatchings (elements1, elements2)
4:     for all matching in newMatchings do
5:         propagateMatching (matching.element1, matching.element2)
6:     end for
7: end function
```

In Algorithmus 2 wird zunächst anhand von Pseudocode gezeigt, wie Korrespondenzen beim paarweisen Vergleich ohne die Integration von S^{3}V Bäumen erkannt werden. Die Operation *findMatchings* wird für jeden Elementtyp mit den zugehörigen Elementen aus beiden zu vergleichenden Dokumenten aufgerufen. *CalculateSimilarities* nimmt den paarweisen Vergleich der als Parameter übergebenen Elementmengen vor und speichert für jedes Paar dessen Ähnlichkeitswert. Diese Operation benötigt den größten Teil der Laufzeit, so dass alle Optimierungen an dieser Stelle ansetzen. Auf Basis der paarweisen Ähnlichkeiten werden die Korrespondenzen durch den Aufruf von *calculateMatchings* bestimmt und abschließend an die Kindelemente propagiert.

Algorithmus 3 zeigt den entsprechenden Code für die mit S^{3}V Bäumen integrierte Version nach der Top-Down Variante. Der bisherige Aufruf von *calculateSimilarities* aus Zeile 2 ist der if-then-else Anweisung aus den Zeilen 2-13 gewichen. Alle Elemente, die nicht Teil der Teil-von Hierarchie sind, müssen wie bisher paarweise verglichen werden. Für die anderen Elemente wird gruppiert nach ihren übergeordneten Elementen (im Algorithmus als *topElements* bezeichnet) vorgegangen. Für jedes übergeordnete Element wird zunächst die Menge der korrespondierenden Elemente ermittelt; *getSimilarElements* bezeichnet den gekapselten Aufruf der Bereichsanfrage im S^{3}V Baum. Die Bereichsanfrage wird nicht für jeden Elementtyp neu angestoßen, die Ergebnisse werden vielmehr einmal ermittelt und dann in einem Cache vorgehalten, so dass sie von nachfolgenden Anfragen wiederverwendet werden können. Der paarweise Vergleich durch Aufruf der Operation *calculateSimilarities* in Zeile 10 erfolgt dann nur noch mit den Elementen, die korrespondierende übergeordnete Elemente aufweisen. Die entsprechenden Elementmengen werden durch Aufruf der Hilfsoperation *getSubElements* bestimmt.

[3]Da die genaue Behandlung von Verschiebungen im Rahmen der Berechnung von Dokumentdifferenzen oft unklar ist, fällt dieser Nachteil nicht unbedingt ins Gewicht.

Algorithmus 3 Korrespondenzen bestimmen in der Top-Down Variante

```
 1: function findMatchings(List elements1, List elements2) : Void
 2:     if elementtype not in hierarchy then
 3:         calculateSimilarities (elements1, elements2)
 4:     else
 5:         for all topElement in topElements do
 6:             correspondingElements = getSimilarElements (topElement)
 7:             subElements1 = getSubElements (elements1, topElement)
 8:             for all correspElement in correspondingElements do
 9:                 subElements2 = getSubElements (elements2, correspElement)
10:                 calculateSimilarities (subElements1, subElements2)
11:             end for
12:         end for
13:     end if
14:     newMatchings = calculateMatchings (elements1, elements2)
15:     for all matching in newMatchings do
16:         propagateMatching (matching.element1, matching.element2)
17:     end for
18: end function
```

Dokumenttypdefinition 1 zeigt abschließend die DTD, auf deren Basis XML-Konfigurationsdateien für die Top-Down Variante der Bestimmung von Dokumentdifferenzen mit dem S^{3}V Baum erstellt werden können. Entsprechende XML-Dateien befinden sich in Anhang B für UML-Klassendiagramme und Anhang D für Simulink-Diagramme.

Die `MainComponent` gibt an, welcher der Elementtypen als übergeordneter Elementtyp behandelt werden soll. In den `Components` werden alle Elementtypen aufgelistet, die bei der Berechnung der Vektoren berücksichtigt werden müssen. In den meisten Fällen ist dies nur der übergeordnete Elementtyp selbst, für die Berechnung von Zwischenergebnissen können jedoch weitere Elementtypen benötigt werden[4]. Die Konfiguration des S^{3}V Baumes findet durch ein mit `TreeConfiguration` bezeichnetes *Tag* statt. Dort werden die Bucketgröße, die Anzahl der zu findenden Nachbarn bei Nachbarsuchen, die ggf. gestaffelten Grenzwerte für die Bereichsanfragen sowie die zu verwendenden Algorithmen für Bucket-Splits und Bereichsanfragen spezifiziert. Außerdem ist es möglich, für die Skalierung der Vektoren Default-Faktoren anzugeben.

Innerhalb des *Tags* `ComponentConfiguration` wird für jeden Elementtyp angegeben, welche Indizes die entsprechenden Vektoren aufweisen. Da Metriken aufeinander aufbauen können, ist es zusätzlich über das Attribut `pass` möglich zu spezifizieren, in welcher Reihenfolge die Berechnung stattfinden soll. Für jeden metrischen In-

[4]Bei Klassen aus UML-Klassendiagrammen können beispielsweise auch Metriken für die Operationen und Attribute berechnet werden, die dann als metrische Indizes in die Vektoren der Klassen eingehen.

Dokumenttypdefinition 1 Top-Down Konfiguration

```
<!DOCTYPE S3VTopDown [
  <!ELEMENT S3VTopDown (MainComponent, Components, TreeConfiguration,
                        ComponentConfiguration+, SkipList)>
  <!ELEMENT MainComponent EMPTY>
  <!ATTLIST MainComponent name CDATA #REQUIRED>
  <!ELEMENT Components (Component+)>
  <!ELEMENT Component EMPTY>
  <!ATTLIST Component name CDATA #REQUIRED>
  <!ELEMENT TreeConfiguration (Parameters, Splitter, Scanner, Weights)>
  <!ELEMENT Parameters EMPTY>
  <!ATTLIST Parameters scaled (true|false) #REQUIRED
                       bucketsize CDATA #REQUIRED
                       neighbours CDATA #REQUIRED
                       threshold CDATA #REQUIRED
                       secondThreshold CDATA #IMPLIED>
  <!ELEMENT Splitter EMPTY>
  <!ATTLIST Splitter name CDATA #REQUIRED>
  <!ELEMENT Scanner EMPTY>
  <!ATTLIST Scanner name CDATA #REQUIRED>
  <!ELEMENT Weights EMPTY>
  <!ATTLIST Weights dictionaryIndex CDATA #REQUIRED
                    dictionaryIndexPart CDATA #REQUIRED
                    metricIndex CDATA #REQUIRED>
  <!ELEMENT ComponentConfiguration (MetricIndices*, DictionaryIndices?)>
  <!ATTLIST ComponentConfiguration nodeType CDATA #REQUIRED>
  <!ELEMENT MetricIndices ((MetricIndex|TempIndex)+)>
  <!ATTLIST MetricIndices pass CDATA #REQUIRED>
  <!ELEMENT MetricIndex EMPTY>
  <!ATTLIST MetricIndex name CDATA #REQUIRED
                        description CDATA #IMPLIED
                        operation CDATA #REQUIRED
                        parameter CDATA #REQUIRED
                        weight CDATA #IMPLIED
                        requires CDATA #IMPLIED>
  <!ELEMENT TempIndex EMPTY>
  <!ATTLIST TempIndex name CDATA #REQUIRED
                      description CDATA #REQUIRED
                      operation CDATA #REQUIRED
                      parameter CDATA #REQUIRED
                      requires CDATA #IMPLIED>
  <!ELEMENT DictionaryIndices (DictionaryIndex+)>
  <!ELEMENT DictionaryIndex EMPTY>
  <!ATTLIST DictionaryIndex parameter CDATA #REQUIRED
                            prefix CDATA #REQUIRED
                            weight CDATA #IMPLIED
                            partWeight CDATA #IMPLIED>
  <!ELEMENT SkipList (Component*)>
]>
```

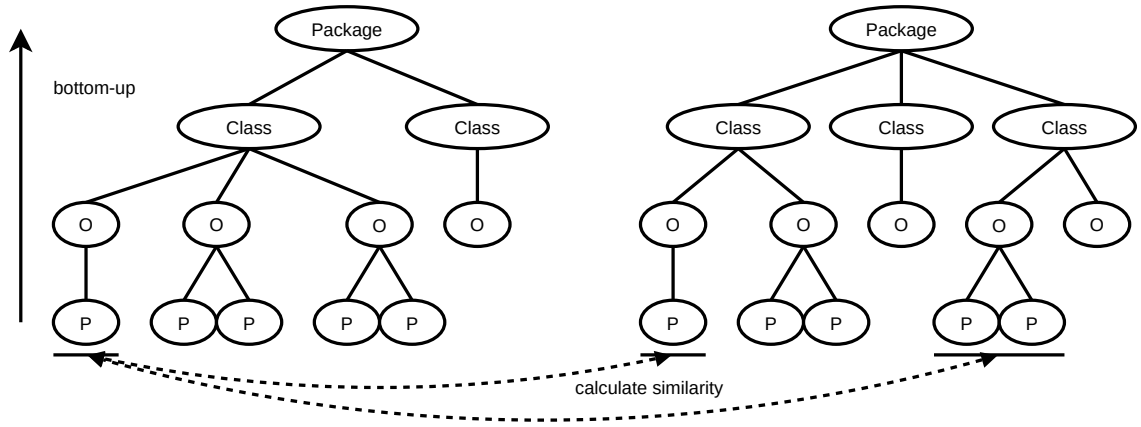

Abbildung 5.3: Schematischer Ablauf des Vergleichs in der Bottom-Up Variante

dex wird neben seinem Namen und einer optionalen Beschreibung angegeben, wie er berechnet und skaliert wird. Jede lexikalische Indexart wird durch Angabe des Pfades, eines Präfixes für den Indexnamen und Skalierungsfaktoren für Namen und Namensteile definiert.

Die in der abschließenden `SkipList` aufgelisteten Elementtypen umfassen alle Typen, die nicht dem als `MainComponent` bezeichneten Elementtyp untergeordnet sind und die somit nach wie vor paarweise verglichen werden müssen. Die explizite Nennung dieser Elementtypen ist nicht zwingend notwendig, sie bietet aber eine Möglichkeit zur Optimierung der Laufzeit, da für alle Elemente der hier genannten Typen nicht nach übergeordneten Elementen gesucht werden muss.

5.4 Integration Bottom-Up

Während in der Top-Down Variante nur für einen Elementtyp ein S^3V Baum erstellt wird, existiert nach der Bottom-Up Variante für jeden Elementtyp ein eigener Baum und es werden für jeden paarweisen Vergleich zunächst die aufgrund des euklidischen Abstands ihrer Vektoren unähnlichen Elemente aus der Menge der zu vergleichenden Elemente entfernt.

Abbildung 5.3 zeigt schematisch den Ablauf des Elementvergleichs gemäß der Bottom-Up Variante. Ob Dokumentelemente über korrespondierende übergeordnete Elemente verfügen, spielt keine Rolle. Die für den paarweisen Vergleich herangezogenen Elemente werden lediglich anhand des Ergebnisses der Bereichsanfrage im entsprechenden S^3V Baum ausgewählt. Im Gegensatz zur Top-Down Variante ist es somit möglich, Verschiebungen von Elementen zu erkennen, denn ähnliche untergeordnete Elemente werden auch dann verglichen, wenn sich ihre übergeordneten Elemente unterscheiden.

Algorithmus 4 zeigt den Pseudocode der Bottom-Up Variante. Die Operation *findMatchings* wird für jeden Elementtyp mit den zugehörigen Elementen aus beiden Dokumenten aufgerufen. Für jedes Element im ersten Dokument wird eine Bereichs-

Algorithmus 4 Korrespondenzen bestimmen in der Bottom-Up Variante

