Fehlertolerante Volltextsuche in elektronischen Enzyklopädien und Heuristiken zur Fehlerratenverbesserung

Dissertation zur Erlangung des
naturwissenschaftlichen Doktorgrades
der Bayerischen Julius-Maximilians-Universität Würzburg

vorgelegt von

Wolfram Eßer

aus
Marburg an der Lahn

Würzburg 2005

Eingereicht am: 17. März 2005

bei der Fakultät für Mathematik und Informatik

1. Gutachter: Prof. Dr. J. Albert
2. Gutachter: Prof. Dr. D. Wotschke

Tag der mündlichen Prüfung: 7. Juli 2005

Danksagung

Die vorliegende Arbeit entstand während meiner Beschäftigung als wissenschaftlicher Mitarbeiter am Lehrstuhl für Informatik II der Universität Würzburg. Mein ganz besonderer Dank gilt an dieser Stelle Herrn Prof. Dr. Jürgen Albert für die interessante Themenstellung und die Möglichkeit, die entwickelten Verfahren in zahlreichen herausfordernden Anwendungsfällen praktisch testen zu können. Die regelmäßigen Gespräche und konstruktiven Diskussionen mit ihm haben meine Dissertation in der vorliegenden Form erst möglich gemacht. Auch Herrn Prof. Dr. Detlef Wotschke möchte ich für seine Bereitschaft zur Übernahme des Zweitgutachtens und die freundliche Zusammenarbeit danken.

Den Herren Dr. Thomas Mager und Dr. Walter Reuß vom Springer-Verlag, Herrn Dr. Peter Reuter als Autor des Werkes *Springer Lexikon Medizin* und Herrn Dr. Hans-Günter Schmidt von der Abteilung Handschriften und alte Drucke der Universitätsbibliothek Würzburg möchte ich an dieser Stelle für die vielen interessanten Diskussionen innerhalb der Projektbesprechungen danken. In diesen Besprechungen wurden die Fundamente einiger hervorstechender Merkmale der Anwendungsfälle gelegt und ich konnte oft – auch abseits des Faches Informatik – neue Erkenntnisse gewinnen.

Weiterhin möchte ich auch den studentischen Hilfskräften, Projektpraktikanten, Diplomanden und Kollegen danken, die in irgendeiner Weise zum Entstehen dieser Dissertation bzw. den darin besprochenen Anwendungsfällen beigetragen haben. Besonders hervorzuheben sind in diesem Zusammenhang Herr Alexander Eckl, Herr Dr. Holger Höhn und mein Vater Herr Dr. Wilfried Körner durch deren Durchsicht manche Argumentationsschwäche der vorliegenden Arbeit bemerkt und behoben werden konnte. Herrn German Tischler möchte ich an dieser Stelle für die Implementierung des Filter-Graphen danken und Frau Almut Kirmse konnte durch ihre sorgfältige Endkontrolle die Qualität der Arbeit im Hinblick auf Tippfehler signifikant heben, wofür ihr ebenfalls mein Dank gebührt.

Ein weiterer Dank muss an die vielen (meist ehrenamtlichen) Entwickler der zahlreichen freien Open-Source-Projekte gehen, deren Tools maßgeblich zur Realisierung der vorliegenden Arbeit beigetragen haben. Exemplarisch seien hier der (Programmier-) Editor vim, die Programmierumgebung eclipse, die Skriptsprache perl und die Bürosoftware OpenOffice genannt.

Abschließend möchte ich mich noch bei meiner Mutter Gudrun, meiner lieben Frau Dorothée und unseren Töchtern Lisa und Caroline für die Liebe, die moralische Unterstützung und das Verständnis – speziell in den letzten besonders arbeitsreichen Monaten – bedanken. Ohne euch wäre diese Arbeit ebenfalls nicht möglich gewesen.

Würzburg, im März 2005 *Wolfram Eßer*

„Fehler sind nützlich,
aber nur, wenn man sie schnell findet.“

John Maynard Keynes
(1883-1946)
Britischer Mathematiker und Ökonom

* * *

„Wer einen Fehler begangen hat
und ihn nicht korrigiert,
begeht einen weiteren Fehler.“

Konfuzius
(551 v.Chr. - 479 v.Chr.)
Chinesischer Philosoph

Inhaltsverzeichnis

Kapitel 1

Einführung

Ausgelöst durch die enormen Fortschritte der Informationstechnologie durchläuft unsere Gesellschaft zurzeit den Wandel hin zur Informations- und Wissensgesellschaft. Speziell mit modernen Techniken der Informationsgewinnung werden Wissensmengen beherrschbar, die früher gar nicht oder nur aufwändig verwaltet und genutzt werden konnten – es sei hier nur auf das „World Wide Web“ als die wohl größte Wissenssammlung der Menschheit und die entsprechenden WWW-Suchmaschinen verwiesen. Da der weitaus größte Teil des menschlichen Wissens in Texten gespeichert sein dürfte, ist das Feld der Informationsbeschaffung (engl. *Information Retrieval*) aus Texten ein besonders interessantes Gebiet – unter anderem, da sich neue Techniken und Entwicklungen hier unmittelbar in erhöhter Nutzbarkeit dieser Quellen auswirken.

Information Retrieval geht in seiner grundsätzlichen Form zurück u.a. auf griechische und römische Schreiber, die – auch ohne Informationstechnologie – größere Werke zunächst neben alphabetischer Ordnung und Inhaltsverzeichnissen auch mit Indexbereichen versahen, um das Auffinden von Artikeln zu erleichtern. Vergleiche hierzu z.B. die älteste komplett überlieferte Enzyklopädie „*Naturalis Historia*“ in 37 Bänden von Plinius dem Älteren (etwa 23 – 79 n.Chr.).

Den größten – und bis heute spürbaren – Schub bekam die Disziplin des Information Retrieval allerdings erst mit Beginn des 20. Jahrhunderts und der Einführung von zunächst mechanischen (Hollerith u.a.) und später elektronischen Rechenmaschinen, denen bereits in den 1950er Jahren erste Untersuchungen zur Nutzung von Computern zur Speicherung und Wiederauffindung von Information folgten (z.B. [TW58]). Mit dem immer noch ungebrochenen Siegeszug des WWW geht die weiterhin zunehmende Verbreitung von PCs an Arbeitsplätzen und in privaten Haushalten einher. So sind der aktuell zum vierten Mal im Rahmen der Initiative D21 von tns-emnid durchgeführten Studie [wwwEmn04] zu-

folge zurzeit bereits ca. 33,9 Millionen Deutsche (53% der Gesamtbevölkerung bzw. 80% der bis zu 29-jährigen) „online“, was das Bedürfnis der industrialisierten Kulturkreise nach aktueller, schnell verfügbarer Information nur weiter unterstreicht.

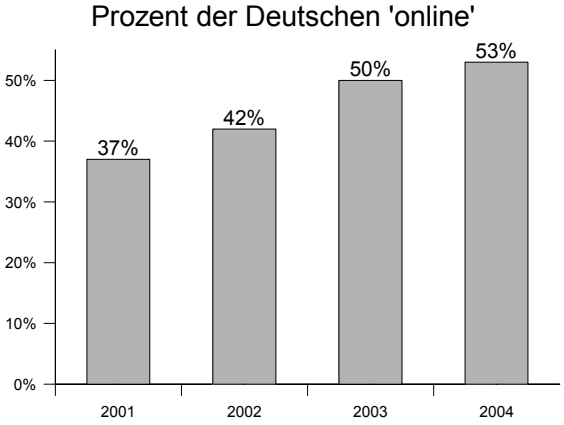


Abb. 1.1. tns-emnid Studie (N)ONLINE-Atlas

In der 2003 vom Allensbacher Institut für Demoskopie durchgeführten Studie „*Allensbacher Computer- und Technik-Analyse 2003*“ [ACTA03] zählten 71% der 10.424 befragten Internet-Nutzer (Alter zwischen 16 und 64 Jahre) Online-Nachschlagewerke zu den beliebtesten Internet-Angeboten. Damit belegt diese Nutzungsart hinter E-Mail und Reiseinformationen den dritten Platz.

Tab. 1.1. Nutzung von Internet-Angeboten – Top-Ten (Quelle: [ACTA03])

Rang	***Prozent***	***Inhalt***
1	90 %	E-Mail
2	72 %	Informationen über Reiseziele
3	**71 %**	**Nachschlagewerke**
4	70 %	Informationen für Schule, Ausbildung, Beruf
5	69 %	Wetterbericht, Reisewetter
6	69 %	Veranstaltungshinweise
7	67 %	Fahr- und Flugpläne
8	67 %	Produktinformationen, Preisvergleiche
9	63 %	Herunterladen von Software
10	63 %	Aktuelle Nachrichten zur Politik

Trotzdem kann das WWW als Informationsquelle nicht alle Bedürfnisse befriedigen. So ist einer der Vorteile des WWW, nämlich die Millionen von Informationserstellern und -anbietern, welche die Inhalte auf den Webseiten pflegen, zugleich auch einer der Hauptnachteile dieses Mediums. So leidet die globale Informationsquelle beispielsweise an so genannter „Linkverrottung“. Dabei handelt es sich um Hyperlinks, deren Ziele nicht mehr existieren, und jüngere Untersuchungen sprechen hier von 50% „Links ohne Ziel“ nach 2 Jahren [Koeh01] bzw. 4 Jahren [Spin03]. Weitere Probleme sind z.B. Aktualität, Richtigkeit und

Vollständigkeit der Information und unklare Haftung bei eventuellen Schäden, die durch falsche Inhalte (z.B. Therapievorschläge) verursacht werden.

So werden beim Bedürfnis nach fundierter, aktueller, wissenschaftlicher Information neben dem klassischen Buch zunehmend digitale Nachschlagewerke am PC genutzt. Ein PC-Nachschlagewerk hat auf diesem Gebiet einige hervorstechende Vorteile gegenüber einem gedruckten Werk: Neben der großen Informationsmenge, die auf wenig Platz untergebracht werden kann, seien hier vor allem die Möglichkeiten genannt, multimediale Inhalte darzubieten (Filme, Bilder, Tondokumente) und die Information mit geeigneten Techniken in vielfacher Weise zugänglich zu machen. Das Speichermedium DVD fasst den unkomprimierten Textinhalt von ca. 1 Million Schreibmaschinen-Seiten – ein Stapel von fast 100 Metern Höhe. Zu Navigationsmerkmalen wie Hyperlinks, Speedtyping und mehrfachen Listenindizes über derselben Informationsmenge sei hier vor allem die Möglichkeit genannt, große Textmengen schnell zu durchsuchen.

Die vorliegende Arbeit beschäftigt sich mit der Konzeption, Implementierung und Analyse von Techniken, die alle darauf abzielen, große wissenschaftliche Textkorpora für deren Benutzer schnell und fehlertolerant durchsuchbar und damit als Wissensquelle nutzbar zu machen. Dabei trägt das Arbeitsgebiet der Fehlerratenverbesserung und Konsistenzerhöhung des Textmaterials untrennbar zu diesem Ziel bei. Denn konsistente Schreibweisen und fehlerfreiere Texte erhöhen die Chancen, Wissen in diesen Texten wiederzufinden. So gliedert sich die vorliegende Arbeit in eben diese zwei Teilbereiche: Fehlertolerante Volltextsuche in elektronischen Enzyklopädien und Heuristiken zur Fehlerratenverbesserung bei redaktioneller Dokumentenverarbeitung.

1.1 Motivation

Soll der Textkorpus von elektronischen Enzyklopädien von einem Benutzer nach einem bestimmten Suchmuster (synonym auch: Suchpattern) durchsucht werden, ergeben sich mehrere Probleme, welche das Ermitteln einer möglichst korrekten Treffermenge erschweren. Verfahren zur Lösung dieser Probleme wurden in einer Kooperation mit dem Springer-Verlag (Heidelberg) bei der Realisierung der elektronischen Version der 12-bändigen Enzyklopädie „*Hagers Handbuch der Pharmazeutischen Praxis*“ (siehe [Bru+00]) entwickelt und erprobt. Einige Beispiele aus dem Kontext dieses Werkes sollen die vorhandenen Probleme aufzeigen. Wenn nicht anders angegeben stehen in Klammern die Häufigkeiten des jeweiligen Suchmusters im Hager-Textkorpus.

- Eine große Anzahl von beteiligten Autoren, wie man sie bei der Erstellung umfangreicher Werke oft benötigt, kann dazu führen, dass unterschiedliche

korrekte Bezeichnungen für identische Fachbegriffe verwendet werden:
`Kalzium` (42) , `Calcium` (3750)

- Tippfehler und Wortvarianten erschweren das Auffinden von Treffern:
`Kronblätter` (600), `Kronbläter` (1), `kronblättrig` (2)
- Abkürzungen (teilweise ohne terminierenden Punkt) werden bei steigender Autorenzahl in unterschiedlicher Form benutzt:
`Prüflösung` (248), `Prüflsg.` (2048), `Prüflsg` (2048+14=2062)
- Zusammenschreibung, Bindestrich-Verwendung und getrennte Schreibweise ist speziell im komposita-lastigen Deutschen problematisch:
`Ethylester` (993), `Ethyl-Ester` (7), `Ethyl Ester` (10)
- Studenten eines Faches und interessierte Laien kennen oft eher die Aussprache statt die korrekte Schreibweise eines Fachausdrucks. Die gezeigten Beispiele wurden alle von Nutzern der WWW-Version von [AB02] gesucht:
`Dishydrose` (0), im Werk vorhanden: `Dyshidrose` (19)
`Nävuszelnävus` (0), im Werk vorhanden: `Naevuszellnaevus` (122)
`Suggilationen` (0), im Werk vorhanden: `Sugillationen` (9)
- Neue deutsche Rechtschreibung: Wie die im Sommer 2004 stattgefundenen erneuten, heftigen Diskussionen [wwwSpie04] über Nachbesserungen an der 1996 beschlossenen deutschen Rechtschreibreform zeigen, sind Schreiber und Leser seit Einführung der Reform trotz (oder gerade wegen) der langen Übergangsfristen noch immer tief verunsichert. Auch bei der Recherche in elektronischen Texten muss der Benutzer nun wissen, ob der durchsuchte Text nach alter oder neuer Rechtschreibung verfasst wurde:
neu: `Darmverschluss` (0), alt: `Darmverschluß` (26)

Aus der oben angegebenen Liste von Punkten ergibt sich die Notwendigkeit, dass ein Suchsystem für umfangreiche, enzyklopädische Texte fehlertolerant einerseits in Richtung des Werkes (Fehler und Inkonsistenzen im Text) und andererseits auch fehlertolerant in Richtung des Benutzers sein sollte, falls dieser eine falsche oder im Werk nicht verwendete Schreibweise eines Begriffes sucht (Fehler im Suchmuster).

1.2 Vorstellung der Anwendungsfälle

Im vorangegangenen Abschnitt wurde bereits darauf hingewiesen, dass die praktische Erprobung der in dieser Arbeit vorgestellten Konzepte und Implementierungen im Rahmen eines Kooperationsprojektes mit dem Springer-Verlag stattfand. Ausgehend von der elektronischen Version von „*Hagers Handbuch*“ wurden

diese Techniken jedoch auch noch in zahlreichen anderen kommerziellen Folgeprojekten verwendet, wodurch der Nachweis der Praxistauglichkeit weiter erhärtet werden konnte.

Die einzelnen Werke werden in der folgenden Auflistung kurz mit ihren Besonderheiten vorgestellt, da im Verlauf dieser Arbeit auf diese Bezug genommen wird. Für eine detailliertere Darstellung der einzelnen Werke und Experimente mit den jeweiligen Texten sei auf Kapitel 9 „Anwendungsfälle“ verwiesen.

Begriffsklärung. In dieser Arbeit stehen die Einheiten zur Beschreibung von Speicherplatzbedarf **Kilobyte** (**KB**), **Megabyte** (**MB**) und **Gigabyte** (**GB**) grundsätzlich für die nicht-metrischen Varianten. Ein Kilobyte sind also nicht 10^3=1000 Byte, sondern:

1KB = 2^{10} Byte = 1024 Byte entsprechend

1MB = 2^{20} Byte = 1024 KB = 1024*1024 Byte = 1.048.576 Byte usw.

Auf die hierfür von der IEC im Jahr 2000 eingeführte und genormte SI-Schreibweise (**KiB**, **MiB**, **GiB**, ...) wird zugunsten besserer Lesbarkeit in dieser Arbeit verzichtet, da die SI-Schreibweise (noch) nicht verbreitet genug scheint (vgl. [IEC00]).

- **HagerROM** – *Hagers Handbuch der Pharmazeutischen Praxis* [Bru+00]: Mit einem Umfang von über 12.000 Seiten in der Druckversion, die 121 MB Speicherbedarf im XML-Format benötigt, welche (nach Entfernen von Layout-Informationen) ca. 62,5 MB reinen Text erzeugt, handelt es sich um das umfangreichste der hier vorgestellten Werke. Hagers Handbuch wurde 2001 erstmals als elektronisches Nachschlagewerk auf CD-ROM unter dem Namen *HagerROM* [BEH+01] vom Springer-Verlag veröffentlicht und seitdem in jährlichem Zyklus erweitert, überarbeitet und neu aufgelegt [BEH+04]. An der Erstellung der Monographie-Texte haben über 600 Autoren mitgewirkt.
- **Altmeyer** – *Springer Enzyklopädie Dermatologie, Allergologie, Umweltmedizin* [AB02]: Mit 1.800 Seiten und einem Schwerpunkt auf dermatologischen Themen wurde nach Drucklegung der Printversion aus den Texten dieses Werkes eine Online-Version erstellt (`http://www.galderma.de`), die über die Webseite des Pharmaunternehmens Galderma seit März 2003 frei für Ärzte, Medizin-Studenten und Apotheker im WWW verfügbar ist, nachdem die Nutzer sich kostenlos via DocCheck authentifiziert haben.
- **Reuter** – *Springer Lexikon Medizin* [Reut04]: Die 2004 erschienene erste Auflage dieses über 2380 Seiten umfassenden medizinischen Nachschlagewerkes steht in direkter Konkurrenz zu etablierten Medizin-Lexika wie z.B. dem *Pschyrembel Klinisches Wörterbuch* oder dem *Roche Lexikon Medizin*. Die elektronische Version des *Springer Lexikon Medizin* erschien 2004 als PC-DVD.

- **Parasitology** – *Parasitology Research & Encyclopedic Reference of Parasitology* [CM04], [Meh01]: Diese beiden englisch-sprachigen Werke, welche ebenfalls im Springer-Verlag erschienen sind, wurden online mit Querverweisen verwoben. Dabei deckt die Enzyklopädie das gesicherte Wissen auf dem Gebiet der Parasitologie ab und die Zeitschrift *Parasitology Research* stellt den jeweils aktuellsten Stand der Forschung dar. Mit dem so entstandenen Gesamtwerk wurde die Anwendbarkeit der in dieser Arbeit entwickelten Techniken auf englische Texte untersucht.
- **Fries-Chronik** – *Histori, Namen, Geschlecht, Wesen, Thaten, gantz leben und sterben der gewesnen Bischoffen zu Wirtzburg und Hertzogen in Franken, Auch was bey einem Jeden in Zeit seiner Regirung, sunderlich gehandelt worden, ergangen vnd beschehen ist* [Frie1582]: Die so genannte *Fries-Chronik* wurde ursprünglich verfasst von Magister Lorenz Fries (1489-1550) und 1574-1582 dann als Prachthandschrift für den Würzburger Fürstbischof Julius Echter (1545-1617) angefertigt. Das 2-bändige, großformatige Werk umfasst ca. 1200 Seiten, welche mit zahlreichen aufwändig colorierten Farbminiaturen illustriert sind. Die Handschrift befindet sich im Besitz der Universitätsbibliothek Würzburg und die 300 interessantesten Seiten wurden im Jahr 2004 zum 1300-jährigen Stadtjubiläum der Stadt Würzburg als Multimedia-DVD mit zoombaren Faksimile-Scans, virtuellen 3D-Buchansichten und durchsuchbarem Volltext herausgegeben.
  Bei diesem Werk wurde untersucht, inwiefern man auch mittelalterliche, frühneuhochdeutsche (FNHD) Texte fehlertolerant durchsuchen kann. Hier entstehen Probleme beim Information Retrieval vor allem dadurch, dass eine normierte, einheitliche Schreibweise im 16. Jahrhundert noch nicht entwickelt war und dass die damals übliche Schreibweise eines Wortes von der heutigen Schreibweise teilweise stark abweichen kann.

Tabelle 1.2 gibt einen Überblick über die wichtigsten Charakteristika der vorgestellten Anwendungsfälle.

Tab. 1.2. Wichtige Merkmale der betrachteten Anwendungsfälle

Werk	***Domäne***	***Sprache***	***Medium***	***Rohtext***
HagerROM	Pharmazie/Chemie/Biologie/Medizin	deutsch	CD	62,5 MB
Altmeyer	Medizin/Dermatologie	deutsch	online	8,2 MB
Reuter	Medizin	deutsch	DVD	18,3 MB
Parasitology	Biologie/Medizin	englisch	online	36,5 MB
Fries-Chronik	Geschichte	FNHD	DVD	1,6 MB

Da eine der wichtigsten Kenngrößen eines Textkorpus bezogen auf eine schnelle Textsuche die Größe des Textes ist, werden die betrachteten Werke hier nun bezüglich des Speicherplatzbedarfs ihres Rohtextes (also Text ohne Layout-Information) grafisch aufgetragen. Der Platzbedarf von 3,97 MB des Rohtextes der „*Luther-Bibel*“ [wwwBibel] (altes und neues Testament) wird zur besseren Vergleichbarkeit ebenfalls mit aufgenommen.

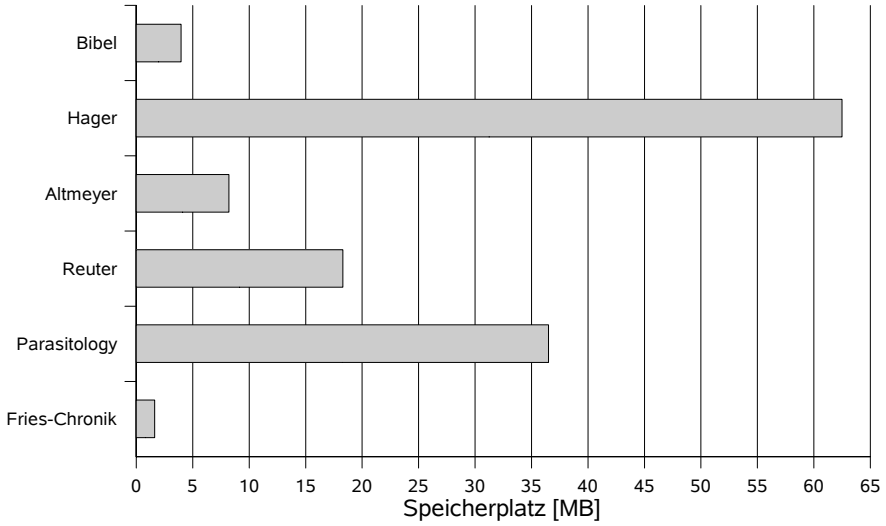


Abb. 1.2. Rohtext-Speicherplatzbedarf der Werke

Die multimediale Lehr- und Lernumgebung DEJAVU (vgl. dazu [Höh02] und [GKHE+03]) und ein für die Firma Audi entwickeltes Informationssystem (vgl. dazu [Behl04]), welche ebenfalls das hier vorgestellte fehlertolerante Volltextsuche-System verwenden, wurden aus Platzgründen nicht in die Liste der vorgestellten Anwendungsfälle aufgenommen. Für weitergehende Informationen zu den speziellen Merkmalen von DEJAVU bezüglich der Integration einer toleranten Textsuche sei auf die Artikel [Ess03] und [EH04] verwiesen.

1.3 Wesentliche Ergebnisse

Die folgende Arbeit stellt Konzepte und deren praktische Umsetzung vor, die es ermöglicht haben, zahlreiche kommerzielle und darunter einige umfangreichere Nachschlagewerke (Enzyklopädien, Lexika, Chroniken) schnell, fehlertolerant und auf das jeweilige Werk angepasst nach Suchmustern zu durchsuchen. Dabei ist genau die Möglichkeit der Anpassung an den jeweiligen Textkorpus eine der herausragenden Stärken des vorgestellten Ansatzes, der ihn von sonst üblicherweise verwendeten Techniken zur fehlertoleranten Volltextsuche unterscheidet.

Durch die im vorhergehenden Abschnitt bereits vorgestellten unterschiedlichen Charakteristika der untersuchten Anwendungsfälle kann gezeigt werden, dass die Adaption auf andere Sprachen (Englisch), Dialekte (Frühneuhochdeutsch) und stark unterschiedliche Textdomänen (Pharmazie, Medizin, Geschichte) leicht möglich ist. Die Anwendung innerhalb der *HagerROM* mit einem ungefähr 16-fachen Umfang des Textes der *Bibel* (siehe vorheriger Abschnitt) hat gezeigt, dass die verwendeten Ansätze auch mit größeren Textmengen gut skalieren.

Da Fehlertoleranz aufgrund erhöhter Laufzeit und größerer Unschärfe der Treffer (verglichen mit einer exakten nicht-fehlertoleranten Suche) immer ein durch den Nutzer zu aktivierendes, optionales Feature sein sollte, wurden zur Verbesserung der exakten Suche zusätzlich diverse Techniken zur Vorab-Korrektur von Tippfehlern und zur Vermeidung von Schreibweisen-Inkonsistenzen untersucht. Dabei stellte sich heraus, dass Fehlertoleranz auch zur Verbesserung von Fehlerraten während der Vorverarbeitung innerhalb des redaktionellen Umfeldes ein hilfreiches Mittel ist. Durch die in dieser Arbeit vorgestellten Heuristiken und Werkzeuge war es möglich, mit vertretbarem menschlichen Überprüfungsaufwand große Mengen von Fehlern aus bereits mehrfach lektorierten Lexika-Texten zu eliminieren.

Nicht immer steht jedoch bei digitalen Nachschlagewerken der Textkorpus im Vordergrund. Um z.B. bei historischen handgeschriebenen Werken dem Anwender das genaue Quellenstudium zu ermöglichen, müssen in solchen Fällen auch die Faksimile-Scans der originalen Buchseiten zur Anzeige vorgehalten werden. Gibt es mehrere Versionen der Buchseiten – z.B. mehrfache Abschriften oder durch Schriftart-Wechsel bedingte unterschiedliche Präsentationsformen (Schriftarten) – bietet sich eine durch den Anwender steuerbare, weiche, grafische Überblendung der vorhandenen Ansichten an. Solch eine Überblendung wurde bei der Fries-DVD realisiert, um das Lesen der für Laien teilweise schwer zu erkennenden Handschrift zu trainieren. Aber auch im redaktionellen Umfeld kann diese Technik verwendet werden, um Unterschiede, die durch Korrekturzyklen entstehen, auf diese Weise sichtbar zu machen.

Die Fries-DVD bot ferner die Herausforderung, Verfahren zu entwickeln, mit denen es möglich sein sollte, den Vorgang des Durchblätterns dieser alten Handschrift am PC virtuell nachzuempfinden. Dazu wurden Techniken des 3D-Modellierens, des Raytracing-Film-Renderns und der Filmformat-Konvertierung so weit automatisiert, dass die zeitaufwändige rechnergesteuerte Generierung der fast 200 Video-Elemente vom PC autark – und damit robust gegen Fehler – durchgeführt werden konnte. Dieser letzte Punkt trägt also zur Fehlerratenverbesserung auf redaktioneller Seite und auf Seite der Software-Entwickler bei, was die Schilderung der besagten Verfahrensweise im Rahmen der vorliegenden Ausarbeitung motiviert.

1.4 Gliederung der Arbeit

Nachdem **Kapitel 1** bereits die Motivation und die wesentlichen Ergebnisse der vorliegenden Arbeit kurz dargestellt und die verschiedenen praktischen Anwendungsfälle vorgestellt hat, wird im Folgenden ein Überblick über die weiteren Kapitel und Abschnitte dieser Ausarbeitung gegeben.

Kapitel 2 beschreibt zunächst die Aufgabenstellung genauer, teilweise mithilfe einiger grundlegender Definitionen. Dabei erfolgt wieder die separate Behandlung von fehlertoleranter Volltextsuche und Fehlerratenverbesserung während der Dokumentenverarbeitung. Auf die in diesem Kapitel enthaltenen Definitionen stützt sich die nachfolgende Ausarbeitung maßgeblich ab.

Ein zentraler Punkt bei der Implementierung eines schnellen Volltextsuche-Systems – unabhängig von eventuell vorhandener Fehlertoleranz – ist die zugrundeliegende Index-Datenstruktur, die das schnelle Ermitteln der Fundstellen zu einem Muster ermöglicht. In **Kapitel 3** werden daher übliche Datenstrukturen für Volltextsuchen vorgestellt, verglichen und bewertet.

Das Bedürfnis nach fehlertolerantem Suchen in Textbeständen hat eine lange Tradition und zahlreiche Ansätze wurden in der Vergangenheit auf diesem Gebiet untersucht. **Kapitel 4** gibt einen Überblick über verschiedene bekannte Ansätze zum fehlertoleranten Information Retrieval, bewertet diese und zeigt Probleme der etablierten Vorgehensweisen auf.

In **Kapitel 5** wird der neu entwickelte Ansatz des „*Weighted Pattern Morphing*“ (WPM) detailliert dargestellt und untersucht. Mit WPM ist es möglich, eine existierende Volltextsuche mit einer optionalen Fehlertoleranz zu versehen, welche – mit erhöhter Laufzeit – sogar vollständig unabhängig von der Struktur des verwendeten Volltextsuche-Index ist. Zur Laufzeit-Verkürzung kann man dem WPM-Algorithmus Informationen über die im Text vorhandenen Zeichenketten geben, welche dann in einem geeigneten zusätzlichen Index abgelegt werden müssen. Kapitel 5 beschreibt weiter, wie man die vom WMP-Algorithmus verwendeten Morph-Regeln gewinnen und die benötigten Regelgewichte automatisch einstellen kann. Eine Analyse von Precision und Recall, welche WPM mit der sonst oft verwendeten Edit-Distanz vergleicht, schließt dieses Kapitel ab.

Da eine optionale Fehlertoleranz ohne zugrundeliegendes Volltextsuche-System für einen Benutzer nicht sinnvoll verwendbar ist, wird in **Kapitel 6** das in den oben beschriebenen kommerziellen Anwendungsfällen zum Einsatz gekommene Volltextsuche-Backend vorgestellt. Dieses wurde in seiner Grundform in der Diplomarbeit von Martin Grimm [Gri01] erarbeitet. Mittlerweile wurde es im Vergleich zur dort beschriebenen Fassung um einige Funktionen erweitert, die in Kapitel 6 detaillierter vorgestellt werden.

Kapitel 7 beschreibt die Konzepte und Implementierungen, mit denen *vor* der Veröffentlichung der Werke Tippfehler und Inkonsistenzen aus den Werktexten eliminiert werden können. Falsch oder inkonsistent geschriebene Worte verhindern oder erschweren, dass eine exakte (nicht-fehlertolerante) Suche eine möglichst umfassende Menge an relevanten Treffern ermittelt. Da in komplexen wissenschaftlichen Werken voll-automatische Korrekturen äußerst riskant sein können, wird hier ein hybrider Ansatz beschrieben, bei welchem dem Fachlektor eine möglichst kurze Liste von potenziell falschen Worten präsentiert wird, die dieser mit einer geeigneten grafischen Oberfläche in falsche und richtige Worte separieren kann.

Da nicht nur bei großen Textwerken, sondern auch bei großen Software-Projekten Fehler gemacht werden können, werden in **Kapitel 8** die Maßnahmen vorgestellt, die aus Sicht des Software-Entwicklers zur Qualitätsverbesserung in den entwickelten Programmen und Skripten beitragen. Es wird dabei teilweise auf Methoden zurückgegriffen, die sich im Umfeld der Extreme-Programming-Strömung entwickelt haben (z.B. Unit-Testing [wwwXP]). Zur Aufgabe der Software-Entwickler bei der Realisierung elektronischer Nachschlagewerke gehört jedoch nicht nur die Verarbeitung der Werktexte, sondern (abhängig vom Werktypus) auch die robuste und konsistente Verarbeitung einer Vielzahl von anderen Datenarten (Bilder und Animationen seien hier als Beispiel genannt). Die Beschreibung des Ablaufes der automatisierten (und dadurch fehlerresistenten) Generierung von 3D-Animationen für ein virtuelles Buch, als Bestandteil eines umfangreichen multimedialen DVD-Projektes, schließen dieses Kapitel ab.

Kapitel 9 stellt die in Kapitel 1 bereits kurz eingeführten Anwendungsfälle für die vorgestellten Ansätze noch einmal detaillierter dar. Anhand von Experimenten wird gezeigt, wie sich die Algorithmen an die jeweiligen Werkspezifika anpassen lassen.

Abschließend werden in **Kapitel 10** die wichtigsten Punkte dieser Arbeit über die bereits gemachten Aussagen von Abschnitt 1.3 hinaus noch einmal vertieft, zusammengefasst und bewertet.

Im **Anhang** ab Seite 209 werden zu einigen aufwändigeren Experimenten ausführlichere Ergebnisse präsentiert, als dies im Hauptteil der Arbeit sinnvoll ist. Weiter finden sich dort Details zu den verwendeten Datenformaten und Steuertabellen. Im Text dieser Ausarbeitung wird auf den jeweils zugehörigen Teil im Anhang verwiesen.

Kapitel 2

Aufgabenstellung & Definitionen

Im nun folgenden Kapitel wird die Aufgabenstellung der Arbeit genauer spezifiziert. Dabei ist das Kapitel nicht als eine Art Pflichtenheft zu verstehen, in dem detailliert alle abzuarbeitenden Punkte angegeben werden. Vielmehr erläutert dieses Kapitel die wesentlichen Bausteine der Arbeit und definiert dabei, wo nötig, Begriffe, die zum verbesserten Verständnis der nachfolgenden Kapitel beitragen.

Wie aus Abschnitt 1.2 ersichtlich ist, wurden die hier vorgestellten Konzepte im Wesentlichen für enzyklopädische Werke (zu denen hier auch Lexika gezählt werden) entwickelt.

> Als **Enzyklopädie** bezeichnet man den Versuch der vollständigen Darstellung des Wissens zu einem bestimmten Thema. Der Begriff Enzyklopädie leitet sich ab von frz. encyclopédie zu lat. encyclopaedia aus altgr. εγκυκλιος und παιδεια, einer unkorrekten Zusammensetzung aus egkýklios, "kreisförmig", "im Kreise herumgehend" und paideía, "Lehre", "Bildung" zu egkyklopaideia, "Grundlehre aller Wissenschaften und Künste", "Wissenschaftskunde". [Wikipedia.de, „Enzyklopädie"]

Enzyklopädien gemeinsam ist also, bedingt durch den Wunsch der vollständigen Darstellung des Wissens zu einem Gebiet, immer der, verglichen mit üblichen Lehrbüchern, große Umfang des Textkorpus. Da die hier entwickelten Techniken teilweise direkt mit dem Endnutzer eines Nachschlagewerkes in Berührung kommen sollen, müssen die Algorithmen also neben den rein funktionalen Anforderungen vor allem trotz des großen Textumfanges die Ergebnisse möglichst schnell präsentieren können.

Im klassischen Information Retrieval wird zwischen zwei großen Klassen von Suchalgorithmen unterschieden: *online* und *offline*. Offline-Algorithmen müssen

vorab Kenntnis über den zu durchsuchenden Text haben und legen für die späteren Suchen geeignete Indexstrukturen an. Ändert sich der zu durchsuchende Text, muss in der Regel auch der Index angepasst werden. Dagegen muss eine Online-Suche vorab keine Kenntnis des zu durchsuchenden Textes haben und sucht daher das gewünschte Suchmuster meist deutlich langsamer.

Da die hier vorgestellte Arbeit sich schwerpunktmäßig mit dem wiederholten Suchen und schnellen Finden von kurzen Suchmustern in vergleichsweise umfangreichen, eher statischen Textmengen beschäftigt, ist es zwingend nötig, Informationen über den Werktext durch Vorverarbeitungsschritte in geeigneten Indexstrukturen abzulegen. Wir betrachten also in dieser Arbeit ausschließlich die erste Klasse von index-basierten Offline-Algorithmen und es wird daher nicht weiter auf obige Online-Verfahren eingegangen. Für eine detaillierte Darstellung verschiedener Online-Suchalgorithmen vergleiche [Ste94].

In dieser Arbeit werden die unterschiedlichen Varianten der Suche jedoch wie folgt unterschieden:

Definition 2-1. Lokale Suche ist eine Suche, bei der Präsentationsoberfläche und zentraler Suchalgorithmus auf demselben physikalischen Rechner ausgeführt werden. Der Rechner muss für die Durchführung der Volltextsuche nicht mit dem Internet verbunden sein.
WWW-Suche bezeichnet im Gegensatz dazu die Suche in einem WWW-basierten Werk, bei dem die Darstellung der Daten üblicherweise mit einem Webbrowser geschieht und die Suche auf einem anderen Rechner (z.B. dem WWW-Server) abläuft, ohne dass der Endnutzer Zugriff auf die verwendeten Suchindizes hat.

2.1 Exakte Volltextsuche

Bevor im folgenden Abschnitt auf die fehlertolerante Volltextsuche eingegangen wird, wird zunächst die Begriffsbestimmung einer exakten Volltextsuche unternommen, auf welche die fehlertolerante Variante aufbaut.

In der Literatur werden oft Textsuche-Algorithmen beschrieben, die in einem Vorverarbeitungsschritt die Zerlegung des Textes in einzelne, linguistische Worte verlangen, worauf über diese Worte und ihre Auftretensstellen ein wortbasierter Index angelegt wird (z.B. mg-System [WMB94]). Sehr häufige Worte (so genannte *Stoppworte*) werden in diesen Index meist nicht aufgenommen, da sie die Indexgröße stark anwachsen lassen, wobei jedoch angenommen wird, dass diese Worte aufgrund ihrer Häufigkeit weniger Informationsgehalt haben. Deutsche Stoppworte sind zum Beispiel: `der`, `die`, `das`, `und`, `oder`, `eine`, `nicht`; Beispiele für englische Stoppworte sind: `the`, `and`, `he`, `she`.

> „**Wort** bezeichnet eine grammatische Einheit oder eine inhaltliche Einheit [...]. Muttersprachler haben ein intuitives Verständnis davon, was in ihrer Sprache ein Wort ausmacht, die Sprachwissenschaft aber tut sich damit schwer, allgemeingültige Kriterien zur Abgrenzung von Wörtern zu finden.“ [Wikipedia.de, „Wort“]

Im Gegensatz zu zeichenketten-basierten Textsuchen, wie sie in dieser Arbeit betrachtet werden, wird bei wortweisen Textsuchen das vom Benutzer eingegebene Suchmuster ebenfalls in Worte zerlegt und dann im vorher aufgebauten Wortindex die passenden Fundstellen ermittelt. Dieses an linguistischen Worten orientierte Vorgehen hat mehrere Nachteile:

1.) Wie aus obigem Zitat zum Begriff *Wort* ersichtlich ist, ist die Auftrennung eines Textes in Worte schon alleine deshalb schwierig, da es keine klare Definition von Wort und damit keine klare Definition von *Wortgrenze* gibt. Dies wird umso schwieriger, wenn man nur aufgrund des Zeichenstroms entscheiden soll, ob eine Stelle einen Worttrenner darstellt oder nicht.
Am Beispiel des Bindestrichs „-“ soll das Problem verdeutlicht werden: Nimmt man den Bindestrich nicht zu den Wortbegrenzern hinzu, ist z.B. `darm-blasen-fistel` im Text ein zusammenhängendes Wort, welches in den Index aufgenommen wird. Eine Suche nach `blase` oder `blasen-fistel` würde hier keinen Treffer erzeugen.
Nimmt man den Bindestrich jedoch zu den Worttrennern hinzu, würden also für das Wort `darm-blasen-fistel` drei Worte in den Wortindex des Textes aufgenommen: `darm`, `blasen` und `fistel`. In diesem Fall würde weder eine Suche nach `blase`, noch eine Suche nach `blasen-fistel` Treffer produzieren.

2.) Die deutsche Sprache, mit ihrer Tendenz ständig neue Worte (so genannte *Komposita*) zu bilden, erschwert das Problem weiter. Würde im Text das Wort `sauerstoffzelt` auftauchen, würde dieses als ein Wort in den Index aufgenommen. Eine Suche nach `sauerstoff` würde an dieser Stelle keinen Treffer produzieren, da Suchmuster und Indexwort nicht identisch sind. Bei einer wortweisen Suche sind also Präfixe oder Teilworte von Worten auffindbar gar nicht oder nur ineffizient auffindbar.

3.) Die Nicht-Aufnahme von Stoppworten in den Wortindex bedeutet, dass das Auffinden einer Kette von Worten (so genannte *Phrase*) erschwert oder unmöglich wird. Lautet ein Fragment des Textes z.B. „`damit ist ciprofloxacin nicht indiziert bei der akuten angina tonsillaris`“, so ist das Suchmuster „`ciprofloxacin nicht indiziert`“ über den Wortindex nicht auffindbar, da im Deutschen das Wort „`nicht`“ üblicherweise ein Stoppwort ist.

Begriffsklärung. Bezogen auf die Suche in Texten ist ein **Suchmuster** eine Zeichenkette, die größtenteils aus Textfragmenten besteht, welche der Endnutzer im (Voll)-Text eines Werkes (siehe unten) finden möchte. Je nach Suchverfahren, welches das Suchmuster als Eingabe entgegennimmt, kann das Suchmuster auch Metazeichen wie Platzhalter oder boolesche Verknüpfungen der Textfragmente enthalten.

Begriffsklärung. In dieser Arbeit bezeichnet **Volltextsuche** eine Textsuche, die einen kompletten Text (oder einen durch den Benutzer spezifizierten Teil davon) nach einem Suchmuster durchsucht und nicht etwa nur seine Überschriften oder Nicht-Stoppworte. Die Volltextsuche liefert eine (eventuell leere) Liste von Trefferpositionen zurück und kann beliebige Teil-Zeichenketten im Text finden.

Dazu werden zunächst die Bestandteile des zu durchsuchenden Textwerkes wie folgt benannt:

Begriffsklärung. Ein **Werk** (auch: **Textkorpus**) besteht aus einem oder mehreren Werkteilen in einer festen Reihenfolge. Ein **Werkteil** besteht aus einem oder mehreren Einträgen in fester Reihenfolge. Ein **Eintrag** (auch: **Artikel**) besteht aus einem oder mehreren Feldern fester Reihenfolge. Ein **Feld** besteht aus einem oder mehreren Zeichen über einem Alphabet Σ in fester Reihenfolge.