```
1:  function findMatchings(List elements1, List elements2) : Void
2:      for all element in elements1 do
3:          similarElements = getSimilarElements (element)
4:          for all similarElement in similarElements do
5:              if similarElement in elements2 then
6:                  calculateSimilarities (element, similarElement)
7:              end if
8:          end for
9:      end for
10:     newMatchings = calculateMatchings (elements1, elements2)
11:     for all matching in newMatchings do
12:         propagateMatching (matching.element1, matching.element2)
13:     end for
14: end function
```

anfrage im S^{3}V Baum gestartet. Bevor mit dem Aufruf von *calculateSimilarities* der paarweise Vergleich zwischen einem Element aus dem ersten Dokument und seinen ähnlichen Elementen im zweiten Dokument aufgerufen wird, wird noch geprüft, ob die Elemente, die von der Bereichsanfrage zurück geliefert wurden, Teil der Elementmenge sind, mit der *findMatchings* aufgerufen wurde. Dies ist z.B. dann nicht der Fall, wenn durch ein vorgelagertes Hashing bereits einige Elemente einander zugeordnet wurden. Diese Elemente brauchen bei der weiteren Suche nach korrespondierenden Elementen nicht mehr betrachtet werden, werden aber dennoch von Bereichsanfragen im S^{3}V Baum gefunden.

Das Konfigurationsformat für die Bottom-Up Variante vereinfacht das Format der Top-Down Variante an einigen Stellen, da keine Sonderbehandlung eines übergeordneten Elementtyps notwendig ist. Dokumenttypdefinition 2 zeigt die entsprechende DTD. Den einzelnen *Tags* kommt dieselbe Bedeutung zu wie in Abschnitt 5.3 beschrieben. Für die Konfiguration der S^{3}V Bäume wird zwischen einer `DefaultConfiguration` und einzelnen `TreeConfigurations` unterschieden. Damit ist es möglich, die Bäume individuell für jeden Elementtyp zu konfigurieren oder Standard-Werte für alle Typen zu verwenden.

Anhang C enthält eine entsprechende XML-Konfigurationsdatei für UML-Klassendiagramme. Die Konfiguration der Bottom-Up Variante ist deutlich aufwändiger als die Konfiguration der Top-Down Variante, da für jeden Elementtyp sämtliche Vektorindizes und ggf. die S^{3}V Bäume zu definieren sind.

Dokumenttypdefinition 2 Bottom-Up Konfiguration