Die obige Abhängigkeitshierarchie wird zum besseren Verständnis hier nun noch als UML-Klassendiagramm dargestellt, wobei UML an dieser Stelle eher als universelles IT-Kommunikationsmittel eingesetzt wird statt zur konkreten Spezifikation von später zu verwendenden Programmiersprachen-Konstrukten. Die Beziehung zwischen den Klassen ist als Komposition ausgeprägt, um zu verdeutlichen, dass z.B. ein Werkteil zu genau einem Werk gehört. Für eine konzentrierte Einführung in UML vergleiche [FS02].

Abb. 2.1. Bestandteile eines Werkes in UML-Notation

Beispiel 2-1. Das Werk *HagerROM* besteht u.a. aus den folgenden *Werkteilen*:
- Drogen (pharmazeutisch wirksame Bestandteile von Pflanzen)
- Stoffe (synthetisch hergestellte Pharmazeutika)
- Impfstoffe
- Blutprodukte

- Wortschatz
- Register

Im Werkteil Drogen befinden sich zum Beispiel die *Einträge*: Abelmoschus, Abelmoschus manihot, Abelmoschi radix (Abelmoschuswurzel).
Im Werkteil Stoffe befinden sich zum Beispiel die *Einträge*: Acetylstein, Acetyldigoxin, Acetylsalicylsäure.
Im Drogen-Eintrag Abelmoschus finden sich u.a. die *Felder*: Systematik (Familie, Tribus, Gattungsgliederung), Verbreitung/Anbaugebiete, Inhaltsstoffe, Drogenliefernde Arten.
Im Stoffe-Eintrag Acetylsalicylsäure finden sich u.a. die *Felder*: Basisangaben (Synonyme, ATC), Darstellung (Synthese, Nebenprodukte), Eigenschaften (UV, ^{1}H-NMR, ^{13}C-NMR, Säure/Basen-Eigenschaften, Löslichkeit), Identität (Erkennung).

Es werden nun einige zentrale Begriffe dieser Arbeit etwas formeller dargestellt, um im folgenden Verlauf der Arbeit auf diese zurückgreifen zu können.

Definition 2-2. Sei Σ ein Alphabet, d.h. eine endliche Menge von Zeichen, und sei $S = s_1 s_2 ... s_n \in \Sigma^+$ eine nicht-leere Zeichenkette mit s_i: $\forall i$: $1 \leq i \leq n$
dann ist $|S| = \texttt{length}(S) = n$ die **Länge der Zeichenkette *S***.

Definition 2-3. Sei Σ ein Alphabet und
sei $S = s_1 s_2 ... s_m \in \Sigma^+$ eine nicht-leere Zeichenkette mit s_i: $\forall i$: $1 \leq i \leq m$
und $R = r_1 r_2 ... r_n \in \Sigma^+$ eine nicht-leere Zeichenkette mit r_i: $\forall i$: $1 \leq i \leq n$
dann gilt **Gleichheit der Zeichenketten *S* und *R***: $S = R$
$\Leftrightarrow m = n$ und $s_i = r_i$, $\forall i$: $1 \leq i \leq n$.

Definition 2-4. Sei Σ ein Alphabet und sei $S = s_1 s_2 ... s_n \in \Sigma^+$ eine nicht-leere Zeichenkette mit $s_i \in \Sigma$, $\forall i$: $1 \leq i \leq n$.
Dann ist $\texttt{substr}(S, off, len) = s_{off} s_{off+1} ... s_{off+len-1}$ mit $1 \leq off \leq n$ und $0 < len \leq n\text{-}off\text{+}1$ die **Teil-Zeichenkette** (auch: **Substring**) der Länge *len* beginnend mit dem *off*-ten Zeichen der Zeichenkette *S*.

Definition 2-5. Sei Σ ein Alphabet und seien $S = s_1 s_2 ... s_n \in \Sigma^+$ und $T = t_1 t_2 ... t_m \in \Sigma^+$ nicht-leere Zeichenketten. Dann wird die zusammengesetzte Zeichenkette $s_1 s_2 ... s_n t_1 t_2 ... t_m$ als **Konkatenation** von S und T bezeichnet. Grafische Notation: $S \circ T$ (auch verkürzt geschrieben als ST).

Aus pragmatischen Gründen werden folgende Einschränkungen zugelassen, welche die obigen Definitionen etwas verallgemeinern, ohne jedoch für den Benutzer zu einer nennenswerten Einschränkung zu führen:

- Groß-/Kleinschreibung wird bei der hier vorgestellten Volltextsuche nicht unterschieden – auch wenn diese als „Exakte Volltextsuche“ bezeichnet wird.

Die Unterscheidung zwischen großen und kleinen Zeichen erhöht den Speicherbedarf von Volltextsuche-Indizes üblicherweise beträchtlich, ohne für den Benutzer hierfür einen entsprechenden Mehrwert zu bieten. Im Gegenteil verhindert eine nicht abschaltbare Unterscheidung zwischen Groß- und Kleinschreibung in manchen Fällen eine erfolgreiche Suche, eine optionale Unterscheidung erhöht entweder die Laufzeit oder den Bedarf für den Speicherplatz des Index.
Die meiste Semantik tragen im Deutschen und Englischen die Zeichen selbst und nicht ihre Groß-/Kleinschreibung, was auch der Grund dafür sein dürfte, dass führende Suchmaschinen im WWW (z.B. google.com, yahoo.com, altavista.com) hier ebenfalls keine Unterscheidung zulassen.

- Die in dieser Arbeit vorgestellte Volltextsuche arbeitet immer eintragsweise. Es werden also Trefferlisten pro Eintrag generiert, die dann zur Gesamt-Trefferliste hintereinander gehängt werden. Das bedeutet z.B., dass ein einzelner Treffer niemals von einem Eintrag in den nächsten reichen kann. Eine geeignete Zerlegung des Komplettwerkes vorausgesetzt, sollte diese Tatsache aber für den Benutzer keinerlei Einschränkung bedeuten, sondern vielmehr seine vorhandenen Erwartungen erfüllen.

Definition 2-6. Sei Σ ein Alphabet und
sei $P = p_1p_2...p_n \in \Sigma^+$ ein Suchpattern (ohne Metazeichen) der Länge n
und $E = e_1e_2...e_m \in \Sigma^+$ ein Eintrag der Länge m
dann ist die Menge von Tupeln $T_{P,E} = \{(off_1,len_1); (off_2, len_2); ...; (off_l, len_l)\}$ die **Liste aller Eintragstreffer** von Suchpattern P auf Eintrag $E \Leftrightarrow$
l=0 [leere Liste]
oder:
`substr`$(E, off_j, len_j) = P, \forall j: 1 \le j \le l$.

Definition 2-7. Sei $W = E_1,E_2, ..., E_m$ ein Werk bestehend aus den Einträgen E_1 bis E_m.
Zu einem Suchpattern P existieren jeweils pro Eintrag die Listen aller Eintragstreffer $T_{P,E1}; T_{P,E2}; ...; T_{P,Em}$ und l_k (mit $1 \le k \le m$) ist die Länge der Liste $T_{P,Ek}$. Dann ist $T_{P,W}$ die **Liste aller Werktreffer** von Suchpattern P auf Werk E mit:
$T_{P,W} = \{(1, off_{1,1}, len_{1,1}); (1, off_{1,2}, len_{1,2}); ... (1, off_{1,l1}, len_{1,l1})$
$; (2, off_{2,1}, len_{2,1}); (2, off_{2,2}, len_{2,2}); ... (2, off_{2,l2}, len_{2,l2})$
$; (m, off_{m,1}, len_{m,1}); (m, off_{m,2}, len_{m,2}); ... (m, off_{m,lm}, len_{m,lm})\}$.

Durch das in Abschnitt 6.3.2 genauer vorgestellte Merkmal der Suche mit Platzhaltern (reguläre Ausdrücke) muss Definition 2-3 (Gleichheit von Zeichenketten) nun noch angepasst werden. In der Grundversion soll die Suche

die beiden Platzhalter Fragezeichen „?“ (steht für ein beliebiges Zeichen) und Sternchen „*“ (steht für beliebig viele oder kein Zeichen) unterstützen, wie aus folgenden Beispielen über dem Alphabet Σ={a,b,c} ersichtlich wird.

Tab. 2.1. Schreibweise für vereinfachte reguläre Ausdrücke über Σ={a,b,c}

vereinfachter regulärer Ausdruck	***findet (falls im Text vorhanden)***
ab	ab
a?b	aab, abb, acb
a*b	ab,aab, abb, acb, aaab, abbccacb, ...

Diese Schreibweise weicht in einem Detail von der üblichen Schreibweise regulärer Ausdrücke ab: Normalerweise iterieren die Platzhalter „?“ und „*“ das Zeichen, welches links von ihnen steht. Zur leichteren Anwendbarkeit der Platzhalter auch durch nicht unbedingt mathematisch geschulte Benutzer iterieren diese „*vereinfachten regulären Ausdrücke*“ hier jedoch immer ein (implizit angegebenes) *beliebiges* Zeichen aus dem Alphabet Σ.

Eine weitere Abweichung (auch bezogen auf Tab. 2.1) der vereinfachten regulären Ausdrücke ergibt sich bezüglich der Semantik des ′*′-Platzhalters. Dieser kann (vor allem wenn mehrfach angewendet) durch kombinatorische Vervielfachung viele, sich teilweise überlappende Treffer in einem Text liefern. Dieses Verhalten oder die von anderen Systemen implementierte „greedyness” ist dem durchschnittlichen Endbenutzer eines Volltextsuche-Systems jedoch nur schwer zu vermitteln. Aus pragmatischen Gründen werden deshalb ′*′-Platzhalter (von links her betrachtet) immer nur eine minimale Zeichenkette bis zum nächsten exakten Treffer im Text überdecken (vgl. [Gri01], S. 64ff).

Für eine genaue Definition von regulären Ausdrücken und den regulären Wortmengen (Sprachen), die diese erzeugen können, sei auf die einschlägige Literatur (wie z.B. [HMU02], IEEE Standard 1003.1 [wwwRegExp] oder [ASU88] ab Seite 114) verwiesen. Basierend auf den dort vorhandenen Definitionen mit obigen Abweichungen spezifizieren wir nun die Gleichheit von Zeichenketten wie folgt:

Definition 2-8. Sei Σ ein Alphabet und sei $S = s_1 s_2 ... s_n \in \Sigma^+$ eine nicht-leere Zeichenkette mit $s_i \in \Sigma$ und $1 \leq i \leq n$ und $R = r_1 r_2 ... r_n \in (\Sigma \cup \{?,*\})^+$ ein vereinfachter regulärer Ausdruck und $L(R)$ die Sprache, die R bezeichnet, dann gilt **Gleichheit der Zeichenketten S und R** $\Leftrightarrow S \in L(R)$. Kommen die für die Platzhalter ′?′ und ′*′ reservierten Symbole bereits im Alphabet Σ vor, so sind diese geeignet umzukodieren.

Diese mächtigere Definition der Gleichheit von Zeichenketten angewendet auf Definition 2-6 ergibt dann eine exakte Volltextsuche, welche Suchmuster mit den Platzhaltern „?“ und „*“ verarbeiten kann.

Auf weitere Merkmale der exakten Volltextsuche wie boolesche Operatoren und die Suche nach Sonderzeichen wird in Abschnitt 6.3 eingegangen.

2.2 Fehlertolerante Volltextsuche

Der Motivationsabschnitt aus dem vorherigen Kapitel hat bereits zahlreiche Gründe dafür angeführt, warum eine Textsuche speziell auf enzyklopädischen Werken nicht nur exakte Treffer liefern können sollte. In diesem Abschnitt wird nun genauer dargelegt, welche Merkmale die benötigte fehlertolerante (synonym auch: unscharfe oder approximative) Komponente des Volltextsuche-Systems haben sollte.

1.) **Optionalität**: Das Merkmal der Fehlertoleranz bei der Textsuche sollte optional vom Benutzer zuschaltbar sein. In zahlreichen Anwendungsfällen ist eine Unschärfe bei den Suchergebnissen nicht gewünscht oder eine durch Toleranzberechnungen bedingte längere Laufzeit nicht akzeptabel. Bei ausgeschalteter Fehlertoleranz sollte die Laufzeit der exakten Suche möglichst nicht negativ durch das Vorhandensein der Fehlertoleranz-Option beeinflusst werden.

2.) **Skalierbarkeit**: Der Grad der Fehlertoleranz sollte durch den Benutzer skalierbar sein, zum Beispiel über Abstufungen in natürlicher Sprache wie z.B. „*keine*“, „*gering*“, „*mittel*“, „*hoch*“. Der Anwender sollte also bei der Auswahl der Toleranzstufe nicht mit internen Parametern des Toleranz-Algorithmus konfrontiert werden.

3.) **Bidirektionalität**: Aufgrund der im Abschnitt zur Motivation dieser Arbeit genannten Beispiele sollte die Fehlertoleranz sowohl mit Fehlern bzw. Inkonsistenzen im Textkorpus (z.B. multiple erlaubte Schreibweisen zu einem Fachwort), als auch mit Fehlern im Suchmuster umgehen können (z.B. Rechtschreibreform oder Unsicherheit des Benutzers bezüglich korrekter Schreibweise von Fachtermini) – vergleiche folgender Punkt.

4.) **Korrektheit des Suchmusters**: Es können unbeabsichtigte Tippfehler in umfangreichen enzyklopädischen Werken sicherlich nie ganz ausgeschlossen werden. Vom Benutzer kann allerdings erwartet werden, dass er sein Suchmuster nach „bestem Wissen“ korrekt formuliert. Es kann also angenommen werden, dass versehentliche Tippfehler (Auslassungen, Zeichendreher, überflüssige Zeichen) im Benutzer-Suchmuster nicht vorkommen. Im Gegensatz dazu sollten Auslassungen von „neutralen“ Zeichen wie z.B. Bindestrichen oder Leerzeichen hier jedoch erlaubt sein.

5.) **Werkspezifität**: Die Fehlertoleranz kann und sollte auf das jeweilige Werk angepasst sein. Dazu gehört neben der Anpassung an die Sprache des Textes auch Anpassungen bezüglich der Domäne innerhalb derer sich das enzyklopädische Werk bewegt. Eine Fehlertoleranz für eine englisch-sprachige Parasiten-Enzyklopädie benötigt z.B. eine anders konfigurierte Fehlertoleranz, als eine mittelalterliche, in Frühneuhochdeutsch formulierte Chronik.

6.) **Distanz**: Werden zum Original-Suchmuster durch tolerantes Information Retrieval verschiedene Varianten des Wortes im Text gefunden, sollte das Fehlertoleranz-Modul die Varianten zuerst angeben, die dem Original-Muster noch am „ähnlichsten“ sind, während dann (absteigend sortiert nach Ähnlichkeit) bis zu einer gewissen Schwelle immer unähnlichere Varianten angegeben werden. Hierzu wird eine geeignete Distanzmessung benötigt, welche den jeweiligen Abstand zwischen dem Original-Wort und seinen Varianten numerisch ausdrückt, wobei hiermit nicht zwingend eine Metrik im mathematischen Sinne gemeint sein muss.

7.) **Filterbarkeit**: Da selbst für Menschen die Bewertung, ob eine Wortvariante noch relevant für ein gegebenes Suchmuster ist (oder nicht), schwierig sein kann, wird es nicht zu verhindern sein, dass bei einer algorithmisch berechneten Fehlertoleranz auch Wortvarianten als „ähnlich“ ermittelt werden, die eine andere Semantik als das Suchmuster haben. Zum Beispiel klingen die Begriffe `Kalzium` und `Kalium` sehr ähnlich und haben bis auf die Löschung eines Zeichens dieselbe Schreibweise – bezeichnen aber zwei unterschiedliche chemische Elemente: **K** mit Ordnungszahl 19 und **Ca** mit Ordnungszahl 20 (im *Periodensystem der Elementarstoffe*).
Daher sollte es für den Benutzer der fehlertoleranten Volltextsuche möglich sein, nach der algorithmischen Ermittlung der „ähnlichen“ Worte mit ihren jeweiligen Fundstellen diejenigen Wortvarianten aus dem Gesamtergebnis herauszufiltern, die er für nicht-relevant hält.

2.3 Redaktionelle Fehlerratenverbesserung

Im vorhergehenden Abschnitt wurde bereits begründet, warum die dort beschriebene Fehlertoleranz ein zur exakten Volltextsuche optional hinzuschaltbares Merkmal sein sollte.

Die exakte Volltextsuche sei nun als Regelfall angenommen und der Benutzer muss erst durch Einschalten der Toleranz aktiv werden, bevor er auch unscharfe Treffer zu seinem Suchmuster erhält. Bei normaler, exakter Suche wirkt sich also jeder Tippfehler, jede inkonsistente Verwendung von Bindestrichen und jede unterschiedliche Anwendung von Getrennt- und Zusammenschreibung eines

Begriffes innerhalb des Werkes negativ auf die Vollständigkeit der Trefferliste aus. Es lohnt sich also, bereits im Vorfeld der Veröffentlichung eines Werkes dieses möglichst von Tippfehlern und inkonsistenten Schreibweisen zu befreien und die Schreibweisen zu vereinheitlichen.

Neben den bei professionell erstellten Nachschlagewerken durch Autoren und Fachlektoren immer stattfindenden mehrfachen Korrekturzyklen werden im Rahmen dieser Arbeit einige Vorgehensweisen untersucht, die die Korrektur von großen Textkorpora unterstützend begleiten können. Dazu werden Methoden und Programme entwickelt, die einem Lektor Listen von potenziell falsch (oder inkonsistent) geschriebenen Worten mit ihrem jeweiligen Kontext und mindestens einem Korrekturvorschlag präsentieren.

Rechtschreibkorrektur-Systeme, wie sie heutzutage in handelsübliche Textverarbeitungsprogramme integriert sind, sind bei der Korrektur von großen, fachspezifischen Textkorpora meist nicht sehr behilflich, weil sie den für diese Werke benötigten Fachwortschatz nicht abdecken (siehe Abb. 2.2). Selbst bei Aufnahme eines Fachwortes in das Benutzerwörterbuch werden dann oft die Abwandlungen des Wortes (z.B. Deklination von Substantiven, Konjugation von Verben usw.) noch immer als Fehler markiert. Außerdem sind diese Systeme nicht in der Lage, inkonsistente Schreibweisen aufzudecken bzw. zu verhindern.

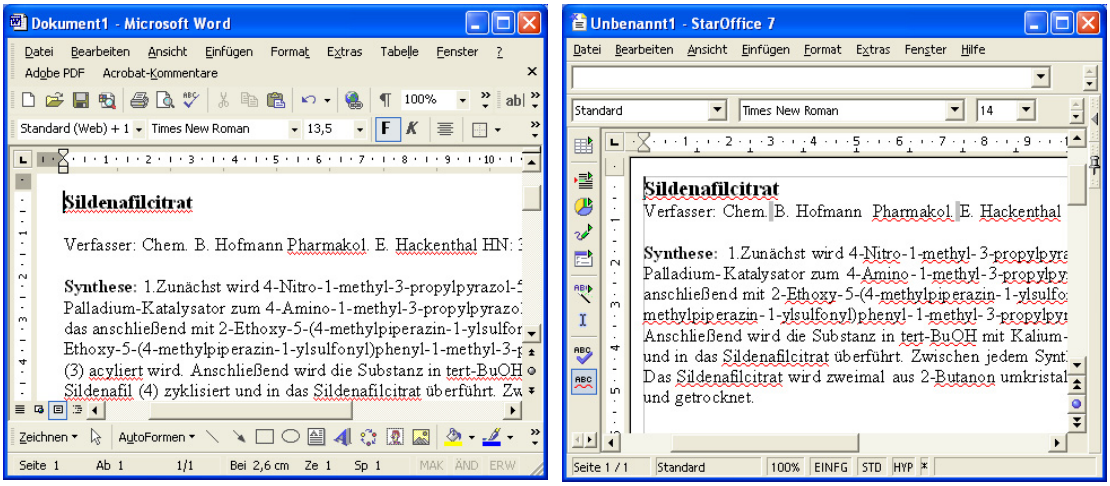


Abb. 2.2. Rechtschreibkorrektur in aktuellen Textverarbeitungsprogrammen

Aber nicht nur das Textmaterial von elektronischen Enzyklopädien und Nachschlagewerken kann von einer automatisierten Qualitätsverbesserung profitieren. Auch das Zusatzmaterial wie Zeichnungen, Bilder, Video- und Audiomaterial und Querverweise (Hyperlinks) lassen sich programmgestützt auf Konsistenz und Plausibilität prüfen. So werden in dieser Arbeit auch einige Techniken vorgestellt, um durch automatisierte Generierungs- und Prüfverfahren Fehler in digitalen Werken außerhalb des Textes zu vermeiden.

Im Rahmen der Beschreibung dieser Fehlerratenverbesserung werden angelehnt an die Begriffe Klasse und Instanz aus der objektorientierten Programmierung die Begriffe *Wortinstanz* und *Wortklasse* verwendet.

Begriffsklärung. Eine **Wortinstanz** bezeichnet jedes Auftreten eines bestimmten Wortes in einem Text. In einer **Wortklasse** dagegen wird mehrfaches Auftreten eines Wortes nur einmal gezählt. Dabei erzeugen gebeugte Wortformen unterschiedliche Wortklassen. **Beispiel**: Der Text „`Das kleine grüne Auto und das große schwarze Auto wurden bei dem Unfall beschädigt.`“ besteht aus 12 Wortklassen und 14 Wortinstanzen und zu den Wortklassen „`Auto`“ und „`das`“ existieren jeweils 2 Wortinstanzen.

Kapitel 3

Datenstrukturen für Volltextsuche-Algorithmen

In der Einleitung von Kapitel 2 wurde bereits die im Information Retrieval herrschende Differenzierung zwischen Online- und Offline-Suchalgorithmen erläutert. Online-Suchalgorithmen finden vor allem dort Anwendung, wo aus Zeit- oder Platzgründen zu einem Text kein Index angelegt werden kann, oder wo die Index-Generierung nicht sinnvoll erscheint, weil der Text nicht als quasi-statisch angesehen werden kann, oder weil es sich aufgrund zu geringer Größe des Textes nicht lohnt. Alles dies gilt für kommerzielle, digitale Enzyklopädien nicht und daher werden Online-Suchalgorithmen in dieser Arbeit nicht betrachtet. Für diesbezügliche Details sei auf die Literatur (z.B. [Ste94]) verwiesen.

Aufgrund der in Abschnitt 2.1 dargelegten Probleme *wortbasierter* Volltextsuchen wird auch auf diese Gruppe von Algorithmen mit ihren Datenstrukturen nicht weiter eingegangen. Im Folgenden werden daher die gebräuchlichen Index-Datenstrukturen vorgestellt, die Volltextsuchen beschleunigen, welche beliebige Substrings in einem Text finden können: *Suffix-Trees*, *Suffix-Arrays*, *q-Gramme* und *q-Samples*.

3.1 Suffix-Tries und Suffix-Trees

Ein Suffix-Tree [McCr76] ist eine Datenstruktur, mit der man zahlreiche String-Probleme effizient lösen kann, da sie den Textinhalt indiziert, ohne ihn vorher in Worte zu zerlegen. Neben der Suche nach Substrings in einem Text können damit zum Beispiel auch Probleme der Bioinformatik (Longest Common Substring) bei der DNA-Analyse schnell gelöst werden.

Definition 3-1. Sei $T=t_1t_2t_3...t_{n-2}t_{n-1}t_n$ eine nicht-leere Zeichenkette der Länge n. Dann ist $P=t_1t_2...t_p$ das **Präfix** (engl. **Prefix**) der Länge p von T (mit $p \leq n$) und $S=t_{n-s+1}t_{n-s+2}...t_n$ das **Suffix** der Länge s von T (mit $s \leq n$).

Dabei gilt: Jeder *Substring* (vgl. Definition 2-4 auf Seite 15) eines Strings T ist gleichzeitig auch *Präfix* genau eines *Suffixes* von T. Diese Eigenschaft ermöglicht bei schnellem Zugriff auf alle Suffixe eines Strings auch die schnelle Suche nach beliebigen Substrings dieses Strings.

Zunächst sei die Methode beschrieben, aus einem Text T seinen Suffix-Trie zu erzeugen. Anschließend wird der Schritt beschrieben, aus einem Suffix-Trie einen Suffix-Tree aufzubauen.

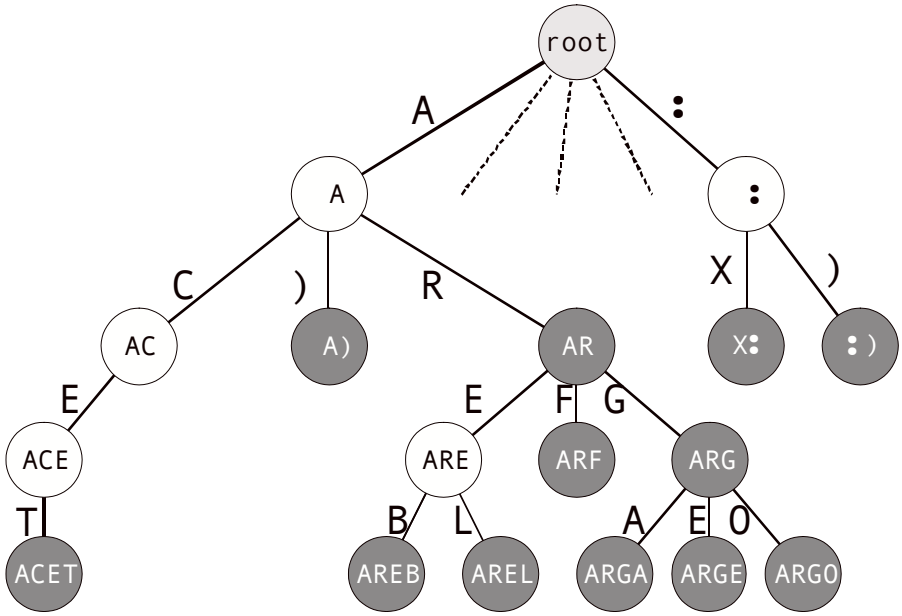


Abb. 3.1. Beispiel-Trie mit kurzen Strings

Der Begriff *Trie* wurde 1960 von FREDKIN aus dem Begriff „Information Re*trie*val“ geprägt [Fred60]. Ein *Trie* ist ein digitaler Suchbaum für Zeichenketten. Aus jedem Knoten können maximal so viele Kanten laufen, wie das Alphabet Symbole besitzt. Auslaufende Kanten eines Knotens stehen dabei stets für jeweils unterschiedliche Symbole. Einige Knoten werden speziell als „String-Ende-Knoten“ ausgezeichnet (in Abb. 3.1 dunkel schattiert), welche dann genau für den String stehen, der entsteht, wenn man die Kantensymbole auf dem Pfad von der Wurzel zu diesem String-Ende-Knoten aneinanderreiht. Soll die Existenz eines Strings in einem Trie überprüft werden, steigt man entsprechend den Symbolen des Suchmusters in den Baum ab. Existiert eine benötigte Kante nicht, so existiert das Suchmuster nicht im Trie. Wurden alle Symbole des Musters abgearbeitet und man befindet sich auf einem Knoten des Tries, so existiert

das Suchmuster an den Startpositionen aller Suffixe des aktuellen Teilbaumes. Kann das Muster nicht im Trie gefunden werden, existiert es auch nicht im Text. Im Trie aus Abb. 3.1 sind zum Beispiel die Strings AR und AREB gespeichert, nicht jedoch die Zeichenkette ARE. Jede Kante im Beispiel-Trie ist mit dem entsprechenden Symbol markiert, welches sie repräsentiert. Für eine vertiefte Diskussion der Trie-Datenstruktur im Allgemeinen sei auf DONALD E. KNUTHS Standardwerk [Knut98] (Vol. 3) ab Seite 492 verwiesen.

Da es bei der Speicherung von Suffixen in einem Trie passieren kann, dass ein (kürzeres) Suffix das Präfix eines anderen (längeren) Suffixes ist, müssen entweder wie oben die entsprechenden Suffix-Ende-Knoten markiert werden, oder alle Suffixe werden mit einem Symbol (z.B. '$') terminiert, welches im Text sonst nicht vorkommt. In den Suffix-Ende-Knoten wird nun üblicherweise noch die Position gespeichert, an der das entsprechende Suffix im Text enthalten war.

Die Anzahl der Kanten von Suffix-Tries über natürlich-sprachlichen Texten lässt sich meist stark reduzieren, indem man Trie-Pfade zusammenfasst, auf denen nur Knoten mit jeweils einer ausgehenden Kante aufgereiht sind. Im Beispiel aus Abb. 3.1 gilt dies für den Pfad von A-AC-ACE-ACET. Dieser Prozess erzeugt eine Datenstruktur, die als *Suffix-Tree* (oder *compact Suffix-Tree*, vgl. [Ste94], S. 90) bezeichnet wird.

Sei T ein Text der Länge n, dann lautet die einfache (brute-force) Methode zum Aufbau des Suffix-Tree Y_n:

```
1: for i=1 to n
2:   Yi = insert(Yi-1, substr(T,i,n))
```

Diese Methode benötigt zwar im schlimmsten Fall eine Laufzeit $O(n^2)$, aber es konnte gezeigt werden, dass bei üblichen Wahrscheinlichkeiten für das Auftreten von Symbolen in natürlichen Sprachen eine durchschnittliche Laufzeit von $O(n \log(n))$ erwartet wird [Ste94].

Nachdem zunächst durch andere Forscher (z.B. [McCr76]) einige schnellere Verfahren für den Aufbau von Suffix-Trees entwickelt wurden, konnte UKKONEN 1992 ein Verfahren vorstellen, welches den Suffix-Tree zu einem Text mit Zeitaufwand $O(n)$ konstruiert [Ukk95]. Dabei wird der Text nur einmal von vorne nach hinten durchlaufen und in jedem Schritt existiert der vollständige Suffix-Tree zum bisher gesehenen Text.

Ein Schwachpunkt von Suffix-Trees ist die Tatsache, dass diese Datenstruktur viel Speicherplatz benötigt (selbst die besten Ansätze benötigen $9^*|T|$) und dass Suffix-Trees schlecht auf externem Speicher verwendet werden können [NBST01].

3.2 Suffix-Arrays

Ein Suffix-Tree über einem natürlich-sprachlichen Text benötigt üblicherweise viel Speicherplatz für die zahlreichen Kantenverbindungen und ihre Beschriftungen. Ein *Suffix-Array* ist eine Datenstruktur, die lediglich die Positionen aller Suffixe zu einem Text nach deren lexikographischer Sortierung in einem Array abspeichert. Diese Darstellung ist zwar kompakt, aber anders als beim Suffix-Tree ist üblicherweise der Zugriff auf den Text noch nötig, um den tatsächlichen Inhalt der Suffixe zu ermitteln. Mithilfe des Suffix-Arrays kann man z.B. eine binäre Suche über alle Suffixe eines Textes leicht realisieren.

> **Beispiel 3-1.** Gegeben sei der Text `good`. Seine vier Suffixe sind nach lexikographischer Sortierung (in Klammern die Startposition): `d`(4),`good`(1),`od`(3),`ood`(2). Das zugehörige Suffix-Array lautet also: [4,1,3,2].

Da ein Suffix-Array einfach mittels Tiefendurchlauf (erst links, dann rechts) aus einem Suffix-Tree konstruiert werden kann und da dieser Suffix-Tree mit Zeitaufwand *O(n)* konstruierbar ist (vgl. vorhergehender Abschnitt), kann also auch das Suffix-Array in *O(n)* aufgebaut werden. Da im Suffix-Array nur Zeiger in den Text gebraucht werden und für übliche Texte 4-Byte Zeiger ausreichen, gilt für Suffix-Arrays eine Speicherplatz-Abschätzung von $4^*|T|$.

Sei nun Text T und das zugehörige Suffix-Array gegeben und man möchte eine exakte Textsuche nach einem Muster durchführen, so muss man nun nicht mehr den kompletten Text durchsuchen. Man führt vielmehr mithilfe des Suffix-Arrays eine binäre Suche nach dem Suffix durch, welches als Präfix das Suchmuster hat. Dabei ist zu beachten, dass eventuell der gesuchte Substring Präfix von mehreren Suffixen sein kann. Ein Teilbaum des Suffix-Tree entspricht also einem Intervall von Suffix-Array-Einträgen. Die Bewegung zu einem Kind-Knoten im Suffix-Tree bedeutet daher im Suffix-Array die Zusammenschiebung der beiden momentanen Intervallgrenzen, wobei dies zwei binäre Suchen erfordert.

3.3 *q*-Gramme und *q*-Samples

Der Begriff *q*-Gramm (in der Literatur auch manchmal *n-Gramm* genannt, [WMB94]) steht vereinfacht ausgedrückt für eine Zeichenkette der Länge *q*. Zerlegt man einen Text in seine *q*-Gramme, so speichert man in einem geeigneten Index meist alle (manchmal auch nur die häufigsten) Substrings der Länge *q* des Textes ab. Beim Aufbau von *q*-Gramm-Indizes wird dabei das Fenster der Breite *q*, mit welchem die *q*-Gramme ermittelt werden, immer jeweils um ein Zeichen weitergeschoben, um zum nächsten *q*-Gramm zu gelangen. Übliche Werte

für q sind z.B. [1,2,3,4]: 1-Gramme, 2-Gramme, 3-Gramme und 4-Gramme, wobei je nach Anwendungszweck auch größere Werte für q möglich sind.

Beispiel 3-2. Der Text T **ABRACADABRA** enthält folgende 4-Gramme (in Klammern angegeben ist jeweils die Startposition):
ABRA(1,8), BRAC(2), RACA(3), ACAD(4), CADA(5), ADAB(6), DABR(7).

Angelehnt an die Bezeichnungen aus der objektorientierten Programmierung kann man von *q-Gramm-Klassen* (im oberen Beispiel: sieben verschiedene) und von *q-Gramm-Instanzen* (im oberen Beispiel: acht verschiedene) sprechen. Ein Text besitzt immer ($|T|$-q+1) viele q-Gramm-Instanzen; die Anzahl seiner q-Gramm-Klassen hingegen lässt sich üblicherweise nicht vorhersagen, denn sie hängt von der Reihenfolge der Symbole im Text ab.

Beispiel 3-3. Aufbauend auf Beispiel 3-2 wird nun im Text mithilfe seines existierenden 4-Gramm-Index nach dem Suchmuster RACADABR gesucht. Dazu wird das Muster ebenfalls in 4-Gramme zerlegt – allerdings ist nun Überlappung nicht mehr notwendig: RACA und DABR werden nun im Index gesucht. Passen die Positionen im Index zum Abstand der 4-Gramme im Suchmuster, so liegt ein Treffer vor: RACA (Position 7 in T), DABR (Position 3 in T), $7 - 3 = 4$. An der Position 3 beginnt also ein Treffer.

Beispiel 3-4. Suchmuster, deren Länge kein ganzzahliges Vielfaches von q ist, werden gesucht, indem sie in überlappende q-Gramme zerlegt werden. Für die Suche nach ACADA wird eine Überlappung von drei (das Fenster wurde um eins weitergeschoben) benötigt. Es werden also die 4-Gramme ACAD(4) und CADA(5) analysiert: $5 - 4 = 1$. An der Position 4 beginnt also ein Treffer.

Enthält das Suchmuster auch nur ein q-Gramm, welches nicht zu der Menge der q-Gramm-Klassen eines Textes gehört, so kann im Text kein Treffer vorhanden sein. Verallgemeinernd lässt sich sagen, dass nach der Zerlegung des Suchmusters in q-Gramme mit der q-Gramm-Klasse angefangen werden sollte, die die wenigsten q-Gramm-Instanzen besitzt. So kann schnell auf immer kürzer werdenden Listen von potenziellen Treffern gearbeitet werden.

Die Suche nach *vereinfachten regulären Ausdrücken* (siehe Abschnitt 2.1, S. 17) ist mit einem q-Gramm-Index trivial. Durch eine Menge von Fragezeichen-Platzhaltern (?) erhöht sich nur der Abstand der q-Gramme links und rechts dieser Platzhalter entsprechend der Anzahl der Fragezeichen. Wird ein Sternchen-Platzhalter (*) in einem Suchmuster verwendet, ändert sich die Bedingung bezüglich der gesuchten q-Gramm-Instanz-Positionen von „ist-gleich“ auf „größer-gleich“. Und abhängig von der Interpretation des *-Platzhalters als „greedy“ oder „nicht-greedy“ werden aus den so ermittelten q-Gramm-Instanzen die konkreten Treffer konstruiert.

Beispiel 3-5. Aufbauend auf den Index aus Beispiel 3-2 soll nun nach dem Muster `ABRA*DABR` mit einem vereinfachten regulären Ausdruck gesucht werden. Zu `ABRA`(1) existiert `DABR`(7) mit größerer Position. Also existiert ein Treffer zum Muster, der sich von Position 1 in T bis einschließlich Position 10 erstreckt. Zu `ABRA`(8) gibt es jedoch keine Instanz der Klasse `DABR` mit einer größeren Position. Daher ist `ABRA`(8) nicht Bestandteil eines Treffers.

Bei der Wahl von q muss abgewogen werden: Ein *größeres* q sorgt für eine durchschnittlich geringere Anzahl von Instanzen je q-Gramm-Klasse, welches zu kürzeren Instanzlisten und damit zu schnellerer Suche führt. Dagegen sorgt ein *kleineres* q für eine geringere Anzahl Klassen und damit für weniger Verwaltungsaufwand für diese allerdings werden die Instanzlisten hierdurch auch länger. Außerdem bestimmt q die minimale Länge, die das Suchmuster haben muss. Existieren zu einem Text nur 4-Gramme, so ist die Suche nach einem Muster der Länge 3 nicht (bzw. vergleichsweise aufwändiger und damit evtl. weniger effizient) möglich.

Die Auswirkung von q auf die minimale Suchmuster-Länge überträgt sich hier auch auf die Bestandteile der regulären Ausdrücke. Existiert (wie in Beispiel 3-2) nur ein 4-Gramm-Index, so kann weder nach dem Muster `AB*DABR` noch nach dem Muster `CADA*A` gesucht werden, da die Teilmuster `AB` und `A` zu kurz sind.

Üblicherweise behilft man sich in diesem Fall damit, dass man auch Indizes für q=1 und andere kleine Werte von q anlegt: z.B. q={1,2,3,4}, was natürlich entsprechend mehr Platzbedarf bedeutet. Der q-Gramm-Suchalgorithmus benutzt dann zur Optimierung der Geschwindigkeit immer möglichst lange q-Gramme (und damit kurze Instanzlisten) und fällt nur dann auf die kurzen q-Gramme (mit langen Instanzlisten) zurück, falls das Muster es erzwingt.

Falls nun also Indizes für verschiedene Werte von q angelegt werden müssen, erhöht sich offensichtlich auch der Speicherbedarf für den Gesamtindex. Nun gibt es zwei Möglichkeiten, den Speicherbedarf (auf Kosten erhöhter Laufzeit) wieder zu verkleinern. Zum einen kann man speziell bei größeren q-Werten diejenigen q-Gramm-Klassen, welche zu wenig Instanzen besitzen nicht in den Index aufnehmen. Dies passiert dann auf der Annahme, dass nach diesen seltenen q-Gramm-Klassen auch selten gesucht wird und in diesen seltenen Fällen nimmt man dann eine erhöhte Laufzeit durch den Zugriff auf kürzere q-Gramm-Listen in Kauf.

Eine zweite Möglichkeit, Speicherplatz zu sparen, sind so genannte *q-Samples* [ST96] (manchmal auch *h-Samples* [Taka94]). Bei der Generierung von q-Samples wird das Fenster zur Generierung nicht jeweils um eine Position weitergeschoben (wie bei q-Grammen), sondern jeweils um einen festen Wert größer als Eins. In Extremfällen überlappen sich die gespeicherten q-Gramme nicht

mehr und man hat daher keine vollständige Information mehr über das Vokabular des Textes. Die q-Samples können also nur der Filterung dienen, indem sie Textstellen herausfiltern, in denen ein Treffer möglicherweise vorhanden sein kann.

Der auf dieser Datenstruktur basierende Volltextsuche-Algorithmus muss dem natürlich Rechnung tragen. Bei der Suche mit einem q-Sample-Index wird zunächst das Pattern in (überlappende!) q-Gramme zerlegt und es werden mithilfe der gespeicherten q-Samples alle die Textstellen ermittelt, in denen genügend Pattern-Fragmente enthalten sind. An diesen Stellen wird dann mit einer nachgeschalteten exakten Online-Suche (ohne Index) im Text überprüft, ob ein exakter Treffer an der fraglichen Stelle auch wirklich vorhanden ist.

Kapitel 4

Bekannte Ansätze zur fehlertoleranten Volltextsuche

Nachdem in Kapitel 3 die wichtigsten Datenstrukturen für exakte Volltextsuchen vorgestellt wurden, wird in diesem Kapitel nun eine Übersicht über bestehende Algorithmen gegeben, die eine fehlertolerante Volltextsuche ermöglichen.

4.1 Phonetische Codes

Das wohl bekannteste Verfahren zur unscharfen phonetischen Suche dürfte der *Soundex*-Algorithmus sein, der bereits 1918 von Robert Russel zu folgendem Zweck entwickelt und unter U.S. Patent No. 1.261.167 patentiert wurde: „*provide an index wherein names are entered and grouped phonetically rather than according to the alphabetical construction of the names*“ [Rus18]. Zeitlich deutlich vor der Erfindung von Computern gelegen, war Russels Erfindung auch zunächst nur dazu gedacht, die Volkszählungen in den USA dahingehend zu unterstützen, Familien mit gleichen Vorfahren aber mittlerweile unterschiedlicher Schreibweise des Familiennamens über den phonetischen Klang der Namen (also orientiert an der Aussprache) einander zuzuordnen.

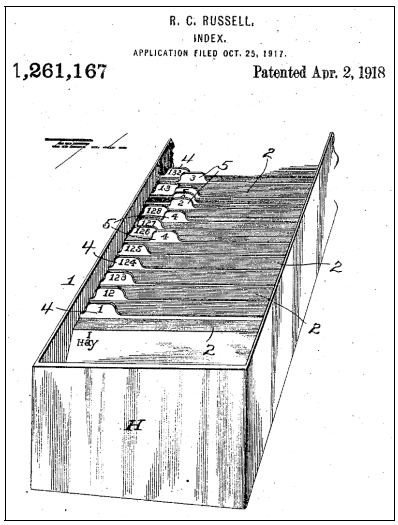

R. C. RUSSELL.
INDEX.
APPLICATION FILED OCT. 25, 1917.
1,261,167
Patented Apr. 2, 1918

Abb. 4.1. Ausschnitt Soundex-Patent

Russel bildete zu diesem Zweck zu jedem Familiennamen einen maximal vierstelligen so genannten Soundex-Code, der nach dem unten stehenden Verfahren gebildet wurde. Anschließend wurden normale Karteikasten-Reiter mit allen existierenden Codes beschriftet und die Karteikarten der entsprechenden Familiennamen dort einsortiert (siehe Abb. 4.1).