```
<!DOCTYPE S3VBottomUp [
  <!ELEMENT S3VBottomUp (Components, DefaultConfiguration,
                         ComponentConfiguration+)>
  <!ELEMENT Components (Component+)>
  <!ELEMENT Component EMPTY>
  <!ATTLIST Component name CDATA #REQUIRED>
  <!ELEMENT DefaultConfiguration (Parameters, Splitter, Scanner, Weights)>
  <!ELEMENT Parameters EMPTY>
  <!ATTLIST Parameters scaled (true|false) #REQUIRED
                       bucketsize CDATA #REQUIRED
                       neighbours CDATA #REQUIRED
                       threshold CDATA #REQUIRED
                       secondThreshold CDATA #IMPLIED>
  <!ELEMENT Splitter EMPTY>
  <!ATTLIST Splitter name CDATA #REQUIRED>
  <!ELEMENT Scanner EMPTY>
  <!ATTLIST Scanner name CDATA #REQUIRED>
  <!ELEMENT Weights EMPTY>
  <!ATTLIST Weights dictionaryIndex CDATA #REQUIRED
                    dictionaryIndexPart CDATA #REQUIRED
                    metricIndex CDATA #REQUIRED>
  <!ELEMENT ComponentConfiguration (TreeConfiguration?, MetricIndices*,
                                    DictionaryIndices?)>
  <!ATTLIST ComponentConfiguration nodeType CDATA #REQUIRED>
  <!ELEMENT TreeConfiguration (Parameters, Splitter, Scanner, Weights)>
  <!ELEMENT MetricIndices ((MetricIndex|TempIndex)+)>
  <!ATTLIST MetricIndices pass CDATA #REQUIRED>
  <!ELEMENT MetricIndex EMPTY>
  <!ATTLIST MetricIndex name CDATA #REQUIRED
                        description CDATA #IMPLIED
                        operation CDATA #REQUIRED
                        parameter CDATA #REQUIRED
                        weight CDATA #IMPLIED
                        requires CDATA #IMPLIED>
  <!ELEMENT TempIndex EMPTY>
  <!ATTLIST TempIndex name CDATA #REQUIRED
                      description CDATA #REQUIRED
                      operation CDATA #REQUIRED
                      parameter CDATA #REQUIRED
                      requires CDATA #IMPLIED>
  <!ELEMENT DictionaryIndices (DictionaryIndex+)>
  <!ELEMENT DictionaryIndex EMPTY>
  <!ATTLIST DictionaryIndex parameter CDATA #REQUIRED
                            prefix CDATA #REQUIRED
                            weight CDATA #IMPLIED
                            partWeight CDATA #IMPLIED>
]>
```

6 Evaluierung der Ergebnisse

Um den mehrdimensionalen Ansatz zur Optimierung der Bestimmung von Dokumentdifferenzen zu evaluieren, muss untersucht werden, inwieweit in der praktischen Anwendung insbesondere für große Dokumente tatsächlich Verbesserungen der Laufzeit erreicht werden können und ob sich die erzielten Differenzergebnisse qualitativ von den Ergebnissen unterscheiden, die durch paarweisen Vergleich aller Elemente zustande kommen. Diese Fragestellungen werden im vorliegenden Kapitel betrachtet.

Außerdem wird auf einige Besonderheiten des S^3V Baumes eingegangen und aufgezeigt, ob die bei der Konstruktion des Baumes getroffenen Annahmen in verschiedenen Test-Szenarien zutreffen. In Abschnitt 6.1 wird zunächst die gewählte Testumgebung vorgestellt, bevor in den Abschnitten 6.2 und 6.3 die wichtigsten Testergebnisse untergliedert nach den beiden Alternativen zur Integration in die Differenzberechnung präsentiert werden. Vor einer abschließenden Bewertung der Ergebnisse zeigt Abschnitt 6.4 noch einige detaillierte Auswertungen bezüglich des S^3V Baumes.

6.1 Testumgebung

Um vergleichbare Ergebnisse zu erhalten und eine gemeinsame Ausgangsbasis nutzen zu können, wurde das Konzept der S^3V Bäume als Teil des Differenztools SiDiff implementiert.

SiDiff

Bei SiDiff[1] handelt es sich um ein generisches Differenzwerkzeug, das anhand des Anwendungsfalls UML-Diagramme detailliert in [Weh04] vorgestellt wird. Der Algorithmus betrachtet die zu vergleichenden Dokumente als Graphen, deren Elemente wie in Abschnitt 5.1 beschrieben gruppiert nach Typen paarweise verglichen werden. Eine vorgelagerte Hashing-Phase ordnet bereits vor dem paarweisen Vergleich die Elemente einander zu, die gemäß ihrer Hashwerte exakt übereinstimmen.

Als interne Datenstruktur für die Dokumente wird das in Abbildung 4.1 auf Seite 41 gezeigte Modell verwendet. Durch den generischen Ansatz müssen für Adaptierungen auf bestimmte Dokumenttypen lediglich die Konfigurationsdateien, die die Ähnlichkeitskriterien für die diversen Elementtypen angeben, angepasst werden. Das

[1] `http://pi.informatik.uni-siegen.de/sidiff/`

Differenzwerkzeug eignet sich somit als Basis für die generische Implementierung von S^{3}V Bäumen.

Testdaten

Bei der Auswahl der Testdaten wurde Wert darauf gelegt, reale Dokumente aus der praktischen Anwendung zu verwenden. Dazu wurden die Dokumente teilweise im Zuge von Reverse Engineering aus dem Quelltext existierender Projekte generiert. Außerdem wurde gefordert, dass die Testdaten eine Heterogenität hinsichtlich der Ähnlichkeit ihrer jeweiligen Vergleichsdokumente aufweisen, so dass sowohl Versionen verglichen werden, bei denen nur wenige Änderungen vorgenommen wurden, als auch Versionen, zwischen denen große Unterschiede vorliegen.

Folgende Test-Szenarien wurden für die Evaluierung des Einsatzes von S^{3}V Bäumen bei der Differenzbestimmung eingesetzt:

Szenario A. Zwei kleinere UML-Klassendiagramme, die HTML Dokumente modellieren und in [OWK03] als Beispiele angegeben wurden. Das größere der Diagramme wird in Abbildung 6.1 gezeigt.

Szenario B. Zwei Versionen eines UML-Klassendiagramms, das während eines Programmierpraktikums an der Universität Kassel entwickelt wurde. Die Versionen wurden aus dem internen Konfigurationsmanagementsystem entnommen und sind nicht durch Reverse Engineering entstanden.

Szenario C. Zwei UML-Klassendiagramme, die aus dem Quelltext des UML-Werkzeugs Fujaba[2] generiert wurden. Es handelt sich um das Package ASG (Abstract Syntax Graph).

Szenario D. Zwei Versionen eines UML-Klassendiagramms, das aus dem Quelltext des in [Hut07] entwickelten SiDiff Plug-ins zur Nachverfolgbarkeit von Modellelementen in Versionshistorien generiert wurde.

Szenario E. Das Package FSA (Fujaba Swing Adapter), ebenfalls aus dem Fujaba-Quelltext generiert.

Szenario F. Zwei Versionen des Basic Package von Fujaba.

Szenario G. Szenario G enthält die größten untersuchten UML-Klassendiagramme. Es handelt sich um das UML Package von Fujaba mit allen zugehörigen Unter-Packages.

Szenario Simulink. Mit dem abschließenden Szenario zweier Matlab Simulink Diagramme wurde untersucht, ob sich der mehrdimensionale Ansatz auf andere Dokumenttypen übertragen lässt. Die verglichenen Diagramme verfügen über je 5 *in-ports*, 3 *out-ports*, einen *bus-creator* und einen *bus-selector* sowie 2 bzw. 3 Subsysteme.

Elementtyp	A	B	C	D	E	F	G
datatype	4	5	6	7	10	13	10
stereotype	0	1	2	4	2	2	2
parameter	2	17	79	118	420	383	1261
operation	13	63	86	98	439	283	1900
attribute	6	97	10	68	14	85	557
class	13	28	53	40	102	122	391
associationEnd	6	54	42	42	74	114	336
association	3	27	21	21	37	57	168
generalization	5	10	15	5	37	26	77
package	1	15	31	21	38	44	74
model	1	1	1	1	1	1	1
Summe	54	318	346	425	1174	1130	4777

Tabelle 6.1: Elementanzahl in den Testdokumenten

In Tabelle 6.1 wird für die Szenarien, die UML-Klassendiagramme zum Gegenstand haben, die Anzahl an enthaltenen Elementen angegeben. Die Elemente werden zusätzlich nach den Elementtypen aufgeschlüsselt, die in UML-Klassendiagrammen auftreten können.

Konfiguration

Für die Testläufe wurden die Konfigurationsdateien verwendet, die in den Anhängen B, C und D abgedruckt sind. Die Konfigurationen für UML-Klassen stimmen mit den in Kapitel 4 entwickelten Werten überein, alle weiteren Konfigurationen orientieren sich an denselben Prinzipien. Für den Vergleich von Simulink-Diagrammen nach der Top-Down Variante wurden Subsysteme als übergeordnete Elementtypen gewählt.

6.2 Top-Down Testläufe

Die Testergebnisse für die Top-Down Variante werden in Tabelle 6.2 gezeigt. Neben den in Spalte Sc aufgeführten Szenarien wurden bei den Testläufen folgende Parameter variiert:

H Sämtliche Szenarien wurden sowohl mit einer vorgelagerten Hashing-Phase als auch ohne Hashing getestet. Testläufe mit Hashing sind in der Tabelle durch ein *t* für *true* markiert. Prinzipiell handelt es sich für den S^3V Baum jeweils um zwei unterschiedliche Testfälle. Da nur die Elemente im S^3V Baum verwaltet werden, die nicht in der vorgelagerten Hashing-Phase zugeordnet werden konnten, variiert die Menge der verwalteten Elemente in diesen beiden Situationen.

[2] `http://www.fujaba.de`

Sc	H	BT	t1	t2	CS	RQ	VD	Match%	CS%	RT_p	RT
A	t	t	1,5	2,3	282	8	99	114,3	64,7	0,2	0,1
	–	t	1,5	2,3	369	8	99	115,8	53,5	0,3	0,2
B	t	t	1,5	2,3	265	23	638	100,0	63,5	0,1	0,4
	–	t	1,5	2,3	4194	23	638	100,0	38,2	2,2	1,6
C	t	t	1,5	2,3	837	50	2630	100,0	65,4	0,4	0,5
	–	t	2,3	–	14703	49	2597	100,0	73,1	2,2	2,3
	–	t	0,0	2,3	11979	57	551	100,0	59,6	2,2	1,9
	–	t	0,5	2,3	11691	53	1300	100,0	58,1	2,2	2,3
	–	t	1,0	2,3	11545	51	1835	100,0	57,4	2,2	2,2
	–	t	1,5	2,3	11545	50	2632	100,0	57,4	2,2	2,0
	–	t	1,8	2,3	12369	49	2597	100,0	61,5	2,2	2,0
	–	–	2,3	–	20105	49	0	100,0	100,0	2,2	2,7
D	t	t	1,5	2,3	888	44	1653	100,5	30,1	1,5	0,4
	–	t	1,5	2,3	5810	44	1666	100,5	17,6	1,8	1,4
E	t	t	2,3	–	14776	97	9894	100,0	18,3	1,9	1,5
	t	t	0,0	2,3	12848	119	2943	100,0	15,9	1,9	1,2
	t	t	0,5	2,3	3650	100	4886	100,0	4,5	1,9	0,7
	t	t	1,0	2,3	3480	97	6622	100,0	4,3	1,9	0,9
	t	t	1,5	2,3	3480	97	9891	100,0	4,3	1,9	1,0
	t	t	1,8	2,3	3704	97	9894	100,0	4,6	1,9	1,0
	t	–	2,3	–	80924	97	0	100,0	100,0	1,9	2,6
	–	t	1,5	2,3	40781	97	9894	99,7	9,2	14,0	3,0
F	t	t	1,5	2,3	11152	121	14099	99,7	5,7	3,3	1,9
	–	t	1,5	2,3	59751	121	14097	89,5	7,5	18,2	3,8
G	t	t	2,3	–	44478	351	137241	99,8	36,6	45,8	21,0
	t	t	0,0	2,3	18812	428	33588	99,7	15,5	45,8	3,6
	t	t	0,5	2,3	17990	280	67896	99,7	14,8	45,8	3,6
	t	t	1,0	2,3	17791	374	96170	99,7	14,6	45,8	4,1
	t	t	1,5	2,3	17475	371	143129	99,7	14,4	45,8	4,2
	t	t	1,8	2,3	17877	354	138358	99,8	14,7	45,8	6,4
	t	–	2,3	–	121511	351	0	100,0	100,0	45,8	63,6
	–	t	2,3	–	504336	351	137241	99,6	8,8	1409,4	133,3
	–	t	0,0	2,3	321508	428	33263	99,6	5,6	1409,4	52,8
	–	t	0,5	2,3	308239	380	65029	99,6	5,3	1409,4	51,9
	–	t	1,0	2,3	307370	374	95224	99,6	5,3	1409,4	51,1
	–	t	1,5	2,3	306806	371	143406	99,6	5,3	1409,4	51,5
	–	t	1,8	2,3	328974	354	138358	99,7	5,7	1409,4	69,4
	–	–	2,3	–	5762913	351	0	100,0	100,0	1409,4	1396,3

Tabelle 6.2: Testergebnisse Top-Down

Darüber hinaus ist ohne vorgelagertes Hashing die Zahl ähnlicher Elemente in der Suchstruktur höher.

BT Die Spalte BT *(Bucket Test)* gibt an, ob ein paarweiser Vektorvergleich stattfindet oder ob alle Elemente aus den Buckets, die bei den Bereichsanfragen nicht ausgeschlossen werden konnten, in den paarweisen Vergleich der Elemente eingehen.

t1 und t2 In den meisten Testläufen wurde mit gestaffelten Grenzwerten gearbeitet. t1 gibt jeweils den kleineren der Grenzwerte an, t2 den größeren. In den Fällen, in denen nur ein Grenzwert verwendet wurde, ist der entsprechende Wert in der Spalte t1 eingetragen.

Die weiteren Spalten präsentieren die Ergebnisse der Testläufe:

CS und CS% Die Spalte CS *(Calculate Similarities)* stellt für jeden Testlauf dar, wie oft die Operation zum direkten Vergleich zweier Elemente aufgerufen wurde. Diese Operation beansprucht bei der Differenzberechnung den größten Teil der Laufzeit. In der Spalte CS% wird der jeweilige Wert ins Verhältnis zu dem Wert gesetzt, der sich ohne die Integration von S^3V Bäumen ergibt.

Die Ergebnisse zeigen, dass die Anzahl an direkten Vergleichen in allen Fällen, in denen der *Bucket Test* durchgeführt wurde, deutlich verringert werden konnte. Vor allem ohne vorgelagertes Hashing ergeben sich für größere Dokumente Einsparungen von über 90%. Außerdem zeigt sich, dass der Einsatz von gestaffelten Grenzwerten eine wirksame Methode darstellt, um die Zahl der notwendigen Direktvergleiche zu reduzieren.

RQ In der mit RQ *(Range Queries)* bezeichneten Spalte ist die Anzahl von Bereichsanfragen aufgeführt, die in den S^3V Bäumen vollzogen wurden. Die Werte steigen mit der Anzahl an Elementen in den Testdokumenten, sind jedoch für unterschiedliche Kombinationen der gestaffelten Grenzwerte relativ homogen.

VD Die Spalte VD *(Vector Distance)* gibt an, wie viele Direktvergleiche zweier Vektoren bei Bereichsanfragen im S^3V Baum notwendig waren. Diese Zahl lässt darauf schließen, wie viele Buckets bei den Bereichsanfragen ausgeschlossen werden konnten. Je kleiner der Wert, desto weniger Buckets mussten betrachtet werden.

Es fällt auf, dass ein kleiner erster Grenzwert bei Verwendung gestaffelter Grenzwerte dazu führt, dass weniger Vektorvergleiche stattfinden. Diese Beobachtung erklärt sich dadurch, dass bei einem kleinen Grenzwert sehr viele Teilbäume des S^3V Baumes bei Bereichsanfragen ausgeschlossen werden können.

Match% In der mit Match% überschriebenen Spalte wird gezeigt, wie viele der Korrespondenzen, die ohne die Integration von S^3V Bäumen gefunden wurden, auch nach der Integration der mehrdimensionalen Suchstruktur ermittelt werden konnten. Mögliche Diskrepanzen sind auf 3 potentielle Fehlerarten zurückzuführen:

- Verschiebungen. Wie in Abschnitt 5.3 beschrieben, können anhand der Top-Down Variante keine Korrespondenzen zwischen verschobenen Elementen erkannt werden. Darauf sind die 10,5%-Abweichung in Szenario F ohne Hashing sowie die Diskrepanzen in Szenario A zurückzuführen.
- Weniger Korrespondenzen. Weniger Korrespondenzen werden außerdem erkannt, wenn bereits zu viele Elemente durch die Bereichsanfragen im S^3V Baum ausgefiltert werden. In diesem Fall werden nicht alle Ähnlichkeitswerte berechnet, die für die Erkennung von Korrespondenzen notwendig sind.
- Mehr Korrespondenzen. Da der Differenzalgorithmus zwei Elemente mit einer hinreichenden Ähnlichkeit nur dann als korrespondierend betrachtet, wenn die Elemente nicht gleichzeitig noch zu weiteren Elementen eine Ähnlichkeit aufweisen, können potentiell falsche Korrespondenzen gefunden werden, wenn nicht alle Direktvergleiche ausgeführt werden.

In keinem der Testläufe trat der Fall auf, dass gleichzeitig bestimmte Korrespondenzen zusätzlich erkannt wurden während andere nicht gefunden wurden.

Bei der Interpretation der in der Spalte Match% gegebenen Werte ist zu berücksichtigen, dass sich in SiDiff bereits ohne S^3V Bäume abhängig davon, ob mit oder ohne Hashing gearbeitet wird, Unterschiede im Differenzergebnis ergeben. Diese Unterschiede sind darauf zurückzuführen, dass sich mit Hashing bereits früh stabile Teile des Ergebnisses bilden. Vor allem bei Dokumenten mit vielen Querverweisen zwischen Elementen (z.B. Assoziationen in UML-Klassendiagrammen) wirkt sich dieser Punkt aus. Die als Match% aufgeführten Werte ergeben sich, wenn die Testläufe getrennt danach verglichen werden, ob Hashing angewendet wurde oder nicht. Außerdem unterliegen Ergebnisse von Differenzberechnungen einer gewissen Subjektivität, da sämtliche Ähnlichkeitskriterien auf Heuristiken zurückgehen.

Mit Ausnahme von Szenario F ohne Hashing liegt die Diskrepanzquote bei weniger als 0,5%. Dieses Ergebnis zeigt, dass trotz einer Einsparung von teilweise bis zu 95% der Direktvergleiche das Differenzergebnis kaum beeinträchtigt wird. Die Diskrepanzen liegen auf demselben Niveau wie die Unterschiede, die sich bereits vor der Integration des S^3V Baumes in den Varianten mit bzw. ohne Hashing ergaben.

RT_p und RT Die letzten beiden Spalten zeigen den Laufzeitvorteil, der sich durch die mehrdimensionalen Suchstrukturen ergibt. RT_p *(Runtime pairwise)* gibt die Laufzeit der Variante ohne Integration des S^3V Baumes an, während die

Hashing	Methode	CS	RQ	VD	Matches	Zeit in ms
true	Paarweise	542			57	150,0
true	Top-Down	270	2	6	57	60,0
false	Paarweise	681			57	151,0
false	Top-Down	237	2	6	57	70,0

Tabelle 6.3: Testergebnisse Simulink

Laufzeit der integrierten Variante in der Spalte RT gezeigt wird. Dabei wird die Zeit, die für den Aufbau der Baumstruktur benötigt wird, nicht berücksichtigt, weil zum einen der entsprechende Aufwand linear mit der Zahl der Elemente wächst und somit für den Vergleich großer Dokumente irrelevant ist und zum anderen je nach Anwendungsfall die Baumstruktur nicht für jeden Vergleich neu aufgebaut werden muss.

Eine Laufzeitverbesserung kann in fast allen Testfällen erreicht werden. Für die größten Dokumente in Szenario G ergibt sich mit Hashing eine Verbesserung von bisher 45,8 Sekunden auf etwa 4 Sekunden je nach Konfiguration der gestaffelten Grenzwerte, ohne Hashing kann eine Verbesserung von bisher über 23 Minuten auf knapp über 50 Sekunden erreicht werden.

Simulink

Tabelle 6.3 zeigt die Ergebnisse, die für das Matlab Simulink Szenario erzielt wurden. Sowohl für den Testfall mit Hashing als auch für den Testfall ohne Hashing stimmt das Differenzergebnis nach der integrierten Variante mit dem Ergebnis ohne S^3V Bäume (in der Tabelle als *Paarweise* bezeichnet) überein. Die Anzahl der notwendigen Direktvergleiche, die in der Spalte CS angegeben wird, konnte jeweils etwa halbiert werden, das gleiche gilt für die Laufzeit.

6.3 Bottom-Up Testläufe

Für die Szenarien mit UML-Klassendiagrammen wurden zusätzlich zu den Top-Down Testläufen Messungen nach der Bottom-Up Variante durchgeführt. Die Ergebnisse zeigt Tabelle 6.4, deren Aufbau mit dem der Tabelle für die Top-Down Testergebnisse übereinstimmt.

Die Qualität der Differenzergebnisse kann gegenüber der Top-Down Variante noch einmal verbessert werden, es sind insgesamt weniger Diskrepanzen vorhanden. Zusätzlich entspricht in allen Situationen, in denen mehr Korrespondenzen erkannt wurden, das Differenzergebnis dem entsprechenden Referenzergebnis mit Hashing. Dies ist darauf zurückzuführen, dass der S^3V Baum gewissermaßen eine erweiterte Hashing-Funktionalität anbietet, da Elemente, die exakt übereinstimmen, in der

Sc	H	BT	t1	t2	CS	RQ	VD	Match%	CS%	RT_p	RT
A	t	t	1,5	2,3	143	112	586	100,0	32,8	0,2	0,1
	–	t	1,5	2,3	175	134	779	110,5	25,4	0,3	0,2
B	t	t	1,5	2,3	92	32	1099	100,0	22,1	0,1	0,4
	–	t	1,5	2,3	558	246	14352	100,0	5,1	2,2	0,8
C	t	t	1,5	2,3	340	120	5939	100,0	26,6	0,4	0,5
	–	t	2,3	–	13648	530	31931	100,0	67,9	2,2	1,7
	–	t	0,0	2,3	1573	509	5369	100,0	7,8	2,2	0,8
	–	t	0,5	2,3	1221	487	10619	100,7	6,1	2,2	0,7
	–	t	1,0	2,3	1125	493	17010	100,0	5,6	2,2	1,1
	–	t	1,5	2,3	1396	512	29673	100,0	6,9	2,2	0,9
	–	t	1,8	2,3	5888	536	31990	100,0	29,3	2,2	1,9
	–	–	2,3	–	20102	546	0	100,0	100,0	2,2	2,3
D	t	t	1,5	2,3	707	248	14996	100,0	24,0	1,5	0,9
	–	t	1,5	2,3	1967	685	52368	100,0	6,0	1,8	1,1
E	t	t	2,3	–	80789	412	165806	100,0	99,8	1,9	2,4
	t	t	0,0	2,3	19819	435	30698	100,0	24,5	1,9	1,3
	t	t	0,5	2,3	19807	433	81711	100,0	24,5	1,9	1,3
	t	t	1,0	2,3	19801	429	112742	100,0	24,5	1,9	2,2
	t	t	1,5	2,3	19376	425	166679	100,0	23,9	1,9	1,7
	t	t	1,8	2,3	40595	418	167093	100,0	50,2	1,9	1,7
	t	–	2,3	–	80921	412	0	100,0	100,0	1,9	2,2
	–	t	1,5	2,3	25339	2261	827644	99,7	5,7	14,0	4,7
F	t	t	1,5	2,3	61568	4867	1285763	99,7	31,3	3,3	5,9
	–	t	1,5	2,3	213247	7875	2119694	99,6	26,7	18,2	11,8
G	t	t	2,3	–	92044	1148	1054187	100,0	75,7	45,8	17,0
	t	t	0,0	2,3	15671	1310	240937	99,7	12,9	45,8	4,8
	t	t	0,5	2,3	13910	1263	460065	99,7	11,4	45,8	5,4
	t	t	1,0	2,3	12563	1218	614702	99,8	10,3	45,8	5,5
	t	t	1,5	2,3	14204	1382	1152551	99,9	11,7	45,8	7,8
	t	t	1,8	2,3	41112	1221	1091765	99,9	33,8	45,8	8,9
	t	–	2,3	–	121453	1147	0	100,0	100,0	45,8	43,6
	–	t	2,3	–	5065938	8072	9896396	99,9	87,9	1409,4	1084,6
	–	t	0,0	2,3	100216	7716	805519	99,7	1,7	1409,4	22,9
	–	t	0,5	2,3	86105	7676	3178348	99,6	1,5	1409,4	27,6
	–	t	1,0	2,3	103518	7876	5592955	99,7	1,8	1409,4	35,6
	–	t	1,5	2,3	118974	8023	9701013	99,8	2,1	1409,4	48,7
	–	t	1,8	2,3	935541	8047	9895289	99,9	16,2	1409,4	85,0
	–	–	2,3	–	5762731	8111	0	100,0	100,0	1409,4	1284,6

Tabelle 6.4: Testergebnisse Bottom-Up

Elementtyp	lexikalisch	metrisch	Summe
datatype	10	4	14
stereotype	2	2	4
parameter	16	1	17
operation	28	10	38
attribute	34	1	35
class	48	25	73
associationEnd	52	1	53
association	20	2	22
generalization	26	0	26
package	49	4	53
model	6	4	10

Tabelle 6.5: Anzahl der Vektorindizes in Test-Szenario A

Suchstruktur zusammen angeordnet werden.

Da für die Integration nach der Bottom-Up Variante das Erkennen von Verschiebungen kein Problem darstellt, ergibt sich kein Ausreißer in Szenario F. In den Werten aus den Spalten RQ und VD schlägt sich nieder, dass nach der Bottom-Up Variante für jeden Elementtyp ein S^3V Baum aufgebaut wird und dementsprechend die Zahl der Bereichsanfragen und direkten Vektorvergleiche deutlich höher ist als in der Top-Down Variante. Dafür kann jedoch die Anzahl der notwendigen Direktvergleiche zweier Elemente (siehe Spalte CS) gegenüber der Top-Down Variante weiter verringert werden – teilweise lassen sich über 98% der Direktvergleiche einsparen.

Die kleinere Zahl an notwendigen Direktvergleichen impliziert eine verbesserte Laufzeit, die sich in der Spalte RT widerspiegelt. Für die größten Dokumente in Szenario G kann die Laufzeit in der Variante ohne Hashing von über 23 Minuten auf etwa eine halbe Minute reduziert werden.

6.4 Verhalten des S^3V Baums

In diesem Abschnitt sollen einige Charakteristika des S^3V Baumes anhand ausgesuchter Test-Szenarien genauer betrachtet werden.

Vektorindizes

Am Beispiel von Szenario A wird zunächst gezeigt, über wie viele Vektorindizes die Vektoren, die in den S^3V Bäumen verwaltet werden, in einem typischen Anwendungsfall verfügen.

Abbildung 6.1 zeigt das größere der beiden UML-Klassendiagramme aus Test-Szenario A. Die Anzahl der Vektorindizes ist in Tabelle 6.5 dargestellt. Die entsprechenden Messungen wurden in der Bottom-Up Variante vorgenommen, so dass

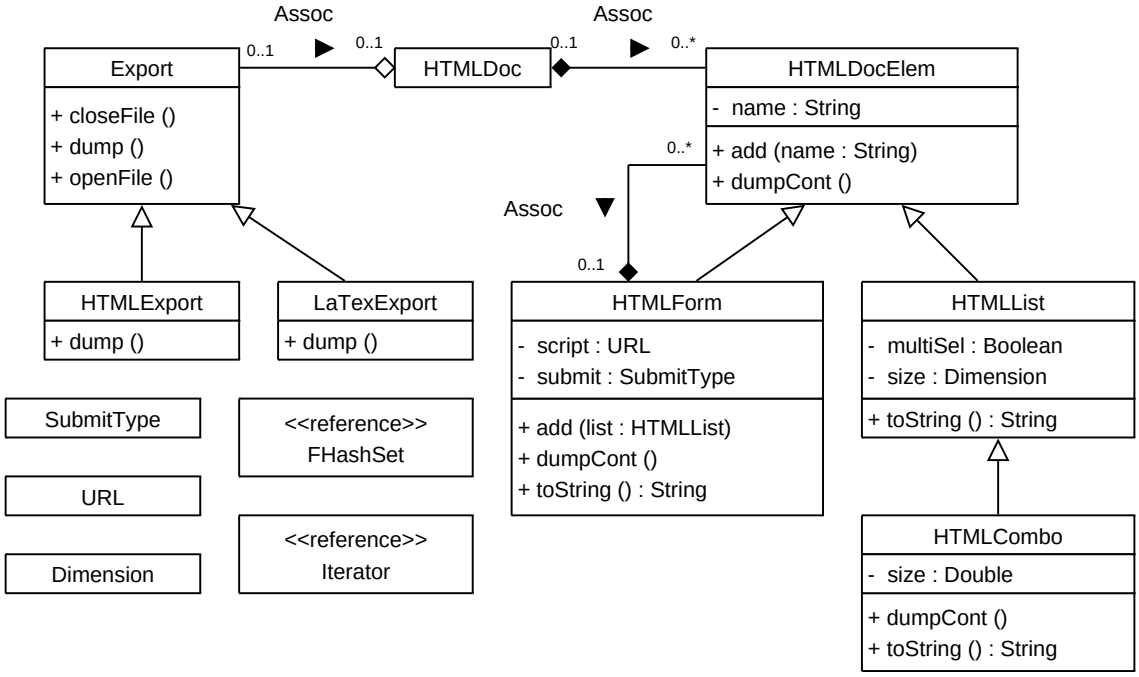

Abbildung 6.1: UML-Klassendiagramm für Szenario A

Vektoren für alle Elementtypen existieren. Die Indizes werden in lexikalische und metrische Indizes aufgeschlüsselt. Bereits in dem sehr kleinen Beispiel fällt auf, dass im Durchschnitt mehr lexikalische als metrische Indizes vorliegen. Da die Anzahl metrischer Indizes konstant ist und die Anzahl lexikalischer Indizes mit der Anzahl an Dokumentelementen wächst, ist dieser Unterschied für alle anderen Test-Szenarien noch größer.

Auffallend in der Tabelle ist weiterhin die große Anzahl lexikalischer Indizes für Elementtypen, bei denen die Namen ihrer untergeordneten Elemente als Indizes verwendet werden. So erklärt sich beispielsweise die große Anzahl an lexikalischen Indizes für den Elementtyp Package, obwohl in den Beispiel-Diagrammen jeweils nur ein Package existiert.

Dank der Auslegung des S^3V Baumes auf hochdimensionale Vektoren und der Tatsache, dass die Vektoren vor allem an den lexikalischen Indizes dünn besetzt sind, stellt die verhältnismäßig große Anzahl an Indizes kein Problem dar.

Bucketgrößen

Bisher wurden alle Testläufe mit einer Bucketgröße von 1 durchgeführt. In diesem Abschnitt soll untersucht werden, wie eine Veränderung der Bucketgröße die Laufzeit beeinflusst.

Eine Variation der Bucketkapazität wirkt sich an zwei Stellen aus – zum einen beim statischen Einfügen in den S^3V Baum, zum anderen bei Bereichsanfragen. Beim statischen Einfügen führt eine größere Bucketkapazität dazu, dass weniger

Bucketgröße	VD Operationen (in Mio.)	RQ Operationen (in Mio.)	Summe (in Mio.)
1	102,26	25,97	128,23
2	163,02	14,33	177,34
3	166,79	11,24	178,03
4	171,34	7,93	179,26
5	173,14	6,17	179,31

Tabelle 6.6: Testergebnisse für unterschiedliche Bucketgrößen in Test-Szenario G

Splits durchgeführt werden müssen und resultiert somit in einer geringeren Laufzeit. Da jedoch der Aufwand zum Aufbau der Bäume lediglich linear im Vergleich zur Zahl der Elemente wächst und für große Dokumente keine Bedeutung hat, werden hier nur die Auswirkungen unterschiedlicher Bucketgrößen auf Bereichsanfragen untersucht.

Auf die Laufzeit von Bereichsanfragen hat eine höhere Bucketkapazität sowohl einen positiven als auch einen negativen Effekt. Diese Effekte werden im Folgenden gegenübergestellt. Da eine höhere Anzahl von Elementen pro Bucket die Höhe des S^3V Baumes verringert, nimmt die Zahl der inneren Knoten, die zu betrachten sind und für die die entsprechenden Teilräume berechnet werden müssen, ab. Andererseits steigt die Anzahl der notwendigen direkten Vektorvergleiche, da für jedes Bucket linear durch die Menge der enthaltenen Vektoren navigiert werden muss.

Um die beiden Effekte vergleichbar zu machen, können sie auf die Anzahl ihrer Elementaroperationen abgebildet werden. Ein direkter Vektorvergleich besteht in einer Addition der quadrierten Differenz der Werte an jedem Vektorindex[3]. Dabei werden nur Vektorindizes betrachtet, an denen in mindestens einem der Vektoren ein anderer Wert als 0 eingetragen ist. Entsprechend lässt sich die Berechnung eines Teilraums an einem inneren Knoten oder Bucket im S^3V Baum ausdrücken. Der Teilraum verändert sich an genau einem der Vektorindizes um die quadrierte Differenz, die sich durch den Split ergibt.

In Tabelle 6.6 werden für Test-Szenario G ohne Hashing und mit den gestaffelten Grenzwerten 1,0 und 2,3 für Bereichsanfragen die Auswirkungen unterschiedlicher Bucketkapazitäten auf die Anzahl notwendiger Additionen bzw. Subtraktionen quadrierter Differenzen dargestellt, die bei der Durchführung der Testläufe gemessen wurden. Die Spalte VD Operationen gibt an, wie viele der Operationen für die Bestimmung von Vektordistanzen benötigt wurden, während die Spalte RQ Operationen die Zahl der Operationen auflistet, die für das Navigieren zu den Buckets gebraucht wurde.

Die Daten zeigen, dass die Anzahl der Berechnungen für Vektordistanzen mit zunehmender Bucketkapazität steigt, während die Zahl der Operationen für die Navigation zu den Buckets sinkt. In der Summe ergeben sich jedoch eindeutig für eine

[3]Da konsequent mit den quadrierten Werten gerechnet wird, entfällt das Ziehen der Wurzel bei der Ermittlung des euklidischen Abstands.

Bucketgröße von 1 die besten Werte. Die nur sehr geringen Unterschiede der Ergebnisse für größere Bucketkapazitäten lassen sich darauf zurückführen, dass in diesen Fällen die S^{3}V Bäume zu wenige Ebenen haben, um viele Buckets bei Bereichsanfragen ausschließen zu können und somit fast alle Vektoren paarweise verglichen werden müssen.

Die Ergebnisse zur Zahl der notwendigen Operationen decken sich mit den Beobachtungen der entsprechenden Laufzeiten und unterstützen somit eine Bucketgröße von 1 im S^{3}V Baum.

Splitstrategie

Eine der Annahmen bei der Entwicklung des S^{3}V Baumes bestand darin, dass es möglich ist, bei hochdimensionalen Vektoren und statischem Einfügen Splitdimensionen und -positionen zu finden, die zum einen die Elementmenge in zwei gleich große Mengen aufteilen und zum anderen eine möglichst große Distanz zwischen den beiden Splitpositionen aufweisen (vgl. Abschnitt 3.2). In diesem Abschnitt wird untersucht, ob diese Annahme zutrifft.

Abbildung 6.2 zeigt den S^{3}V Baum, der für die UML-Klassen aus Szenario A (vgl. Abbildung 6.1) entsteht, als Beispiel. An dem Beispiel lassen sich einige typische Charakteristika der Baumstruktur erkennen:

- Der Baum ist balanciert. Da jeder Split die Elementmenge genau in der Mitte teilt, ist die Höhe des S^{3}V Baumes bei n Elementen durch $O(\log n)$ beschränkt.
- Als Splitdimensionen treten viele unterschiedliche Vektorindizes auf. Darunter fallen sowohl metrische (z.B. SUP für die Anzahl an Superklassen) als auch lexikalische Indizes. Bei den lexikalischen Indizes treten zudem unterschiedliche Indexarten auf: im Beispiel sind dies Klassennamen, Methodennamen und Teile von Klassennamen.
- Die Distanz zwischen den beiden Splitpositionen beträgt in diesem Beispiel an keinem der inneren Knoten 0. Damit kann bei jedem Split gegenüber dem LSD-Baum ein Vorteil hinsichtlich der Bereichsanfragen gewonnen werden, da sich an jedem inneren Knoten für die Teilbäume kleinere Teilräume ergeben.
- Die Distanz zwischen den Splitpositionen auf der untersten Ebene beträgt immer 1. Dies ist zwar theoretisch nicht grundsätzlich der Fall – vor allem dann nicht, wenn mehrfach identische Elemente im S^{3}V Baum verwaltet werden – aber meistens kann für zwei Elemente eine entsprechende optimale Splitdimension gefunden werden.

Wie Tabelle 6.7 zeigt, wirken sich die Vorteile der doppelten Splitposition im S^{3}V Baum in allen Test-Szenarien sowohl in der Top-Down Variante als auch in der Bottom-Up Variante aus.

In der mit Splits überschriebenen Spalte wird angegeben, wie viele Splits beim Aufbau der Bäume durchgeführt wurden. Bei den Ergebnissen nach der Top-Down

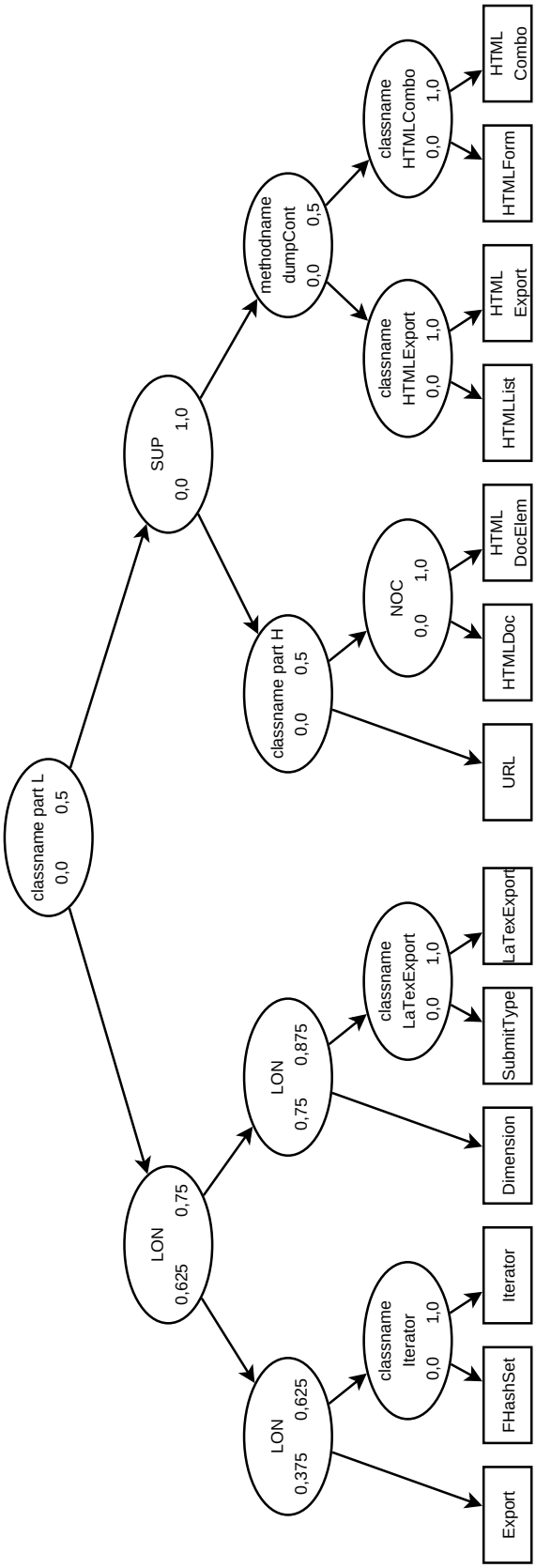

Abbildung 6.2: Aufbau des S^3V Baumes für Test-Szenario A

Szenario	Top-Down				Bottom-Up			
	Splits	Avg	$d = 0$	in %	Splits	Avg	$d = 0$	in %
A	12	0,67	0	0,0	50	0,67	7	14,0
B	27	0,63	1	3,7	307	0,56	79	25,7
C	39	0,60	5	12,8	424	0,60	61	14,4
D	52	0,53	4	7,7	335	0,54	56	16,7
E	101	0,48	17	16,8	1163	0,47	317	27,3
F	121	0,56	19	15,7	1119	0,51	216	19,3
G	390	0,47	57	14,6	4897	0,54	848	17,3

Tabelle 6.7: Testergebnisse für duale Splitpositionen

Variante bezieht sich dies jeweils auf einen Baum, da lediglich einer aufgebaut wird; nach der Bottom-Up Variante ist die Anzahl der Splits über aller Bäume aufsummiert. Die Spalte Avg gibt an, wie groß im Durchschnitt die Differenz zwischen den beiden Splitpositionen ist. In der Spalte $d = 0$ wird aufgeführt, bei wie vielen der Splits der *worst case* zum Tragen kam und keine Splitdimension gefunden werden konnte, an der sich die Splitpositionen unterschieden. Die letzte Spalte setzt diese Zahl ins Verhältnis zur Gesamtzahl der Splits.

Wie die Werte zeigen, kann bei einer – durch die Normierung bedingten – Maximaldistanz von 1 im Durchschnitt eine Distanz von über 0,5 zwischen den Splitpositionen erreicht werden. Dies bedeutet für Bereichsanfragen eine entscheidende Verbesserung gegenüber dem LSD-Baum: An jedem inneren Knoten kann die Größe der Teilräume der darunter liegenden Teilbäume in einer Dimension um durchschnittlich mehr als 0,25 Einheiten pro Teilbaum verringert werden. Besonders bei der Verwendung von gestaffelten Grenzwerten, bei denen der erste Grenzwert relativ klein ist, können somit sehr viele Teilbäume mit den zugehörigen Buckets in der Bereichsabfrage ausgeschlossen werden.

Weiterhin kann festgestellt werden, dass etwa 80% der Splits zu einer Verbesserung im Vergleich zum LSD-Baum führen. Der *worst case*, in dem beide Splitpositionen übereinstimmen, tritt im Durchschnitt nur bei jedem fünften Split auf.

6.5 Bewertung der Ergebnisse

Die Ergebnisse zeigen, dass die Laufzeit der Differenzbestimmung für große Dokumente durch den Einsatz von mehrdimensionalen Suchstrukturen signifikant verbessert werden kann und nur sehr wenige Diskrepanzen im Vergleich zur paarweisen Ähnlichkeitsbestimmung für alle Elemente auftreten. Auch für kleinere Dokumente ist der Laufzeitvorteil vorhanden.

Der Laufzeitgewinn ist in erster Linie auf die stark reduzierte Anzahl an direkten Elementvergleichen zurückzuführen. Im Gegensatz dazu wirkt sich die Anzahl

an Bereichsanfragen im S^3V Baum sowie die damit verbundene Zahl an direkten Vektorvergleichen lediglich marginal auf die Gesamtlaufzeit aus.

Die Ergebnisse der Top-Down und Bottom-Up Testläufe unterscheiden sich nur unwesentlich. Generell ist eher die Bottom-Up Variante vorzuziehen, da zum einen das Problem nicht erkannter Verschiebungen hier nicht auftritt und zum anderen die Anzahl der notwendigen Direktvergleiche von Elementen und damit die Laufzeit noch deutlicher reduziert werden kann als mit der Top-Down Variante.

Das Konzept gestaffelter Grenzwerte hat sich als sehr nützliches Instrument erwiesen, um die Laufzeit weiter zu verringern. Welche Kombination von Grenzwerten genau verwendet wird, spielt dabei eine untergeordnete Rolle – die Ergebnisse für unterschiedliche Kombinationen variieren nur unwesentlich. Der Vorteil liegt hier offensichtlich darin, wie beim Hashing einen zusätzlichen Zwischenschritt einzubauen, der Elemente mit einer sehr hohen Ähnlichkeit bereits früh einander zuordnet und so stabile Teile für das Differenzergebnis schafft.

Im Gegensatz dazu hat sich das Verzichten auf den paarweisen Vektorvergleich als unbrauchbar herausgestellt. Dies liegt vor allem daran, dass gestaffelte Grenzwerte nicht eingesetzt werden können, wenn kein paarweiser Vektorvergleich durchgeführt wird. Außerdem ist durch die Optimierungen bezüglich dünn besetzter Vektoren im S^3V Baum der Aufwand für direkte Vektorvergleiche relativ gering.

Die gewählte Splitstrategie trägt beim Aufbau des S^3V Baumes entscheidend dazu bei, die Laufzeit der Bereichsanfragen zu minimieren. Aufgrund der Hochdimensionalität der Vektoren kann für einen großen Teil der Splits eine Dimension gefunden werden, die mit zwei unterschiedlichen Splitpositionen die Wahrscheinlichkeit erhöht, dass ganze Teilbäume des S^3V Baumes bei Bereichsanfragen nicht betrachtet werden müssen.

Verbesserungsmöglichkeiten

Das größte Verbesserungspotential ergibt sich durch die bisherige Verwendung zweier Ähnlichkeitsmaße für Dokumentelemente. Während Elemente im S^3V Baum anhand der Distanz ihrer Vektoren vorselektiert werden, werden sie anschließend nach anderen Kriterien paarweise verglichen.

Dieses Problem lässt sich nicht vollständig auflösen, da sich nicht alle Kriterien, die für die Ähnlichkeit zweier Elemente herangezogen werden, auf Vektoren abbilden lassen. Dies gilt z.B. für Kriterien, die die Korrespondenzen zwischen untergeordneten Elementen zählen. Auf Vektoren können lediglich die Ähnlichkeiten untergeordneter Elemente abgebildet werden; ob sie bereits einander zugeordnet wurden, lässt sich nicht erfassen. Verbesserungsspielraum besteht allerdings darin, Möglichkeiten zu suchen, mit denen die verwendeten Kriterien für den paarweisen Vergleich so verlustfrei wie möglich auf Vektorindizes abgebildet werden können, um zumindest ein Ähnlichkeitsmaß aus dem anderen generieren zu können.

Darüber hinaus lässt sich durch weitere Experimente mit unterschiedlichen Konfigurationen vermutlich noch etwas weiterer Laufzeitgewinn erzielen, wenn noch mehr Elemente vor den paarweisen Vergleichen aussortiert werden können.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde gezeigt, dass mehrdimensionale Suchstrukturen eingesetzt werden können, um maßgeblich Laufzeit bei der Differenzberechnung nicht-textueller Dokumente einzusparen.

Ansatzpunkt ist die paarweise Vergleichsphase, in der für jede Kombination eines Dokumentelements aus dem ersten Dokument und eines Elements aus dem zweiten Dokument ein Ähnlichkeitswert berechnet wird. Durch die Organisation der Elemente in einer Datenstruktur, die ähnliche Elemente benachbart anordnet, kann diese Phase so verändert werden, dass die Ähnlichkeiten nur noch für benachbarte Elemente explizit berechnet werden. Datenstrukturen aus verschiedenen Bereichen der Informatik wurden in Bezug auf diese Anforderung evaluiert; dabei erwiesen sich mehrdimensionale Baumstrukturen als eine geeignete Lösung für die Problemstellung.

Als Suchbaumstruktur wurde der S^3V Baum entwickelt, der für die Verwaltung hochdimensionaler Vektoren und Bereichsanfragen optimiert ist und es somit ermöglicht, zu einem gegebenen Element alle ähnlichen Elemente zu finden, ohne sämtliche Elemente betrachten zu müssen. Die Datenstruktur orientiert sich am LSD-Baum, nutzt jedoch die Tatsache, dass die zu verwaltenden Datenmengen im Hauptspeicher gehalten werden können. Weiterhin ist es möglich, den Baum statisch aufzubauen, da dynamisches Einfügen oder Löschen im Anwendungsfall der Differenzberechnung nicht auftritt. Unter dieser Voraussetzung lassen sich mit einer zweiten Splitposition und einer Splitstrategie, die auf zwei Splitpositionen zugeschnitten ist, strukturelle Veränderungen zur Effizienzsteigerung von Bereichsanfragen vornehmen. Dies führt dazu, dass weniger Teilbäume bei Bereichsanfragen betrachtet werden müssen und die Baumstruktur balanciert ist. Weitere Optimierungen werden für dünn besetzte Vektoren sowie in der Berechnung von Teilräumen bei Bereichsanfragen umgesetzt.

Für die Abbildung von Dokumentelementen auf die vom S^3V Baum verwalteten Vektoren werden zwei Arten von Indizes herangezogen. Metrische Indizes leiten sich aus Software-Metriken ab und bilden Eigenschaften von Elementen auf Zahlen ab. Lexikalische Indizes dienen dazu, Namensähnlichkeiten zu erfassen. Zu diesem Zweck werden Namen bzw. Namensteile dynamisch als Vektorindizes definiert.

Bei der Integration von S^3V Bäumen in die Differenzberechnung können Abhängigkeiten zwischen Dokumentelementen genutzt werden, um eine vorläufige Zuordnung von Elementen mit korrespondierenden übergeordneten Elementen vorzunehmen. Alternativ lässt sich ein separater Baum für jeden Elementtyp aufbauen, so dass vor jedem paarweisen Vergleich eine Vorauswahl der Elemente durch eine Bereichsanfrage im S^3V Baum stattfindet.

Es konnte gezeigt werden, dass mit Hilfe der mehrdimensionalen Suchstruktur die Anzahl paarweiser Elementvergleiche signifikant reduziert werden kann. Bereichsanfragen wirken sich nur unwesentlich auf die Laufzeit aus, so dass insgesamt ein deutlicher Performancegewinn realisiert wird, der sich vor allem bei großen Dokumenten auswirkt.

Weiterführende Arbeiten

Die vorliegende Arbeit hat die Einsatzfähigkeit mehrdimensionaler Suchstrukturen für die Optimierung der Bestimmung von Dokumentdifferenzen nachgewiesen. An folgenden Stellen bieten sich Erweiterungen und Integrationsmöglichkeiten an, die bisher noch nicht untersucht wurden:

Einheitliches Ähnlichkeitsmaß. Ein Problem bei der Integration von S^3V Bäumen in die Differenzberechnung besteht in der gleichzeitigen Verwendung zweier Ähnlichkeitsmaße für Dokumentelemente. Für Bereichsanfragen in der Baumstruktur wird die euklidische Distanz zweier Vektoren als Maß für die Ähnlichkeit eingesetzt, während der paarweise Vergleich der vorausgewählten Elemente auf Kriterien basiert, die sich teilweise nicht auf Vektoren abbilden lassen.

Zur Definition eines einheitlichen Ähnlichkeitsmaßes muss untersucht werden, ob sich die beiden Maße vereinen lassen bzw. ob eine generische Ableitung der Vektorindizes aus den Kriterien für den paarweisen Vergleich möglich ist.

Weitere Dokumenttypen. Bislang wurde der Einsatz mehrdimensionaler Suchstrukturen für die Optimierung der Bestimmung von Dokumentdifferenzen nur anhand von UML-Klassendiagrammen und Matlab Simulink Diagrammen untersucht. Die Anpassung an weitere Dokumenttypen, die von dem ursprünglichen Differenzalgorithmus verarbeitet werden können, stellt dank des generischen Ansatzes kein Problem dar und lässt sich automatisieren, wenn ein einheitliches Ähnlichkeitsmaß vorhanden ist.

Auffinden von Dokumentelementen in Versionshistorien. Das Konzept einer Datenstruktur für die Verwaltung von Dokumentelementen kann neben der Optimierung des Vergleichs zweier Dokumente auch für die Analyse von Versionshistorien eingesetzt werden. Dazu muss untersucht werden, wie verschiedene Anwendungsfälle durch die Verwaltung von Dokumentelementen aus mehreren Versionen in S^3V Bäumen sinnvoll optimiert werden können.

Dublettenerkennung. Für mehr als zwei Dokumente kann außerdem geprüft werden, inwieweit S^3V Bäume genutzt werden können, um im Sinne von symmetrischen Differenzen gemeinsame Teilmengen von Dokumenten zu identifizieren. Da Dublettenerkennung auf den paarweisen Vergleich aller Elemente in allen zu untersuchenden Dokumenten zurückzuführen ist, besteht durch die Verwendung einer mehrdimensionalen Datenstruktur für die Dokumentelemente Optimierungspotential.

Lernfähigkeit. Eine weitere Möglichkeit zur automatischen Konfiguration der Vektoren besteht darin, die durch paarweisen Vergleich erhaltenen Differenzergebnisse automatisiert zu verarbeiten und so Erkenntnisse für die Skalierung einzelner Vektorindizes zu gewinnen. Dieses Prinzip kann ebenfalls eingesetzt werden, um sinnvolle Grenzwerte für die Ähnlichkeit zweier Vektoren zu ermitteln.

Andere Datenstrukturen. Einige der in Kapitel 2 betrachteten Datenstrukturen scheinen ebenfalls geeignet zu sein, um die Laufzeit der paarweisen Vergleichsphase von Dokumentelementen zu reduzieren. Dazu zählt vor allem das Probabilistische Datalog, mit Hilfe dessen Elementeigenschaften über Fakten und Regeln und Ähnlichkeitskriterien über Wahrscheinlichkeiten modelliert werden können. In einer weiterführenden Arbeit kann untersucht werden, ob in einer entsprechenden Umsetzung tatsächlich Optimierungspotential liegt.

A Bedienung der Software

Da das Konzept der mehrdimensionalen Suchstrukturen in der Berechnung von Dokumentdifferenzen als Teil des Differenzwerkzeugs SiDiff implementiert wurde, wird das gesamte Werkzeug benötigt, um den Quelltext auszuführen. Der Aufruf der auf der beigefügten CD enthaltenen jar-Datei erfolgt per Kommandozeile mit