Zur Bildung der Codes wurden zunächst die Buchstaben des (englischen) Alphabets in die folgenden acht Gruppen aufgeteilt. Dabei ist bemerkenswert, dass alle Vokale in nur einer Gruppe enthalten sind und dass die Konsonanten `h`, `j` und `w` vollständig fehlen.

Tab. 4.1. Einteilung der Buchstaben in Soundex-Gruppen nach [Rus18]

Gruppe	***Buchstaben***	***Bemerkung***
1	`a`, `e`, `i`, `o`, `u`, `y`	Vokale
2	`b`, `f`, `p`, `v`	Lippenlaute (Labiale)
3	`c`, `g`, `k`, `q`, `s`, `x`, `z`	Zisch-, Kehllaute (Sibilant, Guttural)
4	`d`, `t`	stimmlose Dentale
5	`l`	palataler Reibelaut (Frikativ)
6	`m`	Lippen-Nasal
7	`n`	Zungen-Nasal
8	`r`	dentaler Reibelaut (Frikativ)

Anschließend wurden folgende Regeln zur Bildung des Codes angewendet:

1.) Der erste Buchstabe des Namens wird als Buchstabe unverändert übernommen und bildet den ersten Buchstaben des Codes (bei `Meier` also `M`).
2.) Alle folgenden Buchstaben werden als Zahl entsprechend ihrer Gruppe aus obiger Tabelle codiert (Bei `Meier` also z.B. `e` = 1, `r`=8).
3.) Allerdings wird nur der erste Vokal numerisch im Code gespeichert. Alle folgenden Vokale werden verworfen (bei `Meier` verschwindet also das `ie`)
4.) Fallen mehrere aufeinander folgende Buchstaben in dieselbe Gruppe, wird nur das erste Gruppenmitglied codiert.
5.) Die Buchstabenfolge `gh` wird generell verworfen. Die Buchstaben `s` und `z` werden verworfen, falls sie am Ende des Namens stehen.
6.) Der Code wird spätestens nach 4 Stellen abgeschnitten.

Beispiel 4-1. Hier nun einige Namen mit ihren jeweiligen Soundex-Codes:
`Meyer` = **M18** (Mer) `Maier` = **M18** (Mar)
`Smith` = **S614** (Smit) `Smyth` = **S614** (Smyt)
`Peterson` = **P148** (Petr) `Peters` = **P148** (Petr)
aber - trotz ähnlichem Klang:
`Knight` = **K714** (Knit) `Night` = **N14** (Nit)

Da Russels relativ einfacher phonetischer Ansatz zahlreiche Kritik provozierte, hat er etliche, verbesserte Nachfolger hervorgebracht, wie zum Beispiel *Phonix* von Gadd [Gad90], *Metaphone* und *Double Metaphone* von Philips [Phi00] und Hodge & Austins *Phonetex* [HA01b], um nur einige zu nennen. Es existieren sogar einige hybride Algorithmen wie zum Beispiel *Editext* und *Idapist* von Zobel & Dart [ZD96], welche die Fehlerkorrektur-Fähigkeiten der Edit-Distanz (siehe folgender Abschnitt) kombinieren mit der Möglichkeit von phonetischen Codes, ähnlich klingende Wortvarianten zu gruppieren.

Ein Problem von Ansätzen, die sich auf phonetische Codes abstützen, ist die Tatsache, dass phonetische Codes nicht abgestuft verwendbar sind. Das heißt, die im Voraus berechneten Codes gruppieren Worte aus dem Text zusammen, und ein Suchwort ist entweder zu einer kompletten solchen Gruppe ähnlich, oder nicht. Es wird kein Mechanismus zur Verfügung gestellt, Worte mit „mehr“ oder „weniger“ Ähnlichkeit zu suchen. Die Ähnlichkeit ist hier also ein boolescher Wert, der wahr oder falsch sein kann.

Ein weiteres Problem ist, dass die Technik der phonetischen Codes sich eigentlich nur für Gebiete eignet, bei denen die Zerlegung in Worte offensichtlich ist (z.B. Listen von Nachnamen, simple Wörterbücher oder Überschriften von Lexika-Einträgen). Zur Volltextsuche sind sie jedoch ungeeignet: Möchte man phonetische Codes für die Suche in Texten verwenden, so muss man die Texte vorher in Worte zerlegen, denn alle diese Algorithmen arbeiten immer auf Listen von Worten, für welche sie ihre Codes vorab berechnen. Über die Nachteile einer wortweisen Zerlegung wurde bereits oben geschrieben. Eine echte *Volltext*-suche, die jeden beliebigen Substring eines Werkes finden kann, ist mit phonetischen Codes nicht möglich.

Möchte man also mit einer fehlertoleranten Volltextsuche beliebige Fragmente eines Textes auffindbar machen, so führt dies zur Vermeidung von phonetischen Codes und hin zur Verwendung von String-Distanzen. Das wohl bekannteste und meist-verwendete Verfahren zur Repräsentation von solchen Distanzen dürfte die Edit-Distanz sein, welche im folgenden Abschnitt beschrieben wird.

4.2 Edit-Distanz und *q*-Gramm-Distanz

Die 1965 von LEVENSHTEIN [Lev65] eingeführten zwei Metriken *Levenshtein-Distanz* und *Edit-Distanz* stellen den Abstand zweier Zeichenketten S_1 und S_2 als numerischen Wert dar. Diesem Wert zugrunde liegt die minimale Anzahl von atomaren Bearbeitungsschritten (z.B. Tastatur-Anschlägen), die benötigt werden, um S_1 in S_2 zu überführen (und umgekehrt). Damit gibt diese Abstandskenngröße auch eine Aussage darüber, wie ähnlich sich diese beiden Strings sind, und sie wird daher von zahlreichen fehlertoleranten Suchalgorithmen verwendet.

Die atomaren Bearbeitungsschritte, die gezählt werden, sind: `DEL` (Löschen), `INS` (Einfügen) und `SUBST` (Ersetzung) jeweils einzelner Zeichen. Folgende Unterschiede bestehen bei der Zählung der atomaren Aktionen zwischen den beiden von LEVENSHTEIN geprägten Metriken: Die *Edit-Distanz* zählt das Ersetzen eines Zeichens als eine Lösch- und eine Einfüge-Aktion (und damit als zwei Aktionen), während die *Levenshtein-Distanz* für eine Zeichen-Ersetzung nur Kosten=1 berechnet (siehe [Ste94], Seite 40ff).

Tab. 4.2. Kosten atomarer Editier-Aktionen bei Levenshteins LD & ED

Metrik	***DEL***	***INS***	***SUBST***
Levenshtein-Distanz	1	1	1
Edit-Distanz	1	1	2

Im weiteren Verlauf dieser Arbeit wird zwischen den beiden genannten Metrik-Varianten nicht weiter unterschieden, da es hier um die prinzipiellen Möglichkeiten des Ansatzes gehen soll. Diese Möglichkeiten werden jedoch von der unterschiedlichen Bewertung der `SUBST`-Aktion nicht beeinflusst. Es wird im Folgenden also Edit-Distanz (ED) als Oberbegriff für beide Varianten gebraucht. Welche konkreten Kosten die `SUBST`-Aktion benötigt, ist aus dem jeweiligen Kontext zu entnehmen.

Ein übliches Verfahren, die Edit-Distanz zu berechnen, ist die Benutzung von *dynamischer Programmierung* (1953 von RICHARD BELLMAN erstmals beschrieben [wwwBellm]). Bei dynamischer Programmierung werden progressiv Mengen von Werten (oder Kosten) basierend auf bereits berechneten Werten erzeugt, um so Probleme mit sequenziellen Entscheidungsstrukturen und zusammengesetzten Kostenfunktionen zu lösen.

Bei der Berechnung der Edit-Distanz mit dynamischer Programmierung wird hier der Wagner-Fischer-Algorithmus [WF74] betrachtet, welcher sukzessive eine Tabelle mit Werten füllt, bei der in jeder Zelle eine Teillösung steht, welche die Distanz zwischen zwei Präfixen der betrachteten beiden Strings darstellt.

Als Beschriftung für die Tabelle stehen zeilenweise die Zeichen des Quell-Strings $X{=}x_1x_2...x_m$ und spaltenweise die Zeichen des Ziel-Strings $Y{=}y_1y_2...y_n$. Jeweils als 1. Spalte und 1. Zeile wird das „leere Wort“ ε vor die ersten Zeichen von X und Y eingefügt. Diese erste Spalte/Zeile wird aufsteigend mit Kosten für immer jeweils eine weitere Einfügung initialisiert und ist damit nur abhängig von $|X|$ und $|Y|$.

Basierend auf dieser Initialisierung wird die Tabelle zeilenweise (oder auch spaltenweise) mit Werten gefüllt, sodass in einer Zelle $d_{i,j}$ der Tabelle stets die minimalen Kosten für die Überführung von `substr`(X,1,i) in `substr`(Y,1,j) stehen. Der Wert an der Stelle $d_{m,n}$ entspricht damit also `ED`(X, Y).

Zur Berechnung von `ED`(X, Y) sei nun folgende Kostenfunktion w(a,b) gegeben, welche die Kosten für die Transformation von Zeichen `a` in Zeichen `b` berechnet.

w(a,ε) = 1		(Löschen von a)
w(ε,b) = 1		(Einfügen von b)
w(a,b) = 1	falls a≠b	(Substitution a nach b)
w(a,b) = 0	falls a=b	

Der Kostenwert in Tabellen-Zelle $d_{i,j}$ ist dann das Minimum dreier Werte, welche jeweils aus bereits existierenden Kostenwerten berechnet werden:

$$d_{i,j} = \min\{\ d_{i\text{-}1,j}+\mathrm{w}(x_i,\varepsilon),\ \ d_{i,j\text{-}1}+\mathrm{w}(\varepsilon,y_j),\ \ d_{i\text{-}1,j\text{-}1}+\mathrm{w}(x_i,y_j)\ \}$$

Beispiel 4-2. Tabelle zur Berechnung der Edit-Distanz zwischen den Worten X=Wolfram und Y=Wolfgang. Der Wert in der letzten Tabellen-Zeile und der letzten Spalte gibt an: `ED`(Wolfram, Wolfgang)=3. Folgende drei Bearbeitungsschritte sind nötig: `SUBST`(r,g), `SUBST`(m,n) und `INS`(g) am Ende.

Tab. 4.3. Tabelle zur Berechnung Edit-Distanz (Wolfram, Wolfgang)

	ε	W	o	l	f	g	a	n	g
ε	0	1	2	3	4	5	6	7	8
W	1	0	1	2	3	4	5	6	7
o	2	1	0	1	2	3	4	5	6
l	3	2	1	0	1	2	3	4	5
f	4	3	2	1	0	1	2	3	4
r	5	4	3	2	1	1	2	3	4
a	6	5	4	3	2	2	1	2	3
m	7	6	5	4	3	3	2	2	**3**

Wird nun die Edit-Distanz mit einer solchen Tabelle durch dynamische Programmierung berechnet, so bewegt sich der Zeitaufwand in $O(|X|*|Y|)$. Für den Speicherbedarf reicht jedoch bei spaltenweiser Berechnung $O(|X|)$ aus, da immer

nur die komplette zuletzt berechnete Spalte und die Zelle über der aktuellen Zelle zur Berechnung der aktuellen Zelle benötigt wird.

Möchte man nun aus solch einer Tabelle die tatsächlich benötigten Bearbeitungsaktionen ablesen, welche mit minimalen Kosten X in Y überführen, so muss man sich entlang der so genannten „Spur der minimalen Kosten“ in der Tabelle von unten rechts nach oben links bewegen. Details dazu sind in [Ste94] auf Seite 54 dargelegt.

In Abschnitt 3.3 (S. 26) wurde im Kontext von Datenstrukturen zur exakten Suche bereits das Konzept der q-Gramme eingeführt, um damit einen Index zu konstruieren, mit dem eine schnelle Volltextsuche möglich ist. Neben der exakten Suche nach Textfragmenten gibt es jedoch noch die Möglichkeit, mit q-Grammen die Ähnlichkeit von Zeichenketten zu modellieren.

Neben der Edit-Distanz existiert mit der so genannten *q-Gramm-Distanz* ein weiteres interessantes Verfahren, das Abstände zwischen Zeichenketten berechnet. Aufgrund der deutlich selteneren Verwendung in bekannten Ansätzen zur approximativen Suche wird die Betrachtung jedoch vergleichsweise kurz ausfallen, um den Rahmen dieser Arbeit nicht zu sprengen.

Das von Ukkonen in [Ukk92] beschriebene Verfahren benutzt erstmalig die im Folgenden dargestellte q-Gramm-Distanz, um eine approximative Suche zu realisieren. Um den Abstand zwischen zwei Zeichenketten mit dieser Distanz zu berechnen, wird ermittelt, welche q-Gramme in den Zeichenketten vorkommen. Je mehr q-Gramme die Zeichenketten gemeinsam haben, desto ähnlicher werden die Zeichenketten zueinander angesehen.

Definition 4-1. Sei $x=a_1 a_2 \ldots a_n \in \Sigma^*$ eine Zeichenkette über dem Alphabet Σ und sei $v \in \Sigma^q$ ein q-Gramm. Wenn $s=a_i a_{i+1} \ldots a_{i+q-1}$ ein Substring von x ist und $s=v$, dann hat x ein **Vorkommen** von v.
$G(x)[v]$ bezeichnet die **Gesamtzahl aller Vorkommen** von v in x.

Definition 4-2. Seien x, y Zeichenketten über dem Alphabet Σ und sei $q \in \mathbb{N}$ $(q>0)$ gegeben. Dann wird die ***q*-Gramm-Distanz** zwischen x und y definiert als

$$D_q(x,y) = \sum_{v \in \Sigma^q} |G(x)[v] - G(y)[v]|$$

Für alle x, y, z aus Σ^* gilt damit offensichtlich die Eigenschaft $D_q(x,y)=D_q(y,x)$ „Symmetrie“ und die Eigenschaft $D_q(x,y) \leq D_q(x,z)+D_q(z,y)$ „Dreiecksungleichung“ – aber nicht jedoch die Eigenschaft $D_q(x,y)=0 \Leftrightarrow x=y$ „Definitheit“. Die

Definitheit gilt nicht, da ein ungleiches Paar von Zeichenketten trotzdem eine q-Gramm-Distanz von 0 (Null) besitzen kann, wenn die beiden Zeichenketten aus genau denselben q-Grammen aber in anderer Reihenfolge zusammengesetzt sind. Damit ist die q-Gramm-Distanz im mathematischen Sinne keine Metrik, sondern eine so genannte Pseudometrik.

Verglichen mit der Edit-Distanz ist die q-Gramm-Distanz ähnlich mächtig, dabei in der Literatur jedoch deutlich seltener erwähnt. Sie wird daher, wie bereits oben angedeutet, im weiteren Verlauf dieser Arbeit nicht weiter betrachtet.

4.3 Approximative Textsuche

Das Problem der approximativen Textsuche (approximate string matching) kann formal wie folgt dargestellt werden: In einem (langen) Text $T{=}t_1...t_n$ soll jedes Auftreten eines (kürzeren) Suchmusters $P{=}p_1...p_m$ gefunden werden, wobei zwischen Fundstelle und Suchmuster k Abweichungen (Fehler) toleriert werden. Dabei stellt das gebräuchlichste Fehlermodell die im vorhergehenden Abschnitt beschriebene Edit-Distanz (ED) dar. Und in diesem Abschnitt wird ohne Beschränkung der Allgemeinheit diese String-Distanz (mit Kosten 1 für Substitution) daher als Standard-Metrik angenommen.

Wie bereits ausgeführt wurde, werden in dieser Arbeit vor allem Algorithmen für das häufige Durchsuchen von größeren, als statisch angesehenen Texten betrachtet. Der Aufbau eines geeigneten Index in der Vorverarbeitungsstufe bietet sich also an. Für eine Betrachtung von Algorithmen, welche *ohne* vorhandenen Index approximative Suchen durchführen, sei auf Navarros umfangreichen Überblick-Artikel zu genau diesem Thema verwiesen [Nav01].

Existiert nun ein Index zum Textkorpus, hängt die algorithmische Lösung des Problems der approximativen Suche von der Art dieses Index ab. Angelehnt an die Taxonomie, welche Navarro, Baeza-Yates, Sutinen und Tarhio in [NBST01] vorstellen, sollen hier nun die drei von ihnen gebildeten Klassen *Nachbarschaftsgenerierung* (neighborhood generation), *Partitionierung in exakte Suche* (partitioning into exact searching) und *Zwischen-Partitionierung* (intermediate partitioning) beschrieben werden.

Die vier Autoren haben existierende Algorithmen zur approximativen indexgestützten Suche basierend auf der verwendeten Datenstruktur in die drei genannten Algorithmen-Klassen eingeteilt. Alle Original-Arbeiten der jeweiligen Autoren aus Tabelle 4.4 finden sich in genanntem Artikel im Literaturverzeichnis angegeben. Tabellen-Zellen, welche mit „**n.m.**“ markiert sind, stehen für Datenstrukturen, die die zugehörige Algorithmus-Klasse nicht mit der benötigten In-

formation versorgen können, und daher wird diese Kombination von Datenstruktur und Algorithmus-Klasse als „nicht möglich“ angesehen.

Tab. 4.4. Klassifizierung von approximativen Index-Textsuchen (nach [NBST01])

Datenstruktur	***Algorithmus-Klasse***				
	Nachbarschaftsgenerierung	Partitionierung in exakte Suche		Zwischen-Partitionierung	
		Fehler in T	Fehler in P	Fehler in T	Fehler in P
Suffix-Tree	1991: Jokinen & Ukkonen 1993: Ukkonen 1995: Cobbs		1996: Shi		
Suffix-Array	1988: Gonnet				1999: Navarro & Baeza-Yates
q-Gramme	**n.m.**	1991: Jokinen & Ukkonen 1994: Holsti & Sutinen	1997: Navarro & Baeza-Yates		1990: Myers
q-Samples	**n.m.**	1996: Sutinen & Tarhio	**n.m.**	2000: Navarro et al.	**n.m.**

Nachdem die vier Datenstrukturen Suffix-Tree, Suffix-Array, q-Gramme und q-Samples bereits in Kapitel 3 beschrieben wurden, werden jetzt anhand ausgewählter Ansätze die drei Haupt-Algorithmenklassen für approximative Textsuche beschrieben.

4.3.1 Nachbarschaftsgenerierung

Bei der Nachbarschaftsgenerierung (neighborhood generation) wird versucht, ausgehend vom Suchmuster alle Strings zu generieren, die „ähnlich genug“ zu diesem sind (also maximal k Fehler) haben. Die Schreibweise für diese Umgebung ist:

$$U_k(P) = \{x \in \Sigma^* \mid \mathrm{ED}(x,P) \leq k\}$$

Da das Alphabet endlich ist, kann $U_k(P)$ algorithmisch erzeugt werden und die Länge eines Strings aus dieser so genannten k-Nachbarschaft kann höchstens $m+k$ betragen. Nach allen Strings aus der k-Nachbarschaft wird dann mithilfe des Index in exakter Weise gesucht.

Das Problem der Generierung der k-Nachbarschaft ist, dass mit zunehmender Alphabetsgröße und größerem k die Nachbarschaft exponentiell zunimmt. Ukkonen zeigt in [Ukk85] als Grenze auf: $|U_k(P)| = O(m^k * |\Sigma|^k)$. Für eine Alphabetsgröße von z.B. $|\Sigma|$=184, wie sie trotz Nicht-Berücksichtigung von Groß-

und Kleinschreibung für den HagerROM-Textkorpus gilt, eine erlaubte Fehlerzahl von k=3 und eine mittlere Suchmuster-Länge von $|P|$=9 Zeichen würde dies bereits die Erzeugung und anschließende Suche von über 5 Milliarden Nachbarschaftsmustern bedeuten.

Daher ist es sinnvoll, während der Nachbarschaftsgenerierung nur solche Muster zu generieren, die auch im Text vorkommen (können). Dabei kann zum Beispiel ein Suffix-Tree behilflich sein [Ukk93b], wie im Folgenden gezeigt wird.

Während man sich von der Wurzel des Suffix-Trees in alle Teilbäume vorarbeitet, berechnet man jeweils die Edit-Distanz zwischen dem aktuell durch den Tree-Knoten erzeugten Vergleichsmuster N und dem Suchmuster P. Dabei muss die Tabelle zur Berechnung der Edit-Distanz nicht jedes mal von neuem initialisiert und gefüllt werden, denn bei Abstieg oder Aufstieg im Suffix-Tree ändert sich das Vergleichsmuster N immer nur maximal am letzten Zeichen. Daher werden die Spalten der ED-Tabelle in einem Stack verwaltet, und die Berechnung der ED-Werte erfolgt in einem Backtracking-Verfahren jeweils nur mittels Neuberechnung der letzten ED-Tabellen-Spalte.

Als Abbruch-Kriterium für das Backtracking während des Abstiegs in den Tree dient der Fall $\mathrm{ED}(P,N)>k$, worauf der aktuelle Teilbaum des Trees nicht tiefer untersucht werden muss. Falls jedoch $\mathrm{ED}(P,N)\leq k$ ist, so liegt das Präfix der Länge $|N|$ von allen Suffixen, die der aktuelle Knoten repräsentiert, in $U_k(P)$.

4.3.2 Partitionierung in exakte Suche

Die Partitionierung in exakte Suche (partitioning into exact searching) geschieht unter folgender Annahme: In einem Suchmuster, welches mit einigen erlaubten Fehlern im Text vorkommt, müssen auch Stellen (Substrings) enthalten sein, die exakt (also ohne Fehler) im Text vorkommen.

Es wird also das Muster in Substrings zerlegt, welche dann mithilfe des Index schnell und exakt gesucht werden. Es müssen daraufhin also nur wenige Textstellen einer genaueren Untersuchung unterzogen werden, was der Grund dafür ist, dass diese Technik auch als *Filterung* bezeichnet wird: Die potenziell interessanten Textstellen werden von den sicher uninteressanten Textstellen getrennt. Werden genug exakte Fragmente in der richtigen Reihenfolge im Text gefunden, wird die Umgebung dieser Textstelle auf ihre Distanz zum Original-Suchmuster hin untersucht. Ist der Abstand dieser Textstelle zum Suchmuster klein genug, wird die Textstelle in die Trefferliste aufgenommen.

Die Annahme, dass die Fehler Teil des Suchmusters sind und korrigiert werden müssen, oder dass die Fehler Teil des Textes sind, führt zu unterschiedlichen Lösungsansätzen: Der auf q-Samples basierende Ansatz von Sutinen &

Tarhio [ST96] sei hier z.B. als ein Vertreter der Gattung „Fehler im Text“ genannt. Unter die Rubrik „Fehler im Suchmuster“ fällt beispielsweise der Ansatz von Navarro & Baeza-Yates [NB98] mit q-Grammen.

4.3.3 Zwischen-Partitionierung

Die Zwischen-Partitionierung (intermediate partitioning) liegt, wie die Bezeichnung bereits andeutet, zwischen den beiden anderen Algorithmus-Kategorien. Eines der Hauptprobleme der bereits beschriebenen Nachbarschaftsgenerierung ist, dass mit großen Alphabeten und großen Werten für k (Anzahl erlaubter Fehler) die Nachbarschaft schnell sehr groß wird. Die Idee der Zwischen-Partitionierung ist daher, das Suchmuster zunächst in Teile zu zerlegen und dann für diese Teile die jeweiligen Nachbarschaften zu generieren. Da die Teile kürzer sind als das Gesamtmuster, kann man den Wert von k für diese Fragment-Nachbarschaften entsprechend verkleinern, und so sinkt auch die Größe der zu jedem Teil zu generierenden Nachbarschaft.

Nach allen Mitgliedern dieser „Mini“-Nachbarschaften wird dann mit einem geeigneten Index per exakter Textsuche gesucht. Taucht aus jeder dieser Nachbarschaftsmengen mindestens ein Mitglied im Text mit richtigem Textabstand zu den Mitgliedern der anderen Nachbarschaften wieder auf, so kann der entsprechend überdeckte Textteil in die Trefferliste aufgenommen werden.

Sei m die Länge des Suchmusters und sei j die Anzahl der Fragmente, in welche das Original-Suchmuster aufgeteilt wird, so ist die Länge der Fragmente (m/j). Sei nun die Fehlerrate $a{=}k/m$ gegeben, so bleibt diese Fehlerrate identisch, wenn man für die Fragmente mit Länge (m/j) die Fehlerzahl ebenfalls durch j teilt, also $k'{=}\lfloor k/j \rfloor$ Fehler pro Fragment erlaubt.

Myers stellt in [Mye94] einen Ansatz vor, welcher mittels Zwischen-Partitionierung und einem q-Gramm-Index eine approximative Textsuche durchführt.

4.4 Probleme der bekannten Ansätze

Im Abschnitt 4.1 über phonetische Codes wurden bereits die grundsätzlichen Nachteile von Soundex und seinen Nachfolgern beschrieben. Um nun echte *Volltext*suchen durchführen zu können, werden also üblicherweise keine Soundex-Varianten verwendet, sondern Algorithmen aus den drei in den vorhergehenden Abschnitten beschriebenen Kategorien: „Nachbarschaftsgenerierung“, „Partitionierung in exakte Suche“ und „Zwischen-Partitionierung“.

Die Ansätze, die diesen Kategorien zugeordnet sind, benötigen aber eine klar definierte Abstandskenngröße, wobei üblicherweise Levenshteins Edit-Distanz verwendet wird. Die weite Verbreitung dieses Fehlermodells liegt wohl vor allem in der eindeutigen mathematischen Definition, in der leichten und relativ schnellen Berechenbarkeit und in der Kommunizierbarkeit zum Benutzer („*Abstand≤3 entspricht maximal drei Tastatur-Anschlägen*“). Als Vorteil der Edit-Distanz ist sicherlich auch zu bewerten, dass sich damit zufällige Tippfehler oder Veränderungen, die sich bei der Übertragung der Textinformation über einen fehlerbehafteten Kommunikationskanal ergeben, gut modellieren lassen. Denn diese Fehler zerstören aufgrund ihrer zufälligen Natur eventuell die Aussprache und damit den Klang eines Wortes, was wiederum die phonetische Ähnlichkeit zwischen der korrekten und falschen Variante des Wortes ändert.

Trotzdem hat die Edit-Distanz einige Schwächen darin, die Ähnlichkeit von zwei Zeichenketten aus menschlicher Sichtweise zu modellieren.

Beispiel 4-3. Phonetische Ähnlichkeiten
Edit-Distanz(`kalzium`, `calzium`) = 2 (2 Ersetzungen)
aber auch
Edit-Distanz(`kalzium`, `tallium`) = 2 (2 Ersetzungen)

Die Wortpaare aus obigem Beispiel haben dieselben Edit-Distanzen, obwohl ein Mensch sicherlich die beiden ersten Begriffe aufgrund des ähnlichen Klanges als deutlich dichter zueinander ansehen würde als die beiden Begriffe des zweiten Wortpaares.

Da die meisten Sprachen kulturhistorisch gesehen zunächst nur ein akustisches Kommunikationsmittel waren und die grafische Fixierung von Sprache in Form von Schrift sich erst deutlich später entwickelte, unterscheiden sich (bis auf die kleine Ausnahmegruppe der so genannten *Homophone*) unterschiedliche Begriffe zunächst einmal vor allem am Klang. Eine Abstandskenngröße für Zeichenketten, welche den Klang der Worte jedoch komplett vernachlässigt, kann sicherlich die Vorstellung eines Benutzers von „Ähnlichkeit“ und „Nicht-Ähnlichkeit“ von Worten nicht immer treffen.

Beispiel 4-4. Zahlenworte
Edit-Distanz(`4-blättrig`, `vier-blättrig`) = 4 (1 Ersetzung, 3 Einfügungen)
aber
Edit-Distanz(`4-blättrig`, `3-blättrig`) = 1 (1 Ersetzung)

Die Wortpaare aus obigem Beispiel haben sehr unterschiedliche Abstände. Trotzdem spiegeln die Werte der Edit-Distanzen genau das Gegenteil von dem wider, wie ein Mensch den Abstand dieser Begriffe festlegen würde. Die Zeichenketten `4` und `vier` können synonym verwendet werden und sollten daher z.B.

den Distanzwert Null erhalten. Wohingegen die Veränderung einer arabischen Ziffer in eine beliebige andere Ziffer wohl meist vermieden werden sollte.

Beispiel 4-5. Stumme Zeichen
Edit-Distanz(`hydroxyethylstärkelösung`, `hydroxy-ethyl-stärke-lösung`) = 3

Einige Zeichen (z.B. Bindestriche und Leerzeichen) verändern den Klang und die Bedeutung eines Wortes nicht. Es wäre also schlecht, das Auftreten eines solchen Zeichens mit einer Vergrößerung des Wortabstandes zu belegen, denn es könnte passieren, dass ein Anwender nur Worte mit geringem Abstand zu seinem Suchmuster toleriert und ein Wort, welches sich durch einige Bindestriche von diesem unterscheidet, wird dann nicht mehr als Treffer präsentiert.

Beispiel 4-6. Abkürzungen
Edit-Distanz(`prüflösung`, `prüflsg.`) = 4 (3 Löschungen, 1 Einfügung)
und auch
Edit-Distanz(`prüflösung`, `prüföffnung`) = 4 (1 Lösch., 1 Ersetz., 2 Einfüg.)

Wie aus obigem Beispiel ersichtlich ist, sollten auch gängige Abkürzungen als Synonyma mit dem Abstandswert Null belegt werden können, was in der Edit-Distanz so nicht modellierbar ist, denn offensichtlich gleiche Worte bekommen denselben hohen Distanzwert, den auch offensichtlich stark unterschiedliche Begriffe bekommen.

Beispiel 4-7. Rechtschreibreform, Sonderzeichen
Edit-Distanz(`darmverschluß`, `darmverschluss`) = 2
Edit-Distanz(`darmverschluß`, `darmverschleiß`) = 2

Edit-Distanz(`tübingen`, `tuebigen`) = 2
Edit-Distanz(`tübingen`, `tubingen`) = 1
Edit-Distanz(`tübingen`, `taubingen`) = 2

Obige Beispiele verdeutlichen, dass einzelne Zeichen bzw. kleine Zeichenketten mitunter als synonym anzusehen sind. Durch die Rechtschreibreform sind dies z.B. „ß“ und „ss“. Deutsche Umlaute werden von fremdsprachigen Autoren, die diese Zeichen evtl. gar nicht auf ihrer Tastatur haben, durch den lateinischen Basisbuchstaben und ein evtl. angehängtes „e“ quasi angenähert (`ä` → `ae`). Auch wenn das Beispiel hier nur deutsche Probleme aufzeigt, sind diese auch auf andere Sprachen mit lokalen Sonderzeichen übertragbar.

Die Tatsache, dass sich die bekannten Algorithmen zur approximativen Textsuche auf die Edit-Distanz als Abstandskenngröße abstützen bewirkt also, dass diese Algorithmen Treffer als „ähnlich“ zum Muster präsentieren, die es aus Sicht des Lesers nicht sind (`kalzium`, `tallium`). Und sie präsentieren Treffer gar nicht, die aufgrund zu hoher Edit-Distanz aus Sicht des Algorithmus zu weit

vom Suchmuster entfernt sind (prüflösung, prüflsg.), wobei ein Leser vielleicht sein Muster und die ausgelassenen Treffer eher dicht beieinander sehen würde.

Im Beispiel 4-8 werden einige Wortpaare, die allesamt dem Textkorpus des *Springer Lexikon Medizin (SLM)* entstammen, mit ihren Edit-Distanzen gegenübergestellt. Dies geschieht, um zu demonstrieren, dass in realen Texten durchaus Fachbegriffe vorkommen, die sich auch mit größerer Wortlänge (und damit zunehmendem Kontext innerhalb des Wortes) trotzdem mit wenigen einfachen Editier-Aktionen in andere existierende Fachbegriffe desselben Werkes umwandeln lassen. Dabei haben die gewählten Beispiele ausnahmslos weit auseinanderliegende Bedeutungen.

Beispiel 4-8. Geringe Edit-Distanzen zwischen existenten SLM-Worten
In Klammern hinter den Worten steht jeweils die Häufigkeit in [Reut04].
Beim zweiten Wort sind die Aktionen SUBST/INS fett, DEL ist unterstrichen.

eierstockpunktion	(2)	eierstock**f**unktion	(2)	ED=1
intraklavikulär	(2)	in**f**raklavikulär	(3)	ED=1
jodierung	(7)	**c**odierung	(1)	ED=1
lungenvene	(48)	**z**ungenvene	(2)	ED=1
mikrohalluzination	(3)	m**a**krohalluzination	(3)	ED=1
augenerkrankung	(23)	l**u**ngenerkrankung	(42)	ED=2
beckenabschnitt	(10)	**rü**ckenabschnitt	(1)	ED=2
klappensuffizienz	(2)	klappen**in**suffizienz	(44)	ED=2
kleinhirnzäpfchen	(3)	kleinhirnläp**p**chen	(4)	ED=2
nackenlymphknoten	(1)	**be**ckenlymphknoten	(20)	ED=2
bindegewebsknorpel	(4)	bindegewebsknot**te**n	(1)	ED=3
rachenschleimhautentzündung	(5)	**m**a**g**enschleimhautentzündung	(13)	ED=3
wiedereinlagerung	(1)	w**ass**ereinlagerung	(10)	ED=3
bindehautentzündung	(50)	**hir**n**h**autentzündung	(19)	ED=4

Abschließend sei noch ein durchaus realistisches Beispiel konstruiert, welches Kombinationen obiger Probleme beinhaltet und damit die grundsätzliche Schwierigkeit der Edit-Distanz noch einmal unterstreicht:

Beispiel 4-9. Problem-Kombination
Edit-Distanz(kalzium-lösung, calciumlsg.) = 7
und
Edit-Distanz(kalzium-lösung, kalium-löschung) = 3

Das erste Wortpaar ist wohl als Synonym anzusehen und bekommt trotzdem den unpraktikabel hohen Distanzwert von 7. Das zweite Wortpaar beschreibt durch kleine Änderungen ein vollständig anderes Wort (aus Kalzium wurde Kalium – ein anderes chemisches Element, aus Lösung wurde Löschung), welches eine weniger als halb so große Distanz zum Startmuster bekommt.

Navarro macht die Aussage, dass sich die approximativen Textsuchen, die die Edit-Distanz benutzen, leicht auf andere Fehlermodelle adaptieren lassen:

> „Most of the techniques can be easily adapted to other error models.“ ([NBST01], Seite 1).

Gleichzeitig fließen aber indirekte Annahmen über das Fehlermodell direkt in die Algorithmen ein. Zum Beispiel kann bei der Edit-Distanz aus der Tatsache, dass $ED(P,T) \leq k$ direkt eine Aussage über die Längendifferenz von P und T gemacht werden: $\texttt{abs}(|P|-|T|) \leq k$. Diese Tatsache fließt beispielsweise direkt in die schnelle Nachbarschaftsgenerierung mit ein.

Diese Annahme kann jedoch bei einem Fehlermodell, welches z.B. Synonyme, Abkürzungen und Ziffernfolgen mit geringen Abständen belegen möchte, nicht mehr gemacht werden. Folglich können Algorithmen, welche aus dem maximal gewünschten Abstand zu einem Suchmuster auf die Länge des zu suchenden Textes schließen, nicht so leicht auf ein solches Fehlermodell angepasst werden.

Ein weiterer Nachteil der Edit-Distanz ist die stets vorhandene Symmetrie-Eigenschaft dieses Abstandes: Bezogen auf zwei Zeichenketten a und b gilt hier stets: $ED(a,b)=ED(b,a)$. In bestimmten Anwendungsfällen kann es aber erwünscht sein, „günstig“ (mit geringem Abstand) von einer Schreibweise a eines Wortes zur Schreibweise b zu gelangen, aber die Rückrichtung von b nach a soll „weniger günstig“, also mit höherem Abstand, belegt sein.

Beispiel 4-10. Vorteile durch Verzicht auf Symmetrie
Ist im Suchmuster (z.B. `lymfknoten`) des Anwenders der Substring „`f`“ enthalten, so steht dieser eventuell nur stellvertretend für den Laut „f“ und es sollten verschiedene (auch seltenere und komplexe) Schreibweisen dieses Lautes gesucht werden (z.B. „`ph`“ oder „`v`“). Hat der Anwender im Suchmuster (z.B. `lymphknoten`) jedoch bereits eine eher „komplexe“ Schreibweise des Lautes „f“ gewählt, sollte diese nur unter erschwerten Bedingungen in andere Schreibweisen überführt werden.

Aufgrund der in diesem Abschnitt dargelegten Probleme der Edit-Distanz, String-Ähnlichkeiten aus einer eher semantisch-phonetischen Sicht zu modellieren und aufgrund der Tatsache, dass die üblichen Ansätze zur approximativen Textsuche auf dieses Abstandskenngröße aufbauen, soll nun im folgenden Kapitel ein neues Verfahren zur fehlertoleranten Volltextsuche vorgestellt und entwickelt werden: *Weighted Pattern Morphing*.

Kapitel 5

Weighted Pattern Morphing

Im vorangegangenen Kapitel wurden die Schwächen von üblichen Techniken zur fehlertoleranten Volltextsuche dargelegt. Insbesondere wurden die Einschränkungen von phonetischen Codes und von auf der Edit-Distanz basierenden Ansätzen beschrieben.

In diesem Kapitel wird nun die Konzipierung eines neuen Verfahrens beschrieben, welches die Nachteile der bereits beschriebenen Verfahren umgeht. Dazu wird auf eine existierende exakte Volltextsuche ein optional zuschaltbares, fehlertolerantes Modul aufgesetzt, welches keine negativen Auswirkungen auf die Laufzeit der exakten Suche hat. Das Modul berücksichtigt, dass in bestimmten Kontexten auch längere Zeichenketten als synonym oder ähnlich zueinander angesehen werden können und dass sich bei Austausch solcher Substrings die Distanz zum Original-Muster nur wenig erhöht. Auch werden Zeichenketten zu Phonem-Gruppen zusammenfassbar sein, und innerhalb dieser Gruppen von ähnlich klingenden Wortfragmenten ist der Austausch von Zeichenketten im Original-Suchmuster zu geringen Kosten möglich.

Da das neue Verfahren die Gestalt (griech. *Morphe*) des Suchmusters (engl. *pattern*) kontrolliert verändert, dabei aber nur solche Änderungen durchführt, die vom Anwender noch als tolerabel angesehen werden, wird das Verfahren als *Weighted Pattern Morphing (WPM)* bezeichnet.

5.1 Konzeptbildung

In diesem Abschnitt wird nun zunächst über formale Definitionen eine abstrakte Konzeptbildung des *Weighted Pattern Morphing* durchgeführt. Da es dabei um die iterative Anwendung von Regeln geht, die Zeichenketten verändern, wurde

wo möglich auf Schreibweisen zurückgegriffen, wie sie im Bereich der formalen Sprachtheorie und im Compilerbau verwendet werden. Als Einführung in diese Thematik seien die Standardwerke [ASU88] und [Harr78] genannt.

Definition 5-1. Eine **Submorph-Regel** $r=(\beta,\delta,g)\in(\Sigma^*\times\Sigma^*\times\mathbb{N})$ überführt die Zeichenkette β in die Zeichenkette δ mit dem so genannten **Gewicht** $g\in\mathbb{N}$ als Maß für den Abstand von β und δ, wobei $\beta\neq\delta$. Grafische Notation: $\beta\xrightarrow{g}\delta$.

Definition 5-2. Eine **Submorph-Regelmenge** $R=\{r_1,r_2,\ldots,r_n\}\subset(\Sigma^*\times\Sigma^*\times\mathbb{N})$ ist eine endliche Menge von Submorph-Regeln r_i, $1\leq i\leq n$.

Definition 5-3. Ein **Morph** ist ein Paar $U=(u,x)\in(\Sigma^+\times\mathbb{N}_0)$, wobei $u\in\Sigma^+$ die nicht-leere, endlich lange **Zeichenkette** des Morphs und $x\in\mathbb{N}_0$ die **Gewichtssumme** des Morphs ist. Ein Spezialfall ist der **Start-Morph** $S=(s,0)$, $s\in\Sigma^+$ beliebig.

Definition 5-4. Regelanwendung.
Sei $r=(\beta,\delta,g)$ eine Submorph-Regel, sei $U=(\alpha\beta\gamma,\ x)$ ein Morph dessen nicht-leere Zeichenkette aus der Konkatenation der Zeichenketten $\alpha,\beta,\gamma\in\Sigma^*$ besteht und sei weiter Position $p=|\alpha|\in\mathbb{N}_0$. Die *Regelanwendung* $A=(U,r,p)$ ist die Überführung des Morphs U in den Morph $V=(\alpha\delta\gamma,\ x+g)$. D.h. durch die Anwendung der Regel r auf der Zeichenkette von U wird an der Position p der Substring β durch den String δ ersetzt und die Gewichtssumme x um g erhöht.
Grafische Notation: $U\xRightarrow[r,p]{}V$.

Definition 5-5. Kette von Regelanwendungen.
Seien $U=(u,x)$, $W=(w,z)$ zwei Morphs und $R=\{r_1,\ldots,r_n\}$ eine Submorph-Regelmenge. Eine *Kette von j Regelanwendungen* ist die Überführung von Morph U in den Morph W durch die sequenzielle Anwendung der Regeln $r_{i1},r_{i2},\ldots,r_{ij}$ an den Positionen $p_1,p_2,\ldots,p_j$ mit: $i_1,i_2,\ldots,i_j$ ist eine Folge von Zahlen (wobei $1\leq i_k\leq n\ \ \forall k:1\leq k\leq j$).
Für die angewendeten Regeln sei: $r_l=(\beta_l,\delta_l,g_l)\in R$, $1\leq l\leq j$.
Damit ergibt sich für den Morph W die Gewichtssumme $z=x+\sum_{l=1}^{j}g_l$.
Grafische Notation: $U\xRightarrow[r_{i_1},p_1/\ldots/r_{i_j},p_j]{j}W = U\xRightarrow[r_{i_1},p_1]{}V_1,\ V_1\xRightarrow[r_{i_2},p_2]{}V_2,\ \ldots,\ V_{j-1}\xRightarrow[r_{i_j},p_j]{}W$
Als Spezialfall existiert die leere Kette von Regelanwendungen: $S\xRightarrow{0}S$

Definition 5-6. Eine **Morpher-Sprache** $L(R,S)$ wird erzeugt von einer Submorph-Regelmenge $R=\{r_1, r_2, \ldots, r_n\}$ aus einem Start-Morph $S=(s,0)$.
Dabei ist $L(R,S)\subset(\Sigma^*\times\mathbb{N}_0)$ die Menge von Morphs mit:
$L(R,S)=\{ W=(w,z)\in(\Sigma^*\times\mathbb{N}_0) \mid \exists$ eine Kette von Regelanwendungen aus R mit $S\xrightarrow[r_{i_1},p_1/\ldots/r_{i_j},p_j]{j} W$ mit $j\in\mathbb{N}_0\}$

Aufgrund ihrer Konstruktion umfasst eine Morpher-Sprache immer mindestens einen Morph: den Start-Morph. Abhängig von der Regelmenge und der Zeichenkette des Start-Morphs enthält die Morpher-Sprache nur den Start-Morph, oder den Start-Morph und *endlich* viele weitere Morphs, oder den Start-Morph und *unendlich* viele weitere Morphs, wie Beispiel 5-1 illustriert.