```
java -jar sidiff.jar <file1> <file2> <hash> <method> <config>.
```

Dabei bezeichnen `file1` und `file2` die Eingabedateien, die verglichen werden sollen und im XMI-Format vorliegen müssen. Über den Parameter `hash` wird gesteuert, ob ein vorgelagertes Hashing stattfinden soll; als Werte sind `true` und `false` zulässig. Der Parameter `method` gibt an, nach welcher Methode beim paarweisen Vergleich von Dokumentelementen vorgegangen werden soll. Mögliche Methoden sind `iterative` für den paarweisen Vergleich aller Elemente, `topDown` für die mit S^3V Bäumen integrierte Version nach der Top-Down Variante sowie `bottomUp` für die Bottom-Up Variante. Der abschließende Parameter `config` gibt für die integrierten Varianten die Konfigurationsdatei an, die nach den in Kapitel 5 angegebenen DTDs gebildet wird.

Neben Beispiel-Konfigurationsdateien und Beispiel-Dokumenten für den Vergleich ist auf der beigefügten CD der Quelltext zu finden, der im Rahmen dieser Arbeit entwickelt wurde.

B Konfigurationsdatei für UML-Klassendiagramme (Top-Down Variante)

Nachfolgend ist die Konfigurationsdatei für die Optimierung des Vergleichs von UML-Klassendiagrammen mit dem S^3V Baum nach der Top-Down Variante abgedruckt.