Beispiel 5-1. Morpher-Sprachen zum Start-Morph S=(kalzium,0)

R	***L(R,S)***
{b $\xrightarrow{2}$ p}	{(kalzium,0)}
{k $\xrightarrow{1}$ c, z $\xrightarrow{1}$ c}	{(kalzium,0), (calzium,1), (kalcium,1), (calcium,2)}
{i $\xrightarrow{3}$ ie}	{(kalzium,0), (kalzieum,3), (kalziieum,6), (kalziiieum,9), ...}

O.B.d.A. seien die Elemente von $L(R,S)=\{(w_1,z_1),(w_2,z_2),\ldots\}$ aufsteigend nach ihrer jeweiligen Morph-Gewichtssumme sortiert (bei Gleichheit nach Zeichenkette). Mit $|L|$, der Anzahl von Morphs in L, gilt also: $z_i\le z_k,\ 1\le i\le k\le|L|$. Diese Sortierung kann z.B. mittels „Sortieren durch Einfügen“ des jeweils generierten Morphs in die Menge der bisher generierten Morphs erfolgen.

Es soll nun gezeigt werden, dass $|L|$ zwar möglicherweise unendlich sein kann, es zu einer gegebenen Gewichtssumme g aber niemals unendliche viele Morphs innerhalb von L geben kann.

Hilfssatz 5-1: Sei $R=\{r_1, r_2, \ldots, r_n\}$ eine Regelmenge und $S=(s,0)$ ein Start-Morph. Dann gibt es in der Morpher-Sprache $L(R,S)$ zu einem gegebenen Gewicht g nur *endlich viele* oder *keine* Morphs, deren Gewichtssummen identisch sind mit g.

Beweis: Angenommen es existiere eine Gewichtssumme g, sodass *unendlich* viele Morphs aus $L(R,S)$ den Wert g als Gewichtssumme erhalten würden.
Da das minimale Gewicht einer Regel 1 ist (vgl. Def. 5-1) und die Gewichtssumme eines Morphs W aus einer Kette von j sequenziellen Regelanwendungen $S\xrightarrow[r_{i_1},p_1/\ldots/r_{i_j},p_j]{j} W$ entstanden sein muss (für ein $j\in\mathbb{N}_0$), können maximal endlich viele einzelne Regeln in einer Kette der Länge g angewendet worden sein.

Um nun mit maximal g sequenziellen Regelanwendungen mit einem minimalen Regelgewicht von 1 unendlich viele Morphs mit demselben Gewicht g zu erzeugen, müsste entweder die Regelmenge R unendlich groß sein, oder die Zeichenkette s des Start-Morphs müsste unendlich lang sein. Ersteres ist jedoch ein Widerspruch zu Def. 5-2 und letzteres steht im Widerspruch zu Def. 5-3.

Aus diesen Widersprüchen folgt, dass nur endlich viele Morphs eine vorgegebene Gewichtssumme g innerhalb einer Morpher-Sprache $L(R,S)$ haben können. ■

Da die Generierung der Morpher-Sprache (einer möglicherweise unendlichen Menge von Morphs) auch gegebenenfalls unendlich lange dauern würde, werden für die praktische Umsetzung des Morph-Konzeptes nun einige Limitierungen gewählt.

Definition 5-7. Eine **Morpher-Konfiguration** ist ein 4-Tupel $K=(S,a,t,b)$ mit:

$S=(s,0)$	Start-Morph
$a\in\mathbb{N}_0$	maximale Anzahl erlaubter Regelanwendungen, ausgehend von S
$t\in\mathbb{N}_0$	vorgegebene Gewichtssumme
$b\in\mathbb{N}$	vorgegebener Morph-Index (in sortiertem L)

Definition 5-8. Ein **Morpher** ist ein Paar $M=(R,K)$ mit:

R	Submorph-Regelmenge
K	Morpher-Konfiguration

Definition 5-9. Morpher-Ergebnis mit Ziel-Morphs:
Zu einem Morpher $M=(R,K)$ mit $K=(S,a,t,b)$ sei $E(M)\subset L(R,S)$ das Morpher-Ergebnis. Die Elemente aus der Morpher-Sprache $L(R,S)=\{(w_1,z_1),(w_2,z_2),...\}$ seien aufsteigend nach Gewichtssumme sortiert.
Das Morpher-Ergebnis ist die Menge bestehend aus den Ziel-Morphs:

$$E(M)=\{W=(w,z)\in L(R,S)\,|\,(\mathrm{i})\wedge(\mathrm{ii})\wedge(\mathrm{iii})\}$$ mit den Bedingungen:

(i) für die Anzahl j angewendeter Submorph-Regeln, die S in W überführen: $j\le a$

(ii) das Gewicht z ist die minimale Gewichtssumme, um S in W zu überführen.

(iii) für das Gewicht z gilt: $\begin{cases} z\le t & \text{falls } b>|L| \\ z\le \min(t,z_b) & \text{sonst} \end{cases}$

In Definition 5-9 garantiert Bedingung (i), dass auf den Ziel-Morphs (ausgehend von Start-Morph S) nicht mehr als j Submorph-Regelanwendungen durchgeführt wurden. Bedingung (ii) verhindert, dass im Morpher-Ergebnis derselbe Morph-String (mit unterschiedlichem Gewicht) mehrfach auftaucht und damit andere eventuell interessante Morph-Strings – aufgrund von Bedingung (iii) – nicht mehr in die Ergebnismenge aufgenommen werden können. Bedingung (iii) berücksichtigt einerseits die aus der Morpher-Konfiguration vorgegebene Ge-

wichtssumme ($z \leq t$) und nimmt bezogen auf den Parameter b alle die Morphs noch mit in die Ergebnismenge auf, die dasselbe Gewicht wie der Morph mit Index b in L besitzt ($z \leq z_b$). Durch die Minimum-Bildung dieser beiden Teilbedingungen kann es natürlich vorkommen, dass der Morph-Index b oder das Gewicht t erreicht werden (vgl. Abb. 5.1). Die Endlichkeit des Morpher-Ergebnisses ist durch Hilfssatz 5-1 gezeigt, da in die Menge nur endlich viele Gewichtsniveau-Stufen mit jeweils endlich vielen Morphs aufgenommen werden.

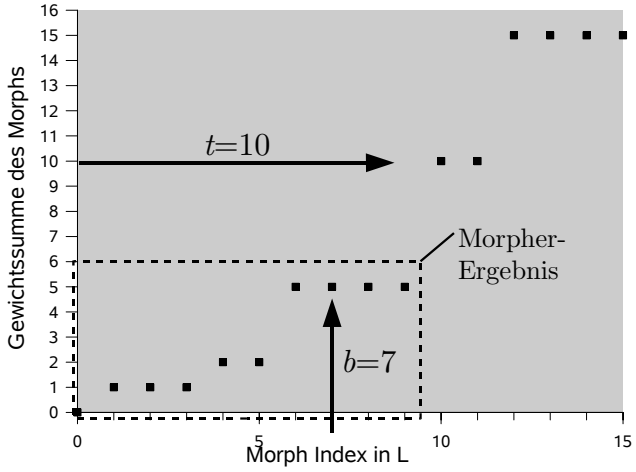


Abb. 5.1. Beispiel: Treppenstufen in $L(R,S)$ durch identische Morph-Gewichte und Bildung des Morpher-Ergebnisses durch Parameter t und b.

In der Morpher-Sprache $L(R,S)$ sind die Morphs aufsteigend nach Gewichtssumme sortiert, wobei es zu einem Gewichtswert eventuell mehrere Morphs geben kann. Das Diagramm, welches über alle Morphs aus L den Index in L auf die Gewichtssumme des Morphs abbildet, ähnelt also einer Treppenfunktion (vgl. Abb. 5.1). Da über die Reihenfolge der Morphs mit gleicher Gewichtssumme keine Aussage gemacht wurde, würde ein striktes Abschneiden der Morphs ab Index b dazu führen, dass aufgrund der Stufen-Eigenschaft eventuell interessante Morphs mit demselben Gewicht nicht in das Morpher-Ergebnis aufgenommen würden. Daher garantiert in Definition 5-9 die Teilbedingung ($z \leq z_b$) aus Bedingung (iii), dass alle Morphs auf demselben Gewichtsniveau ebenfalls in das Morpher-Ergebnis übernommen werden. Diese Teilbedingung kommt jedoch nur zum Einsatz, wenn der Start-Morph S und die anderen Parameter aus der Morpher-Konfiguration entsprechend gewählt wurden.

Mithilfe einer Regelmenge bestehend aus gewichteten Regeln und mit geeigneten Parametern a, t und b als Bestandteil der Morpher-Konfiguration können nun Ketten von Regelanwendungen sequenzielle atomare Substring-Ersetzungen durchführen und so Varianten des Original-Suchmusters (die Zeichenkette des Start-Morphs) generieren. Aus dieser möglicherweise unendlich großen Morpher-Sprache wird über die limitierende Morpher-Konfiguration eine endliche Aus-

wahl von Morphs in das Morpher-Ergebnis übernommen: Die erzeugten Morphs erhalten mit ihrer Gewichtssumme ein Maß für den Abstand zum Start-Morph und so können die Morphs, deren Abstand akzeptabel erscheint, anschließend in einem Textkorpus gesucht werden. Durch dieses Vorgehen erhält man eine Suche, die nicht nur bei exakter String-Übereinstimmung Treffer liefert sondern auch bei Strings, die zum Suchmuster in konfigurierbarer Weise „ähnlich genug“ sind: Man erhält also eine fehlertolerante Volltextsuche.

Auf Basis der oben stehenden Definitionen wird das Konzept des WPM-Verfahrens nun wie folgt zusammengefasst:

1.) Es wird eine Regelmenge festgelegt, deren gewichtete so genannten Submorph-Regeln atomare Substring-Ersetzungen beschreiben. Dabei dürfen linke und rechte Seite der Regel aus mehr als einem Symbol bestehen.

2.) Basierend auf einem Original-Suchmuster des Benutzers wird der so genannte Start-Morph mit dem Gewicht 0 gebildet.

3.) Der Start-Morph bildet zusammen mit einigen limitierenden Parametern die so genannte Morpher-Konfiguration.

4.) Durch sequenzielles Anwenden von Submorph-Regeln an unterschiedlichen Positionen der Zeichenkette des Start-Morphs (und der daraus entstehenden Morphs) werden Varianten des Original-Suchmusters gebildet, deren Gewichtssummen als Maß für die Ähnlichkeit zum Original-Suchmuster gelten.

5.) Aus der potenziell unendlich großen Menge der möglichen Morphs werden nur diejenigen ausgewählt, die innerhalb der durch die Morpher-Konfiguration gesetzten Grenzen fallen.

6.) Die so ausgewählten Morphs werden an eine nachgeschaltete exakte Volltextsuche zur Ermittlung etwaiger Trefferpositionen im Textkorpus übergeben.

Gesucht wird nun also ein Algorithmus, der zu einer gegebenen Submorph-Regelmenge und einer gegebenen Morpher-Konfiguration das entsprechende Morpher-Ergebnis berechnet. Die Zeichenketten aller Morphs aus dieser Ergebnismenge sollen dann mit einer exakten Volltextsuche gesucht werden, um die fehlertoleranten Treffer zur Zeichenkette des Start-Morphs zu ermitteln.

5.2 Vorbereitungen für das WPM-Verfahren

In diesem Abschnitt wird nun aufbauend auf die Definitionen aus dem vorhergehenden Abschnitt das Verfahren beschrieben, eine Morpher-Ergebnismenge zu berechnen. Dazu wird zunächst dargelegt, wie die Submorph-Regeln erstellt und

für den WPM-Algorithmus aufbereitet werden. Anschließend wird der eigentliche WPM-Algorithmus erläutert.

5.2.1 Erstellung der Submorph-Regelmenge

Obwohl die Submorph-Regeln eigentlich nur Quell-String, Ziel-String und Gewicht beinhalten, wurden die Regeln zur leichteren Wartbarkeit in so genannte *Submorph-Gruppen* (auch: Phonem-Gruppen oder Synonym-Gruppen) eingeteilt. Diese Einteilung ist vor allem bei der manuellen Erstellung der Regeln hilfreich, da man hier dann mit kleinen, übersichtlichen Tabellen arbeiten kann (vgl. Tab. 5.1). In diesen Tabellen stehen in den Zeilen die Quell-Strings, in den Spalten die Ziel-Strings und in den Zellen die Gewichte.

Die Gewichte regeln dabei, mit welchen „Kosten“ die Zeichenketten ineinander überführt werden dürfen. Tabelle 5.1 zeigt beispielsweise, dass der Substring „c“ mit Gewicht 1 in den Substring „k“ überführt werden darf, welches der Submorph-Regel $\texttt{c} \xrightarrow{1} \texttt{k}$ entspricht.

Tab. 5.1. Beispiel-Tabellen mit Gewichten zu Submorph-Gruppen (c/g/k/..) und Ziffern

	c	**g**	**k**	...
c	–	–	1	...
g	10	–	10	
k	1	15	–	
...	...			

	1	**ein**	**2**	...
ein	1	–	–	...
1	–	1	–	
zwei	–	–	1	
...	...			

Wie aus der Tabelle ersichtlich wird, ist nicht jeder mögliche Submorph auch mit einem Gewicht belegt. Ein „–“ als Gewicht bedeutet, dass dieser Submorph nicht erlaubt ist. Weiterhin ist die Tabelle nicht zwingend symmetrisch bezüglich der Diagonalen von oben links nach unten rechts. So lassen sich asymmetrische String-Ersetzungen steuern. In Tabelle 5.1 wird beispielsweise die Ersetzung von „k“ nach „g“ mit Gewicht 15 belegt, aber die Rückrichtung „g“ nach „k“ nur mit Gewicht 10. Dies ist ein Vorteil gegenüber der Edit-Distanz, welche grundsätzlich alle Editier-Aktionen nur symmetrisch durchführt (vgl. Bsp. 4-10, S. 44). Der Verzicht auf Symmetrie bewirkt jedoch auch, dass die WPM-Gewichtssumme keine Metrik (Distanz) im mathematischen Sinne ist, da hierfür die Symmetrie-Eigenschaft stets gegeben sein muss (vgl. z.B. [Gerh02], S. 75). Diese Beobachtung hat jedoch keinerlei negative Auswirkung auf die praktische Nutzbarkeit des WPM-Verfahrens.

Die Zusammenfassung der Submorph-Regeln zu Gruppen und die Speicherung dieser Gruppen in TAB-separierten Textdateien macht neben der einfachen au-

tomatisierten Weiterverarbeitung dieser Tabellen auch die manuelle Erstellung und Bearbeitung der Submorph-Regeln leicht möglich. Die verwendeten Steuerdateien können mit Standard-Tabellenkalkulationsprogrammen wie Microsoft Excel oder StarCalc aus der Bürosuite OpenOffice/StarOffice bearbeitet werden, wie Abbildung 5.2 illustriert. So kann die Bearbeitung der Regeln nicht nur vom Software-Entwickler sondern auch von einem Werkautor, einem Fachlektor oder einem betreuenden Produkt-Manager durchgeführt werden. Es wäre prinzipiell sogar möglich, dem Endnutzer hier Eingriffsmöglichkeiten zu gewähren und ihm die Möglichkeit zu geben, eigene Regeln zu erstellen oder vorhandene Regeln zu überarbeiten (Änderung des Gewichtes) oder zu löschen. Die Möglichkeit der Benutzer-Regeln wurde in den in dieser Arbeit vorgestellten Projekten jedoch noch nicht umgesetzt und bleibt so Teil des Ausblick-Kapitels.

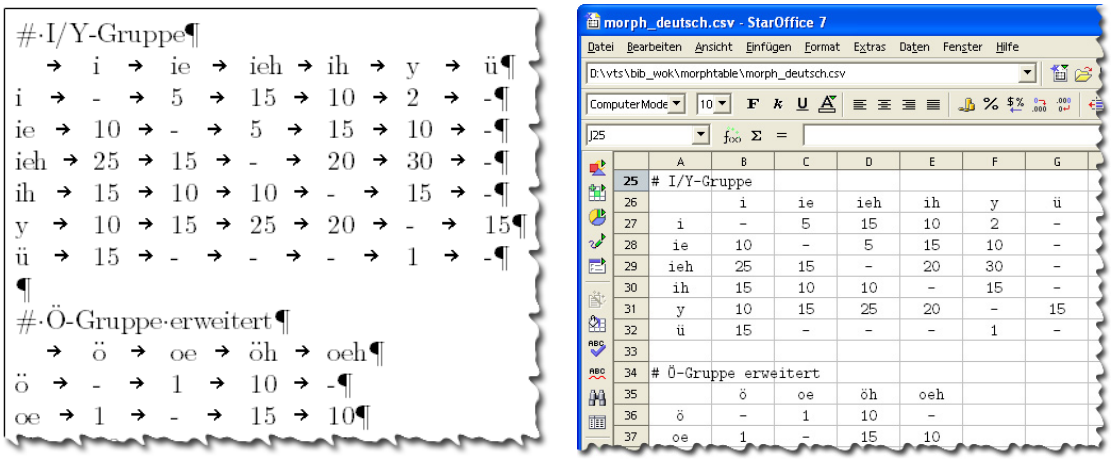


Abb. 5.2. Bearbeitung von Submorph-Gruppen mit Standard-Office-Software

Da die Erstellung der Submorph-Regeln und dabei vor allem die Definition zusammengehöriger Quell- und Ziel-Strings eine gute Kenntnis der zugrunde liegenden Sprache und eine Vorstellung über verschiedene Schreibweisen gleicher Klangfragmente erfordert, wurde darauf verzichtet, die Submorph-Regeln vollständig automatisch zu erstellen. Für die deutsche Sprache, gemischt mit lateinischen und griechischen Termini, wurden daher ca. 360 Submorph-Regeln manuell definiert. Details dazu finden sich im Anhang.

Neben der für Muttersprachler noch bewältigbaren Aufgabe, Quell- und Ziel-String zu ähnlich klingenden oder synonym verwendeten Buchstabengruppen zu finden, ist die Aufgabe, passende (und konsistente) Gewichte für diese Paare zu definieren, nicht so einfach und eher zeitaufwändig. Daher wird in der vorliegenden Ausarbeitung auch ein Verfahren präsentiert, welches die Gewichte einer bestehenden Submorph-Regelmenge einer Feinjustierung unterzieht und die Regeln so automatisiert an einen Textkorpus anpasst (siehe dazu Abschnitt 5.8 ab Seite 78).

In dieser Arbeit soll auf die Regelgenerierung nicht weiter eingegangen werden, es soll vielmehr beschrieben werden, wie der WPM-Algorithmus bei gegebener Submorph-Regelmenge aus einem Start-Morph eine Menge von Ziel-Morphs generiert. Dazu müssen die Submorph-Regeln zunächst in ein vom WPM-Algorithmus schneller verarbeitbares Format umgewandelt werden.

5.2.2 Linearisierung der Submorph-Regelmenge

In folgenden Abschnitt wird das Vorgehen beschrieben, wie man von der oben beschriebenen tabellarischen Darstellung der Submorph-Regeln, welche vor allem für die manuelle Erstellung und Pflege optimiert ist, nun zu einer Darstellung gelangt, die für den Algorithmus einen schnellen Zugriff auf passende Regeln bietet. Diese Formatänderung wird *Regel-Linearisierung* genannt.

Die Linearisierung erfolgt mit dem in der Sprache perl [wwwPerl] verfassten Skript `morphini.pl`. Dieses Skript extrahiert aus den oben vorgestellten, TAB-separierten Regel-Textdateien die einzelnen Regeln und generiert daraus für verschiedene Programmiersprachen (perl und C++) Anweisungen zur Initialisierung der linearisierten Regellisten. Diese generierten Quellcode-Dateien werden dann in das WPM-Programm mit eincompiliert (oder als perl-Modul zur Laufzeit eingebunden). Im Weiteren vorgestellt werden nun nur die C++-Anweisungen; für andere Programmiersprachen funktioniert dies analog.

Die vom perl-Skript generierten C++-Anweisungen lassen sich in vier Bereiche gliedern, welche im Folgenden genauer erklärt werden:

1.) *Definition von Maximalwerten.*
Hier werden z.B. die Integer-Variablen `maxFromLength` und `maxToLength` gesetzt, welche dem Algorithmus später mitteilen, wie lang ein Quell- oder Ziel-String in dieser Submorph-Regelmenge maximal werden kann. Auch die Gesamtzahl der Regeln `lastArrayIndex` wird hier festgelegt.

2.) *Speicher-Reservierung* (optional, in perl nicht benötigt).
Hier wird Speicherplatz für die linearisierte Darstellung aller gefundenen Regeln belegt.

3.) *Initialisierung der Regelintervalle* (1. Zeichen der Quell-Strings).
Nach Ermittlung aller Regeln aus allen Tabellen werden die Regeln sortiert. Erster Sortierschlüssel ist der alphanumerische Wert des Quell-Strings der Regel. Zweiter Sortierschlüssel ist das Gewicht der Regel. Es werden dann Regelintervalle gebildet und mit Start- und Stopp-Indexwerten belegt, die dem Algorithmus darüber Auskunft geben, in welchem Bereich Regeln liegen, die Quell-Strings haben, die mit einem bestimmten Zeichen beginnen.

Beispiel aus C++:

```
startcount[(unsigned char)'k'] = 208;
stopcount [(unsigned char)'k'] = 219;
startcount[(unsigned char)'l'] = 220;
stopcount [(unsigned char)'l'] = 221;
```

Findet der Algorithmus also in einem String ein „k“, evtl. gefolgt von einigen weiteren Zeichen, so müssen nur die Regeln im Intervall [208, 219] überprüft werden, ob ihre Quell-Strings dem Substring des Musters beginnend mit diesem „k“ entsprechen. Alle anderen Regeln brauchen nicht berücksichtigt werden – denn ihr erstes Zeichen des Quell-Strings passt bereits nicht. Die Regelintervalle dienen also vor allem der Beschleunigung, indem sie den Suchraum der anwendbaren Regeln einschränken.

4.) *Attribute für alle Submorph-Regeln setzen.*
Jede Regel speichert die folgenden sechs Attribute. Die Attribute werden für alle Regeln jeweils in einem Array gleichen Typs verwaltet. Bis auf `from`, `to` und `penalty` werden alle Regelattribute vom perl Skript automatisch berechnet.

`char*`	`from`	der Quell-String
`char*`	`to`	der Ziel-String
`int`	`penalty`	das Gewicht
`bool`	`sameprefix`	ist `from` Präfix von `to`?
`bool`	`samepostfix`	ist `from` Postfix von `to`?
`int`	`fromlength`	Länge des Quell-Strings
`int`	`tolength`	Länge des Ziel-Strings

Das folgende Beispiel zeigt den C++-Quellcode, der für zwei Regeln aus der „k“-Phonemgruppe gebildet wurde.

Beispiel 5-2. Regel 208 und Regel 219 (Deutsch)

```
//////////////////// i=208
from[i] = new char[2];       strcpy(from[i], "k");
to[i]   = new char[2];       strcpy(to[i]  , "c");
penalty[i]    = 1;
sameprefix[i] = 0;           samepostfix[i] = 0;
fromlength[i] = 1;           tolength[i++]  = 1;
//////////////////// i=219
from[i] = new char[2];       strcpy(from[i], "k");
to[i]   = new char[3];       strcpy(to[i]  , "kk");
penalty[i]    = 10;
sameprefix[i] = 1;           samepostfix[i] = 1;
fromlength[i] = 1;           tolength[i++]  = 2;
```

Beispiel 5-2 zeigt, wie bei Regel 219 die Wahrheitswerte von `sameprefix` und `samepostfix` gesetzt werden. Dies dient vor allem dazu, diese Regeln am Suchmuster-Anfang (`sameprefix=1`) bzw. am Suchmuster-Ende (`samepostfix=1`) nicht anzuwenden. Denn da hinter den WPM-Algorithmus eine exakte Volltext-

suche geschaltet wird, die jeden Substring im Textkorpus findet, würden dem Anwender durch eine solche Regelanwendung an den Suchmuster-Rändern keine neuen Fundstellen gezeigt werden, die er durch sein Original-Suchmuster nicht auch schon bekommen hätte, was Beispiel 5-3 illustriert.

Beispiel 5-3. Anwendung des `samepostfix`-Attributes
Sei das Suchmuster **`kxxxxxxx`** für das eine fehlertolerante Volltextsuche durchgeführt werden soll. Würde nun obige Regel 219 (ohne Berücksichtigung des `samepostfix`-Attributes) angewendet, so würde der Ziel-Morph **`kk`**`xxxxxxx` entstehen. Dieser Morph hätte aber bei einer Substring-Suche nur weniger oder gleichviele Treffer verglichen mit dem Original-Suchmuster. Würden gleichviele Treffer gefunden, so würden diese Trefferstellen in den Texten genau alle Trefferstellen des Original-Suchmusters überdecken. Daher wird die Anwendung dieser Regel durch `samepostfix=1` am Anfang des Suchmusters verhindert.
Aber bei einem Suchmuster `xxxx`**`k`**`xxxx`, welches den Quell-String der Regel 219 nicht am Rand hat, kann der entstehende Morph `xxxx`**`kk`**`xxxx` jedoch plötzlich zu Treffern führen, die vorher nicht gefunden werden konnten.

5.2.3 Filterung unwichtiger Morphs

Trotz manuell generierter Submorph-Regelmengen mit optimierten Gewichten kommt es beim Ersetzen von Substrings in Suchmustern zwangsläufig zu Morphs, welche man nicht mehr als sinnvolle Worte bezeichnen kann und die daher meist auch nicht in den Werken vorkommen. Diese Morphs sind ein Nebeneffekt, der zunächst nicht als störend erscheint, da die hinter dem Morpher angeordnete exakte Volltextsuche nur die Morphs findet, die auch im Text vorkommen und alle anderen Morphs dem Endnutzer nicht präsentiert werden.

Beispiel 5-4. Bei der Suche nach `zytostatikum` (46 Treffer) in *HagerROM* produzieren folgende zwei Morphs ebenfalls Treffer im Werk: `cytostatikum` (16) und `cytostaticum` (5). Die hinterlegte Submorph-Regelmenge produziert jedoch beispielsweise auch die folgenden Morphs, die nicht im Werk enthalten sind:

`zytostatikkum`
`zytostathikum`
`zytostattikum`
`zitostatikum`
...

Ein q-Gramm-basiertes exaktes Volltextsuche-Verfahren kann schnell entscheiden, ob ein Suchwort überhaupt im Text vorkommt oder nicht (vgl. [Ess03b], S. 440). Dies liegt darin begründet, dass nach Aufteilen des Suchbegriffes in gültige (d.h. in den Index aufgenommene) q-Gramme, die möglichst lang sind und sich möglichst wenig überlappen, als temporäre Ergebnis-Trefferliste die Trefferliste des q-Gramms gesetzt wird, welches die kürzeste Trefferliste hat. Dies ist

also das q-Gramm, das am seltensten im Text vorkommt. Im Extremfall ist diese Liste leer – es existiert also ein q-Gramm im Suchmuster, welches nicht im Text vorkommt. An dieser Stelle kann der exakte Suchalgorithmus bereits terminieren, denn das gesamte Suchmuster kann also auch nicht im Text vorkommen (vgl. [Gri01], Abschnitt 5.3.1). Aber selbst wenn alle q-Gramme eines Suchbegriffes Trefferlisten mit Länge ungleich Null besitzen, wird mit der kürzesten Trefferliste begonnen, und falls die Reihenfolge der q-Gramme so im Text nicht vorkommt, kann der Algorithmus sehr schnell zu immer kürzeren und schließlich leeren Treffermengen kommen.

Es wäre also denkbar, dass man alle vom Morpher generierbaren Morphs auch an die q-Gramm Suche weiterreicht und sich auf deren Fähigkeit verlässt, nicht existente Morphs schnell auszufiltern. Allerdings gibt es zwei Nachteile bei diesem Vorgehen:

1.) Da der Morpher am Suchmuster an mehreren Positionen mehrfache Regelanwendungen durchführen muss, bietet es sich an, dies mit einem rekursiven (Backtracking)-Algorithmus zu lösen (siehe folgender Abschnitt). So kann der Codeteil, der bereits Morphs durch Anwendung von Submorph-Regeln generiert hat, diese wieder rekursiv an sich selbst übergeben, um weitere Regelanwendungen durchzuführen. Dabei wird aber – speziell bei längeren Suchmustern – unnötig Rechenzeit verschwendet, wenn an einem Morph nur noch im hinteren Teil Veränderungen durchgeführt werden, wobei gleichzeitig der vordere Teil des Morphs nicht Bestandteil des Textes ist. Denn dieser Morph wird auch durch beliebige Änderungen am Ende nie im Text gefunden. Es wird also unnötig Rechenzeit verbraucht.

2.) Der Anwender soll festlegen können, dass von allen generierten Morphs nur die b besten auch wirklich gesucht werden (vgl. Def. 5-7 und Def. 5-9, Seite 48). Dies sind die Morphs mit den niedrigsten Strafgewichten. Daher kann es passieren, dass besser bewertete Morphs, die nicht im Text vorkommen, solche Morphs nach hinten verdrängen, die schlechter bewertet wurden – aber im Text vorkommen. Bei niedrig gewähltem b kann es dann passieren, dass diese interessanten Morphs durch ihr relativ höheres Strafgewicht überhaupt nicht gesucht und damit dem Endnutzer nicht als gefundene Variante seines Suchwortes präsentiert werden.

Als Lösungsmöglichkeit für die obigen Probleme bietet sich an, frühzeitig solche Morphs auszufiltern, die nicht im Text vorkommen können. Um hierzu aber nicht den umfangreichen q-Gramm-Index und die vergleichsweise aufwändige nachgeschaltete Volltextsuche zu benutzen, bietet sich eine *Trie-Datenstruktur* an, welche Ketten von Zeichen aus dem Textkorpus speichert. Dabei soll in der Datenstruktur aber nur die Tatsache vermerkt werden, dass diese Zeichenkette mindestens einmal im Werk vorkommt. Es handelt sich also um einen *boole-*

schen Trie, der z.B. keine genauen Positionsangaben zu den gespeicherten Zeichenketten verwaltet. Es werden also nur q-Gramm-Klassen, nicht aber q-Gramm-Instanzen (vgl. Abschnitt 3.3) im Trie gespeichert.

Da diese Datenstruktur der Verkürzung der Laufzeit des Morphers dienen soll ist es vorteilhaft, die Struktur so kompakt wie möglich zu halten, damit sie während der Morph-Generierung komplett im Arbeitsspeicher gehalten werden kann. Es wird daher darauf verzichtet, einen kompletten Suffix-Trie aufzubauen – stattdessen werden alle vorkommenden q-Gramme des Textes im Trie gespeichert.

Damit der Trie selbst für große Werke im Arbeitsspeicher gehalten werden kann lässt sich an zwei Parametern, die während der Trie-Generierung festgelegt werden, die Filter-Genauigkeit und damit auch der Speicherplatzbedarf verringern. Beide Parameter steuern direkt die Anzahl unterschiedlicher q-Gramm-Klassen, die im Trie gespeichert werden müssen. So kann das q für den Filter-Trie größer gewählt werden als für den Index der nachgeschalteten exakten q-Gramm-Volltextsuche.

- **Alphabet-Umfang.** Über diesen Parameter lässt sich einstellen, wieviele Symbole überhaupt nur vom Trie unterschieden werden. Im Text häufig vorkommende Zeichen werden exakt abgespeichert, Gruppen von seltenen Zeichen werden jeweils zu einem Metazeichen zusammengefasst und nur das Metazeichen wird im Trie abgespeichert. Zum Beispiel kann ein Metazeichen für Ziffern, eines für griechische Zeichen und eines für mathematische Operatoren verwendet werden. Für große Werke wie *HagerROM* werden zum Beispiel nur 32 verschiedene Symbole unterschieden, obwohl *HagerROM* über 160 verschiedene Symbole im Textkorpus verwendet (vgl. Kapitel 9 ab Seite 169).
- **Trie-Tiefe.** Die maximale Trie-Tiefe steht im direkten Zusammenhang mit q. Für große Werke kann die maximale Trie-Tiefe auf q=5 oder q=6 beschränkt werden. Für Werke mit geringerem Umfang sind auch höhere Werte wie z.B. q=7 oder q=8 vertretbar (vgl. auch Kapitel 9 und [Ess04], S. 345).

Soll nun mit dem booleschen Filter-Trie (maximale Tiefe sei q) ermittelt werden, ob eine Zeichenkette P möglicher Bestandteil eines Werkes ist, so wird die Zeichenkette in möglichst viele (überlappende!) solche Fragmente der Länge q zerlegt. Ist auch nur eines dieser Fragmente nicht im Trie enthalten, so kann die Zeichenkette P nicht im Werk enthalten sein. Ist $|P| \leq q$, so kann das komplette Muster im Trie gesucht werden. Ist es enthalten, so ist auch P im Werk enthalten. Gilt jedoch $|P| > q$ und sind alle Fragmente im Trie abgelegt, so ist P nur *möglicherweise* im Werk enthalten. Seine Fragmente kommen eventuell im Werk nie genau in der Reihenfolge vor, wie sie in P enthalten sind. Der Filter-Trie kann also nur bezüglich Nicht-Existenz eine hinreichende Aussage machen.

Wurde P durch eine Regelanwendung an Position p erzeugt, so trifft der im folgenden Abschnitt vorgestellte WPM-Algorithmus mithilfe des Tries folgende Entscheidungen:

1.) Sind alle Fragmente von P im Trie enthalten, so wird
   (a) P zur temporären Ergebnismenge des Morphers hinzugefügt.
   (b) P rekursiv an die Morpher-Methode übergeben, um an anderen Positionen weitere Veränderungen durchzuführen.

2.) Ist auch nur ein Fragment von P nicht im Trie enthalten, so wird überprüft, ob alle Fragmente von `substr`(P,1,p) im Trie enthalten sind. Wenn ja, wird analog zu 1.)(b) P rekursiv an die Morpher-Methode übergeben, um an anderen Positionen weitere Regelanwendungen durchzuführen. Allerdings wird P selber nicht zur temporären Ergebnismenge hinzugefügt.

Besteht der zu überprüfende Morph nur aus Symbolen des Alphabets, so sind die Pfade, mit denen seine Fragmente im Trie überprüft werden müssen, eindeutig. Eine Schwierigkeit ergibt sich jedoch, falls im fraglichen Morph auch Fragezeichen-Platzhalter (`?`) enthalten sein dürfen. In diesem Fall muss man an der Stelle, an der das Fragezeichen im Prüffragment enthalten ist, mit einem Backtracking-Verfahren in den Trie absteigen und im worst-case jeden Teilbaum unterhalb des Zeichens, welches vor dem Fragezeichen steht, überprüfen.

Die Vorgehensweise, alle q-Gramm-Fragmente des zu überprüfenden Morphs einzeln mit dem Trie auf Existenz zu überprüfen, ist aus Sicht der Laufzeit-Optimierung verbesserungsfähig. Da die Fragmente sich stark überlappen, wird bei Prüfung eines Folgefragments zunächst (bis auf das erste und letzte Zeichen) eine identische Zeichenfolge überprüft.

> **Beispiel 5-5. Redundanz bei Trie-Filterung**
> Sei der zu überprüfende Morph `aaba` und ein Trie mit den 3-Grammen eines Textes gegeben. So sind die beiden 3-Gramme des Morphs: `aab` und `aba`. Diese haben den Substring `ab` (einmal als Präfix, einmal als Suffix) gemeinsam. Auch wenn sich die Pfade im Trie durch das Präfix `a` des ersten Fragments unterscheiden, so werden (falls *beide* Fragmente im Trie enthalten sind) doch zwei Pfade mit den gleichen Verzweigungsentscheidungen im Trie abgelaufen. Je größer das Alphabet und je tiefer der Trie, desto mehr redundante Berechnungen dieser Art müssen durchgeführt werden.

Damit also nicht bei jedem neuen Prüffragment wieder von der Wurzel an mit entsprechendem Rechenaufwand in den Trie abgestiegen werden muss, werden im Trie nun Zeiger von den Blättern der letzten Ebene auf die vorletzte Ebene gelegt. Bei diesen Zeigern wird bei der Wahl des Zielknotens jeweils das vorne um ein Symbol verkürzte Präfix des Quellknotens berücksichtigt.

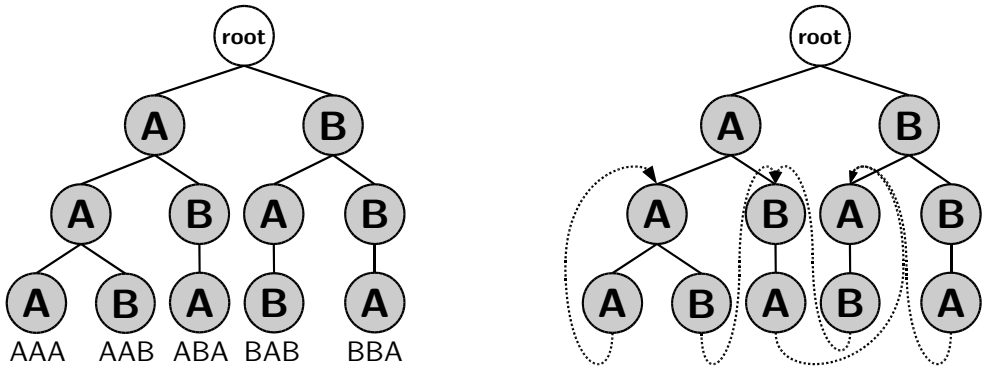


Abb. 5.3. Beispiel-Umwandlung eines 3-Gramm Filter-Tries in einen Filter-Graphen, Σ={A,B}

Eine derart veränderte Datenstruktur weist nun also *Zyklen* (graphentheoretisch: *Kreise*) auf und ist damit keine Baumstruktur mehr, denn ein Graph ist genau dann ein Baum, wenn er keinen Kreis enthält und zusammenhängend ist (vgl. [Dies00], S. 13).

Durch die Kanten, die überlappende String-Sequenzen miteinander verbinden, erinnert der zu einem Filter-Graphen umgewandelte Filter-Trie somit also stark an die in der Kodierungstheorie wichtigen DE BRUIJN Graphen.

Definition 5-10. Eine **(n, k) de Bruijn Folge** ist die kürzeste, zirkuläre Folge $a_0a_1...a_{t-1}$ von Buchstaben aus $\Sigma=\{0,...,n\text{-}1\}$, so dass jedes Wort w der Länge k über Σ in der Form $w=a_ia_{i+1}...a_{i+k-1}$ für ein einziges $i\in\{0,...,t\text{-}1\}$ geschrieben werden kann.
Für den Fall $i\geq t\text{-}1\text{-}k$ kann diese Form als $a_ia_{i+1}...a_{t-1}a_0...a_{i+k-t-1}$ interpretiert werden, so dass die Folge zyklisch ist. *(nach [CH94], S. 265)*

Definition 5-11. Ein **de Bruijn Graph** (oder *de Bruijn Diagramm*) zu einer gegebenen *(n, k) de Bruijn Folge* ist ein gerichteter Graph mit der Knotenmenge V, bestehend aus allen n^{k-1} Worten der Länge k-1 aus dem Alphabet Σ und der Kantenmenge E. Dabei laufen die gerichteten Kanten genau von allen Knoten mit Worten der Form $b_1b_2...b_{k-1}$ zum zugehörigen Knoten mit dem Wort der Form $b_2b_3...b_k$. Die Kanten werden mit $b_1b_2...b_k$ beschriftet. *(nach [CH94], S. 266)*

Streng nach Definition entspricht der oben eingeführte Filter-Graph daher also aus mehreren Gründen nicht einem DE BRUIJN Graphen:

1.) Im Filter-Trie (und damit im Graphen) sind nicht *alle* Zeichenketten der Länge q über dem Alphabet gespeichert, sondern nur solche, die auch im Text vorkommen.

2.) Die Knoten im Trie stehen (auf den oberen Ebenen) auch für Worte mit geringerer Länge als q – im DE BRUIJN Graphen sind alle Strings gleich lang.

3.) Die Überlappungskanten werden nur auf der untersten Trie-Ebene hinzugefügt (beim DE BRUIJN Graphen gehen sie aus jedem Knoten heraus).

4.) Schließlich existieren auf allen Ebenen mit Rang$<q$ auch Kanten zwischen Strings unterschiedlicher Länge, deren Überlappung nur durch Anhängen eines Symbols am Ende entsteht und nicht zusätzlich durch Löschen eines Symbols am Anfang, wie bei DE BRUIJN Graphen.

Zur weiteren Verdeutlichung der Unterschiede wird bezugnehmend auf Abbildung 5.3 ein DE BRUIJN-Graph über dem Alphabet Σ={A,B} und Stringlänge q=3 in Abbildung 5.4 dargestellt.

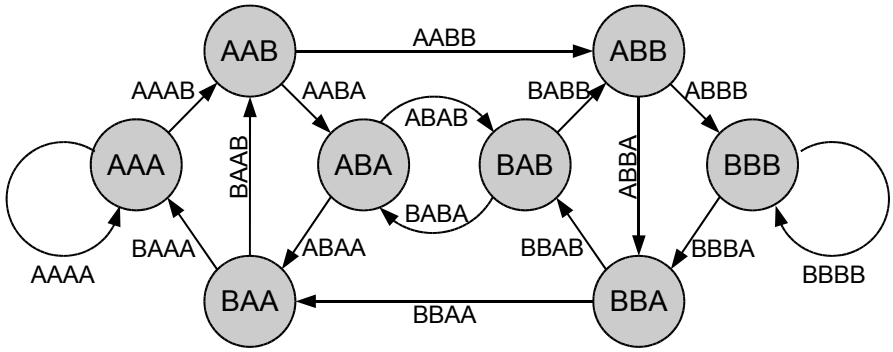


Abb. 5.4. Beispiel eines DE BRUIJN Graphen (Σ={A,B}, q=3)

Mit dem aus dem Filter-Trie gewonnenen Filter-Graphen ist es nun möglich, die (Nicht)-Existenz von generierten Morphs zu prüfen, ohne diese vorher in überlappende q-Gramm-Fragmente zerlegen zu müssen. Außerdem muss beim Testen nun nicht mehr redundant von der Wurzel ausgehend mehrfach in den Trie verzweigt werden. Es kann nun einfach Zeichen für Zeichen überprüft und dabei in den Filter-Graphen abgestiegen werden. Gelangt man auf der untersten Ebene an, so folgt man der Kante, die eine Ebene höher wieder in den Graphen zeigt und setzt die Überprüfung des nächsten Zeichens an diesem Knoten fort.