```
<?xml version="1.0"?>

<S3VTopDown>

   <MainComponent name="class" />

   <Components>
      <Component name="class" />
      <Component name="attribute" />
      <Component name="operation" />
   </Components>

   <TreeConfiguration>
      <Parameters scaled="true" bucketsize="1" neighbours="100"
        threshold="2.3" />
      <Splitter name="S3vDefaultSplitter" />
      <Scanner name="S3vDualRangeScanner" />
      <Weights dictionaryIndex="1.0" dictionaryIndexPart="1.0"
        metricIndex="1.0" />
   </TreeConfiguration>

   <ComponentConfiguration nodeType="class">
      <MetricIndices pass="1">
         <MetricIndex name="LON" description="length of name"
           operation="CountAttributeValueTextLength" parameter="name"
           weight="1.0" />
         <MetricIndex name="NAG" description="number of getters"
           operation="CountChildrenValueStartsWith"
           parameter="operation,name,get" weight="0.5" />
         <MetricIndex name="NAS" description="number of setters"
           operation="CountChildrenValueStartsWith"
           parameter="operation,name,set" weight="0.5" />
         <MetricIndex name="NAM" description="number of abstract methods"
           operation="CountChildNodesByValue"
           parameter="operation,isAbstract,true" weight="0.5" />
```

```
<MetricIndex name="NOCM" description="number of constructors"
  operation="CountClassConstructors" parameter="" weight="0.5" />
<MetricIndex name="NCV" description="number of class variables"
  operation="CountChildNodesByValue"
  parameter="attribute,ownerScope,classifier" weight="0.5" />
<MetricIndex name="NIV"
  description="number of instance variables"
  operation="CountChildNodesByValue"
  parameter="attribute,ownerScope,instance" weight="0.5" />
<MetricIndex name="NOM" description="number of methods"
  operation="CountChildNodesTypes" parameter="operation"
  weight="0.0" />
<MetricIndex name="NOA" description="number of attributes"
  operation="CountChildNodesTypes" parameter="attribute"
  weight="0.0" />
<MetricIndex name="NAPUB"
  description="number of public attributes"
  operation="CountChildNodesByValue"
  parameter="attribute,visibility,public" weight="1.0" />
<MetricIndex name="NAPRI"
  description="number of private attributes"
  operation="CountChildNodesByValue"
  parameter="attribute,visibility,private" weight="1.0" />
<MetricIndex name="NAPRO"
  description="number of protected attributes"
  operation="CountChildNodesByValue"
  parameter="attribute,visibility,protected" weight="1.0" />
<MetricIndex name="NAPAC"
  description="number of package-visible attributes"
  operation="CountChildNodesByValue"
  parameter="attribute,visibility,package" weight="1.0" />
<MetricIndex name="NMPUB" description="number of public methods"
  operation="CountChildNodesByValue"
  parameter="operation,visibility,public" weight="1.0" />
<MetricIndex name="NMPRI" description="number of private methods"
  operation="CountChildNodesByValue"
  parameter="operation,visibility,private" weight="1.0" />
<MetricIndex name="NMPRO"
  description="number of protected methods"
  operation="CountChildNodesByValue"
  parameter="operation,visibility,protected" weight="1.0" />
<MetricIndex name="NMPAC"
  description="number of package-visible methods"
  operation="CountChildNodesByValue"
  parameter="operation,visibility,package" weight="1.0" />
<MetricIndex name="SUP" description="number of superclasses"
  operation="CountIncomingEdges" parameter="generalization_src"
  weight="1.0" />
<MetricIndex name="NOC"
  description="number of children in class hierarchy"
  operation="CountIncomingEdges" parameter="generalization_tgt"
  weight="1.0" />
```

```
   </MetricIndices>
   <MetricIndices pass="2">
      <TempIndex name="PPI"
        description="pre-processing for inheritance based metrics"
        operation="CalculateInheritance" parameter=""
        requires="SUP,NOC,MDECL,ADECL" />
   </MetricIndices>
   <MetricIndices pass="3">
      <MetricIndex name="NMA" description="number of methods added"
        operation="CountMethodsAdded" parameter="" requires="PPI"
        weight="0.5" />
      <MetricIndex name="NMOI"
        description="number of methods overloaded, inherited methods
        are considered" operation="CountMethodsOverloadedInheritance"
        parameter="" requires="PPI" weight="0.5" />
      <MetricIndex name="NMOO"
        description="number of methods overloaded, without considering
        inheritance" operation="CountMethodsOverloaded" parameter=""
        requires="PPI" weight="0.5" />
      <MetricIndex name="NMO"
        description="number of methods overridden"
        operation="CountMethodsOverridden" parameter="" requires="PPI"
        weight="0.5" />
      <MetricIndex name="NMI" description="number of methods inherited"
        operation="CountMethodsInherited" parameter="" requires="PPI"
        weight="0.5" />
      <MetricIndex name="NIA"
        description="number of attributes inherited"
        operation="CountAttributesInherited" parameter=""
        requires="PPI" weight="0.5" />
   </MetricIndices>
   <DictionaryIndices>
      <DictionaryIndex parameter="name" prefix="class:" weight="1.0"
        partWeight="0.5" />
      <DictionaryIndex parameter="methods, name" prefix="method:"
        weight="0.5" partWeight="0.25" />
   </DictionaryIndices>
</ComponentConfiguration>

<ComponentConfiguration nodeType="attribute">
   <MetricIndices pass="1">
      <TempIndex name="ADECL"
        description="attribute declaration with type"
        operation="CalculateAttributeDeclaration" parameter="" />
   </MetricIndices>
</ComponentConfiguration>

<ComponentConfiguration nodeType="operation">
   <MetricIndices pass="1">
      <TempIndex name="MDECL"
        description="method declaration with parameter types"
        operation="CalculateMethodDeclaration" parameter="" />
```

```
      </MetricIndices>
   </ComponentConfiguration>

   <ComponentConfiguration nodeType="*">
      <MetricIndices pass="1">
          <TempIndex name="ANC"
            description="ancestor according to tree view, nodetype
            specified" operation="GetAncestorNode"  parameter="class"/>
      </MetricIndices>
   </ComponentConfiguration>

   <SkipList>
      <Component name="datatype"/>
      <Component name="association"/>
      <Component name="associationEnd"/>
      <Component name="package"/>
      <Component name="model"/>
      <Component name="stereotype"/>
   </SkipList>

</S3VTopDown>
```

C Konfigurationsdatei für UML-Klassendiagramme (Bottom-Up Variante)

Nachfolgend ist die Konfigurationsdatei für die Optimierung des Vergleichs von UML-Klassendiagrammen mit dem S^3V Baum nach der Bottom-Up Variante abgedruckt.

```
<?xml version="1.0"?>

<S3VBottomUp>
   <Components>
      <Component name="model" />
      <Component name="package" />
      <Component name="generalization" />
      <Component name="association" />
      <Component name="associationEnd" />
      <Component name="class" />
      <Component name="attribute" />
      <Component name="operation" />
      <Component name="parameter" />
      <Component name="stereotype" />
      <Component name="datatype" />
   </Components>

   <DefaultConfiguration>
      <Parameters scaled="true" bucketsize="1" neighbours="100"
        threshold="2.3" />
      <Splitter name="S3vDefaultSplitter" />
      <Scanner name="S3vDualRangeScanner" />
      <Weights dictionaryIndex="1.0" dictionaryIndexPart="1.0"
        metricIndex="1.0" />
   </DefaultConfiguration>

   <ComponentConfiguration nodeType="model">
      <MetricIndices pass="1">
         <MetricIndex name="model:LON" description="length of name"
           operation="CountAttributeValueTextLength" parameter="name"
           weight="1.0" />
         <MetricIndex name="model:NOP" description="number of packages"
           operation="CountNeighbourEdges" parameter="packages"
           weight="1.0" />
         <MetricIndex name="model:NOS" description="number of stereotypes"
```

```
        operation="CountNeighbourEdges" parameter="allstereotypes"
        weight="1.0" />
      <MetricIndex name="model:NOD" description="number of datatypes"
        operation="CountNeighbourEdges" parameter="datatypes"
        weight="1.0" />
   </MetricIndices>
   <DictionaryIndices>
      <DictionaryIndex parameter="name" prefix="model:" weight="1.0"
        partWeight="0.5" />
      <DictionaryIndex parameter="packages, name" prefix="package:"
        weight="0.5" partWeight="0.25" />
   </DictionaryIndices>
</ComponentConfiguration>

<ComponentConfiguration nodeType="package">
   <MetricIndices pass="1">
      <MetricIndex name="package:LON" description="length of name"
        operation="CountAttributeValueTextLength" parameter="name"
        weight="1.0" />
      <MetricIndex name="package:NOP" description="number of packages"
        operation="CountOutgoingEdges" parameter="packages"
        weight="1.0" />
      <MetricIndex name="package:NOA"
        description="number of associations"
        operation="CountNeighbourEdges" parameter="assocs" weight="1.0"
        />
      <MetricIndex name="package:NOC" description="number of classes"
        operation="CountNeighbourEdges" parameter="classes"
        weight="1.0" />
   </MetricIndices>
   <DictionaryIndices>
      <DictionaryIndex parameter="name" prefix="package:" weight="1.0"
        partWeight="0.5" />
      <DictionaryIndex parameter="classes, name" prefix="class:"
        weight="0.5" partWeight="0.25" />
   </DictionaryIndices>
</ComponentConfiguration>

<ComponentConfiguration nodeType="generalization">
   <DictionaryIndices>
      <DictionaryIndex parameter="generalization_src, name"
        prefix="src:" weight="1.0" partWeight="0.5" />
      <DictionaryIndex parameter="generalization_tgt, name"
        prefix="tgt:" weight="1.0" partWeight="0.5" />
   </DictionaryIndices>
</ComponentConfiguration>

<ComponentConfiguration nodeType="association">
   <MetricIndices pass="1">
      <MetricIndex name="association:LON" description="length of name"
        operation="CountAttributeValueTextLength" parameter="name"
        weight="1.0" />
```

```
      <MetricIndex name="association:NOAE"
        description="number of associationEnds"
        operation="CountNeighbourEdges" parameter="connection"
        weight="1.0" />
   </MetricIndices>
   <DictionaryIndices>
      <DictionaryIndex parameter="name" prefix="association:"
        weight="1.0" partWeight="0.5" />
      <DictionaryIndex parameter="connection, name"
        prefix="associationEnd:" weight="0.5" partWeight="0.25" />
   </DictionaryIndices>
</ComponentConfiguration>

<ComponentConfiguration nodeType="associationEnd">
   <MetricIndices pass="1">
      <MetricIndex name="associationEnd:LON"
        description="length of name"
        operation="CountAttributeValueTextLength" parameter="name"
        weight="1.0" />
   </MetricIndices>
   <DictionaryIndices>
      <DictionaryIndex parameter="name" prefix="associationEnd:"
        weight="1.0" partWeight="0.5" />
      <DictionaryIndex parameter="aggregation" prefix="aggregation:"
        weight="1.0" partWeight="0.0" />
      <DictionaryIndex parameter="isNavigable" prefix="isNavigable:"
        weight="1.0" partWeight="0.0" />
      <DictionaryIndex parameter="multiplicity" prefix="multiplicity:"
        weight="1.0" partWeight="0.0" />
      <DictionaryIndex parameter="ordering" prefix="ordering:"
        weight="1.0" partWeight="0.0" />
      <DictionaryIndex parameter="visibility" prefix="visibility:"
        weight="1.0" partWeight="0.0" />
      <DictionaryIndex parameter="connection, name"
        prefix="association:" weight="1.0" partWeight="0.5" />
      <DictionaryIndex parameter="target, name" prefix="class:"
        weight="1.0" partWeight="0.5" />
   </DictionaryIndices>
</ComponentConfiguration>

<ComponentConfiguration nodeType="class">
   <MetricIndices pass="1">
      <MetricIndex name="LON" description="length of name"
        operation="CountAttributeValueTextLength" parameter="name"
        weight="1.0" />
      <MetricIndex name="NAG" description="number of getters"
        operation="CountChildrenValueStartsWith"
        parameter="operation,name,get" weight="0.5" />
      <MetricIndex name="NAS" description="number of setters"
        operation="CountChildrenValueStartsWith"
        parameter="operation,name,set" weight="0.5" />
      <MetricIndex name="NAM" description="number of abstract methods"
```

```
  operation="CountChildNodesByValue"
  parameter="operation,isAbstract,true" weight="0.5" />
<MetricIndex name="NOCM" description="number of constructors"
  operation="CountClassConstructors" parameter="" weight="0.5" />
<MetricIndex name="NCV" description="number of class variables"
  operation="CountChildNodesByValue"
  parameter="attribute,ownerScope,classifier" weight="0.5" />
<MetricIndex name="NIV"
  description="number of instance variables"
  operation="CountChildNodesByValue"
  parameter="attribute,ownerScope,instance" weight="0.5" />
<MetricIndex name="NOM" description="number of methods"
  operation="CountChildNodesTypes" parameter="operation"
  weight="0.0" />
<MetricIndex name="NOA" description="number of attributes"
  operation="CountChildNodesTypes" parameter="attribute"
  weight="0.0" />
<MetricIndex name="NAPUB"
  description="number of public attributes"
  operation="CountChildNodesByValue"
  parameter="attribute,visibility,public" weight="1.0" />
<MetricIndex name="NAPRI"
  description="number of private attributes"
  operation="CountChildNodesByValue"
  parameter="attribute,visibility,private" weight="1.0" />
<MetricIndex name="NAPRO"
  description="number of protected attributes"
  operation="CountChildNodesByValue"
  parameter="attribute,visibility,protected" weight="1.0" />
<MetricIndex name="NAPAC"
  description="number of package-visible attributes"
  operation="CountChildNodesByValue"
  parameter="attribute,visibility,package" weight="1.0" />
<MetricIndex name="NMPUB"
  description="number of public methods"
  operation="CountChildNodesByValue"
  parameter="operation,visibility,public" weight="1.0" />
<MetricIndex name="NMPRI"
  description="number of private methods"
  operation="CountChildNodesByValue"
  parameter="operation,visibility,private" weight="1.0" />
<MetricIndex name="NMPRO"
  description="number of protected methods"
  operation="CountChildNodesByValue"
  parameter="operation,visibility,protected" weight="1.0" />
<MetricIndex name="NMPAC"
  description="number of package-visible methods"
  operation="CountChildNodesByValue"
  parameter="operation,visibility,package" weight="1.0" />
<MetricIndex name="SUP" description="number of superclasses"
  operation="CountIncomingEdges" parameter="generalization_src"
  weight="1.0" />
```

```
      <MetricIndex name="NOC"
        description="number of children in class hierarchy"
        operation="CountIncomingEdges" parameter="generalization_tgt"
        weight="1.0" />
   </MetricIndices>
   <MetricIndices pass="2">
      <TempIndex name="PPI"
        description="pre-processing for inheritance based metrics"
        operation="CalculateInheritance" parameter="" />
   </MetricIndices>
   <MetricIndices pass="3">
      <MetricIndex name="NMA" description="number of methods added"
        operation="CountMethodsAdded" parameter="" requires="PPI"
        weight="0.5" />
      <MetricIndex name="NMOI"
        description="number of methods overloaded, inherited methods
        are considered" operation="CountMethodsOverloadedInheritance"
        parameter="" requires="PPI" weight="0.5" />
      <MetricIndex name="NMOO"
        description="number of methods overloaded, without considering
        inheritance" operation="CountMethodsOverloaded" parameter=""
        requires="PPI" weight="0.5" />
      <MetricIndex name="NMO"
        description="number of methods overridden"
        operation="CountMethodsOverridden" parameter="" requires="PPI"
        weight="0.5" />
      <MetricIndex name="NMI" description="number of methods inherited"
        operation="CountMethodsInherited" parameter="" requires="PPI"
        weight="0.5" />
      <MetricIndex name="NIA"
        description="number of attributes inherited"
        operation="CountAttributesInherited" parameter=""
        requires="PPI" weight="0.5" />
   </MetricIndices>
   <DictionaryIndices>
      <DictionaryIndex parameter="name" prefix="class:" weight="1.0"
        partWeight="0.5" />
      <DictionaryIndex parameter="methods, name" prefix="method:"
        weight="0.5" partWeight="0.25" />
   </DictionaryIndices>
</ComponentConfiguration>