5.3 Der WPM-Algorithmus

Basierend auf den vorhergehenden Abschnitten wird im folgenden Abschnitt nun der eigentliche Kern-Algorithmus vorgestellt. Der Arbeitsablauf des WPM-Verfahrens wird in Abbildung 5.5 schematisch dargestellt.

Der Kern-Algorithmus generiert mithilfe der linearisierten Submorph-Regelmenge und dem Filter-Graphen über dem Textkorpus zu einer gegebenen Morpher-Konfiguration $K=(S\,,a\,,t\,,b)$ eine Menge von Morphs, die mit der nachgeschalteten exakten Volltextsuche zu suchen sind. Dabei ist $S=(s\,,0)$ der Start-Morph;

$a \in \mathbb{N}_0$ die maximale Anzahl erlaubter Regelanwendungen, ausgehend von S; $t \in \mathbb{N}_0$ die vorgegebene Gewichtssumme und $b \in \mathbb{N}$ der vorgegebene Morph-Index (vgl. Def. 5-7 auf S. 48).

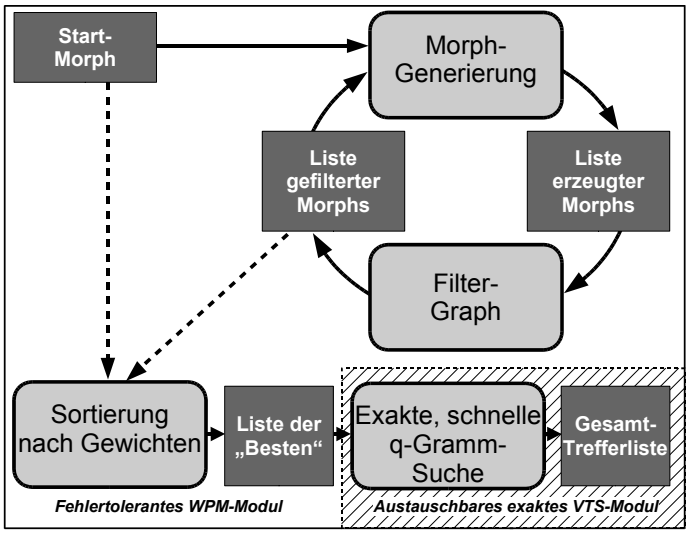


Abb. 5.5. Schematischer Arbeitsablauf des *Weighted Pattern Morphing Verfahrens* mit nachgeschalteter exakter q-Gramm Volltextsuche

1: **Input:** Zeichenkette s,
Parameter a,
Parameter t,
Parameter b
2: **Output:** Morpher-Ergebnis E
3: Erzeuge Start-Morph $S=(s\,,0)$
4: Erzeuge Morpher-Konfiguration $K=(S\,,a\,,t\,,b)$
5: Setze $E=\{S\} \cup$ `morphSub`$(K,1)$ // *siehe Algorithmus 5-2*
6: Für jedes Morph-Paar $W=(w,z)$ und $V=(v,x)$ mit $w=v$ und $W,V \in E$
6.1: Lösche aus E den Morph V falls $x \geq z$ oder
6.2: Lösche aus E den Morph W falls $z>x$
7: Falls $|E|>b$
7.1: Bestimme das Gewicht γ des b. Morphs in der nach Morph-Gewicht aufsteigend sortierten Menge E.
7.2: Lösche aus E alle Morphs $W=(w,z) \in E$ mit $z>\gamma$.
8: $\Rightarrow$FERTIG

Algorithmus 5-1. Öffentliche Schnittstelle zum Morpher

Die öffentliche Schnittstelle des Morphers delegiert also die eigentliche Generierung der Morphs an eine nach außen nicht sichtbare Methode `morphSub`, welche die Morphs durch rekursive Selbstaufrufe berechnet. In Schritt 6 werden gemäß Bedingung (ii) von Definition 5-9 nur Morphs mit minimalen Gewichtskosten

bei identischen Zeichenketten beibehalten. Wie in Definition 5-9 bereits festgelegt, steuert Parameter b nicht die genaue Anzahl der Morphs, die der Algorithmus zurückgeben soll. Denn in der nach Gewicht sortierten Liste der generierten Morphs kann es vorkommen, dass der Morph an Position b dasselbe Gewicht besitzt wie ein oder mehrere seiner Nachfolger. Diese Morphs mit demselben Gewicht werden daher aus Konsistenzgründen in Schritt 7 ebenfalls im Morpher-Ergebnis belassen und alle Morphs mit größerem Gewicht werden gelöscht. Die rekursive Methode `morphSub` wird in Algorithmus 5-2 beschrieben.

Neu an Algorithmus 5-2 ist die Tatsache, dass die Submorph-Regeln je Rekursionsstufe nur an einer festen Position p angewendet werden dürfen, wobei p bei jedem rekursiven Aufruf immer um genau eins erhöht wird. Diese Technik verhindert effizient, dass der Algorithmus sich in einer Endlos-Schleife fängt, wenn Regelzyklen bestehen wie zum Beispiel `x`→`y`, `y`→`z` und `z`→`x`.

Die Schritte 4 bis 6 verhindern, dass die Rekursion tiefer absteigt, falls eines der drei Abbruchkriterien erfüllt ist. Schritt 7 gibt den aktuellen Morph-String ohne jegliche Regelanwendung an eine tiefere Rekursionsebene weiter. Dies ist nötig, da es möglich sein muss, den vorderen Teil des Start-Morphs unverändert zu lassen und erst im hinteren Teil Submorph-Regeln anzuwenden.

Die Schritte 8 und 9 bauen die Schleife auf, die genau über die Regeln iteriert, welche als ersten Buchstaben ihres `from`-Strings den Buchstaben besitzen, der in s an Position p steht. Damit wird die Zahl der genauer auf Anwendbarkeit zu untersuchenden Regeln stark eingeschränkt, um die Geschwindigkeit zu erhöhen. Schritt 9.1 verhindert die Anwendung von Regeln, deren Präfix- und Suffix-Flag nicht zur aktuellen Startposition passt. Schritt 9.2 wendet passende Regeln an und erzeugt dabei den temporären Morph V, dessen Morph-String v dann mithilfe des Filter-Graphen untersucht wird.

Passiert v den Filter-Graphen, so wird V in die Ergebnismenge aufgenommen und V an die nächste Rekursionsebene mit angepasstem Morph-Zähler a und angepasster Startposition p übergeben. Passiert der komplette String v den Filter-Graphen jedoch *nicht*, so wird v' (der linke v-Teil bis p) mit dem Filter-Graphen überprüft: Da tiefere Rekursionsebenen ihre Regelanwendungen bei $(p+1)$ oder später starten, werden sich die Symbole bis einschließlich p nicht mehr verändern. Falls v' den Graphen passiert, könnte es also sein, dass folgende Rekursionsebenen im hinteren Teil von v durch Regelanwendungen noch einen String erzeugen, der vom Filter-Graphen in seiner Gesamtheit als gültig eingestuft wird. Daher wird in diesem Fall zwar V nicht zur Ergebnismenge hinzugefügt, aber V wird an die nächste Rekursionsebene mit verringertem a und erhöhtem p übergeben.

1: **Input:** Morpher-Konfiguration $K=(S, a, t, b)$ mit Start-Morph $S=(s, g)$, 1-basierte Startposition p für Regelanwendungen auf String s
2: **Output:** Liste E von Morphs, aufsteigend sortiert nach Gewicht
3: Lokale Rückgabeliste $E=\varnothing$
4: Wenn $p=(|s|+1)$ ⇒FERTIG *// noch Buchstaben übrig?*
5: Wenn $a\leq 0$ ⇒FERTIG *// noch Regelanwendngn. erlaubt?*
6: Wenn $g\geq$t ⇒FERTIG *// Gewicht schon zu groß?*
7: E = E ∪ `morphSub`$(((s, g)\ a, t, b), p+1)$ *// Rekursion nur mit p++*
8: k=`substr`$(s, p, 1)$ *// Buchstabe an Startposition*
9: ∀ i: `startcount`$[k] \leq i \leq$ `stopcount`$[k]$ *// iteriere Submorph-Regeln*
9.1: Wenn (Startposition am Anfang von s ist und `samepostfix`$[i]$ gesetzt ist) oder wenn (Startposition am Ende von s ist und `sameprefix`$[i]$ gesetzt ist) dann nächste Iteration aus Schritt 9.
9.2: Wenn in s ab Position p der `from`-Teil der i. Regel enthalten ist und wenn die Summe g+`penalty`$[i] \leq t$ ist, dann wende die i. Regel an:
▶ Erzeuge String v=`substr`$(s, 1, p\text{-}1)$ *// links von Regelanwendung*
∘`to`$[i]$ *// Regel-Zielstring*
∘`substr`$(s, p+|$`from`$[i]|, |s|\text{-}(p+|$`from`$[i]|)+1)$ *// rechts*
▶ Prüfe v mit Filter-Graph auf Gültigkeit. Ist v gültig, dann erzeuge neuen Morph $V=(v, g+$`penalty`$[i])$
- $E = E \cup V$
- Wenn $(g+$`penalty`$[i])<t$ dann rufe rekursiv auf: $E = E \cup$ `morphSub`$((V, a\text{–}1, t, b), p+1)$
▶ Falls v vom Filter-Graph nicht als gültig eingestuft wurde, dann prüfe von v die ersten p Zeichen v'=`substr`$(v, 1, p)$. Falls v' als gültig eingestuft wird und wenn $(g+$`penalty`$[i])<t$ dann rufe rekursiv auf: $E = E \cup$ `morphSub`$((V, a\text{–}1, t, b), p+1)$.
10: ⇒ FERTIG

Algorithmus 5-2. Algorithmus `morphSub` zur rekursiven Generierung der Morphs

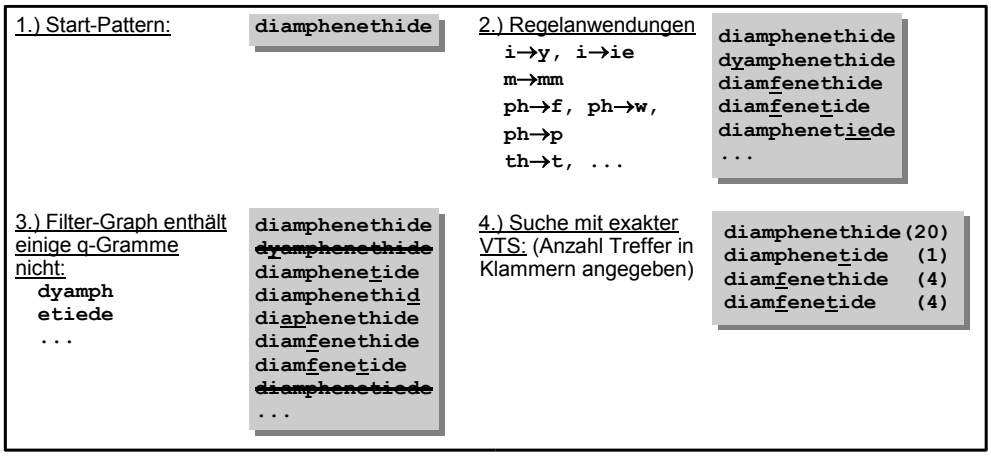


Abb. 5.6. Beispiel einer WPM-Suche nach ″diamphenethide″

In Abbildung 5.6 wird anhand des Suchbegriffs `diamphenethide` beispielhaft illustriert, wie Regelanwendungen, Filter-Graph und exaktes Volltextsuche-System zusammen eine fehlertolerante Volltextsuche durchführen. Die Angaben zu den Trefferanzahlen in Klammern beziehen sich auf *Parasitology*, das Kombinationswerk bestehend aus [CM04] und [Meh01].

5.4 Anwendung von Spezial-Submorphs

In der Fachliteratur findet man die Alltagserfahrung bestätigt, dass selbst aufwändig lektorierte und professionell hergestellte Fachbücher meist noch zahlreiche Wortfehler beinhalten. Karen Kukich zitiert in ihrem umfangreichen Übersicht-Artikel *Techniques for Automatically Correcting Words in Text* [Kuk92] Arbeiten, welche in maschinell erfassten Texten eine Wortfehler-Rate von 1% (Schreibmaschinen-Experten) bis zu 3,2% (Anfänger) in umfangreichen Textkorpora ermittelt haben [ebd. S. 389]. In einer eigenen Arbeit analysierte die Autorin Texte im Umfang von 40.000 Worten (genauer: Strings), die mit Fernschreibern für Gehörlose und Schwerhörige erstellt wurden. In diesen Texten ermittelte sie eine Wortfehler-Rate von 6% [Kuk92b].

Dabei lassen sich die Fehler in maschinell erfassten Texten einteilen in *typografische Fehler* und *kognitive Fehler*. Die eher technisch bedingten typografischen Fehler entstehen z.B. durch versehentliche Auslassung oder Vertauschung von Buchstaben (z.B. `Kronbläter`, `Kronbältter`) oder aufgrund von Buchstabenverdopplung durch Prellen von Tasten (z.B. `Kronblättter`). Kognitive Fehler hingegen entstehen dadurch, dass der Schreiber die korrekte Schreibweise eines Wortes nicht kennt und unbewusst eine klanglich ähnliche Schreibweise wählt (z.B. `Stafylokocken`, `nähmlich`). Oft ist die Klassifizierung eines vorliegenden Wortfehlers schwer oder sogar unmöglich: Handelt es sich bei den Worten `Wallnuss` und `Phamazie` um typografische Fehler durch ein verdoppeltes „`l`“ bzw. ein ausgelassenes „`r`“ – oder handelt es sich um kognitive Fehler?

Bei handschriftlich erstellten Texten gibt Kukich einen Bereich von 1,5%–2,5% fehlerhafter Worte an, die in umfangreichen Aufsatzsammlungen ermittelt wurden ([Kuk92] S. 390). Dabei sind Wortfehler in dieser Art von Texten fast ausschließlich kognitive Fehler, da (abgesehen von so genannten Flüchtigkeitsfehlern) technisch bedingte Tippfehler bei Handschriften nicht vorkommen.

Das in der vorliegenden Arbeit bereits beschriebene Verfahren zum *Weighted Pattern Morphing* stützt sich bei der Generierung der Morphs ausschließlich auf die vorgegebene Submorph-Regelmenge mit ihren bisher eher phonetisch orientierten Regeln. In diese Menge wird aufgrund der bisher beschriebenen Vorgehensweise zum Beispiel kein Submorph `a`→`x` aufgenommen, da diese beiden

Buchstaben weder klanglich noch optisch Gemeinsamkeiten haben. Die bisher beschriebenen Regeln können also vor allem kognitive Fehler (im Text oder im Suchmuster) ausgleichen, und typografische Fehler sind nur in den folgenden Fällen eingeschränkt korrigierbar:

- **Verdopplung von Buchstaben** ändern den Klang eines Wortes nur verhältnismäßig marginal. Daher enthält die Submorph-Regelmenge üblicherweise auch Regeln mit doppelten Buchstaben wie z.B. `t`→`tt`, `tt`→`t`, `oo`→`o` usw.
- Typografische **Einfügungen von Buchstaben** können nur dann korrigiert werden, wenn es einen entsprechenden kognitiven/phonetischen Fehler gibt. Ein angehängtes Dehnungs-h sei hierzu als Beispiel genannt: `uh`→`u`.
- Für typografische **Löschungen von Buchstaben** gilt analog zu den Einfügungen, dass hierzu meist nur ausnahmsweise identische phonetische Regeln bestehen: `t`→`th`, `sh`→`sch`.
- Die **Ersetzung von einzelnen Buchstaben** durch andere einzelne Buchstaben wird nur von wenigen phonetischen Submorph-Regeln abgedeckt. In diesen Fällen handelt es sich um Buchstaben, die ähnlich klingen und daher bei phonetisch motivierten Regeln angewendet werden: `k`→`c`, `ä`→`a`, `p`→`b` usw. Beliebige Substitution von Buchstaben lässt sich damit natürlich nicht korrigieren.

Vertauschung von Buchstaben führt jedoch (außer bei Doppelbuchstaben) fast immer zu einem stark veränderten Wortklang. Darum gibt es hierfür bei den phonetisch orientierten Submorph-Regeln üblicherweise keine Entsprechungen. Zufällige Tippfehler können also mit der bisher beschriebenen Regelmenge nicht korrigiert werden.

Im folgenden Abschnitt wird nun erläutert, wie mit so genannten *starken Submorph-Regeln* die Fähigkeiten einer auf der Edit-Distanz basierten Fehlertoleranz simuliert werden können.

5.4.1 Simulation der Edit-Distanz

Mit den *starken Submorph-Regeln* werden die sonst eher strikten Submorphs um Regeln ergänzt, die das Original-Suchwort stärker verändern können, um z.B. (zufällige) Tippfehler im Suchwort oder im Text ausgleichen zu können.

Tab. 5.2. Starke Submorph-Regeln zur Simulation der Edit-Distanz

Submorph	***Bedeutung***
$c \rightarrow \varepsilon$ $(c \in \Sigma)$	Jedes Zeichen im Suchwort kann gelöscht werden, womit diese Regel der Edit-Distanz-Aktion **DEL** entspricht. Dabei entspricht ε wieder dem leeren Wort.
$xy \rightarrow yx$ $(x,y \in \Sigma)$	Beliebige benachbarte Symbole im Suchwort können vertauscht werden (**SWAP**). Diese Kategorie von Tippfehlern wird von der Standard-Edit-Distanz mit Abstand 2 belegt (zwei SUBST Aktionen).
$\varepsilon \rightarrow$?	An jeder Stelle im Suchwort kann ein Fragezeichen als Platzhalter für ein beliebiges Zeichen eingefügt werden. Damit entspricht diese Regel der Aktion **INS** bei der Berechnung der Edit-Distanz.
$c \rightarrow$? $(c \in \Sigma)$	Jedes Zeichen im Suchwort kann durch den Fragezeichen-Platzhalter ersetzt werden. Dies entspricht der Edit-Distanz-Aktion **SUBST**.

Mit den drei starken Submorph-Regeln $c \rightarrow \varepsilon$, $\varepsilon \rightarrow$? und $c \rightarrow$? ist es nun also möglich, auch alle (unscharfen) Veränderungen an einem String vorzunehmen, welche bei der Berechnung der Edit-Distanz implizit durchgeführt werden können.

Auf Implementierungsseite ist eine Regel bisher die Ersetzung eines konstanten Strings durch einen anderen konstanten String gewesen. Da sich die neuen Regeln nicht in dieses Schema eingliedern, werden sie außerhalb der im vorhergehenden Abschnitt beschriebenen Morph-Rekursion angewendet. Es wird daher eine Klasse von „Spezial-Morphs“ eröffnet, bei der jeder der beschriebenen Spezial-Morphs in einer eigenen Methode abgearbeitet wird. Neben der leichten zukünftigen Erweiterbarkeit hat diese getrennte Behandlung noch einen weiteren Vorteil: Da die starken Submorphs nun unvermeidbar auch alle im Abschnitt 4.4 detailliert beschriebenen Nachteile der Edit-Distanz mit sich bringen, soll ihr Einsatz nur sehr kontrolliert erfolgen. Das bedeutet, dass auf niedrigen Fehlertoleranz-Stufen die besonders stark verändernden Submorph-Regeln wie $\varepsilon \rightarrow$? und $c \rightarrow$? noch nicht angewendet werden. Erst wenn der Endnutzer auch Interesse an Morphs hat, die weiter vom Suchmuster entfernt sind, werden auch diese Regeln angewendet. Durch eine Herausnahme dieser Regeln aus der Morph-Rekursion kann weiterhin pro Toleranzstufe festgelegt werden, ob ein Begriff, auf dem bereits starke Submorph-Regeln angewendet wurden, zusätzlich

noch den rekursiven Veränderungen durch die normale Submorph-Regelmenge unterzogen wird.

Beispiel 5-6. Anwendung starker Submorphs in *HagerROM [BEH+04]*
Im Folgenden werden zu den vier Arten der starken Submorphs aus dem Textkorpus von *HagerROM* jeweils fünf Wortpaare aufgeführt, die (vom 1. Wort zum 2. Wort) jeweils eine Löschung, Vertauschung, Einfügung oder Ersetzung benötigen, welche durch normale (phonetische) Submorph-Regeln nicht abgedeckt wird.

DEL:	3-jähriger(1)	3jähriger(5)	
	aufbereitungsmonographie(730)	aufbreitungsmonographie(2)	
	beobachtet(2955)	bobachtet(3), beobachet(5)	
	drogen-extraktverhältnis(1)	droge-extraktverhältnis(4)	
	zwölffingerdarmgeschwüre(16)	zwölffingerdarmgeschüre(1)	
SWAP:	abwechselnden(7)	abwechslenden(1)	
	antibiotikatherapie(12)	antibiotikatherpaie(1)	
	beobachtet(2955)	bebobachtet(2)	
	carboxymethylcellulose(47)	carboxymethylcelullose(1)	
	dosisabhängig(839)	dosisbahängig(1)	
INS:	accidenteller(1)	accident?eller(1)	?∈ {'i'}
	beeinflussen(236)	beeinfluss?en(2)	?∈ {'t'}
	bestandteile(1148)	bestand?teile(2)	?∈ {'e', 't'}
	brenzcatechin(55)	brenzcatech?in(1)	?∈ {'z'}
	methylendioxyphenyl(27)	methylendioxy?phenyl(1)	?∈ {')'}
SUBST:	carbaminsäure(30)	carbamins?ure(31)	?∈ {'ä', 'a'}
	antiöstrogen(21)	anti?strogen(51)	?∈ {'ö', 'e'}
	conversionsenzym(26)	conversion?enzym(27)	?∈ {'s', '-'}
	ninhydrin-sprührg.(7)	ninhydrin?sprührg.(8)	?∈ {'-', ' '}
	placenta-schranke(10)	placenta?schranke(22)	?∈ {'-', 'r'}

Wie Beispiel 5-6 illustriert, kommen in einem umfangreichen Textkorpus wie *HagerROM* tatsächlich auch alle aufgeführten Fehlerarten mehrfach vor, die nun mit den starken Submorph-Regeln abgedeckt werden. Die optionale Anwendung dieser Regeln sollte also grundsätzlich möglich sein, aber nicht dem Normalfall entsprechen. Es fällt weiter auf, dass speziell in längeren Worten Bindestriche und Leerzeichen zur Erhöhung der Lesbarkeit dieser Worte inkonsistent eingesetzt werden:

3-jähriger⇒3jähriger
ninhydrin-sprührg.⇒ninhydrin sprührg.
placenta-schranke⇒placentaschranke

Im folgenden Abschnitt werden daher noch einige weitere Spezial-Submorphs vorgestellt, die unter anderem diese Klasse von Wortveränderungen abdecken werden.

5.4.2 Weitere Spezial-Submorphs

Dieser Abschnitt stellt nun weitere Spezial-Morphs vor, die sich weder zu den normalen *String*→*String*-Ersetzungsregeln noch zu den oben eingeführten starken Submorphs zählen lassen.

Bindestriche und Leerzeichen innerhalb von Fachtermini werden oft zur Erhöhung der Lesbarkeit von zusammengesetzten Begriffen mit großer Wortlänge gebraucht. Nun könnte man die Einfügung oder Löschung eines Bindestriches oder die Ersetzung eines Leerzeichens zu einem Bindestrich natürlich grundsätzlich auch immer mit den starken Submorphs $c \rightarrow \varepsilon$, $\varepsilon \rightarrow$ `?` und $c \rightarrow$ `?` abdecken. Allerdings bringen diese unscharfen Regeln dann verstärkt Worte in das Morpher-Ergebnis, die eine andere Semantik besitzen als das Original-Suchmuster (vgl. Beispiel 4-8 auf Seite 43). Um dies zu vermeiden, wurden die folgenden beiden Spezial-Submorphs in das WPM-Verfahren aufgenommen.

Tab. 5.3. Spezial-Submorphs für Bindestriche und Leerzeichen

Submorph	***Bedeutung***
$\varepsilon \rightarrow$ _	Einfügen eines **Leerzeichens** an einer beliebigen Stelle im String. Das Leerzeichen wurde zur besseren Visualisierung hier als Unterstrich '_' dargestellt.
$\varepsilon \rightarrow$ -	Einfügen eines **Bindestriches** an einer beliebigen Stelle im String.

Durch die Aufnahme dieser beiden Regeln zusätzlich zu den bereits bestehenden starken Submorph-Regeln ist es nun möglich, Leerzeichen und Bindestriche bereits in niedrigen Fehlertoleranz-Stufen einzufügen und die Einfügung von Fragezeichen-Platzhaltern erst auf höheren Toleranzstufen vorzunehmen. Analog gilt dies natürlich auch für die Löschung von Leerzeichen und Bindestrichen.

Beispiel 5-7. Bindestriche und Leerzeichen in *HagerROM* [BEH+04]
In diesem Beispiel werden nun einige Suchmuster aus dem Textkorpus der *HagerROM* aufgelistet, die durch Anwendung der Leerzeichen- oder Bindestrich-Regel wiederum auf ein Wort aus demselben Textkorpus abgebildet werden können.

`acetonitrilanteil` (1)	`acetonitril-anteil` (1)
`alkoholunlösliche` (9)	`alkohol-unlösliche` (3)
`anisaldehydlösung` (20)	`anisaldehyd-lösung` (13)
`acetylcholininduzierte` (4)	`acetylcholin-induzierte` (1)
	`acetylcholin induzierte` (6)
`ammoniumglycyrrhizinat` (6)	`ammonium glycyrrhizinat` (1)
`carboanhydrasehemmende` (2)	`carboanhydrase-hemmende` (9)
	`carboanhydrase hemmende` (2)

5.5 Überdeckungsfilter

Die im vorhergehenden Abschnitt eingeführte starke Submorph-Regel $c \rightarrow$ `?` ($c \in \Sigma$) kann den generierten Morph potenziell sehr weit vom Original-Suchwort entfernen. Im Gegensatz dazu kann es aber auch passieren, dass durch eine große Zahl von Kontext-Symbolen um die Position der Anwendung des starken Morphs herum trotz des Fragezeichen-Platzhalters wieder nur genau dieselben Fundstellen im Text Treffer produzieren. Dem Anwender würde also ein Morph mit Ergebnissen präsentiert, die schon bekannt sind, da immer auch nach dem Original-Suchmuster gesucht wird. Zusätzlich erschwert der Fragezeichen-Platzhalter die Lesbarkeit des Morphs.

Ein ähnlicher Fall liegt vor, wenn die starke Submorph-Regel $c \rightarrow \varepsilon$ ($c \in \Sigma$) am Anfang oder am Ende einer Zeichenkette angewendet wird. Durch das Löschen von Zeichen am Rand eines Strings entsteht ein String kürzerer Länge, der aber Substring des Original-Strings ist. Dieser Morph-String liefert bei einer Substring-Volltextsuche daher immer gleichviele oder mehr Treffer verglichen mit dem Original-String. Werden aber gleichviele Treffer produziert, so kennt der Anwender alle Fundstellen bereits durch die Fundstellen des Original-Strings.

Generell lässt sich also formulieren, dass die Treffer eines Morphs, der gegenüber dem Original-Suchwort ein oder mehrere Fragezeichen-Platzhalter durch die Regel $c \rightarrow$ `?` erhalten hat und an dessen Rändern beliebig viele Zeichen durch die Regel $c \rightarrow \varepsilon$ gelöscht wurden, dieselben Fundstellen im Text erzeugt wie das Original-Suchmuster – und eventuell noch einige zusätzliche Fundstellen mehr. Ein solcher Morph würde also in jedem Fall die Trefferstellen des Original-Suchmusters überdecken und wird daher auch *überdeckender Morph* genannt.

Beispiel 5-8. Überdeckende Morphs
Das Original-Suchmuster sei der String `patienten`, welcher im Werk *HagerROM* (Teil Drogen+Stoffe) gesucht wird. Dann sind einige der entstehenden Morphs mit der Anzahl ihrer Fundstellen die folgenden:

`patienten`	(5027)	Original-Suchmuster
`atienten`	(5027)	überdeckender Morph durch Löschung am Anfang
`patiente`	(5027)	überdeckender Morph durch Löschung am Ende
`pateinten`	(3)	keine Überdeckung!
`p?tienten`	(5027)	überdeckender Morph durch Fragezeichen-Platzhalter
`patient?n`	(**5332**)	überdeckender Morph durch Fragezeichen-Platzhalter

Bei dem Szenario aus Beispiel 5-8 sollen also die ersten drei überdeckenden Morphs (`atienten`, `patiente`, `p?tienten`) dem Anwender nicht gezeigt werden, denn diese produzieren dieselbe Trefferzahl (5027) wie das Original-Suchmuster – es muss sich aufgrund der Überdeckungseigenschaft der Strings um dieselben Trefferstellen handeln. Der letzte überdeckende Morph (`patient?n`) hat jedoch

mit 5332 eine höhere Trefferzahl als das Original-Suchmuster, denn hierin sind auch die Treffer zum Suchmuster `patientin` enthalten. In den Trefferstellen des überdeckenden Morphs `patient?n` befinden sich also neue Informationen für den Anwender, und dieser Morph soll daher angezeigt werden.

Beschrieben wird nun der Algorithmus `coversHits`, der zwei gegebene Strings auf die Überdeckungseigenschaft hin überprüft. Da das Prädikat `coversHits` nur in einer Richtung auf Überdeckung überprüft, wird in der Beschreibung des Algorithmus der String mit potenziell mehr Trefferstellen M (more) heißen und der String mit potenziell weniger Trefferstellen L (less). Vor der Algorithmus-Beschreibung jedoch ein Beispiel, welches die erwarteten Ergebnisse des Prädikates verdeutlichen soll.

Beispiel 5-9. Erwartete Ergebnisse des Prädikates `coversHits(M,L)`

`coversHits(aab, xaaxaabxxx) == true`	Substring in der Mitte
`coversHits(aabb, xaaxaabx) == false`	kein Substring
`coversHits(xa, xaaxaabxxx) == true`	Substring am Anfang
`coversHits(xxx, xaaxaabxxx) == true`	Substring am Ende
`coversHits(a?a, xaaxaabxxx) == true`	Fragezeichen im more-String
`coversHits(axa, xaa?aabxxx) == false`	Fragezeichen im less-String
`coversHits(a?a, xaa?aabxxx) == true`	Fragezeichen an gleicher Stelle
`coversHits(axa, axa) == true`	Identität

Realisiert wird der Algorithmus über einen endlichen Automaten mit vier Zuständen: (0). Präfix überlesen; (1). Gleichheit der Symbole; (2). Endzustand Überdeckung; (3). Endzustand keine Überdeckung.

In Algorithmus 5-3 laufen die Variablen l und m die Strings L und M zeichenweise ab, wobei der Zugriff auf das gerade aktuelle Zeichen als L_l und M_m dargestellt wird. Da M möglicherweise ein Substring von L ist wird in Zustand 0 zunächst ein eventuell vorhandenes Präfix von L überlesen, um dann in Zustand 1 die identischen Zeichen zu überprüfen. Wurde jedoch zu früh in Zustand 1 gewechselt, wird über die Puffervariable *prefix_restore_l* der alte Wert von l wieder hergestellt und wieder zurück in Zustand 0 verzweigt.

1: **Input:** String $M \in \Sigma^*$, String $L \in \Sigma^*$
2: **Output:** `true`, falls M die Trefferstellen von L potenziell überdecken könnte
`false`, sonst.
3: Zustand des endlichen Automaten $Z := 0$
4: Cursor in L ist $l := 1$; Cursor in M ist $m := 1$
prefix_restore_l $:= 1$
5: Solange ($Z \neq 2$ und $Z \neq 3$)
5.1: Wenn ($Z = 0$) dann *// Präfix von L-String überlesen*
▶ Wenn ($l > |L|$ oder $m > |M|$) dann ($Z := 3$; gehe nach Schritt 5)
▶ sonst wenn ($M_m = L_l$ oder $M_m = '?'$) dann
$m{+}{+}$; $l{+}{+}$
prefix_restore_l $:= l$
$Z := 1$; gehe nach Schritt 5
▶ sonst wenn $L_l = '?'$ dann ($Z := 3$; gehe nach Schritt 5)
▶ sonst $l{+}{+}$ *// implizit: $M_m \neq L_l$*
5.2: Wenn ($Z = 1$) dann *// Gleichheit von L und M*
▶ Wenn $m > |M|$ dann ($Z := 2$; gehe nach Schritt 5)
▶ sonst wenn $l > |L|$ dann ($Z := 3$; gehe nach Schritt 5)
▶ sonst wenn ($M_m = L_l$ oder $M_m = '?'$) dann
$m{+}{+}$; $l{+}{+}$
▶ sonst wenn $L_l = '?'$ dann ($Z := 3$; gehe nach Schritt 5)
▶ sonst *// implizit: $M_m \neq L_l$*
$m := 1$
$l :=$ *prefix_restore_l*
$Z := 0$; gehe nach Schritt 5
6: Wenn ($Z = 2$) dann `coversHits`$:=$`true`; $\Rightarrow$ FERTIG *// Endzustand*
7: Wenn ($Z = 3$) dann `coversHits`$:=$`false`; $\Rightarrow$ FERTIG *// Endzustand*

Algorithmus 5-3. Prädikat `coversHits(M,L)` für überdeckende Morphs

Mit Algorithmus 5-3 alleine kann jedoch noch keine Entscheidung getroffen werden, ob ein spezieller überdeckender Morph nicht angezeigt werden soll. Erst nach erfolgter exakter Suche und der damit möglichen Ermittlung der Trefferanzahlen kann diese Entscheidung getroffen werden. Daher ist die Anwendung des Überdeckungsfilters auch außerhalb des eigentlichen Morphers anzuordnen.

5.6 Verfahren zur Messung der Performanz

Werden die Parameter verändert, die das Verhalten eines Algorithmus steuern, so benötigt man im Allgemeinen ein Kriterium zur Messung der durch diese Änderung erfolgten Verbesserung (oder Verschlechterung) des Algorithmus. Bei ei-

nem Suchverfahren benötigt man dazu zunächst eine Menge von Test-Suchmustern und anschließend Kenngrößen, die Veränderungen im Verhalten des Verfahrens aufzeigen. Eindeutig messbare Kenngrößen könnten z.B. die Laufzeit oder der Bedarf an Arbeitsspeicher zur Laufzeit sein. Bei einem fehlertoleranten Suchverfahren interessiert aber vor allem auch, ob das Verfahren nun „bessere" Varianten zu den jeweiligen Test-Suchmustern findet. Die Bedeutung des Wortes „besser" in Bezug auf die Performanz des WPM-Verfahrens wird im Verlauf dieses Abschnitts erläutert.

1: **Input:** Textkorpus $T \in \Sigma^*$
2: **Output:** Liste von Test-Suchmustern $M \subset \Sigma^*$
3: Setze Rückgabeliste $M = \varnothing$
4: Fasse T in Leerzeichen (_) ein $T := \_ \circ T \circ \_$
5: Setze Suchposition $p := 1$; $minWL := 9$; $maxWL := 30$
6: Suche in T ab Position p nächste zusammenhängende Zeichenkette W, die vorne und hinten von einer nicht-leeren Folge von Symbolen aus der Menge
$X = \{$ = + \s . ! ? , ; : ( ) [ ] { } / " “ ” „ ± × ® ° † ‡ … ~ ' * • \xA0 % $\}$
begrenzt ist. Dabei steht \s für die Menge der sog. Whitespace (Leerzeichen, Tabulator, Zeilenumbruch, Wagenrücklauf) und \xA0 steht für das geschützte Leerzeichen (non-breaking space).
7: Ist $W = \varepsilon$ (das leere Wort) gehe zu Schritt 12
8: Setze p auf das Zeichen, welches dem letzten Zeichen von W in T folgt
9: Wenn ($W \notin M$ und $|W| \geq minWL$ und $|W| \leq maxWL$) dann $M := M \cup W$
10: Wenn W mindestens einen Bindestrich (-) enthält dann
10.1: Fasse W in Bindestriche ein $W := \text{-} \circ W \circ \text{-}$
10.2: Setze Fragment-Position $f := 1$
10.3: Suche in W ab Position f nächste zusammenhängende Zeichenkette F, die vorne und hinten von einer nicht-leeren Folge von Bindestrichen (-) begrenzt ist.
10.4: Ist $F = \varepsilon$ (das leere Wort) gehe zu Schritt 11
10.5: Setze f auf das Zeichen, welches dem letzten Zeichen von F in W folgt
10.6: Wenn ($F \notin M$ und $|F| \geq minWL$ und $|F| \leq maxWL$) dann $M := M \cup F$
11: Wenn $p < |T|$ gehe zu Schritt 6
12: $\Rightarrow$ FERTIG

Algorithmus 5-4. Generierung von Test-Suchmustern aus einem Textkorpus

Zum Zeitpunkt des Entwurfes eines Suchverfahrens stehen üblicherweise noch keine Suchmuster fest, die später von den Benutzern des Verfahrens gebraucht werden. Daher wurde bei der Entwicklung und Verbesserung des *Weighted Pattern Morphing* der umfangreiche Textkorpus des Werkes *HagerROM* als Quelle von praxisnahen Test-Suchmustern verwendet. Um diese Suchmuster zu erhalten, wurden nach dem in Algorithmus 5-4 beschriebenen Verfahren Worte aus dem Textkorpus extrahiert.

Algorithmus 5-4 geht zweistufig vor: Zuerst wird der Text in Worte zerlegt, die durchaus den Bindestrich noch beinhalten dürfen. Befindet sich die Länge eines solchen Wortes innerhalb der gesetzten Grenzen, so wird es der Wortmenge hinzugefügt (falls es noch nicht enthalten war). Dann wird das Wort einer weiteren Untersuchung unterzogen: Enthält das Wort mindestens einen Bindestrich, so wird es anschließend noch an den Bindestrichen aufgetrennt und die Fragmente werden dann der Ergebnismenge hinzugefügt – falls die Wortlänge im gewünschten Bereich ist und das Fragment noch nicht in der Ergebnismenge enthalten war.

Die minimale Wortlänge von 9 Zeichen (Schritt 9 und Schritt 10.6 von Algorithmus 5-4) ist motiviert durch die Erfahrungen mit der Online-Volltextsuche der *Springer Enzyklopädie Dermatologie*. Hier gaben die Benutzer Suchmuster mit einem Längenmedian von 9 Zeichen ein (vgl. Kapitel 9), und die Test-Suchmuster sollten daher grundsätzlich länger als dieser Median sein. Neun Zeichen verschaffen einem Wort auch genug Länge, um bei einigen wenigen Buchstaben-Veränderungen (Submorphs) noch genug Kontext im Wort zu erhalten, um seine Semantik nicht (zu stark) zu verändern. Die Obergrenze von 30 Zeichen ist motiviert durch die Tatsache, dass längere Worte auch eine längere Laufzeit bewirken, und durch diese Grenze sollte daher die Gesamtlaufzeit des Tests nach oben beschränkt werden. Suchmuster größerer Länge sind in der Praxis äußerst unüblich, wie ebenfalls Untersuchungen aus der WWW-Suche der *Springer Enzyklopädie Dermatologie* gezeigt haben (vgl. Kapitel 9).

Die so gewonnene Wortmenge M umfasst z.B. für *HagerROM*: ≈197.000 verschiedene Wortklassen, welche für ≈1,4 Millionen Wortinstanzen stehen (vgl. Begriffsklärung auf S. 21) und wird nun daraufhin untersucht, welche der Worte aus M mithilfe des WPM-Verfahrens bei hoher Toleranz in so genannte *gültige Ziel-Morphs* gewandelt werden können. Gültige Ziel-Morphs sind Morphs, die wiederum Bestandteil der Wortmenge M sind. Die nachfolgenden Messungen werden auf diese Menge der „interessanten“ Test-Suchmuster $M' \subset M$ beschränkt, um die Zeit für Testläufe über viele Muster zu verkürzen. Für *HagerROM* umfasste die Menge M' ≈14.000 verschiedene Wortklassen.

Die Messung der Performanz einer bestimmten Morpher-Konfiguration erfolgt nun durch Suchen aller Worte (oder z.B. jedes 3. Wortes) aus M' mit dem WPM-Algorithmus und anschließender Ermittlung der Anzahl der gültigen Ziel-Morphs, die generiert werden können. Dabei steht eine größere Anzahl generierter gültiger Ziel-Morphs für eine bessere Performanz und damit für eine „bessere“ Morpher-Konfiguration.

Das in diesem Abschnitt beschriebene Verfahren zur Performanz-Messung wird in einigen Experimenten aus Kapitel 9 angewendet. Auch die automatische Justage der Regelgewichte (Abschnitt 5.8) und Experimente zur Ermittlung plau-

sibler Fehlertoleranz-Stufen (folgender Abschnitt) greifen auf dieses Verfahren zurück.

5.7 Stufen der Fehlertoleranz

Laut Definition 5-7 (S. 48) besteht eine Morpher-Konfiguration neben der vom Benutzer vorgegebenen Zeichenkette des Start-Morphs S aus den drei Parametern:

- a – maximale Anzahl Regelanwendungen für die Abbildung $S \Rightarrow Morph$
- t – vorgegebenes Gewicht
- b – vorgegebener Morph-Index (in der nach Gewichtssumme sortierten Liste)

Ein durchschnittlicher Anwender, der von den Interna der fehlertoleranten Suche keine Kenntnis hat, wäre wohl damit überfordert, sinnvolle Parameter-Kombinationen für eine fehlertolerante Suche anzugeben. Daher werden in der Schnittstelle zum Anwender die drei so genannten *Toleranzstufen gering*, *mittel* und *hoch* eingeführt, welche jeweils eine Kombination der drei Parameter beinhalten (vgl. auch [Ess04]). Um zu sinnvollen Parameter-Kombinationen für die drei Toleranzstufen zu gelangen, werden auf dem Textkorpus der *HagerROM* einige Performanz-Experimente durchgeführt, deren Ergebnisse hier nun beschrieben werden. Als Submorph-Regelmenge dient dabei eine Menge von 361 manuell erzeugten Regeln mit manuell erzeugten Gewichten im Intervall von 1 bis 30. Ein Verfahren zur automatischen Erzeugung der Gewichte wird im folgenden Abschnitt beschrieben. Für eine genaue Auflistung der Submorph-Zeichenketten für die deutsche Sprache mit lateinischen und griechischen Fachworten sei auf den Anhang ab Seite 209 verwiesen.

Aus der Menge M' von ca. 14.000 überhaupt morphbaren Worten (vgl. vorhergehender Abschnitt) aus dem Wortschatz der *HagerROM* werden zufällig 4596 Worte ausgewählt, um die Laufzeit der zahlreichen Testreihen weiter zu verringern. Für unterschiedliche Parametersätze von a, t und b wird dann ermittelt, wieviele gültige Ziel-Morphs (also Worte, die Bestandteil des gesamten Wortschatzes sind) der Algorithmus erzeugen kann. Jeweils ein Parameter wird pro Experiment variabel gehalten, die anderen beiden Parameter stehen auf fixen Werten. Der Algorithmus wird für die Dauer der Tests keine starken Submorph-Regeln (Löschung, Vertauschung etc.) anwenden und sich nur auf die Anwendung der normalen Submorph-Regelmenge beschränken.

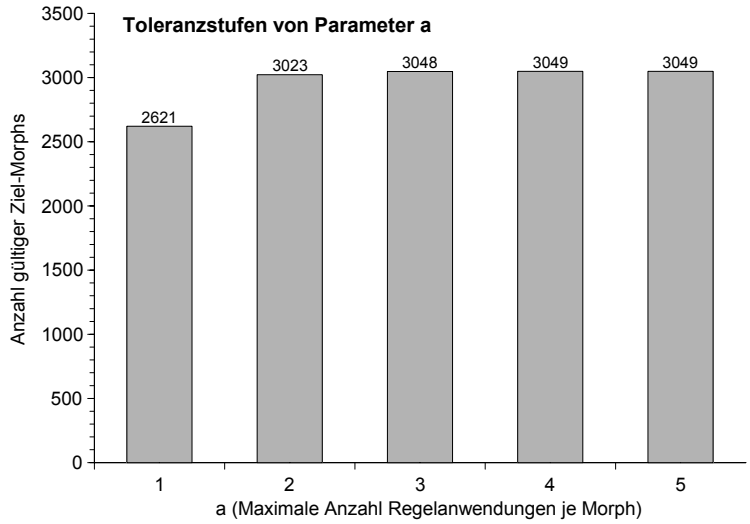


Abb. 5.7. Performanz-Messungen zu a={1,2,3,4,5}, t=20, b=200

Das erste Experiment ist in Abb. 5.7 wiedergegeben und zeigt, dass die Performanz bereits bei nur einer angewendeten Submorph-Regel einen merklichen Sprung macht und ab drei angewendeten Submorph-Regeln eine Sättigung erreicht. Mit nur einer angewendeten Morph-Regel werden bereits 85% der bei fünf angewendeten Morph-Regeln erreichbaren gültigen Ziel-Morphs generiert.

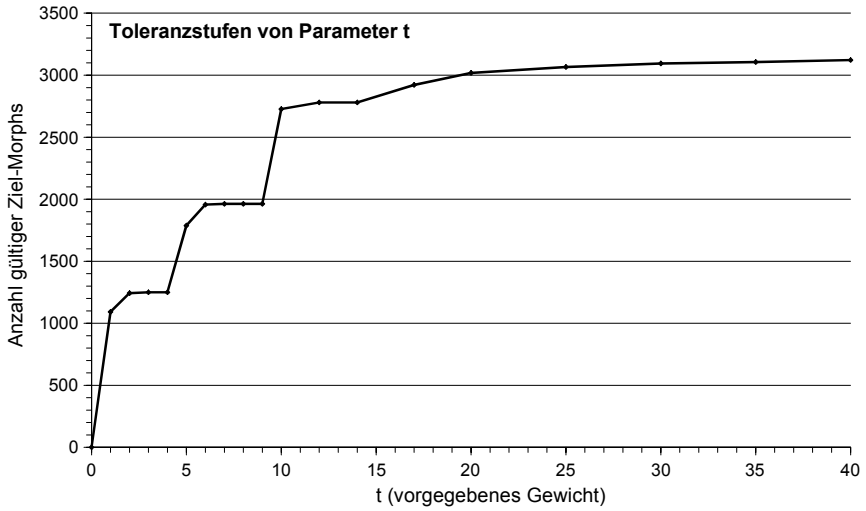


Abb. 5.8. Performanz-Messungen zu a=2, t={1,...,40}, b=200

Diese Beobachtung deckt sich mit den Ergebnissen, die Karen Kukich in [Kuk92] aufführt. Dort werden verschiedene Autoren zitiert, die in unterschiedlichen Untersuchungen großer Textkorpora jeweils 69% (Mitton), 80% (Damerau) bzw. 94% (Pollok und Zamora) der falsch geschriebenen Worte als so genannte „single error misspellings“ klassifizieren [ebd. S. 388].

Innerhalb des Intervalls [1, ..., 30] für die Gewichte der Submorph-Regeln wird nicht jeder mögliche Wert als Regelgewicht vergeben. Es werden vielmehr die Werte [1, 2, 5, 10, 15, 20, 30] stellvertretend für die Abstandsklassen [*sehr niedrig, niedrig, ..., hoch, sehr hoch*] vergeben. Diese eher „informellen Gewichtsklassen" lassen sich bei der manuellen Einstellung der Gewichte leichter zuordnen und aus dieser vergröberten Sicht erklären sich daher die Treppenstufen, die in Abb. 5.8 erscheinen. Die Abbildung zeigt auch, dass bei t=30 bereits eine Sättigung der Performanz eintritt. Daher wird der Bereich unterhalb dieses Wertes in die nachher festgelegten Toleranzstufen eingeteilt.

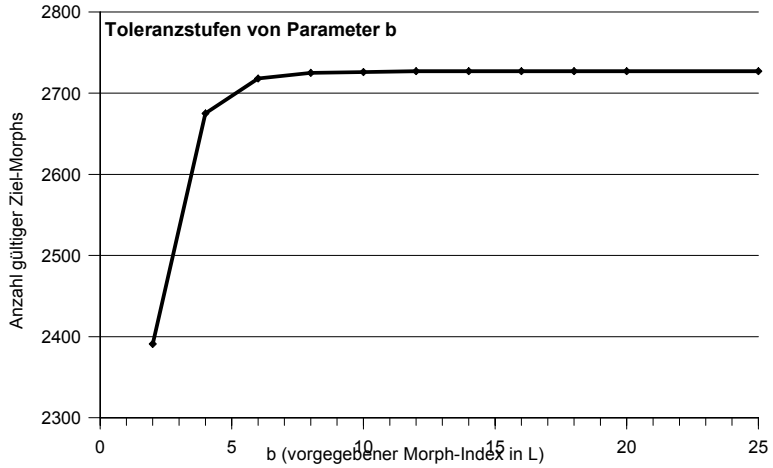


Abb. 5.9. Performanz-Messungen zu a=2, t=20, b={2,...,20}

Die Ergebnisse des letzten Experimentes zum Parameter b (Abb. 5.9) zeigen, dass auch hier bei entsprechend großen Werten eine Sättigung der Performanz eintritt, ab der keine weiteren gültigen Ziel-Morphs generiert werden können.

Auf Basis der oben stehenden Experimente zu den Parametern a, t und b werden daher die folgenden Standard-Parametersätze den drei für den Endnutzer verfügbaren Toleranzstufen *gering*, *mittel* und *hoch* zugeordnet.

Tab. 5.4. Parametersätze für drei Stufen der Fehlertoleranz

Parameter	***gering***	***mittel***	***hoch***
a – max. Anz. Regelanwendungen	2	3	4
t – vorgegebenes Gewicht	10	20	30
b – vorgegebener Morph-Index	10	15	20

Die Separation der Spezial-Submorphs von den normalen Submorph-Regeln eröffnet nun die Möglichkeit, auf verschiedenen Stufen der Fehlertoleranz verschiedene Arten von Spezial-Submorphs anzuwenden. Außerdem kann für jede

Art von Submorph festgelegt werden, ob Morphs, die durch ihre Anwendung entstehen, zusätzlich noch der Anwendung der normalen Submorph-Regeln unterzogen werden sollen. Tabelle 5.5 gibt eine mögliche Verteilung der vorgestellten Submorph-Arten auf die drei Toleranzstufen wieder.

Tab. 5.5. Anwendung von Spezial-Submorphs auf Stufen der Fehlertoleranz

Submorph-Art	***gering***	***mittel***	***hoch***
normale Submorph-Regeln (R)	ja	ja	ja
$c \rightarrow \varepsilon$, ($c \in \Sigma$) DEL	ja	ja	ja+R
$xy \rightarrow yx$, ($x,y \in \Sigma$) SWAP	ja	ja	ja+R
$\varepsilon \rightarrow$ _ (Leerzeichen)	ja	ja	ja+R
$\varepsilon \rightarrow$ - (Bindestrich)	ja	ja	ja+R
$\varepsilon \rightarrow$? INS	nein	ja	ja+R
$c \rightarrow$?, ($c \in \Sigma$) SUBST	nein	ja	ja+R

Der auf Seite 61 angegebene Algorithmus 5-1 „*Öffentliche Schnittstelle zum Morpher*“ verwendet bisher nicht die neu eingeführten Merkmale der *Spezial-Submorphs* und der *Fehlertoleranz-Stufen*. Daher wird nun ein entsprechend angepasster Algorithmus vorgestellt.

In Algorithmus 5-5 (siehe unten) werden abhängig von der Toleranzstufe λ die Methoden `morphDel`, `morphSwap`, `morphInsSpace`, `morphInsDash`, `morphInsQuest`, `morphSubstQuest` aufgerufen. Diese generieren aus dem in ihrer Konfiguration übergebenen Start-Morph S eine Menge von Morphs, indem der entsprechende Spezial-Submorph sukzessive an jeder Zeichenposition in der Zeichenkette s von S angewendet wird. Aufgrund ihrer Trivialität wird auf eine genauere Beschreibung dieser Algorithmen hier verzichtet.

In Schritt 11 von Algorithmus 5-5 werden bei höchster gewählter Toleranzstufe die durch die Spezial-Submorphs erzeugten und in F gesammelten Morphs jeweils noch einer rekursiven Anwendung der normalen Submorph-Regeln unterzogen. Darauf wird die Menge F mit der Menge E vereinigt und in Schritt 13 werden wie bei der ersten Variante der Morpher-Schnittstelle (Algorithmus 5-1) Morphs mit nicht-minimalen Gewichtskosten ausgefiltert.

1: **Input:** Zeichenkette s,
Fehlertoleranz-Stufe $\lambda \in \{0,1,2\}$
2: **Output:** Morpher-Ergebnis E
3: Bestimme (abhängig von λ) die Werte für a, t, b
4: Erzeuge Start-Morph $S=(s\,,0)$ und Morpher-Konfiguration $K=(S\,,a\,,t\,,b)$
5: Setze $E=\{S\} \cup$ `morphSub`$(K,1)$ *// siehe Algorithmus 5-2*
6: Setze $F=$`morphDel`(K)
7: Setze $F=F \cup$ `morphSwap`(K)
8: Setze $F=F \cup$ `morphInsSpace`(K)
9: Setze $F=F \cup$ `morphInsDash`(K)
10: Falls $(\lambda>0)$
10.1: Setze $F=F \cup$ `morphInsQuest`(K)
10.2: Setze $F=F \cup$ `morphSubstQuest`(K)
11: Falls $(\lambda=2)$: für alle Morphs $W \in$ F
11.1: Erzeuge temporäre Konfiguration $K'=(W\,,a\,,t\,,b)$
11.2: Setze $F=F \cup$ `morphSub`$(K',1)$ *// siehe Algorithmus 5-2*
12: Setze $E=E \cup F$
13: Für jedes Morph-Paar $W=(w,z)$ und $V=(v,x)$ mit $w=v$ und $W,V \in E$
13.1: Lösche aus E den Morph V falls $x \geq z$ oder
13.2: Lösche aus E den Morph W falls $z>x$
14: Falls $|E|>b$
14.1: Bestimme das Gewicht γ des b. Morphs in der nach Morph-Gewicht aufsteigend sortierten Menge E.
14.2: Lösche aus E alle Morphs $W=(w,z) \in E$ mit $z>\gamma$.
15: $\Rightarrow$FERTIG

Algorithmus 5-5. Öffentliche Schnittstelle zum Morpher mit Spezial-Morphs

5.8 Automatische Justage der Regelgewichte

Die Erstellung einer Submorph-Regel besteht aus zwei Teilen: Erstens der Festlegung von Quell- und Ziel-String und zweitens der Festlegung eines geeigneten Gewichtes.

Die Auswahl von Quell- und Ziel-String in einer natürlichen Sprache ist für einen Muttersprachler mit etwas Fachwissen aus der Domäne des Textkörpers i.A. eine zu bewältigende Aufgabe: Es müssen Paare von Zeichen gefunden werden, die innerhalb der Sprache ähnlich klingen aber unterschiedlich geschrieben werden (z.B. `ö`→`oe`), ergänzt um typische Problemfälle (z.B. `ß`→`ss`) und Abkürzungen der Textdomäne (z.B. `lösung` → `lsg`.) – vergleiche Abschnitt 5.2.1. Für diese Aufgabe wird also linguistisches Wissen, Kenntnisse zum Klang von Buchstabengruppen und Hintergrundwissen aus der Textdomäne benötigt. Daher ist diese Aufgabe auf absehbare Zeit von Menschen deutlich besser zu lösen als von

Computern. Sämtliche für diese Arbeit verwendeten Regel-Zeichenketten wurden daher manuell erstellt.

Bei der Festlegung der Regelgewichte ist das Problem jedoch anders gelagert: Hier ist nicht das Setzen eines einzelnen Regelgewichtes als kritisch anzusehen, sondern vielmehr das Verhältnis der Gewichte einer ganzen Submorph-Regelmenge zueinander. Denn erst im Zusammenspiel der Regeln, die aufgrund ihrer Einzel-Gewichte die Gewichtssumme eines Morphs ausmachen, liegt die Stärke des WPM-Verfahrens.

In einigen vorhergehenden Abschnitten wurde bereits darauf hingewiesen, dass die Festlegung der Regelgewichte einer großen Menge von Regeln für Menschen eine komplexe und zeitraubende Aufgabe ist. Daher wurde im Rahmen dieser Arbeit ein Verfahren entwickelt, welches zu einer gegebenen Submorph-Regelmenge automatisch die Gewichte einer Feinjustierung unterzieht und sie auf einen Textkorpus anpasst. Dieses *Morph-Feedback* genannte Verfahren wird in diesem Abschnitt vorgestellt (vgl. auch [Ess04b]).

Beim Morph-Feedback wird die in Abschnitt 5.6 vorgestellte Performanz-Messung verwendet, um zu ermitteln, wie stark die veränderten Regelgewichte die Fähigkeit des WPM-Algorithmus beeinflussen, gültige Ziel-Morphs zu generieren.

Die bisher eingeführte Performanz-Messung generiert die Suchmuster aus Textkorpus-Worten und ermittelt dann zu jedem solchen Muster, wieviele Morphs generiert werden können, die wieder Worte aus demselben Textkorpus sind. Diese Vorgehensweise beruht vor allem auf der Tatsache, dass während der Entwicklung und Verbesserung eines Suchverfahrens oft keine (oder nicht ausreichend viele) Suchmuster von Endnutzern zur Verfügung stehen, an denen man das Verfahren verbessern kann. Man ist also auf synthetische Suchmuster angewiesen, was dazu führen kann, dass die Änderungen an den Verfahrensparametern – in diesem Fall also die Justage der Regelgewichte – auch nur eine Verbesserung für solche synthetischen Einsatz-Szenarien bewirken und eine merkliche Aufwertung des Verfahrens für den Endnutzer fraglich bleibt.

Mit dem Werk von P. Altmeyer: *Springer Enzyklopädie Dermatologie, Allergologie, Umweltmedizin* [AB02] steht ein umfangreiches Nachschlagewerk kostenlos im World Wide Web zur Verfügung, welches das hier vorgestellte WPM-Verfahren zur fehlertoleranten Volltextsuche benutzt. Wie bei Werken mit medizinisch-therapeutischen Bestandteilen üblich, ist der Zugang auf Ärzte und Studenten der Medizin beschränkt, deren Überprüfung mittels DocCheck (`http://www.doccheck.de`) stattfindet. Trotz dieser Zugriffshürde wurden seit dem offiziellen Start des Angebotes im Frühjahr 2003 bis Dezember 2004 im Schnitt pro Monat ca. 2.400 Volltextsuchen von den Benutzern des Online-Nachschlagewerkes durchgeführt. Die von den Benutzern gesuchten Muster ha-

ben eine durchschnittliche Länge von 11,6 Zeichen, wobei aufgrund der starken Längenschwankungen (längstes Muster: 150 Zeichen) der Längenmedian von 9 Zeichen wohl die aussagekräftigere Größe sein dürfte.

Diese (anonymen) Suchanfragen konnten also aus den Protokolldateien des Webservers extrahiert werden, und nach Entfernung von redundanten Suchmustern wurden über 13.000 unterschiedliche Begriffe gewonnen, aus denen wiederum 5.424 Muster W zufällig ausgewählt wurden, welche $9 \leq |W| \leq 30$ als gewählte Längenlimitierungen einhalten (Details siehe Kapitel 9 „Anwendungsfälle“). Mit diesen Benutzer-Suchmustern kann nach Durchführung des Morph-Feedback-Verfahrens überprüft werden, ob die rein synthetisch basierte Justage der Regelgewichte zur Verbesserung der Performanz auch die WPM-Performanz verbessert, wenn ausschließlich mit echten Benutzer-Suchmustern gemessen wird.

In diesem Abschnitt wurde bisher beschrieben, wie die Verbesserungen gemessen und wie die Allgemeingültigkeit des Morph-Feedback-Verfahrens überprüft werden kann. Nun sollen die einzelnen Schritte zur automatischen Justage der Regelgewichte im Detail dargelegt werden.

Zuerst werden aus dem Textkorpus Worte extrahiert, die pro Substring über ausreichend Zeichen-Kontext verfügen, aber nicht zu lang sind. In den durchgeführten Experimenten sind das mindestens 9 Zeichen (=Längenmedian der Online-Suchmuster) und höchstens 30 Zeichen. Anschließend wird jedes dieser Muster per WPM-Verfahren fehlertolerant gesucht, und es wird zu jedem generierten Morph, der wieder Bestandteil des Textwortschatzes ist protokolliert, welche Submorph-Regeln an der Bildung dieses so genannten *gültigen Ziel-Morphs* beteiligt waren. Nachdem alle Suchmuster abgearbeitet sind, wird für jede Submorph-Regel die Anzahl ihrer Verwendungen bei der Bildung der gültigen Ziel-Morphs gezählt: Einige Regeln erzeugen viele, andere dagegen weniger gültige Ziel-Morphs, und manche Regeln kommen eventuell überhaupt nie zur Anwendung.

Aufgrund der ermittelten Häufigkeit ihrer Anwendung bekommen die Regeln nun neue Gewichte: Die am häufigsten verwendeten Regeln bekommen das minimale Regelgewicht (nach Definition 5-1: der Wert 1), die nie verwendeten Regeln das maximale Gewicht (in unserer Regelmenge: der Wert 30). So entsteht eine neue Regelmenge mit auf den Textkorpus angepassten Gewichten.

Zusammenfassend lässt sich das hier angewendete Morph-Feedback-Verfahren mit den folgenden Schritten beschreiben:

1.) Extraktion von Worten mit ausreichender Länge aus dem Textkorpus

2.) Zur Beschleunigung des Gesamttests werden durch einen Testlauf mit hoher Toleranz die Muster ermittelt, die überhaupt gültige Ziel-Morphs erzeugen

können. Bemerkung: Dieser Schritt ist jedoch nur zur schnelleren Generierung der zahlreichen Messpunkte in Abb. 5.10 und Abb. 5.11 nötig.

3.) Suche aller dieser Worte mit dem fehlertoleranten WPM-Verfahren

4.) Ermittlung der gültigen Ziel-Morphs zu jedem Wort und Protokollierung der hilfreichen Regeln, die bei der Entstehung der gültigen Ziel-Morphs zur Anwendung kamen

5.) Vergabe neuer Regelgewichte aufgrund der Anzahl der hilfreichen Anwendungsfälle jeder Regel. Maximales Gewicht für niemals hilfreiche Regeln.

6.) Optional: Überprüfung der gesteigerten Performanz der neuen Submorph-Regelmenge mithilfe der Benutzer-Suchmuster

Zunächst wird also das Morph-Feedback-Verfahren auf *synthetische* Suchmuster angewendet. Diese werden dem Textkorpus des Werkes [AB02] entnommen und aus den über 62.000 Wortklassen des Textes werden durch eine Vorab-Analyse 3.475 Wortklassen (durchschnittliche Länge 12,03 Zeichen) ermittelt, die bei hoher Toleranzstufe überhaupt zu gültigen Ziel-Morphs verändert werden können. Dieser Schritt dient der Beschleunigung des anschließenden Performanz-Experimentes, da hiermit die Muster, welche selbst bei hoher Toleranz resistent gegen Bildung von gültigen Ziel-Morphs sind, nicht auch noch mit zahlreichen restriktiveren Toleranz-Parametern wiederholt überprüft werden müssen.

Tabelle 5.6 gibt exemplarisch einige Worte aus dem Textkorpus mit den daraus generierten gültigen Ziel-Morphs wieder. Zusätzlich werden die Regeln aufgeführt, welche an der Bildung der jeweiligen Morphs beteiligt waren, und in Klammern hinter jedem Wort stehen die Auftretenshäufigkeiten des Begriffes in den Ursprungs-Quellen des Original-Textes. Wie aus der Tabelle ersichtlich ist, deuten einige Wortpaare auf typografische Fehler oder inkonsistente Schreibweisen hin. So entstand die Idee, mithilfe des WPM-Verfahrens Techniken zur Fehlerkorrektur zu entwickeln, welche in Kapitel 7 beschrieben werden.

Durch die ausschließliche Anwendung von Submorph-Regeln (ohne Spezial-Morphs) ist die weitaus größte Zahl der Ziel-Morphs mit dem Suchmuster aus dem Text semantisch gleich oder zumindest sehr ähnlich zu sehen. Die beiden letzten Zeilen der Tabelle sollen jedoch belegen, dass es auch Beispiele gibt, bei denen Suchmuster und gültiger Ziel-Morph nicht dieselbe Bedeutung haben (`verdicken`⇒`fertigen` und `vermehren`⇒`wärmeren`), auch wenn sie phonetisch durchaus nahe beieinander liegen. Dies sind meist Fälle, in denen mehr als zwei Submorph-Regeln zur Anwendung kamen, die für sich betrachtet trotzdem wichtig sind. Zum Beispiel ist an der Bildung des eigentlich falschen Morphs `verdicken`⇒`fertigen` die Regel `d`→`t` beteiligt. Diese Regel alleine angewendet wird jedoch auch für das richtige Paar `intermittend`⇒`intermittent` benötigt. Tabelle 5.6 zeigt weitere, ähnlich gelagerte Fälle.

Tab. 5.6. Beispiel-Suchmuster aus [AB02] und ihre gültigen Ziel-Morphs mit beteiligten Regeln

Suchmuster [AB02]	***gültige Ziel-Morphs [AB02]***	***Regeln***
20-nägel-dysthrophie (1)	20-nägel-dystrophie (3) zwanzig-nägel-dystrophie (8)	th→t th→t, 20→zwanzig
abgeblaßte (2)	abgeblasste (2)	ß→ss
aciclovir (72)	acyclovir (9)	i→y
aetiologie (2)	ätiologie (1215)	ae→ä
aktinomykose (28)	aktinomycose (2) actinomycose (2)	k→c k→c, k→c
amitriptylin (10)	amitryptilin (2) amytriptilin (1)	i→y, y→i i→y
calciummangel (2)	kalziummangel (1)	c→k, c→z
galanthamin (1)	gallanthamin (1)	l→ll
hemangioma (43)	hämangioma (18) haemangioma (10)	e→ä e→ae
intermittend (3)	intermittent (5)	d→t
norvegica (10)	norwegica (1)	v→w
proliverative (1)	proliferative (53)	v→f
verdicken (2)	**fertigen** (2)	v→f, d→t, ck→g
vermehren (9)	**wärmeren** (1)	v→w, e→ä, eh→e

Nach Anwendung des WMP-Algorithmus auf alle 3.475 synthetischen Suchmuster zeigt Tabelle 5.7 eine exponentiell fallende Anzahl von gültigen Ziel-Morphs, an deren Bildung die jeweilige Submorph-Regel der Tabellen-Zeile beteiligt war. Die Summe über die 2. Spalte der Tabelle übersteigt die Zahl der Suchmuster, da ein gültiger Ziel-Morph eventuell von mehreren Regeln generiert wurde und in diesem Fall jede solche Regel mitgezählt wird.

Trotz des exponentiellen Abfalls der Anzahlwerte in der 2. Spalte sollten die neuen Regelgewichte eher eine abgeschwächte Verteilung erhalten. Daher wurden für die Neuberechnung der Gewichte die Quadratwurzeln der Anzahlwerte ermittelt und in die weitere Kalkulation einbezogen. Ein Parameter φ wurde eingeführt, um empirisch die Kurve der neuen Gewichte über die nach Anzahlwerten sortierten Regeln flacher zu gestalten, damit die Gruppe der „häufig“ hilfreichen Regeln das minimale Gewicht „1“ erhalten kann.

Tab. 5.7. Einstellung neuer Gewichte für Submorph-Regeln durch automatische Justage

Listenpos. der Regel	*Anzahl gültiger Ziel-Morphs*	*Quelle*	*Ziel*	*Altes, manuelles Gewicht* w_i	*Neues, berechnetes Gewicht* $\bar{w}_i$
1.	497	k	c	1	1
2.	490	c	k	1	1
3.	161	z	c	5	1
4.	158	c	z	5	1
5.	95	ie	y	10	1
6.	94	ß	ss	1	1
7.	86	y	ie	15	1
8.	82	th	t	1	1
43.	16	f	ph	5	5
79.	4	ck	c	20	13
97.	2	zwanzig	20	10	15
134.	1	tz	c	10	16
361.	0	w	ph	25	30

Die neuen Submorph-Regelgewichte lassen sich zum Beispiel nach folgendem Schema berechnen, wobei andere Schemata ebenfalls denkbar sind.

1.) Setze μ_{max} auf die Anzahl gültiger Morphs, an deren Generierung die Regel auf Listenposition 1 beteiligt war. In Tab. 5.7 ist das die Regel k→c mit 497 gültigen Ziel-Morphs.

2.) Setze S_{max1} und S_{max2} auf die vorgegebenen Gewichtswerte t, welche auf den Toleranzstufen *mittel* und *hoch* festgelegt sind. Mit den Werten aus dem vorhergehenden Abschnitt ergibt sich: S_{max1}=20, S_{max2}=30.
Wähle φ=0,9 als Parameter, um die neue Gewichtskurve abzuflachen und setze $\delta = \frac{\sqrt{\mu_{max}}}{S_{max1}} - \varphi$ als globale Stufenhöhe.

3.) Sei μ_i die Anzahl gültiger Ziel-Morphs von Regel r_i. Dann berechnet sich das neue Gewicht $\bar{w}_i$ der Regel r_i als:

$$\bar{w}_i = \begin{cases} S_{max2} & \text{wenn } \mu_i = 0 \\ S_{max1} - \frac{\sqrt{\mu_i}}{\delta} & \text{sonst} \end{cases}$$

Um nun zu überprüfen, inwiefern die neu berechneten Regelgewichte die Performanz des WPM-Verfahrens beeinflussen, wurde neben der manuellen und der justierten noch eine dritte Submorph-Regelmenge mit identischen Quell- und

Ziel-String aber mit zufällig aus dem Intervall [1,...,30] gewählten Regelgewichten generiert. Bei festem Parameter a=3 (max. Anzahl Submorph-Regelanwendungen pro Morph) und b=200 (festgelegter Morph-Index in Morpher-Sprache) wurde der Parameter t (festgelegtes Gewicht) über das Intervall [0,1,2,...,50] variiert. Die aus den Messwerten generierten Performanz-Kurven sind aus Abbildung 5.10 ersichtlich.

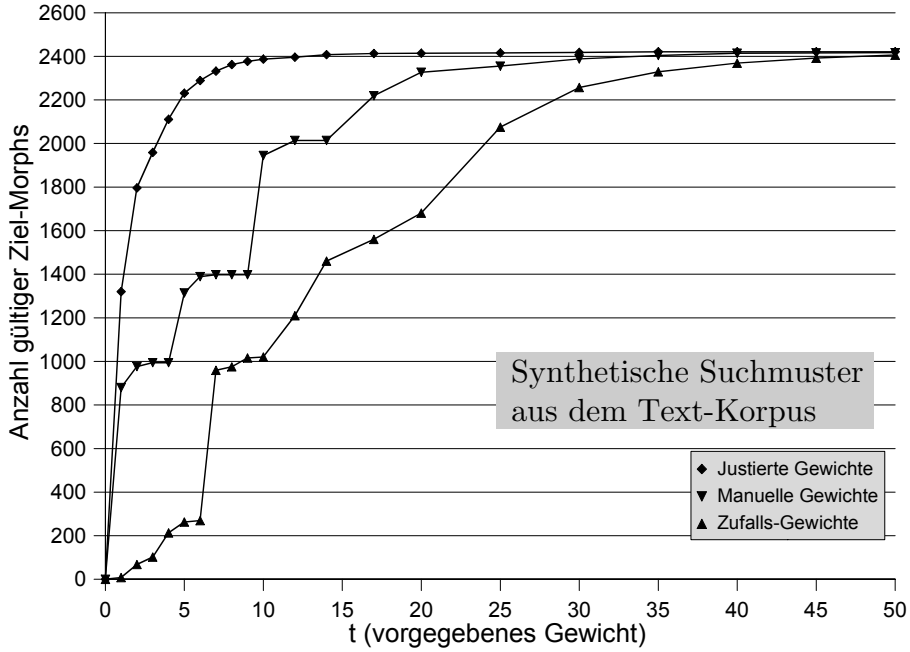


Abb. 5.10. Verbesserte Performanz der automatisch justierten Gewichte gemessen mit synthetischen Suchmustern aus dem Textkorpus

Abbildung 5.10 zeigt, dass die Performanz-Kurve mit den Zufalls-Gewichten einen eher flachen Verlauf besitzt. Die im Rahmen dieses Experiments maximal mögliche Performanz von ca. 2.400 gültigen Ziel-Morphs wird erst spät bei hohem Wert für t erreicht. Die Kurve der ursprünglichen, manuell eingestellten Gewichte hat dagegen einen steileren Anstieg, der einige Treppenstufen aufweist (zur Deutung der Treppenstufen siehe vorhergehender Abschnitt). Die Kurve der durch das Morph-Feedback-Verfahren justierten Gewichtswerte zeigt wiederum einen deutlich steileren Verlauf als die beiden anderen Kurven und strebt schnell gegen den Sättigungswert, ohne auf Zwischen-Niveaus zu verharren.

Wie oben bereits einführend erläutert, werden die mit synthetischen Suchmustern justierten Submorph-Regelgewichte nun mit Suchmustern aus den Protokolldateien des Webservers zum Online-Werk [AB02] daraufhin untersucht, ob auch mit diesen Suchmustern eine gesteigerte Performanz beobachtet werden kann. Wäre dieses der Fall, so kann geschlossen werden, dass man auch ohne Kenntnis tatsächlicher Benutzer-Suchmuster generell die WPM-Regelgewichte

auf die hier beschriebene Art justieren kann und so auf einen Textkorpus anpasst, dass Endnutzer von einer gesteigerten Performanz profitieren.

Wie oben bereits beschrieben, wird das Experiment mit 5.424 unterschiedlichen Benutzermustern durchgeführt. Wieder wird bei festem Parameter a=3 (max. Anzahl Submorph-Regelanwendungen pro Morph) und b=200 (festgelegter Morph-Index in Morpher-Sprache) der Parameter t (festgelegtes Gewicht) über das Intervall [0,1,2,...,50] variiert. Die resultierenden Performanz-Kurven sind in Abbildung 5.11 dargestellt.

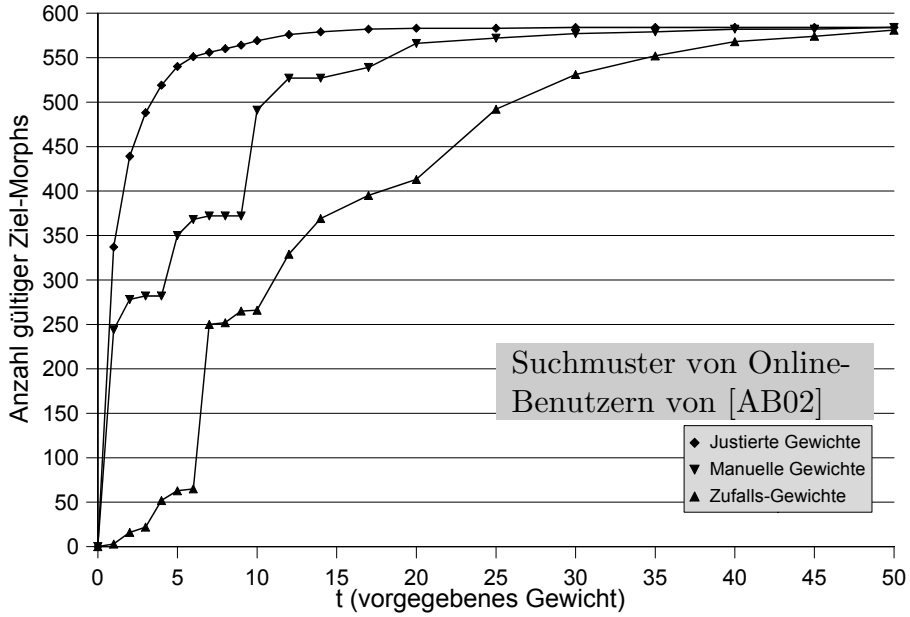


Abb. 5.11. Die justierte Regelmenge aus Abb. 5.10 verbessert auch die Performanz des WPM-Verfahrens auf Benutzer-Suchmustern

Die Kurven aus Abbildung 5.11 zeigen einen zur Abbildung 5.10 analogen Verlauf, wobei die absoluten Werte der Anzahl gültiger Ziel-Morphs aufgrund der unterschiedlich großen Testmuster-Mengen variiert. Wieder dominiert die Kurve der justierten Regelgewichte die beiden anderen Kurven deutlich.

Es zeigt sich also, dass die Technik der automatischen Justage der Submorph-Regelgewichte dazu geeignet ist, mit synthetischen Suchmustern aus dem Textkorpus die Performanz des WPM-Verfahrens nicht nur für diese Suchmuster sondern auch für echte Benutzer-Suchmuster zu verbessern. Dieses Ergebnis ist umso interessanter, da Benutzer-Suchmuster während der Entwicklung eines Suchverfahrens meist gar nicht oder nicht in ausreichendem Maße zur Verfügung stehen.

5.9 Mehrsprachigkeit

Standen im letzten Abschnitt die Gewichte der Submorph-Regeln und ihre automatische Justierung im Mittelpunkt, so beschreibt dieser Abschnitt, wie die ursprünglich nur für deutsche Textkorpora (mit lateinischen und griechischen Fachworten) vorgesehene Regelmenge nun auf englische Texte angepasst wird.

Auch wenn eine solche Schätzung schwierig durchzuführen ist, wird die Anzahl der auf der Welt gesprochenen Sprachen von verschiedenen Quellen auf 6.000 bis 8.000 verschiedene Sprachen geschätzt (siehe [wwwLehmann], Kap. 1.1). Soll die Wichtigkeit einer Sprache gemessen werden an der Anzahl der Menschen, die diese Sprache sprechen, so ist die Zahl der Zweitsprecher nur sehr schwierig zu ermitteln. Die Anzahl der Muttersprachler dagegen lässt sich z.B. durch Volkszählungen deutlich genauer bestimmen.

Tab. 5.8. Rangliste der Sprachen nach Anzahl der Muttersprachler in Mio. [wwwLehmann]

Nr.	***Sprache***	***Mutter-sprachler***	***Zweit-sprecher***	***Summe***
1	Mandarin	874	185	1059
2	Hindi	366	121	487
3	Spanisch	358	95	453
4	Englisch	341	167	508
5	Bengali	207	4	211
6	Portugiesisch	176	15	191
7	Russisch	167	110	277
8	Arabisch	150	50	200
9	Japanisch	125	1	126
10	Deutsch	100	28	128

Englisch liegt also bezüglich weltweiter Anzahl der Muttersprachler auf Rang 4 hinter Mandarin (v.a. China), Hindi (v.a. Indien) und Spanisch. Aufgrund der für wissenschaftliche Publikationen der westlichen Welt herausragenden Stellung der englischen Sprache soll das für wissenschaftliche Texte konzipierte WPM-Verfahren hier an den Quasi-Sprach-Standard Englisch angepasst werden.

Zur Anwendung kommt die so angepasste fehlertolerante Volltextsuche in der Online-Version des Kombinationswerkes aus *Parasitology Research & Encyclopedic Reference of Parasitology* ([CM04] und [Meh01]), und die Anpassung wird dabei vor allem durch Erzeugung von neuen Regeln mit speziellen englischen Quell- und Ziel-Strings erreicht.

Wie in den meisten anderen Sprachen auch, gab es auch im Englischen zunächst nur die mündlich überlieferte Aussprache, ehe zur Fixierung des gesprochenen Wortes die Schrift entwickelt wurde. Als Ausnahmen seien hier etwa Türkisch mit seiner phonetischen Orthographie oder Esperanto als künstlich entwickelte Sprache genannt. Diese Art der Entwicklung der englischen Orthographie bewirkt, dass es im Englischen oft schwierig ist, vom Wortklang zur Schreibweise und umgekehrt von der Schreibweise zur Aussprache zu gelangen, welches wiederum in Bezug auf ein digitales Volltextsuche-System Fehlerquelle für Wortfehler im Werk aber auch im Suchmuster sein kann.

Um die für die englischen WPM-Submorph-Regeln nötigen Quell- und Ziel-Strings zu ermitteln, werden die Ergebnisse des Artikels *Hou tu pranownse Inglish* von ROSENFELDER [wwwRos00] verwendet. Der Autor hat dort für das von ihm entwickelte Programm *sound: The Sound Change Applier* eine Menge von Regeln zusammengetragen, welche zu einem gegebenen amerikanisch-englischen Wort seine Aussprache ermitteln:

> „Everybody agrees that English spelling is horrible. [...] The purpose of this page is to describe those rules -- to explain the system behind English spelling, the rules that tell you how to pronounce a written word correctly over 85% of the time.“ [ebd.]

Ähnlich wie beim WPM-Verfahren geschieht die Berechnung der Aussprache durch Anwendung passender Regeln auf das Ursprungswort. Iterative Anwendung dieser Regeln führt schließlich zum Ziel-String, in welchem die Aussprache in einer Art phonetischem Alphabet kodiert ist. Die Regeln werden von ROSENFELDER in einer Textdatei gespeichert, die von seinem *sound*-Programm zur Laufzeit eingelesen und ausgewertet wird. Die Form einer solchen Regel ist $x/y/z$, wobei x für den Quell-String, y für den Ziel-String und z für den Kontext steht. Beispielsweise wandelt die Regel `c/g/V_V` ein `c` in ein `g`, wenn das `c` zwischen zwei Vokalen steht. Im z-Teil der Regeln muss immer ein Unterstrich für die Position des Quell-Strings stehen, sodass z.B. die Regel `gn/nh/_` immer unabhängig vom Kontext angewendet werden kann.

Im Englischen gibt es (wie auch z.B. im Deutschen) mehr Phoneme als das aus dem Lateinischen stammende Basisalphabet Buchstaben besitzt. Der übliche Ausweg für die Orthographie ist dann, wie ROSENFELDER schreibt, der Umweg über Digraphen – also Kombinationen von Buchstaben, die untrennbar für einen bestimmten Laut stehen (im Englischen z.B. `th`, `sh`; im Deutschen z.B. `ch`, `ck`). Die Darstellung der für das Englische benötigten Phoneme in der normierten IPA-Schreibweise (IPA=International Phonetic Association, [wwwIPA]) wird von ROSENFELDER mittels einfacher ISO Latin-1 Buchstaben durchgeführt. So kann er seine Ersetzungsregeln in einfachen 8-Bit Textdateien speichern, ohne

auf Multi-Byte Unicode-Dateien zurückgreifen zu müssen. Dabei verwendet er teilweise diakritische Zeichen, wie die Beispiele in Tabelle 5.9 zeigen.

Tab. 5.9. Beispiele für phonetische Schreibweisen von Vokalen und Konsonanten in ROSENFELDERS „General American Dialect" mit IPA-Schreibweise, Latin-1-Schreibweise und Beispiel-Worten (aus [wwwRos00]).

IPA	*ROSENFELDER*	*Beispiel*
æ	***â***	rat
v	***v***	vine
aw	***ôw***	crowd, loud
oj	***öy***	boy, droid
θ	***+***	thin
ð	***±***	this

In seinem Artikel trägt ROSENFELDER zunächst informell über 50 Regelgruppen zusammen, welche jeweils einen oder mehrere Buchstaben (oft in festgelegtem Kontext) in Phoneme wandeln. Beispielhaft werden hier Regel 4 bis 8 des Abschnittes „*The notorious* ***gh***" wiedergegeben:

4. Before a vowel, gh becomes ***g***: ghost = ***göst***
5. gh turns a preceding single vowel long: right = ***rït***
6. aught and ought become ***òt***: daughter = ***dòt@r***, sought = ***sòt***
7. Any other ough becomes ***ö***: dough = ***dö***
8. Elsewhere, gh is simply dropped: freight = ***frät***

Gegen Ende seines Artikels zeigt ROSENFELDER die Möglichkeit auf, mithilfe dieser Regeln auch die umgekehrte Richtung zu gehen und von der Aussprache zur korrekten Schreibweise eines Wortes zu gelangen. Auch wenn dieser Weg vom Autor als deutlich schwerer und fehlerträchtiger angesehen wird, gibt er eine Tabelle an, welche zu einem Phonem die verschiedenen möglichen Schreibweisen auflistet.

Diese Tabelle zusammen mit den von ihm vorher zusammengetragenen Regelgruppen wird benutzt, um die für die WPM-Regeln benötigten unterschiedlichen Schreibweisen zu einem Phonem (oder einer Phonem-Gruppe) zu ermitteln. Zusätzlich wurden noch Submorph-Regeln für englische Zahlen (20⇔twenty) und typische Problemfälle aufgrund unterschiedlicher Schreibweise von amerikanischem und britischem Englisch aufgenommen (Tabelle 5.10).

Tab. 5.10. Unterschiedliche Schreibweisen von Worten in britischem und amerikanischem Englisch (nach [TWWCP02]).

britisch / amerikanisch	*Beispiel (brit.)*	*Beispiel (amerik.)*
-ou- / -o-	behaviour	behavior
-ae- / -e-	gynaecology	gynecology
-oe- / -e-	diarrhoea	diarrhea
-ph- / -f-	sulphite	sulfite
-pp- / -p-	worshipped	worshiped (worshipped)
-ll- / -l-	cancelled	canceled (cancelled)
-l- / -ll-	skilful	skillful
-c- / -k-	sceptic	skeptic
-que / -ck	cheque	check
-xion / -ction	inflexion (inflection)	inflection
-ce / -se	defence	defense
-re / -er	centre	center
-gue / -g	catalogue	catalog (catalogue)

Aus Tabelle 5.10 wird offensichtlich, dass zahlreiche Submorph-Regeln, die bereits für die deutsche Sprache hilfreich sind, auch im Englischen wichtig für die Überwindung von britisch-amerikanischen Dialekt-Unterschieden sind (z.B. ph⇔f, pp⇔p, c⇔k), aber es finden sich auch Paare, die spezifisch für britisches und amerikanisches Englisch sind (z.B. ou⇔o, que⇔ck, gue⇔g).