<ComponentConfiguration nodeType="attribute">
   <MetricIndices pass="1">
      <MetricIndex name="attribute:ALON" description="length of name"
        operation="CountAttributeValueTextLength" parameter="name"
        weight="1.0" />
      <TempIndex name="ADECL"
        description="attribute declaration with type"
        operation="CalculateAttributeDeclaration" parameter="" />
   </MetricIndices>
   <DictionaryIndices>
```

```
      <DictionaryIndex parameter="name" prefix="attribute:"
        weight="1.0" partWeight="0.5" />
      <DictionaryIndex parameter="changeability"
        prefix="changeability:" weight="0.5" partWeight="0.0" />
      <DictionaryIndex parameter="ownerScope" prefix="ownerScope:"
        weight="0.5" partWeight="0.0" />
      <DictionaryIndex parameter="visibility" prefix="visibility:"
        weight="0.5" partWeight="0.0" />
      <DictionaryIndex parameter="typeof, name" prefix="class:"
        weight="0.5" partWeight="0.25" />
   </DictionaryIndices>
</ComponentConfiguration>

<ComponentConfiguration nodeType="operation">
   <MetricIndices pass="1">
      <MetricIndex name="operation:MLON" description="length of name"
        operation="CountAttributeValueTextLength" parameter="name"
        weight="1.0" />
      <MetricIndex name="operation:NOP"
        description="number of parameters"
        operation="CountChildNodesTypes" parameter="parameter"
        weight="0.0" />
      <TempIndex name="MDECL"
        description="method declaration with parameter types"
        operation="CalculateMethodDeclaration" parameter="" />
      <MetricIndex name="operation:LOD"
        description="text length of method declaration"
        operation="CountMethodDeclarationLength" parameter=""
        weight="1.0" />
      <MetricIndex name="operation:NOPint"
        description="number of int-params"
        operation="Count2StepNeighbourNodesByValue"
        parameter="parameters,paramType,name,Integer" weight="1.0" />
      <MetricIndex name="operation:NOPstring"
        description="number of string-params"
        operation="Count2StepNeighbourNodesByValue"
        parameter="parameters,paramType,name,String" weight="1.0" />
      <MetricIndex name="operation:NOPfloat"
        description="number of float-params"
        operation="Count2StepNeighbourNodesByValue"
        parameter="parameters,paramType,name,Float" weight="1.0" />
      <MetricIndex name="operation:NOPother"
        description="number of class-params"
        operation="Count2StepNeighbourTypes"
        parameter="parameters,paramType,class" weight="1.0" />
      <MetricIndex name="operation:NOPin"
        description="number of in-params"
        operation="CountChildNodesByValue"
        parameter="parameter,kind,in" weight="0.5" />
      <MetricIndex name="operation:NOPinout"
        description="number of inout-params"
        operation="CountChildNodesByValue"
```

```
        parameter="parameter,kind,inout" weight="0.5" />
   </MetricIndices>
   <MetricIndices pass="3">
      <MetricIndex name="NMOF"
        description="number of overloads of this method in class"
        operation="CountMethodOverloadFrequency" parameter=""
        requires="PPI" weight="1.0" />
   </MetricIndices>
   <DictionaryIndices>
      <DictionaryIndex parameter="name" prefix="operation:"
        weight="1.0" partWeight="0.5" />
      <DictionaryIndex parameter="visibility" prefix="visibility:"
        weight="0.5" partWeight="0.0" />
      <DictionaryIndex parameter="ownerScope" prefix="ownerScope:"
        weight="0.5" partWeight="0.0" />
      <DictionaryIndex parameter="isAbstract" prefix="isAbstract:"
        weight="0.5" partWeight="0.0" />
      <DictionaryIndex parameter="returns, name" prefix="returns:"
        weight="0.5" partWeight="0.0" />
      <DictionaryIndex parameter="parameters, name" prefix="parameter:"
        weight="0.5" partWeight="0.25" />
   </DictionaryIndices>
</ComponentConfiguration>

<ComponentConfiguration nodeType="parameter">
   <MetricIndices pass="1">
      <MetricIndex name="parameter:LON" description="length of name"
        operation="CountAttributeValueTextLength" parameter="name"
        weight="1.0"/>
   </MetricIndices>
   <DictionaryIndices>
      <DictionaryIndex parameter="name" prefix="parameter:"
        weight="1.0" partWeight="0.5"/>
      <DictionaryIndex parameter="kind" prefix="kind:" weight="0.5"
        partWeight="0.0" />
      <DictionaryIndex parameter="paramType, name" prefix="class:"
        weight="0.5" partWeight="0.25" />
   </DictionaryIndices>
</ComponentConfiguration>

<ComponentConfiguration nodeType="stereotype">
   <MetricIndices pass="1">
      <MetricIndex name="stereotype:LON" description="length of name"
        operation="CountAttributeValueTextLength" parameter="name"
        weight="1.0" />
      <MetricIndex name="stereotype:NOC"
        description="number of classes" operation="CountNeighbourEdges"
        parameter="stereotypes" weight="1.0" />
   </MetricIndices>
   <DictionaryIndices>
      <DictionaryIndex parameter="name" prefix="stereotype:"
        weight="1.0" partWeight="0.5"/>
```

```
      </DictionaryIndices>
   </ComponentConfiguration>

   <ComponentConfiguration nodeType="datatype">
      <MetricIndices pass="1">
         <MetricIndex name="datatype:LON" description="length of name"
           operation="CountAttributeValueTextLength" parameter="name"
           weight="1.0" />
         <MetricIndex name="datatype:NOO"
           description="number of operations"
           operation="CountNeighbourEdges" parameter="returns"
           weight="1.0" />
         <MetricIndex name="datatype:NOP"
           description="number of parameters"
           operation="CountNeighbourEdges" parameter="paramType"
           weight="1.0" />
         <MetricIndex name="datatype:NOA"
           description="number of attributes"
           operation="CountNeighbourEdges" parameter="typeof" weight="1.0"
           />
      </MetricIndices>
      <DictionaryIndices>
         <DictionaryIndex parameter="name" prefix="datatype:" weight="1.0"
           partWeight="0.5" />
      </DictionaryIndices>
   </ComponentConfiguration>
</S3VBottomUp>
```

D Konfigurationsdatei für Simulink-Diagramme (Top-Down Variante)

Nachfolgend ist die Konfigurationsdatei für die Optimierung des Vergleichs von Simulink-Diagrammen mit dem S^3V Baum nach der Top-Down Variante abgedruckt.

```
<S3VTopDown>

   <MainComponent name="Subsystem" />

   <Components>
      <Component name="Subsystem" />
   </Components>

   <TreeConfiguration>
      <Parameters scaled="true" bucketsize="1" neighbours="100"
        threshold="1.0" />
      <Splitter name="S3vDefaultSplitter" />
      <Scanner name="S3vDualRangeScanner" />
      <Weights dictionaryIndex="1.0" dictionaryIndexPart="1.0"
        metricIndex="1.0" />
   </TreeConfiguration>

   <ComponentConfiguration nodeType="Subsystem">
       <MetricIndices pass="1">
           <MetricIndex name="NOB" description="number of blocks"
             operation="CountOutgoingEdges"
             parameter="SystemContainsBlock" weight="1.0"/>
      </MetricIndices>
      <DictionaryIndices>
         <DictionaryIndex parameter="Name" prefix="Subsystem:"
           weight="1.0" partWeight="0.5" />
      </DictionaryIndices>
   </ComponentConfiguration>

   <ComponentConfiguration nodeType="*">
      <MetricIndices pass="1">
          <TempIndex name="ANC"
            description="ancestor according to tree view, nodetype
            specified" operation="GetAncestorNode" parameter="Subsystem"/>
      </MetricIndices>
   </ComponentConfiguration>
```

```
   <SkipList>
   </SkipList>

</S3VTopDown>
```

Literaturverzeichnis

[AVL62] G.M. Adelson-Velskii and E.M. Landis. An algorithm for the organization of information. *Soviet Math. Dokl.*, 3:1259–1262, 1962.

[Bay72] Rudolf Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Inf.*, 1:290–306, 1972.

[Bec04] Dave Beckett. RDF/XML Syntax Specification. Website, 2004. `http://www.w3.org/TR/rdf-syntax-grammar/`. Abrufdatum: 19.02.2007.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.

[BM98] Rudolf Bayer and Volker Markl. The UB-tree: Performance of Multidimensional Range Queries. Technical report, 1998.

[Bur91] Margarete Burkart. Dokumentationssprachen. *Grundlagen der praktischen Information und Dokumentation*, pages 143–182, 1991.

[Chi01] Yves Chiaramella. *Information retrieval and structured documents*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[CKPT92] Douglass R. Cutting, David R. Karger, Jan O. Pedersen, and John W. Tukey. Scatter/Gather: a cluster-based approach to browsing large document collections. In *SIGIR '92: Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 318–329, New York, NY, USA, 1992. ACM Press.

[CST00] Nello Cristianini and John Shawe-Taylor. *An introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, New York, NY, USA, 2000.

[FB74] Raphael A. Finkel and Jon Louis Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4:1–9, 1974.

[Fre60] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[Fuh96] Norbert Fuhr. Information Retrieval. Skriptum zur Vorlesung. Technical report, Universität Dortmund, Fachbereich Informatik, 1996.

[Fuh00] Norbert Fuhr. Probabilistic datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society of Information Science*, 51(2):95–110, 2000.

[Gus97] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA, 1997.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM Press.

[Hen90] Andreas Henrich. *Der LSD-Baum: eine mehrdimensionale Zugriffsstruktur und ihre Einsatzmöglichkeiten in Datenbanksystemen.* PhD thesis, FernUniversität Hagen, 1990.

[HK00] Jiawei Han and Micheline Kamber. *Data mining: concepts and techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[HSW89] A. Henrich, H. W. Six, and P. Widmayer. The LSD tree: spatial access to multidimensional and non-point objects. In *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, pages 45–53, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[Hut07] Hermann Hutter. Nachverfolgbarkeit von Modellelementen in Versionshistorien. Master's thesis, University of Siegen, 2007.

[JMF99] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[Kel03] Udo Kelter. Objektorientierte Modellierung. *Lehrmodul Praktische Informatik*, 2003.

[Kel06] Udo Kelter. Dokumentdifferenzen. *Lehrmodul Praktische Informatik*, 2006.

[KWN05] Udo Kelter, Jürgen Wehren, and Jörg Niere. A Generic Difference Algorithm for UML Models. In *Software Engineering*, pages 105–116, 2005.

[Lan99] Michele Lanza. Combining Metrics and Graphs for Object Oriented Reverse Engineering. Master's thesis, University of Bern, Switzerland, 1999.

[LS98] O. Lassila and R. Swick. Resource Description Framework (RDF) model and syntax specification. Website, 1998. `http://www.w3.org/TR/1998/WD-rdf-syntax-19981008/`. Abrufdatum: 14.02.2007.

[OWK03] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of uml diagrams. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236, New York, NY, USA, 2003. ACM Press.

[Pea01] K. Pearson. On Lines and Planes of Closest Fit to Systems of Points in Space. *Philoshophical Magazine*, 2:559–572, 1901.

[SWY75] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[UML07] Unified Modeling Language: Superstructure (UML 2.1.1 Superstructure Specification, February 2007); OMG. Website, 2007. `http://www.omg.org/docs/formal/07-02-03.pdf`. Abrufdatum: 03.03.2007.

[Weh04] Jürgen Wehren. Ein XMI-basiertes Differenzwerkzeug für UML-Diagramme. Master's thesis, University of Siegen, Germany, 2004.

[WK06] Sven Wenzel and Udo Kelter. Model-Driven Design Pattern Detection Using Difference Calculation. In *Proc. of the 1st International Workshop on Pattern Detection For Reverse Engineering (DPD4RE), co-located with 13th Working Conference on Reverse Engineering (WCRE'06)*, Benevento, Italy, October 2006.