Durch die beiden oben beschriebenen Quellen wurden für die englische Sprache 30 Phonem-Gruppen mit über 900 WPM-Submorph-Regeln gewonnen, von denen hier nun die Phonem-Gruppe „o/ou/ow“ exemplarisch aufgeführt wird (siehe unten stehende Tabelle 5.11). Diese soll den englischen „o/ou“-Laut (in Worten wie z.B. road, though, bureau, snow, toe, caught, court) in verschiedene Schreibweisen überführen.

Dabei wird bewusst in Kauf genommen, dass hierdurch teilweise Schreibweisen ineinander überführt werden können, die Muttersprachler eventuell deutlich in der Aussprache differenzieren können. Im wissenschaftlichen Bereich – speziell bei internationalen Journals – werden die Texte jedoch oft von Nicht-Muttersprachlern verfasst, die ähnlich klingende Schreibweisen eventuell eher verwechseln. Und auch auf Seiten des Endnutzers muss man davon ausgehen, dass ein englisch-sprachiges internationales Journal (wie *Parasitology Research*), welches über das WWW nutzbar ist, auch zahlreiche Zweitsprecher anzieht, die sich der korrekten englischen Schreibweise eines Begriffes nicht immer sicher sind.

Tab. 5.11. Beispiel-Tabelle mit Gewichten zur englischen Submorph-Gruppe (o/ou/ow/..)

	o	a	ou	oo	ow	oa	oe	aw	au	eau	ough	augh
o	-	1	5	1	5	5	5	5	5	5	5	1
a	1	-	5	15	5	5	5	5	5	5	5	5
ou	1	1	-	15	1	2	2	2	2	2	2	1
oo	1	5	-	-	15	15	15	15	15	15	15	15
ow	1	1	1	15	-	2	2	2	2	2	2	2
oa	1	1	2	15	2	-	2	2	2	2	2	2
oe	1	1	2	15	2	2	-	2	2	2	2	2
aw	1	1	2	15	2	2	2	-	2	2	2	2
au	1	1	2	15	2	2	2	2	-	2	2	2
eau	1	1	2	15	2	2	2	2	2	-	2	2
ough	1	1	2	15	2	2	2	2	2	2	-	2
augh	1	1	1	15	2	2	2	2	2	2	2	-

Mit den neu gewonnenen englischen Submorph-Regeln wurden nun sämtliche Worte aus dem Wortschatz des englisch-sprachigen Kombinationswerkes aus *Parasitology Research & Encyclopedic Reference of Parasitology* ([CM04] und [Meh01]) daraufhin untersucht, ob mittels WPM ein Wort gefunden werden kann, welches wiederum Bestandteil desselben Wortschatzes ist. Abschließend werden hier nun einige der gefundenen Worte aufgeführt. Für eine detailliertere Auflistung sei auf das Kapitel 9 „Anwendungsfälle“ verwiesen.

Beispiel 5-10. Morphs aus *Parasitology* (mit englischen WPM-Regeln)

1-day-old (6)	one-day-old (1)
acknowledgements (359)	acknowledgments (4), acknowlegements (1)
continuously (57)	continously (1)
favorable (29)	favourable (29)
hematophagous (12)	hematophageous (1)

5.10 Messung von Precision und Recall

Neben objektiv messbaren Kenngrößen wie Laufzeit und Speicherbedarf ist auch die „Qualität“ eines fehlertoleranten Suchverfahrens eine interessante Größe. Im Gegensatz zu den beiden anderen genannten Kenngrößen beinhaltet die Quali-

tätsbewertung einer Fehlertoleranz aber zu leicht subjektive Erwartungen und die Ergebnisse sind eventuell schwer übertragbar oder nicht reproduzierbar. Zu unterschiedlich können die (subjektiven) Anforderungen und Erwartungen an eine fehlertolerante Suche sein.

Wie soll nun also die „Qualität“ einer fehlertoleranten Suche möglichst objektiv und reproduzierbar gemessen werden? Im Informatik-Forschungszweig *Information Retrieval* hat sich zur Bewertung und zum Vergleich von (fehlertoleranten) Suchmethoden die Messung von *precision and recall* etabliert. Precision steht hierbei für *Präzision* bzw. *Genauigkeit* eines Suchergebnisses und Recall für dessen *Vollständigkeit*, wobei im Folgenden die englischen Begriffe bevorzugt verwendet werden. Diese beiden Parameter sind üblicherweise voneinander abhängig: Während man die Suchschärfe des Algorithmus von exakt nach tolerant verändert, nimmt die Vollständigkeit der Suchergebnisse zu, wobei die Genauigkeit abnimmt. Es kommen also bei immer toleranteren Suchen nach einem Muster q mehr und mehr gewünschte Ergebnisse, aber im gleichen Zug eben auch zunehmend unerwünschte Ergebnisse vor (vgl. dazu Abb. 5.13).

Dieser Abschnitt erläutert zunächst, wie Precision und Recall definiert werden. Anschließend werden einige Kritikpunkte bezüglich einer solchen Messung dargelegt, worauf die Messung von Precision und Recall mit WPM-Verfahren und Edit-Distanz auf dem *HagerROM*- und dem *Altmeyer*-Textkorpus durchgeführt wird.

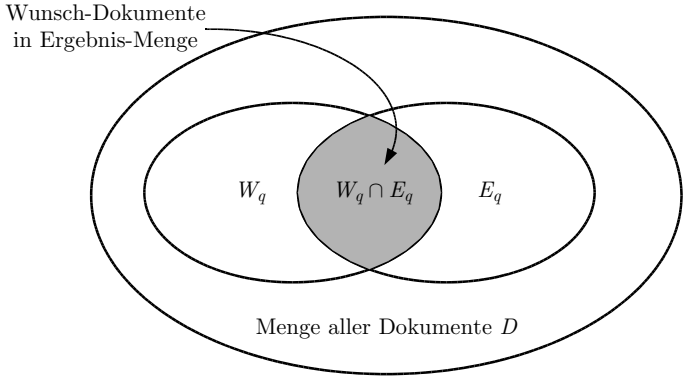


Abb. 5.12. Dokumentmengen bei der Berechnung von Precision und Recall

Nach der üblichen Auffassung benötigt man für die Messung von Precision und Recall zunächst eine Menge von Suchbegriffen und eine Menge von Dokumenten D. Zu jedem Suchbegriff q wird die gewünschte, vollständige Menge der Wunsch-Dokumente W_q festgelegt. Mit einem fehlertoleranten Suchalgorithmus werden dann bei unterschiedlichen Toleranzstufen diejenigen Dokumente ermittelt, welche der Algorithmus als Ergebnismenge E_q zu dieser Suchanfrage zurückliefert (vgl. Abb. 5.12). Dann berechnen sich die Werte für die Precision P und den Recall R nach Definition 5-12.

Definition 5-12. Precision und **Recall** berechnen sich wie folgt:
Sei D die Menge aller Dokumente im System
$W_q \subseteq D$ die Menge der (apriori) Wunsch-Dokumente zu einer Anfrage q
$E_q \subseteq D$ die Menge der vom Algorithmus gelieferten Dokumente q

Precision $P = \frac{|W_q \cap E_q|}{|E_q|}$ ist das Verhältnis der Anzahl gewünschter (und auch gelieferter) Dokumente zur Anzahl insgesamt gelieferten Dokumente (gewünscht oder nicht).

Recall $R = \frac{|W_q \cap E_q|}{|W_q|}$ ist das Verhältnis der Anzahl gewünschter (und gelieferter) Dokumente zu Anzahl der a priori gewünschten Dokumente.

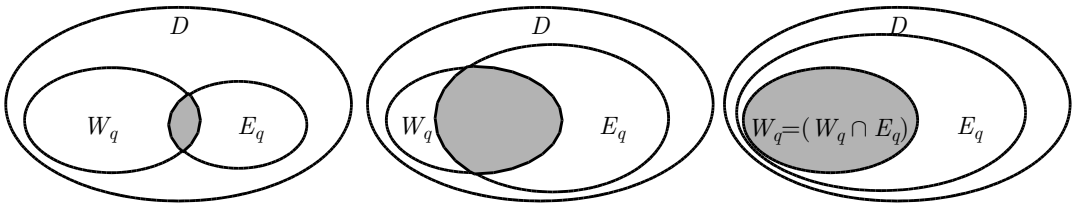


Abb. 5.13. Beispielhafte Entwicklung der Ergebnismenge E_q eines typischen Suchsystems bei zunehmender Fehlertoleranz. Die Menge der Wunsch-Dokumente W_q und die Menge aller Dokumente D bleiben konstant).

Ein ideales System zur Volltextsuche würde also bei zunehmender Toleranz zunehmend die Mengen W_q und E_q zur Überdeckung bringen, bis schließlich W_q=E_q gilt. Diese Anforderung können reale Systeme in der Regel jedoch nicht erfüllen. Vielmehr wird mit zunehmender Toleranz bei der Suche nicht nur die Menge $W_q \cap E_q$ größer sondern auch die Ergebnismenge E_q, sodass sich eine Entwicklung ähnlich Abb. 5.13 ergibt. Bei der Entwicklung eines Suchsystems gilt es also abzuwägen, ob eher auf Vollständigkeit der Suchergebnisse und damit möglichst weiter Überdeckung der Menge W_q abgezielt wird, oder ob das System auf möglichst hohe Reinheit (wenig unerwünschte Suchmuster-Varianten) der Ergebnisse optimiert wird.

5.10.1 Kritik an Precision und Recall

Auch wenn die Messung von Precision und Recall ein etabliertes Vorgehen zur Qualitätsbewertung von unscharfen Suchverfahren ist, so gibt es doch zahlreiche Kritikpunkte an diesem Verfahren. Die Kritik stützt sich vor allem auf die apriori unter Laborbedingungen stattfindende Festlegung der Menge W_q von gewünschten Dokumenten zu einem Suchmuster q.

> "An information need cannot be fully expressed as a search request that is independent of innumerable presuppositions of context - context that itself is impossible to describe fully, for it includes among other things the requester's own background of knowledge." [Swa88]

SWANSON kritisiert hier also die Tatsache, dass bei Messungen von Precision und Recall nicht berücksichtigt wird, dass verschiedene Benutzer eines Suchsystems bei der Suche nach ein und demselben Pattern abhängig von ihrem Wissenshintergrund und dem Kontext ihres Informationsbedürfnisses durchaus unterschiedliche Mengen von „gewünschten Dokumenten" haben können. Diese Mengen wiederum können von der für das Experiment durch Experten der jeweiligen Textdomäne festgelegten Menge der „gewünschten Dokumente" abweichen. BAEZA-YATES und RIBERO-NETO äußern in *Modern Information Retrieval* vergleichbare Kritik:

> "Recall and precision are based on the assumption that the set of relevant documents for a query is the same, independent of the user. However different users might have a different interpretation of which document is relevant and which is not." [BR99], S. 83

So kann also ein Verfahren, welches die im Labor festgelegte Menge an Wunsch-Dokumenten schlecht überdeckt, für einen Benutzer trotzdem die ideale Treffermenge produzieren und umgekehrt ein im Labor erfolgreiches Verfahren verschiedene Benutzer eventuell nicht zufrieden stellen. Ein weiterer Kritikpunkt liegt in einem anderen Aspekt der Auswahl von W_q begründet:

> "[...] the proper estimation of maximum recall for a query requires detailed knowledge of all the documents in the collection. With large collections, such knowledge is unavailable." [ebd.], S. 81

Die Autoren beschreiben hier das Problem, dass in wirklich umfangreichen Werken (wie z.B. Enzyklopädien, Journal-Sammlungen oder digitalen Bibliotheken) niemand eine vollständige Aussage darüber machen kann, welche Dokumente zu einer Anfrage gewünscht wären, da niemand wirklich alle Dokumente kennt.

Ein weiteres Problem ist, dass bei den üblichen Verfahren zur unscharfen Volltextsuche ab einem gewissen Punkt alles auf den Vergleich von Zeichenketten hinausläuft. Diese technisch-syntaktische Sicht, die zur Implementierung nötig ist, kann jedoch eine semantische Sicht auf Suchanfragen (z.B. Verwendung von Synonymen etc.), wie sie aber bei der Auswahl der Menge W_q stattfindet, niemals ganz befriedigen.

5.10.2 Experimente zu Precision und Recall

Trotz der oben aufgeführten Kritik werden in diesem Abschnitt nun die Ergebnisse einiger Precision-und-Recall-Experimente zum WPM Verfahren aufgeführt, um WPM in einer standardisierten Form vergleichbar mit anderen Ansätzen zu machen.

In den folgenden Experimenten wird WPM mit der weit verbreiteten Edit-Distanz (siehe Abschnitt 4.2) verglichen. Dieser Vergleich ist insofern zunächst schwierig, weil in der für diese Arbeit entwickelten Implementierung das WPM-Verfahren, von einem Suchmuster ausgehend, möglichst sinnvolle ähnliche Suchmuster generiert und dann per Substring-Suche überprüft, ob (und wenn ja an welchen Stellen) das Muster im Text vorkommt. Im Gegensatz dazu benötigt eine einfache Edit-Distanz-Suche, wie sie zu Vergleichszwecken hier implementiert wurde, immer Quell- und Ziel-Wort, um die Distanz zwischen beiden zu berechnen. Voraussetzung dafür ist dann jedoch, dass der Text vorher in Worte zerlegt wird, was wiederum nicht einer Substring-Suche entsprechen würde.

Aus diesem Grund wurde zu den zu untersuchenden Texten zunächst jeweils der Wortschatz generiert, wobei als Worttrenner die üblichen Sonderzeichen (. , ; ? ! usw.) gewählt wurden. Worte mit Bindestrich wurden komplett aufgenommen, und zusätzlich wurden alle ihre an mindestens einen Bindestrich grenzenden Fragmente in den Wortschatz aufgenommen. Das Verfahren läuft also analog dem Verfahren zur Generierung der synthetischen Test-Suchmuster bei der Messung der Performanz ab (vgl. Abschn. 5.6, S. 71).

Anschließend wurden, um die zu überprüfende Menge der Suchmuster-Varianten kontrollierbar groß zu halten und trotzdem eine Substring-Suche mit einer Wortschatz-Suche zu simulieren, alle Worte aus dem Wortschatz ebenfalls als „gefundene Suchmuster-Varianten“ gezählt, die vor und nach der eigentlichen Suchmuster-Variante noch bis zu drei zusätzliche Zeichen enthalten. Wenn also der Algorithmus z.B. zum Suchmuster `ätherisch` zunächst nur die Variante `aetherisch` generiert hat, wurden aus dem Wortschatz die Worte `aetherische`, `aetherischen` und `aetherischer` ermittelt und ebenfalls zur Menge der Suchmuster-Varianten hinzugefügt. Da diese Art der Wortschatz-Substringsuche auch für das WPM-Verfahren angewendet wurde, ist es möglich, die beiden Ansätze WPM und Edit-Distanz einem fairen Vergleich zu unterziehen.

Um einen Teil der oben aufgeführten Kritik an Precision und Recall abzuschwächen wurde bezüglich eines konkreten Suchmusters auf eine stark subjektive Klassifizierung ganzer Dokumente in „*gewünscht*“ und „*nicht gewünscht*“ verzichtet und statt dessen die deutlich einfachere Klassifizierung der Suchmuster-Varianten in „*synonym*“ und „*nicht synonym*“ vorgenommen. Es wird also beispielsweise als positiv bewertet, wenn ein Verfahren auf der Suche nach `kalzium` auch

alle Dokumente anzeigt, in denen `calcium` auftritt. Aber es wird als negativ bewertet, wenn auch alle Dokumente angezeigt werden, in denen `kalium` auftritt.

Zusammenfassend besteht das hier angewendete Verfahren zur Messung von Precision und Recall von WPM und Edit-Distanz aus folgenden Schritten:

1.) Wortschatz-Generierung (Zerlegung des Textkorpus in Worte)

2.) Festlegung der Menge der Suchmuster (siehe unten)

3.) Ermittlung der Ergebnisworte zu jedem Suchmuster mit beiden Verfahren

   (a) *WPM*: Mit den Toleranzstufen T={gering, mittel, hoch} jeweils Generierung aller Morphs mit exakter Übereinstimmung zu einem Wortschatz-Wort und anschließende Analyse, welche Worte aus dem Wortschatz diese Treffer-Morphs mit bis zu drei Zeichen davor und dahinter als Substring enthalten. Diese Liste heißt *Ergebnisworte* (des WPM).

   (b) *Edit-Distanz*: Ermittlung aller Wortvarianten aus dem Wortschatz, die zum Suchmuster eine maximale Distanz D={1,2,3} haben und anschließende Analyse, welche Worte aus dem Wortschatz diese Wortvarianten mit bis zu drei Zeichen davor und dahinter als Substring enthalten. Diese Liste heißt *Ergebnisworte* (der ED).

4.) Durchsicht aller so gefundenen Ergebnisworte daraufhin, ob sie zu dem jeweils vorgegebenen Suchmuster als *gewünschtes Ergebniswort* gelten können. Dabei werden beide Ansätze mit jeweils allen untersuchten Toleranzstufen berücksichtigt. Die dabei entstehende Liste heißt *Wunsch-Worte*.

5.) Zu beiden Ansätzen und jeweils auf den drei Toleranzstufen wird jeweils ein Precision- und ein Recall-Wert berechnet. Aus oben erwähnten Gründen stehen jetzt die Mengen W_q und E_q nicht mehr für Mengen von Dokumenten, sondern für Mengen von Worten: E_q die Ergebniswortliste und W_q die Wunsch-Wortliste jeweils zu einem Suchmuster q. Zur Berechnung von P und R wird jeweils über alle n Suchmuster hinweg aufsummiert:

$$\text{Precision } P = \frac{\sum_{q=1}^{n} |W_q \cap E_q|}{\sum_{q=1}^{n} |E_q|} \quad \text{und Recall } R = \frac{\sum_{q=1}^{n} |W_q \cap E_q|}{\sum_{q=1}^{n} |W_q|}$$

Die oben beschriebenen Messungen werden an den Texten der Werke *HagerROM* und *Altmeyer* durchgeführt. Bezüglich dieses Experimentes ist das Werk *HagerROM* aufgrund seines Textumfanges und der dadurch vorhandenen großen Vielfalt von Schreibvarianten interessant. Bei Messungen bezüglich des Werkes *Altmeyer* soll ausgenutzt werden, dass durch seine Präsenz im WWW zahlreiche von Benutzern eingegebene Suchmuster über die Logfiles der WWW-Server in Erfahrung gebracht werden können. Die Auswahl der Suchmuster für dieses Experiment findet also nicht willkürlich sondern durch die WWW-Benutzer selber statt. Dabei werden speziell die Benutzer-Suchmuster untersucht, die

bei exakter Suche im Werk *keine* Treffer produzieren, denn hier können die beiden Verfahren zeigen, inwiefern falsche Benutzermuster korrigiert werden können und wie groß dabei der Anteil unerwünschter Ergebnisworte wird.

Tab. 5.12. Kenngrößen der Experimente zur Messung von Precision und Recall auf den Texten zu *HagerROM* und *Altmeyer*

	HagerROM	***Altmeyer***
Anzahl Suchmuster	50	60
Anzahl gewünschter Suchmuster-Varianten	265	170
Anzahl Worte im Wortschatz	307.737	93.775

Zunächst wurden die 50 Suchmuster mit dem oben beschriebenen Verfahren auf dem Textkorpus der *HagerROM* gesucht. Exemplarisch sind in Tabelle 5.13 einige Suchmuster mit ihren gewünschten Varianten aufgeführt (die komplette Liste befindet sich im Anhang B ab Seite 211).

Tab. 5.13. Beispiele von Precision-Recall-Suchmustern für den *HagerROM*-Textkorpus

Suchmuster	***gewünschte Suchmuster-Varianten (HagerROM)***
50fach	fünfzigfache, fünfzigfach
adenocarcinom	adeno-carcinoma, adenokarzinomen, adenokarzinom, adenocarcinoma, adenocarcinome, adenocarcinomen
antiarrhythmika	antiarrhythmica, antiarrythmika, antiarrhythmikum, antiarrhythmic, antiarrhythmikums, anti-arrhythmia, anti-arrhythmic, antiarrhythmisch, antiarrhythmische, antiarrhythmischen, antiarrhythmischer, i-antiarrhythmikum, i-antiarrhythmika, iv-antiarrhythmik
hämorrhoiden	hämorroiden, haemorrhoiden, hämorrhoidalen, hémorrhoidal

Auf den drei WPM-Toleranzstufen (gering, mittel und hoch) bzw. mit den drei Edit-Distanzen (1, 2 und 3) werden die in Tabelle 5.14 aufgeführten Anzahlen von gewünschten und nicht gewünschten Suchmuster-Varianten gefunden. Am Beispiel für WPM mit geringer Fehlertoleranz (vgl. erste Datenzeile von Tab. 5.14) berechnen sich bei einer Anzahl von 265 gewünschten Suchmuster-Varianten der Precision-Wert zu $P = (201/307) = 0{,}6547$ und der Recall-Wert zu $R = (201/265) = 0{,}7584$.

Tab. 5.14. Ergebnisse der Precision-Recall-Messung (*HagerROM*) mit Anzahl erwünschter und nicht erwünschter Suchmuster-Varianten

	#erw.	*#n.erw.*	*#Summe*	*Precision*	*Recall*
WPM gering	201	106	307	65,5%	75,8%
WPM mittel	212	168	380	55,8%	80,0%
WPM hoch	226	264	490	46,1%	85,3%
ED=1	130	77	207	62,8%	49,1%
ED=2	222	529	751	29,6%	83,8%
ED=3	255	3117	3372	7,6%	96,2%

Aus den in Tabelle 5.14 aufgeführten Precision- und Recall-Werten ergibt sich der in Abbildung 5.14 dargestellte Graph, bei welchem der Nullpunkt der Abszisse zur besseren Darstellung unterdrückt wurde. Da ein optimales fehlertolerantes Suchsystem bei 100% Precision und 100% Recall arbeiten würde, sind also Messpunkte besser, die sich weiter oben-rechts befinden. Die Abbildung zeigt, dass mit den ausgewählten Suchmustern das WPM-Verfahren auf dem *HagerROM*-Textkorpus der Edit-Distanz überlegen ist.

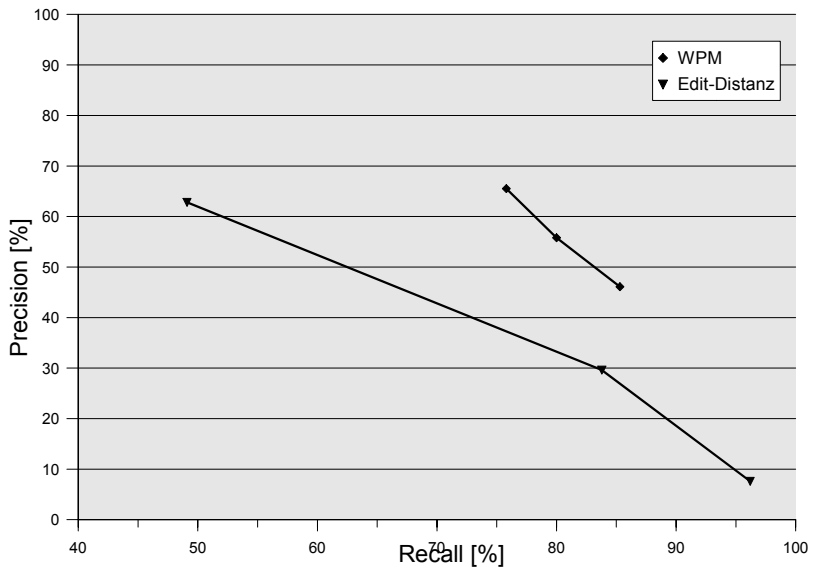


Abb. 5.14. Vergleich von WPM und Edit-Distanz durch Messung von Precision und Recall (HagerROM)

Die Verbindungen zwischen den Messpunkten im Precision-Recall-Diagramm dienen nur der besseren grafischen Darstellung, denn sie gruppieren sichtbar die zu einer Messreihe gehörigen Punkte. Streng genommen liegen zwischen den Messpunkten keinerlei interpolierbare Werte.

Die Suchmuster für den *HagerROM*-Textkorpus müssen manuell ausgewählt werden, da hier keine Benutzer-Suchmuster zur Verfügung stehen. Auch wenn hier größte Sorgfalt bezüglich einer objektiven Auswahl herrschte und so kein Verfahren speziell bevorteiligt oder benachteiligt wurde, werden die Suchmuster für den *Altmeyer*-Textkorpus auf eine noch objektivere Art gewonnen: Aus den Protokolldateien des WWW-Servers, über den die Online-Benutzer ihre Suchen durchführen.

In knapp 22 Monaten sammelten sich in den Protokollen über 53.000 Suchanfragen nach über 13.000 verschiedenen Suchmustern (Längenmedian: 9 Zeichen). Von diesen fast 13.000 Benutzer-Suchmustern produzieren 5.691 Suchmuster bei exakter Suche keine Treffer im Werk. Nachdem per Vorfilter von diesen Null-Treffer-Suchmustern diejenigen mit weniger als neun Zeichen aussortiert werden und außerdem diejenigen mit eingebetteten Leerzeichen entfernt werden, bleiben 1804 Suchmuster ohne Treffer übrig. Muster mit Leerzeichen müssen entfernt werden, weil die Edit-Distanz nur zwischen zwei vorgegebenen Worten den Abstand berechnen kann. Da bei diesem Experiment die Ziel-Worte für diese Berechnungen aber aus dem Wortschatz kommen, enthalten diese prinzipbedingt keine Leerzeichen. Denn die Worte für den Wortschatz werden erst durch Aufspaltung des Textkorpus an Leerzeichen (und anderen Wortbegrenzern) gewonnen.

Um nun den Aufwand zur Kategorisierung der zahlreichen generierten Suchmuster-Varianten in gewünschte und nicht gewünschte Varianten kontrollierbar zu halten, wurden jedoch nicht alle 1804 Benutzer-Suchmuster in das Experiment einbezogen, sondern (nach alphabetischer Sortierung) nur jedes 30. Suchmuster, was zu 60 Benutzer-Suchmustern führt (vgl. Tab. 5.15).

Tab. 5.15. Beispiele von Precision-Recall-Suchmustern für den *Altmeyer*-Textkorpus

Suchmuster	***gewünschte Suchmuster-Varianten (Altmeyer)***
`acanthose`	`akanthose, acanthosis, akanthosis`
`erythematosqamös`	`erythematosquamöse, erythematosquamösen, erythematosquamosa, erythemato-squamöse`
`kummulativ`	`kumulativ, kumulativen, kumulative, cumulative, kumulation`
`pigmentnävus`	`pigmentnaevus`
`zytizerkose`	`zystizerkose`

Wie im oben aufgeführten Fall der *HagerROM* werden nun auch für den Textkorpus von *Altmeyer* hier die Anzahlen der gemessenen erwünschten und nicht erwünschten Suchmuster-Varianten aufgeführt (vgl. Tab. 5.16). In einem vorge-

schalteten Analyseschritt wurden zu den 60 Benutzer-Suchmustern 170 gewünschte Suchmuster-Varianten ermittelt. Diese Zahl liegt im Vergleich zur *HagerROM* wohl vor allem deshalb niedriger, weil der Textkorpus des *Altmeyer*-Werkes nur ca. 12% des Umfanges von *HagerROM* hat. Entsprechend sind bei diesem Werk zu den einzelnen Fachbegriffen auch jeweils durchschnittlich weniger gewünschte Suchmuster-Varianten anzutreffen.

Tab. 5.16. Ergebnisse der Precision-Recall-Messung (*Altmeyer*) mit Anzahl erwünschter und nicht erwünschter Suchmuster-Varianten

	#erw.	***#n.erw.***	***#Summe***	***Precision***	***Recall***
WPM gering	88	86	2	97,7%	50,6%
WPM mittel	117	114	3	97,4%	67,1%
WPM hoch	152	144	8	94,7%	84,7%
ED=1	87	83	4	95,4%	48,8%
ED=2	143	129	14	90,2%	75,9%
ED=3	332	167	165	50,3%	98,2%

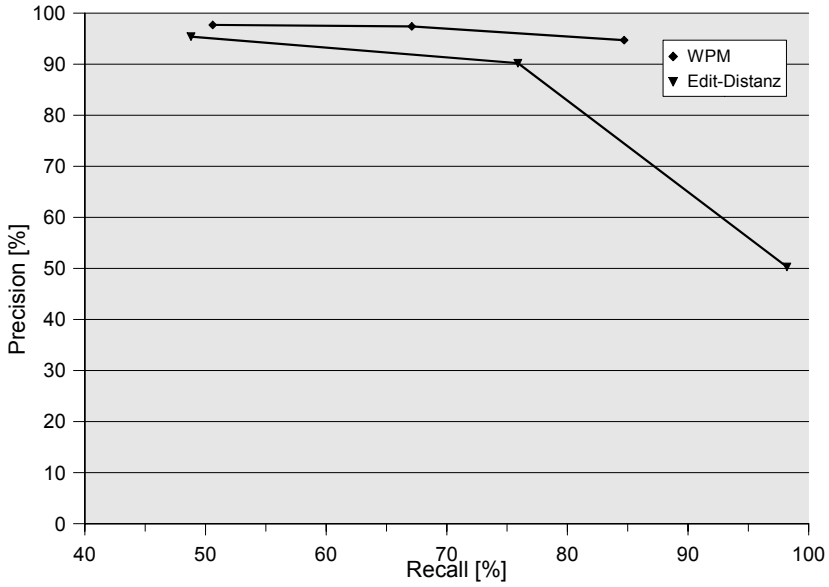


Abb. 5.15. Vergleich von WPM und Edit-Distanz durch Messung von Precision und Recall (Online-Altmeyer)

Aus den in Tabelle 5.16 aufgeführten Werten ergibt sich wiederum ein Diagramm, welches in Abbildung 5.15 wiedergegeben ist. Der Nullpunkt der Abszisse wurde wieder zur besseren Darstellung unterdrückt. Auch bei dem Experi-

ment mit Benutzer-Suchworten liegt die Kurve des WPM-Verfahrens oberhalb und rechts der Edit-Distanz-Kurve, womit belegt wird, dass WPM bei jeweils besserem Recall auch eine bessere Precision hat.

Bei beiden Experimenten dominiert das WPM-Verfahren also das Edit-Distanz-Verfahren. Bei Abbildung 5.14 (*HagerROM*) ist zwar der Abstand zwischen WPM und Edit-Distanz größer, jedoch sind die absoluten Werte schlechter als im Vergleich zu Abbildung 5.15 (*Altmeyer*). Es fällt weiter auf, dass in beiden Experimenten mit der Edit-Distanz Recall-Werte erreichbar sind, die mit dem WPM-Verfahren nicht erreicht werden können. Dies liegt vor allen Dingen in selteneren Wortfehlern begründet, für die es keine Submorph-Regel beim WPM-Verfahren gibt. Wird zum Beispiel im Suchmuster versehentlich für ein ′x′ ein ′p′ eingegeben, so gibt es hierfür keine direkte Submorph-Regel. Und auch die auf höheren Fehlertoleranz-Stufen zunehmend angewendeten Spezial-Submorphs (vgl. Abschnitt 5.4) können nicht immer helfen, wenn diese Fehlerart mehrfach zu korrigieren ist. Denn in der momentanen Implementierung werden zugunsten höherer Precision-Werte die Spezial-Submorphs nur sehr konservativ eingesetzt, um das Suchergebnis nicht mit als unerwünscht vermuteten Suchwort-Varianten zu „überschwemmen“.

Kapitel 6

Implementierung des Volltextsuche-Systems

Wie bereits im vorangegangenen Kapitel beschrieben, arbeitet das WPM-Verfahren als Frontend für ein exaktes Volltextsuche-System. Die für die Implementierung dieser Arbeit hinter den WPM-Algorithmus geschaltete *exakte Volltextsuche* wird nun in diesem Kapitel beschrieben, wobei der besseren Lesbarkeit halber oft schlicht nur die Bezeichnung *Volltextsuche* verwendet wird. Dabei werden hier die Aspekte der Volltextsuche, die bereits in anderen Arbeiten veröffentlicht wurden, nur kurz angerissen und mit entsprechenden Literatur-Verweisen versehen. Die Merkmale der Volltextsuche, die jedoch auf bisher nicht beschriebenen Neuerungen beruhen, werden etwas detaillierter dargestellt.

6.1 Sonderzeichen und chartab

Bevor für eine schnelle Volltextsuche ein Index generiert werden kann, muss entschieden werden, wie die Texte zu formatierten sind, die anschließend durchsuchbar sein sollen. Es muss also festgelegt werden, welche Bit-Kombinationen für welche Glyphe der jeweiligen Schriftsprache stehen. Dabei wird bei dem hier vorgestellten System zwischen externer (Eingangsformat) und interner (Indexformat) Repräsentierung der Strings unterschieden.

Als Eingangsformat für die Index-Generierung wurde die durch das WWW gebräuchliche „Hypertext Markup Language“ HTML gewählt, denn für verschiedene andere Eingangsformate (z.B. XML, SGML, MS-Word) lassen sich leicht Konverter in das offene HTML-Format erstellen. Auch lassen sich die Eingangstexte so leicht mit einem handelsüblichen WWW-Browser kontrollieren, bevor

sie an die Index-Generierung übergeben werden. In HTML können alle von modernen PCs darstellbaren Zeichen leicht codiert werden, wobei der Nachteil von HTML jedoch der relativ große Platzbedarf ist, wenn hier Sonderzeichen in Alternativ-Schreibweisen (ö = `&ouml;` = `&#246;` = `&#x00F6;`) eingebettet sind.

Um die Vielfalt der Codierungsmöglichkeiten von Sonderzeichen in HTML beherrschbar zu machen, wurde mit der so genannten `chartab` eine Übersetzungstabelle eingeführt. Mit dem zugehörigen perl-Modul `chartab.pm` ist es leicht möglich, die verschiedenen Darstellungsformen eines Sonderzeichens ineinander umzuwandeln. So gibt es dort bedingt durch die unterschiedlichen Bedürfnisse verschiedener Projekte zum Beispiel auch Übersetzungsspalten für den unter MS-Windows gebräuchlichen Symbol-Font und für die Sonderzeichen-Darstellungen des 3B2 Druck- und Satz-Systems der Firma Advent.

Weitere zwei Spalten zu den Sonderzeichen der `chartab` geben an, welches zu einem gegebenen großen Sonderzeichen die Kleinschrift-Variante ist und umgekehrt. Dies ist wichtig, da die Volltextsuche nicht zwischen Groß- und Kleinschreibung unterscheiden soll und daher für den Indexaufbau ein beispielsweise im Text vorkommendes großes Delta (Δ) zu einem kleinen Delta (δ) umgewandelt werden muss.

Eine weitere Spalte ordnet den im Deutschen nicht gebräuchlichen diakritischen Zeichen ihr jeweiliges lateinisches Basiszeichen zu (z.B. `à`→`a`, `â`→`a`, `ç`→`c`, `ñ`→`n` usw.). Diese Spalte wurde beim Indexaufbau benutzt, um fremdsprachige Zeichen auch zusätzlich über ihr normales lateinisches Basiszeichen zu finden. Für eine komplette Format-Beschreibung der `chartab`-Tabelle sei jedoch auf den Anhang C (ab Seite 216) verwiesen.

Die bei HTML übliche Entity- (`&ouml;`) oder Character-Reference-Schreibweise (`&#246;`) für Sonderzeichen ist für eine Speicherung der Zeichenketten im Volltextsuche-Index zu speicheraufwändig. Jedoch für umfangreiche wissenschaftliche Werke mit zahlreichen Sonderzeichen (z.B. mathematische, griechische, osteuropäische) ist der sonst üblicherweise in Deutschland verwendete 8-bit Zeichensatz ISO-Latin-1 (ISO-8859-1, abkürzend: Latin-1) bei weitem nicht mehr ausreichend. Für den daher eigentlich konsequenten Schritt hin zum Unicode-Zeichensatz [wwwUnicode] hätte man die Wahlmöglichkeit zwischen zwei üblichen Speichermethoden für die einzelnen Unicode-Zeichen: „Double-Byte Character-Set“ (DBCS, 16-bit) oder das „Unicode Transformation Format 8“ (UTF-8) [RFC3626]. Beide Alternativen sind jedoch mit Nachteilen für den Indexaufbau der Volltextsuche behaftet:

1.) **DBCS:** Hiermit würde für alle Zeichenketten im Index doppelter Speicherplatz belegt (im Vergleich zum 8-bit ISO-Latin-1-Zeichensatz). Außerdem könnte implemetierungstechnisch gesehen dann z.B. ein 16-bit Zeiger auf einen Text-Speicherblock statt 64KB nur noch 32KB Text überdecken.

2.) **UTF-8** ist ein Speicherformat mit unterschiedlicher Byteanzahl für verschiedene Zeichen. Zeichen mit niedrigem Unicode benötigen zur UTF-8-Darstellung nur 1 Byte, Zeichen mit höherem Unicode benötigen dagegen bis zu sechs Byte. Da also die UTF-8 kodierten Zeichen keine feste Bitlänge haben, ist ihre Verwendung mit einigem Mehraufwand verbunden: Random-Access in Texten (z.B. Sprung zum 100. Symbol) und Stringlängen-Bestimmung werden mit UTF-8 deutlich komplizierter und langsamer.

Für die Implementierung der hier beschriebenen Volltextsuche wird daher ein anderer Weg gewählt: Unter den getesteten Anwendungsfällen benötigt selbst das Werk mit dem umfangreichsten Alphabet (*HagerROM*) nur ca. 240 verschiedene Unicode-Zeichen und ca. 180 verschiedene Unicode-Zeichen nach Umwandlung aller Zeichen in die jeweils entsprechende Klein-Darstellung. Es bietet sich also an, aus dem Raum der insgesamt möglichen Unicode-Zeichen eine Abbildung auf ein 8-bit Alphabet durchzuführen, um die Zeichenketten effizient und platzsparend im Index speichern zu können. Als 8-bit Zeichenkodierung wurde der ISO-Latin-1-Zeichensatz gewählt, da hier mit Standard-Editoren die üblichen deutschen und englischen Zeichen unmittelbar lesbar und editierbar sind. Benötigt das Alphabet eines Werkes nach Wandlung in Kleinbuchstaben mehr als 255 verschiedene Zeichen – das Zeichen 0 wird als Stringterminator reserviert – so sind die entsprechenden Datenstrukturen der Volltextsuche zur Speicherung von q-Grammen und Submorph-Regeln anzupassen.

Dabei werden Zeichen mit einem Unicode größer als 255 auf nicht benutzte Zeichen im Latin-1 Bereich 1-255 abgebildet. Da auch die normalen deutschen Zeichen vorher in ihre Kleinschreibweise gewandelt werden ('A' wird zu 'a'), sind sämtliche großen Latin-1-Zeichen frei zur Speicherung von klein-gewandelten Nicht-Latin-1-Zeichen (wie z.B. mathematischen oder griechischen Sonderzeichen). Die Abbildung muss natürlich dynamisch geschehen, denn durch Aktualisierungen und Korrekturen am Textkorpus eines Werkes kann es vorkommen, dass z.B. bisher nicht verwendete Latin-1-Zeichen nun verwendet werden oder dass bestimmte Unicode-Zeichen gar nicht mehr verwendet werden.

Daher wird bei jeder Index-Generierung mit dem perl-Skript `check_sonderzeichen.pl` zunächst überprüft, welche Zeichen überhaupt im zu verarbeitenden Textkorpus verwendet werden. Dabei werden in einem ersten Lauf alle (nach der Wandlung von Groß- in Kleinsymbole) verwendeten kleinen Latin-1-Zeichen als belegt markiert. In einem zweiten Lauf werden dann alle Nicht-Latin-1-Zeichen auf die noch freien Speicherplätze im Intervall [32-255] verteilt. Erst wenn die freien Speicherplätze in diesem Bereich nicht mehr ausreichen, werden die freien Speicherplätze der Kontrollzeichen [1-31] benutzt. Reichen auch diese nicht aus, dann hat das zu verarbeitende Werk eine kleingeschriebene Symbolmenge von über 255 Zeichen und ist mit dem hier vorgestellten Volltextsuche-System erst nach Anpassung der internen Datenstrukturen

durchsuchbar. Das Ergebnis ist ein für den momentanen Textkorpus des betrachteten Werkes gültiges 8-bit Suchalphabet.

Das perl-Skript `check_sonderzeichen.pl` generiert nach jedem Lauf als Ausgabe ein perl-Modul mit dem Namen `suche_alphabet.pm`, welches perl-Code enthält, der durch reguläre Ausdrücke aus HTML-Zeichenketten mit beliebigen im Werk vorkommenden HTML-Sonderzeichen eine Zeichenkette konform zum momentanen Suchalphabet erzeugt. Die von diesem dynamisch erzeugten perl-Modul vorgenommenen Ersetzungen kommen bei der Index-Generierung zum Einsatz, nachdem aus den HTML-Texten die HTML-Layout-Tags (wie z.B. `<b>` für Fettschrift) entfernt werden. So wird ein HTML-Fragment, wie in Beispiel 6-1 gezeigt, in reinen Text gemäß dem momentan festgesetzten Suchalphabet umgewandelt, und die Volltextsuche sucht mit ihrem Index auf den so gebildeten Texten. Für das Beispiel galt eine Abbildung β→`L`, da alle Großbuchstaben – und damit auch das Zeichen L – zunächst nicht belegt und damit für die Aufnahme sonstiger Sonderzeichen frei sind).

Beispiel 6-1. Abbildung HTML⇒Suchalphabet am Beispiel eines Fragmentes aus *HagerROM* (Eintrag: *Petasitidis rhizoma*)

Unicode: Sonstige Verbindungen. Aus dem alkoholischen Auszug wurde β-Sitosterol (umgerechnet 2,6 g [...]

HTML:
```
<i>Sonstige Verbindungen. </i>Aus dem alkoholischen Auszug
wurde <i>&#946;</i>-Sitosterol (umgerechnet 2,6 g [...]
```

Suche:
```
sonstige verbindungen. aus dem alkoholischen auszug
wurde L-sitosterol (umgerechnet 2,6 g [...]
```

Bezüglich der Laufzeit ist die hier vorgestellte Benutzung eines internen Suchalphabetes unkritisch, da lediglich zu Beginn der Volltextsuche einmalig das Benutzer-Suchmuster in die interne Darstellung abgebildet werden muss. Ab dann kann der Algorithmus mit normalen String-Funktionen die so entstandene Zeichenkette weiterverarbeiten. Bei fehlertoleranten Volltextsuchen muss am Ende des Suchprozesses eventuell eine Rückwandlung von einzelnen Sonderzeichen innerhalb von Treffer-Morphs erfolgen. Beide Schritte sind aber, bezogen auf den eigentlichen Rechenaufwand der Volltextsuche vernachlässigbar.

Damit die zahlreichen Sonderzeichen nun für Benutzer suchbar sind, unabhängig davon, ob sie leicht auf der Tastatur eingegeben werden können, interpretiert der Parser der Volltextsuche Zeichenketten in spitzen Klammern als Sonderzeichen. So kann das im obigen Beispiel vorkommende β-Sitosterol durch das Suchmuster `<beta>-sitosterol` gefunden werden. Für weniger erfahrene Benutzer gibt es eine geeignete grafische Benutzeroberfläche, um diese Sonderzeichen-Namen in ein Suchmuster einzufügen.

6.2 Der *q*-Gramm-Index

Die Datenstruktur der *q-Gramme* (manchmal auch *n-Gramme* genannt) zum Aufbau eines Volltextsuche-Index wurde bereits in Abschnitt 3.3 eingeführt (vgl. auch [Ste94], S. 34). Die herausragende Stärke von *q*-Gramm-Indizes liegt vor allem in der Durchführbarkeit einer tatsächlichen *Volltext*suche. Das heißt, dass der indizierte Text nach jedem beliebigen Substring effizient durchsucht werden kann. Diese Eigenschaft steht im starken Gegensatz zu allen wortbasierten Ansätzen, deren Nachteile bereits in Abschnitt 2.1 eingehend erläutert wurden.

Die für die momentane Implementierung des exakten Backends verwendete Volltextsuche hat bereits einige, jeweils von starken Verbesserungen geprägte, Implementierungszyklen durchlaufen. Die erste funktionierende Version verfügte bereits über einen *q*-Gramm-Index und wurde für die *CD Klinische Dermatologie* [Schm97] (der digitalen Version des Buches [Schm94]) von Martin Riedl im Rahmen seiner Diplomarbeit [Ried98] entwickelt. Auf den aus heutiger Sicht nicht besonders leistungsfähigen Windows-PCs (üblich waren um die 200 MHz CPUs) zeigte sich bereits das enorme Potenzial dieser Indexart.

Für größere Werke wie z.B. *HagerROM* [BEH+01] und die mittlerweile parallel entwickelte Fehlertoleranz mittels WPM war der erste Ansatz jedoch nicht skalierbar genug. Daher entwickelte Martin Grimm im Rahmen seiner Diplomarbeit [Gri01] Kompressionsmöglichkeiten für den *q*-Gramm-Index und ein verbessertes exaktes Volltextsuche-System. Das von ihm entwickelte Speicherformat für den *q*-Gramm-Index und seine Algorithmen waren Basis für das in der hier vorliegenden Arbeit verwendete exakte Volltextsuche-Backend.

> „Durch oben beschriebenes Format konnte die Datenmenge der abgelegten *q*-Gramme beim Hager von ca. 517 MB auf ca. 335 MB reduziert werden, d.h. auf etwa 65% des Originalvolumens.“ [Gri01], S. 37

Das Datenformat für den Trie-Header und die komprimiert abgelegten *q*-Gramm-Positionen ist in der genannten Diplomarbeit detailliert beschrieben und durch Experimente in seiner Leistungsfähigkeit bestätigt.

6.3 Merkmale der Volltextsuche

Im folgenden Abschnitt sollen nun die Merkmale der exakten Volltextsuche beschrieben werden, die sich direkter auf den Benutzer auswirken als das darunter liegende *q*-Gramm-Indexformat.

Da die q-Gramm Volltextsuche exakt jeden Substring des Werkes wiederfinden kann, werden Leerzeichen im Suchmuster wörtlich genommen. Das Muster

```
dies ist ein text
```

findet also alle die Stellen im Text, an denen die vier Worte durch insgesamt drei Leerzeichen getrennt sind und genau in dieser Reihenfolge auftreten. Aus pragmatischen Gründen stellt die Vor-Verarbeitungsphase der Index-Generierung dabei sicher, dass Ketten von Whitespace (also Leerzeichen, Umbrüche, Tabulatoren usw.) in den Original-Texten jeweils zu einem Leerzeichen in der internen Textrepräsentation der Volltextsuche verschmolzen werden. Die Volltextsuche interpretiert Leerzeichen nicht implizit als booleschen Und-Operator und weicht hierdurch also von etablierten Suchmaschinen im WWW ab, die bei oben angegebenem Beispiel-Suchmuster alle Dokumente auffinden würden, in denen die vier Worte in beliebiger Reihenfolge und evtl. auch weit auseinanderliegend auftreten.

6.3.1 Boolesche Operatoren

Um nun aber komplexere Suchmuster an die Suche zu übergeben, wurden die folgenden *booleschen Operatoren* in die Abfragesprache eingeführt (vgl. Tab. 6.1). Im strengen Sinne ist der dritte Operator (# Und-Nicht) nicht zu den booleschen Operatoren zu zählen. Die Semantik eines unären Nicht-Operators ist jedoch dem Benutzer nur schwer zu vermitteln. Daher wurde die in der Tabelle erläuterte Funktion des Und-Nicht aufgenommen.

Tab. 6.1. (Boolesche) Operatoren der Abfragesprache der exakten Volltextsuche

Op.	***Beispiel***	***Bedeutung***
`&`	`wolfs&kraut`	*Binärer Und-Operator:* Die Teilmuster links und rechts vom Operator müssen im Dokument vorkommen, wobei deren Reihenfolge unwichtig ist.
`|`	`wolfs|kraut`	*Binärer Oder-Operator:* Mindestens das links oder rechts vom Operator stehende Teilmuster muss im Dokument vorkommen, wobei deren Reihenfolge unwichtig ist.
`#`	`wolfs#kraut`	*Binärer Und-Nicht-Operator:* Findet alle Dokumente, in denen zwar das links vom Operator stehende Teilmuster vorkommt aber nicht das rechte. Vertauschung der Operatoren verändert daher die Semantik des Suchmusters.

Der Und-Operator kann in seiner Funktionalität durch einen in spitzen Klammern folgenden numerischen Ausdruck auf einen maximalen Abstand seiner Teilmuster eingestellt werden. So findet das Suchmuster `wolf&<20>kraut` alle Dokumente in denen die beiden Worte `wolf` und `kraut` vorkommen, wenn deren Trefferpositionen (also die Positionen der 'w' und 'k') maximal 20 Zeichen auseinander liegen (vgl. [Gri01] S. 44ff).

6.3.2 Reguläre Ausdrücke

Bereits in Abschnitt 2.1 im Kapitel „Aufgabenstellung und Definitionen“ wurde darauf hingewiesen, dass die Volltextsuche mit regulären Ausdrücken arbeiten können soll. Dabei kommen die für durchschnittliche Benutzer leichter verständlichen vereinfachten regulären Ausdrücke zum Einsatz (vgl. ebenfalls Abschnitt 2.1).

Beispiel 6-2. Vereinfachte reguläre Ausdrücke mit Beispiel-Treffern

`wolf?kraut`	wolf<u>s</u>kraut	(2)
`wolf*kraut`	wolf<u>s</u>kraut	(2)
	wolf<u>strapp</u>kraut	(6)
	wolf<u>smilch, maulwurfs</u>kraut	(1)

Die Implementierung dieser vereinfachten regulären Ausdrücke mit einer exakten q-Gramm Volltextsuche ist prinzipiell einfach, da lediglich überprüft werden muss, ob die Offsets der q-Gramm-Fragmente im Suchmuster mit den Offsets der q-Gramme im Suchindex übereinstimmen. Dabei erhöht jedes Fragezeichen den Offset der rechten q-Gramme um den Wert 1 und jedes Sternchen bedingt statt einer „=“ Prüfung der rechten Offsets eine „≥“ Prüfung. Für Details zur Implementierung sei auf [Gri01] verwiesen, welcher für den `?`-Platzhalter die Klasse `CFixedCatNode` ([ebd], S. 62) und für den `*`-Platzhalter die Klasse `CStarCatNode` ([ebd], S. 63)implementiert hat.

Deutlich erweiterte Funktionalität der regulären Ausdrücke unter weitgehender Beibehaltung der hohen Suchgeschwindigkeit der q-Gramm Volltextsuche hat Daniel Schreckling in seiner Diplomarbeit [Schr04] entwickelt.

6.3.3 Werkteilsuche und Feldfilter

Ein weiteres Merkmal, das die Benutzer bei der Volltextsuche eines umfangreichen Werkes erwarten, dürfte die Einschränkung des Suchraumes sein. Sehr fle-

xibel ist dabei der Ansatz, zwischen Werkteilen und Feldern zu unterscheiden (vgl. Abb. 2.1 und Bsp. 2-1 ab Seite 14).

Ein *Werkteil* ist dabei ein zusammenhängendes Intervall von `FileIDs` (eindeutige Dateinummer). Bei einem Buch könnte jedes Hauptkapitel ein Werkteil sein, bei *HagerROM* werden zum Beispiel die Werkteile „Drogen“ (pflanzliche Wirkstoffe) und „Stoffe“ (synthetische Wirkstoffe) unterschieden. Mittels eines multiple-choice Dialoges kann der Benutzer so seine Volltextsuche zum Beispiel auf den Werkteil „Drogen“ limitieren und Treffer aus dem Werkteil „Stoffe“ ausblenden.

Ein *Feld* ist dagegen ein semantischer Bereich innerhalb einzelner Dateien oder Dokumente, welche von der Volltextsuche indiziert werden. Um für die Algorithmen die Feldgrenzen erkennbar zu machen, sind in den Quelldateien eindeutige Start- und Stop-Marker gespeichert, welche für den Benutzer sogar unsichtbar sein können. Jeder dieser Marker trägt einen eindeutigen Namen, der aus einer festen Gruppe von gültigen Feldnamen gewählt werden kann. Jeder Werkteil darf dabei einen eigenen Satz von Feldnamen besitzen. Im Werk *HagerROM* besitzt der Werkteil Drogen zum Beispiel u.a. die Felder „Basisangaben“, „Charakteristik“, „Gesetzliche Bestimmungen“, „Pharmakologie“.

Die Werkteilsuche lässt sich einfach dadurch realisieren, dass beim Einlesen der q-Gramm-Listen aus dem Index großräumig die `FileID`-Intervalle von nicht gewünschten Werkteilen übersprungen werden. Prototypisch wurde das in [Gri01] bereits für fest einkompilierte Grenzen der *HagerROM*-Werkteile Drogen und Stoffe realisiert. Die Volltextsuche wurde jedoch durch ein deutlich flexibleres Verfahren erweitert, welches die Werkteile aus der bei der Generierung abfallenden Steuerdatei, der so genannten *Namelist*, entnimmt (siehe Beispiel 6-3). Über einen der Volltextsuche übergebenen Bitvektor kann dann festgelegt werden, welche Werkteile durchsucht werden sollen. Die Kommandozeilen-Version der Volltextsuche würde bei dieser Namelist mit dem Schalter `-w 5` die Werkteile „Drogen“ und „Impfstoffe“ durchsuchen, wobei auch eine explizite Nennung der Werkteile mit Namen möglich ist: `-w "drogen,impfstoffe"`. Für eine komplette Auflistung der möglichen Parameter der Kommandozeilen-Suche sei auf Anhang D (ab S. 220) verwiesen.

Beispiel 6-3. Anfang einer *HagerROM*-Namelist mit 20 Werkteilen
Vor den Werkteilen stehen jeweils 1. und letzter `FileID` und Anzahl `FileIDs`.

```
v5
20     PartCount
1      6269        6269      Drogen
6270   11818       5549      Stoffe
11819  11919       101       Impfstoffe
11920  11950       31        Blutprodukte
[...]
```

Im Gegensatz zur Werkteilsuche beziehen sich die Bereiche der Feldsuche nicht auf `FileID`-Intervalle sondern auf `Offset`-Intervalle innerhalb der vom Index gespeicherten Textdateien. Daher muss für das in der Grimm-Suche noch nicht enthaltene Merkmal „Feldsuche“ ein separater Index angelegt werden. Die Überprüfung eines Treffers bezüglich Einhaltung der gewünschten Feldgrenzen in den inneren Schleifen der q-Gramm-Suche verbietet sich einerseits aus Laufzeit-Gründen und andererseits aus Gründen der Korrektheit: Würde jedes q-Gramm beim Laden daraufhin überprüft, ob es innerhalb der gewünschten Feldgrenzen liegt, müsste eine unnötig große Anzahl von q-Grammen überprüft werden, die im weiteren Verlauf der Suche sowieso aus anderen Gründen ausgefiltert werden (z.B. falscher Offset zu anderem q-Gramm). Außerdem kann erst ganz am Ende der Kern-Suche wirklich sicher entschieden werden, ob ein Treffer komplett innerhalb der gewünschten Felder liegt, weil erst dann sein Offset und seine komplette Länge feststehen – denn ein Treffer kann aus vielen q-Gramm Fragmenten bestehen (vgl. Bsp. 6-4).

Beispiel 6-4. Problemfall bei zu frühem Feldgrenzen-Test
Angenommen der Benutzer hat nach dem Muster `wolf*kraut` gesucht und dabei einige Felder als unerwünscht markiert. Dabei liegen bezüglich einer konkreten Textdatei sowohl alle q-Gramme von „`wolf`“ als auch alle q-Gramme von „`kraut`“ innerhalb der vom Benutzer gewünschten Felder.
Eine zu frühe Entscheidung, diese Fragment-Fundstelle(n) als Feldtreffer zu behalten, könnte sich aber als falsch herausstellen, wenn der `*`-Platzhalter ein nicht-gewünschtes Feld überstreicht, welches zwischen gewünschten Feldern liegt.
Daher kann die Entscheidung, ob ein Treffer komplett innerhalb gewünschter Felder liegt, erst getroffen werden, wenn der Treffer komplett zusammengesetzt ist.

Aus diesem Grund wird das Merkmal der Feldsuche als Trefferfilter *hinter* der eigentlichen Kern-Suche angeordnet. Hier stehen alle Treffer mit ihren Offsets und Längen fest – jetzt kann korrekt entschieden werden, ob ein Treffer komplett in vom Benutzer gewünschten Feldern liegt.

Bei der Generierung des Feldgrenzen-Index muss berücksichtigt werden, dass die Markierungen für die Feldgrenzen zunächst in den mit Layout-Tags versehenen HTML-Quelltexten stehen, die Zählung der Feld-Offsets sich jedoch zur Verwendung innerhalb der Volltextsuche auf die bereinigten Texte beziehen muss. Schließlich hat die Volltextsuche keinerlei Information über Anzahl, Lage und Länge der Layout-Tags innerhalb der HTML-Dateien.

Aus Geschwindigkeitsgründen werden die Feldgrenzen während der Bereinigung der Suchtexte in der binären Arraystruktur gespeichert. In dieser Arraystruktur entspricht jede Datensatz-Zeile einer `FileID` und jede Datensatz-Spalte dem Beginn eines Feldes. Alle Arrayzeilen haben gleiche Breite (fixed record length), sodass schneller und wahlfreier Zugriff auf die Felder jeder gültigen `FileID` gewährleistet ist. Das Ende eines Feldes berechnet sich aus dem Beginn des nächs-

ten Feldes, warum auch ein künstliches „Rest“-Feld eingeführt werden muss, welches den Offset des letzten Zeichens einer jeden Datei (und damit ihre Länge) markiert.

Ebenfalls aus Geschwindigkeitsgründen werden zur Laufzeit aneinander angrenzende und vom Benutzer als gewünscht markierte Felder bei der Initialisierung des Feldfilters zu einem so genannten *Feldblock* „verschmolzen“. Dies bewirkt, dass z.B. bei fünf aneinander grenzenden Feldern nicht fünf Tests, sondern nur ein einziger gemacht werden muss. Diese Einsparung multipliziert sich mit der Anzahl der von der Kern-Suche gelieferten Treffer, denn jeder Treffer muss den Feldfilter passieren. Voraussetzung hierfür ist natürlich, dass die Felder in den HTML-Quelldateien grundsätzlich immer genau aneinandergrenzen und sich keine Texte zwischen den Feldern befinden, die keinem Feld zugeordnet sind.

Das grafische Benutzerinterface von Werkteilsuche und Feldfilter ist in den Abbildungen von Abschnitt 9.1 dargestellt, in dem der Anwendungsfall *HagerROM* detaillierter vorgestellt wird.

6.3.4 Ranking

Ein ebenfalls wichtiges Merkmal einer Textsuche stellt das so genannte *Ranking* dar – also die Bildung einer Rangfolge unter den Dokumenten mit mindestens einem Treffer. Aus der Rangfolge bildet sich dann die Reihenfolge, in der die Trefferdokumente dem Benutzer angezeigt werden: Wichtige Dokumente sollen oben stehen, unwichtigere Dokumente sollen folgen.

Der Stellenwert dieses Merkmals ist erkennbar, wenn man die Geschichte der momentan erfolgreichsten WWW-Suche `google.com` betrachtet: Innerhalb weniger Jahre verwies der „Newcomer“ Ende der 1990er Jahre die damals etablierten Suchmaschinen wie z.B. `yahoo.com` und `altavista.com` auf die Plätze. Dies geschah vor allen Dingen, weil die neue Firma die Treffer mit ihrem *PageRank* (vgl. [BP98], [Pre02]) genannten Algorithmus besser sortierte als die Mitbewerber, und so die Benutzer (bei genügend spezifischen Suchmustern) die gesuchte Information meist gleich auf der ersten Seite erhielten (vgl. [wwwPgRnk]).

Einige übliche Bewertungsstrategien zur Bildung einer Rangfolge unter den Trefferdokumenten zu einer Textsuche werden in der folgenden Liste, ohne Anspruch auf Vollständigkeit, aufgeführt:

- **Absolute Häufigkeit**: Zählt das Vorkommen des Suchmusters in einem Dokument. Je mehr Treffer ein Dokument hat, desto wichtiger wird das Dokument bewertet – unabhängig von der Länge des Dokumentes.

- **Relativierte Häufigkeit**: Berechnet die relative Häufigkeit des Vorkommens der einzelnen Suchbegriffe bezogen auf die Dokumentlänge (sog. Suchmuster Dichte).
- **TF·IDF**: Bedingt durch den Umstand, dass einige Teile eines Suchmusters (z.B. Fachworte) aufgrund ihrer Gesamthäufigkeit im Text potenziell mehr Bedeutung tragen als andere Teile des Suchmusters (z.B. Stoppworte), wird die Wichtigkeit eines Dokumentes als Produkt aus *term frequency* und *inverse document frequency* der Suchbegriffe ausgedrückt (vgl.[BR99], S.29).
- **Dokumentstruktur**: Unterschiedliche Bewertung der Trefferstellen aufgrund ihrer Position in der Dokumentstruktur: Z.B. Treffer in Überschriften oder in unmittelbarer Nähe von Abbildungen höher bewerten, als Treffer im Fließtext. Hierzu müssen jedoch zusätzlich zum reinen Dokumenttext noch Informationen über die Dokumentstruktur vorgehalten werden (vgl. Feldfilter im vorherigen Abschnitt).
- **Link-Popularität**: Zählt, wieviele Hyperlinks auf ein Dokument verweisen. Je mehr eingehende Verweise ein Dokument besitzt, desto wichtiger wird es eingestuft. Diese und die folgende Strategie funktioniert nur bei Textkorpora, in denen die einzelnen Dokumente umfassend mit Hyperlinks verbunden sind.
- **PageRank**: Ähnlich „Link-Popularität“, zählt aber nicht nur in einer Ebene die eingehenden Links zu einem Dokument, sondern berechnet rekursiv mehrere Ebenen rückwärts auch die Wichtigkeit der jeweils verweisenden Dokumente. Eingehende Links von ihrerseits bedeutenden Dokumenten werden höher eingestuft als eingehende Links von unwichtigen Dokumenten.

Die beiden letzten Strategien der obigen Liste sind beim Ranking von Internet-Dokumenten besonders hilfreich, da speziell in diesem Medium die Qualität der Seiten oft stark schwankt. Ein im WWW veröffentlichtes Schüler-Referat über `"Primfaktorzerlegung"` ist beispielsweise qualitativ der Veröffentlichung eines Hochschulprofessors zu demselben Suchmuster in einem renommierten wissenschaftlichen Journal wahrscheinlich unterlegen. Trotzdem stehen beide Dokumente gleichberechtigt im WWW nebeneinander und haben möglicherweise gleiche Länge und gleiche Anzahl Suchmuster-Treffer. Hier können Kenngrößen wie Link-Popularität und PageRank entscheidende Hinweise auf die relative Wichtigkeit eines Dokumentes geben.

In einem von gleichrangigen Wissenschaftlern bearbeiteten Werk wie zum Beispiel einer digitalen Enzyklopädie sind aber die einzelnen Artikel auch eher gleichrangig anzusehen. Und aus der Tatsache, dass ein Artikelautor hier mehr Links in sein Dokument einfügt als ein anderer sollte bezüglich des Volltextsuche-Rankings nicht zu schwer wiegen. In digitalen Enzyklopädien bieten sich daher also eher Ranking-Strategien wie *Dokumentstruktur* oder *TF/IDF* an.

Für das in dieser Arbeit vorgestellte exakte Volltextsuche-System wurde von André Schaupp im Rahmen seiner Diplomarbeit ein konfigurierbarer Ranker entwickelt, der verschiedene Bewertungsstrategien enthält [Scha03]. Eine angepasste Version seiner Ranking-Algorithmen wurde z.B. in der Online-Version von *Altmeyer* [AB02] integriert, und einige Abbildungen in Abschnitt 9.2 zeigen Bildschirmfotos der Online-Enzyklopädie.

6.3.5 Anzeige des Treffer-Kontextes

Ein weiteres Merkmal, das durch sein Vorhandensein in populären WWW-Suchmaschinen mittlerweile von den Benutzern als Standard bei Textsuchen vorausgesetzt wird, ist die Kontext-Anzeige. Dabei werden zu einigen oder allen Fundstellen des Suchmusters im Text jeweils einige Worte aus dem Umfeld des Treffers angezeigt. Durch diese Kontext-Anzeige kann der Benutzer häufig noch vor dem Aufruf des betreffenden Dokumentes entscheiden, ob das Dokument ihn wirklich im Rahmen seiner aktuellen Recherche interessiert (vgl. fett formatierte Hervorhebungen der Suchworte in den Kontexten von Abb. 6.1).

Abb. 6.1. Kontext-Anzeige bei einer Suche mit Google

Für die Anzeige von Kontexten benötigt man aber zusätzlich zur reinen Textinformation auch Teile des Layouts, da das Weglassen solcher Attribute wie „hochgestellt“ oder „Schriftart-Wechsel“ den Kontext ggfs. stark verfälschen können (H2O ist nicht dasselbe wie H_2O). Üblicherweise wird jedoch in den sowieso umfangreichen Volltextsuche-Indizes die Layout-Information verworfen und nur eine bereinigte Form des Textes abgelegt. Die Differenz zwischen der Position eines Zeichens im Volltextsuche-Index und der Position des entspre-

chenden Zeichens in den präsentierbaren Texten mit Layout nimmt aber zu, je höher der Offset des Zeichens gerechnet vom Dokumentanfang ist (vgl. Abb. 6.2). Daher ist ein Verfahren gesucht, mit dem möglichst schnell zu einer gegebenen Position im bereinigten Text (der Trefferstelle) die entsprechende Position im Text mit Layout (Zentrum eines Kontext-Fragmentes) gefunden werden kann.

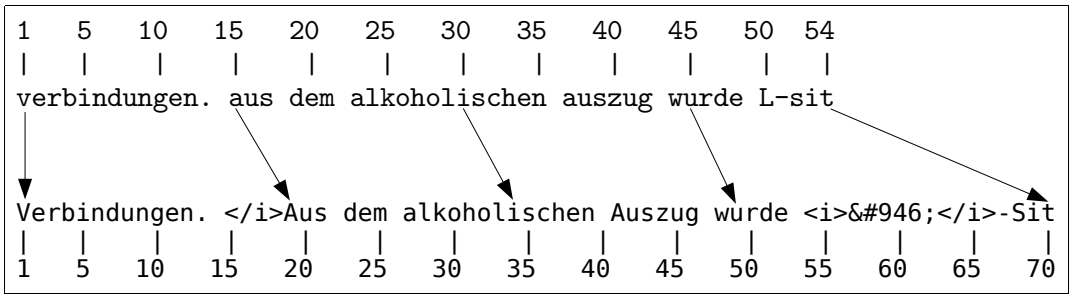


Abb. 6.2. Zunehmende Offset-Differenzen einzelner Zeichen zwischen bereinigtem Text (oben) und Text mit HTML-Layout (unten)

Ein naiver Ansatz zur Berechnung der Kontext-Position aus einer Trefferposition wäre, jeweils in der entsprechenden HTML-Datei ab Anfang die Zeichen zu zählen, wobei in spitzen Klammern stehende HTML-Tags überlesen und eventuell vorhandene Sonderzeichen (`&#nnn;`) jeweils nur als ein Zeichen gezählt werden. Da die Kontext-Generierung aber für viele Dokumente und pro Dokument für viele Treffer geschehen muss, wäre ein solcher Ansatz zu rechenaufwändig und würde die Gesamtlaufzeit der Volltextsuche merklich negativ beeinflussen.

Daher wurde ein Ansatz mit einem so genannten *Stützstellen-Index* entwickelt, der in einem Vorverarbeitungsschritt für jede Textdatei in einem Index die Zuordnung von jedem n-ten Zeichen im bereinigten Text zum Offset des entsprechenden Zeichens in der HTML-Datei speichert. Für n=15 würde der Index für das Beispiel aus Abb. 6.2 also der Reihe nach die Offsets [`19,34,49,...`] speichern. Für einen gegebenen Volltextsuche-Treffer kann dann durch einfache Berechnung sofort die Position ausgerechnet werden, an der im Index die Position der nächsten Stützstelle im HTML-Text liegt. Nun muss der oben erwähnte „naive“ Ansatz nur noch daraufhin angepasst werden, nicht mehr jeweils von Beginn der Datei an zu zählen, sondern von der ermittelten Stützstelle aus.

Ein weiterer Punkt, der bei der Konstruktion von Kontexten berücksichtigt werden muss, ist die Verschachtelung von HTML-Tags. Schneidet man ein beliebiges Stück HTML-Text aus einer HTML-Datei heraus, so kann dieses HTML-Fragment nur dann korrekt angezeigt werden, wenn bekannt ist, welche Format-Tags vor dem Fragment alle geöffnet waren. Außerdem sollten am Ende des Fragments alle noch offenen Fragmente geschlossen werden, um mehrere Kontexte hintereinander hängen zu können. Beispiel 6-5 illustriert das Problem und seine Lösung.

Beispiel 6-5. Tag-Hierarchie und Kontext-Bildung
Angenommen, aus dem folgenden HTML-Text soll um das Wort „fetter“ ein Kontext-Fragment mit insgesamt drei Worten herausgetrennt werden:

HTML: `Das <u>ist <b>ein <i>fetter, kursiver</i></b> Text.</u>`

Layout: Das ist **ein *fetter, kursiver*** Text.

Durch Verwendung des Stützstellen-Index wird der Kontext schnell gefunden:

HTML: `ein <i>fetter, kursiver`

Layout: ein *fetter, kursiver* **(falsches Layout)**

Um das korrekte Layout des Kontextes zu erhalten, muss die Tag-Hierarchie vor und nach dem Kontext entsprechend dem HTML-Text angepasst werden:

HTML: `<u><b>ein <i>fetter, kursiver</i></b></u>`

Layout: **ein *fetter, kursiver*** **(korrektes Layout)**

Ein naiver Ansatz wäre also, zur Laufzeit durch Parsen der kompletten HTML-Datei vor dem Kontext alle noch offenen HTML-Tags zu ermitteln und diese Tags dann vor das Kontext-Fragment zu schreiben, bevor der Kontext am Bildschirm ausgegeben wird. Dies würde jedoch die durch den Stützstellen-Index gesparte Zeit wieder sinnlos konsumieren, denn wieder müsste die gesamte HTML-Datei von vorne bis zum Treffer analysiert werden.

Daher wird parallel zum Stützstellen-Index noch ein *Tag-Stack-Index* abgelegt. In diesem befindet sich zu jeder Stützstelle die Information, welche Tags an dieser Stützstelle gerade alle geöffnet sind. Da die Öffnungsreihenfolge der Tags ebenfalls wichtig ist, bietet sich in der Analysephase eine Stack-Datenstruktur an. Zu jeder Stützstelle wird dann im Tag-Stack-Index der momentane Inhalt des Tag-Stacks abgespeichert. Nun muss bei Konstruktion eines Kontextes nur noch der an der Stützstelle gespeicherte Tag-Stack mit dem (kleinen) Stück von der Stützstelle bis zum wirklichen Beginn des Kontext-Fragmentes aktualisiert werden. Alle auf dem Stack liegenden HTML-Tags werden dann vor das Fragment geschrieben.

Da bei der Trefferausgabe evtl. mehrere Kontexte hintereinander ausgegeben werden, muss abschließend noch berücksichtigt werden, dass Folgekontexte in ihrer Tag-Struktur nicht gestört werden sollen. Daher muss jeder Kontext an seinem Ende alle von ihm geöffneten und noch nicht geschlossenen HTML-Tags wieder schließen. Am Ende eines jeden Kontextes muss der Tag-Stack also leer sein. Da die Kontext-Fragmente in der Regel eher kurz sind, ist hier eine lokale Analyse ausreichend schnell.

6.4 Oberflächen-Anbindung (JNI und Servlet)

In diesem Abschnitt wird die Anbindung der Volltextsuche an die Benutzeroberfläche beschrieben, wobei hier mit „Volltextsuche“ das gesamte bisher beschriebene System einschließlich fehlertolerantem Modul gemeint ist. Bisher wurde in Abschnitt 6.3.3 nur kurz darauf hingewiesen, dass die Volltextsuche per Kommandozeile bedienbar ist (Details zur Kommandozeilen-Version in Anhang D). Einem Endnutzer kann diese komplizierte Art von Schnittstelle natürlich nicht zugemutet werden – sie ist eher für Test- und Entwicklungszwecke gedacht. Abbildung 6.3 illustriert die verschiedenen Interaktionsmöglichkeiten, die mit der Kern-Suche realisiert wurden und deren Erläuterungen in diesem Abschnitt folgen.

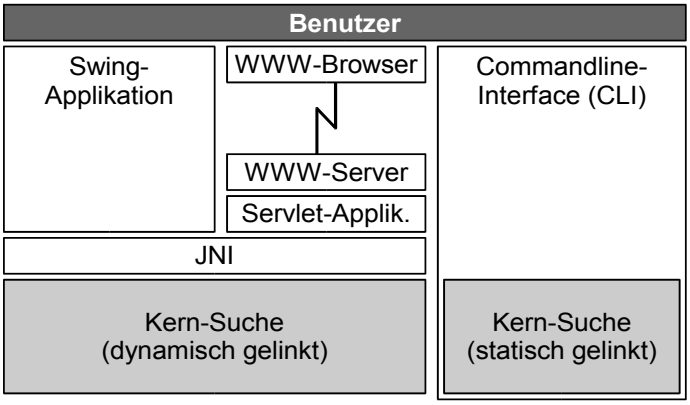


Abb. 6.3. Unterschiedliche Anbindungsarten der Kern-Suche an Benutzeroberfläche

Die in Kapitel 9 genauer vorgestellten Anwendungsfälle für das fehlertolerante Volltextsuche-System wurden entweder als CD-ROM/DVD-Projekte zum lokalen Einsatz (z.B. *HagerROM*) oder als Online-Version für das WWW konzipiert (z.B. *Altmeyer*). Als Entwicklungssprache für die Oberfläche bot sich im ersten Fall Java und in zweiten Fall HTML (mit CGI für die interaktiven Elemente) an. Die Volltextsuche ihrerseits wurde aus Performanz-Gründen jedoch komplett in der Sprache C++ entwickelt. Verschiedene Experimente, die q-Gramm-Suche als „100% pure Java“ zu implementieren, brachten jeweils eine nicht akzeptable Verlängerung der Laufzeit ca. um den Faktor 7 mit sich.

Daraus ergibt sich die Notwendigkeit, die nativ kompilierte Volltextsuche an eine heterogene Art von Oberflächen anzubinden: Lokale Benutzung und WWW-Benutzung. Für beide Szenarien steht jedoch im Java-Umfeld eine jeweils passende Technologie zur Verfügung: Die Anbindung einer Bibliothek an eine Java-(Swing)-Oberfläche geschieht mittels „*Java Nativecode Interface*“ (JNI) und die Server-Anbindung geschieht über JNI und „*Java Servlets*“. Die Volltextsuche muss dazu in eine native Bibliothek kompiliert werden: Unter

Windows eine Dynamic-Link-Library (`*.dll`) und unter GNU/Linux eine shared library (`*.so`). Da die Kern-Suche *plattformunabhängig* in C++ entwickelt wurde, lässt sie sich problemlos sowohl unter GNU/Linux für den Serverbetrieb innerhalb eines WWW-Werkes, als auch unter Windows für den Betrieb innerhalb eines lokal installierten Werkes übersetzen.

Bei JNI wird zunächst auf Java-Seite eine Java-Klasse implementiert, die als Schnittstellen-Klasse aus Sicht der anderen Java-Klassen agiert. In dieser Schnittstellen-Klasse werden dann Methoden mit dem Schlüsselwort `native` markiert, und ohne weitere Implementierung dieser Methoden wird die Klasse mit dem Java-Compiler in eine `class`-Datei übersetzt. Mit dem von Sun mitgelieferten Entwicklerwerkzeug `javah` wird nun die so entstandene `class`-Datei analysiert und eine passende C/C++-Header-Datei generiert, in der für jede der oben erwähnten native-Methoden nun eine C++ Methoden-Deklaration enthalten ist.

Der Entwickler muss nun lediglich diese „leeren“ C++-Methoden mit dem entsprechenden C++-Quellcode füllen und die native Bibliothek kompilieren. Innerhalb von C++ stehen dabei spezielle Datentypen und Objekte zur Verfügung, die einen Datenaustausch mit Javas Virtual-Machine ermöglichen, sodass Parameter von der Java-Seite an den nativen Code übergeben und Ergebnisse vom nativen Code zurück an die Virtual-Machine geliefert werden können. Es ist sogar möglich, innerhalb des nativen Codes Java-Exceptions zu generieren, die von normalen Java-Klassen abgefangen und behandelt werden können. Für eine detaillierte Beschreibung von JNI sei auf die Literatur verwiesen (z.B. [Ull04], Kap. 23).

Bei der Anbindung der fehlertoleranten Volltextsuche an das WWW wurde um die via JNI genutzte C++-Bibliothek herum ein Java-Servlet implementiert. Sun stellt mit `javax.servlet.HttpServlet` ein Interface zur Verfügung, welches über zu implementierenden Methoden (z.B. `init()`, `service()`, `destroy()`) die Java-Klasse direkt von einem Webserver aufrufbar macht, der eine so genannte Servlet-Container-Funktionalität bereit stellt. Diese Funktionalität kann z.B. im Webserver *Apache* [wwwApache] als Plugin-Modul, oder über den in Java programmierten Webserver und Servlet-Container *Apache Tomcat* [wwwTomcat] erfolgen. Das Servlet läuft also im Kontext des Webservers und wird beim ersten Zugriff via `init()`-Methode geladen. Alle an das Servlet gerichteten Anfragen werden vom Webserver über die Schnittstellen-Methode `service()` an das Servlet weitergereicht. Im Gegensatz zu extern angesiedelten CGI-Programmen hat dies aus Sicht der Performanz den Vorteil, dass nicht bei jeder Anfrage eine aufwändige Initialisierungsphase des Programms durchlaufen werden muss. Das Servlet generiert als Antwort auf die seiner `service()`-Methode übergebenen Parameter eine HTML-Seite, die es an den Webserver zurückliefert.

Der als Servlet-Container fungierende Webserver entscheidet aufgrund der angeforderten URL, ob er eine statische HTML-Seite an die WWW-Client zurückliefert (z.B. `http://www.mein-server.de/html/index.html`) oder ob er eine dynamische, von einer Servlet-Klasse generierte HTML-Seite liefert (z.B. `http://www.mein-server.de/servlet/servesidesearch.Suche`).
Details zur Servlet-Programmierung finden sich z.B. in [Ull04] (Kapitel 17).

Abbildung 6.3 (oben) illustriert die unterschiedlichen Anbindungsarten der Suche, wobei für die Bereiche dynamische und statische Bindung derselbe Quellcode verwendet wird und lediglich die Compiler- bzw. Linker-Optionen angepasst werden müssen.

Kapitel 7

Fehlerratenverbesserung auf redaktioneller Seite

Die bisherigen Kapitel der vorliegenden Arbeit beschäftigten sich damit, auf den als statisch angesehenen wissenschaftlichen Texten eine fehlertolerante Volltextsuche zu realisieren und diese mit etablierten Verfahren zu vergleichen. Es wurde auch darauf hingewiesen, dass das zu lösende Problem nicht nur auf der Seite des Benutzers liegt, der sich z.B. der korrekten Schreibweise seines Suchmusters nicht sicher ist. Vielmehr sind gerade bei umfangreichen Texten (wie Enzyklopädien) Tippfehler und inkonsistente Schreibweisen niemals ganz auszuschließen. Beide Fehlerarten können aber bei Recherchen in digitalen Texten das Auffinden relevanter Textstellen verhindern, wie bereits im Abschnitt „Motivation“ dieser Arbeit dargelegt wurde.

Im folgenden Kapitel werden deshalb Maßnahmen vorgestellt, mit denen bereits in der Produktionsphase eines umfangreichen digitalen Werkes Tippfehler und inkonsistente Schreibweisen erkannt und verbessert werden können. Neben einigen neu entwickelten Techniken wird dazu u.a. auch die bereits vorgestellte fehlertolerante Volltextsuche mit WPM-Modul zur Ermittlung von Korrekturvorschlägen benutzt.

Dazu wird u.a. auch die Entwicklung eines Programms beschrieben, welches Mengenoperationen (Vereinigung, Schnitt, etc.) und Filter (z.B. nach Anzahl oder Wortlänge) auf XML-formatierten Wortlisten zur Verfügung stellt. So bietet dieses Werkzeug die Möglichkeit, aus einem Grundstock von Wortlisten Wörterbücher für unterschiedliche Einsatzzwecke zu „destillieren“ und mit einem solchen Wörterbuch Korrekturvorschläge zu möglicherweise falsch geschriebenen Worten zu erzeugen.

7.1 Grundsätzliche Überlegungen

In Abbildung 2.2 (Seite 20) wurde bereits das Problem üblicher Rechtschreibkorrektur-Funktionen von Office-Textverarbeitungen gezeigt. Die Rechtschreibkorrektur arbeitet bei diesen Systemen in der Regel mit standardisierten Wörterbüchern, die für übliche Geschäftskorrespondenz meist befriedigend arbeiten, für wissenschaftliche Fachaufsätze aber nicht ausreichend sind. In solchen Texten ist der Anteil von Fachworten zu hoch, als dass hier zufriedenstellend gearbeitet werden kann: Zu oft muss das System eigentlich korrekte Worte als fehlerhaft markieren. In der Masse der falsch markierten aber korrekt geschriebenen Worte gehen die wenigen wirklichen Fehler unter, und nur eine aufwändige Trainingsphase könnte den kompletten Verzicht auf diese Funktion verhindern.

In den folgenden Abschnitten dieses Kapitels wird das Vorgehen beschrieben, die Erstauflage eines umfangreichen medizinischen Lexikons (*Springer Lexikon Medizin. Die DVD* [Reut04b]) von möglichst vielen Tippfehlern oder inkonsistenten Schreibweisen zu befreien und so dessen Qualität zu verbessern. Weil das Verfahren auf anderen Werken noch nicht eingesetzt wurde, ist die Untersuchung der Übertragbarkeit des Verfahrens auf andere Texte Bestandteil des Ausblicks dieser Arbeit.

Aus Sicherheitsgründen führt der hier vorgestellte Ansatz keine autonomen Korrekturen am Text durch, sondern konzentriert sich darauf, dem Autor des Werkes (oder einem Lektor) möglichst gut gefilterte Listen von potenziell falsch geschriebenen Worten mit ihren jeweiligen Korrekturvorschlägen zu präsentieren. Dabei zieht das System umfangreiche Texte aus der Domäne des zu korrigierenden Textes heran und extrahiert aus diesen den jeweiligen Wortschatz. Aufgrund von Kenngrößen wie Worthäufigkeit, Wortlänge und Auftreten in unterschiedlichen Werken werden dabei Annahmen über die vermutete Korrektheit der Worte gemacht. Zusammen mit numerischen Werten für die Ähnlichkeit zwischen Worten ergeben sich daraus die Korrekturvorschläge.

Der hier vorgestellte Ansatz wird daher nicht in der Lage sein, korrekt geschriebene Worte in falschem semantischen Kontext zu ermitteln („Das ist ein Beispiele“) – es soll vielmehr untersucht werden, inwiefern sich die zur fehlertoleranten Volltextsuche entwickelten Algorithmen auch zur redaktionellen Fehlerkorrektur eignen. Ein Verfahren, welches versucht, korrekte Worte in falscher Verwendung (sog. real-word errors) zu erkennen, findet sich z.B. in [HB03].

Es wird auch nicht versucht, Groß- und Kleinschreibung zu korrigieren, da die unternommenen Korrekturen weniger auf den ungestörten Lesefluss, als auf die Verbesserung der Qualität und Vollständigkeit von Volltextsuche-Ergebnissen abzielt. Da die zum Einsatz kommende Volltextsuche aber unabhängig von

Groß- und Kleinschreibung arbeitet, werden Tippfehler dieser Art ignoriert. Auch Zeichensetzungs- oder Grammatik-Fehler werden nicht untersucht, womit sich der Ansatz insgesamt von Verfahren abgrenzt, die z.B. mit Techniken aus den Forschungszweigen „Künstliche Intelligenz“ bzw. „Maschinelles Lernen“ versuchen, über ein vorgeschaltetes Textverständnis oder eine Kontext-Analyse diese Kategorie von Fehlern zu korrigieren (z.B. [GR99]).

7.2 Geschwindigkeitsvergleich Java/perl

Die Analyse und Verarbeitung großer Textmengen war lange Zeit vor allem der genau für diesen Zweck entwickelten Skriptsprache perl (*Practical Extraction and Report Language*, vgl. [wwwPerl], [WCS96]) vorbehalten. Verbreitete objektorientierte Hochsprachen wie Java [Krü04] oder C++ [Stro97] boten praktische Merkmale wie (fast) beliebig lange Zeichenketten, dynamisch wachsende Arrays und Hash-Maps oder ausgereifte reguläre Ausdrücke auf Unicode-Strings im Basisumfang der Sprache gar nicht oder nur teilweise an.

Seit der Java-Version 1.4, welche die Firma Sun im Jahr 2002 veröffentlichte, besitzt nun auch Java native Funktionalität für reguläre Ausdrücke (größtenteils syntaktisch kompatibel zu perl5) und erfüllt so die üblichen Ansprüche an eine Programmiersprache zur automatisierten Verarbeitung von Texten. Denn Unicode-Support, (fast) beliebig lange Strings, dynamische Container-Datenstrukturen usw. stellt Java schon seit Beginn Verfügung. In seinem Buch *Mastering Regular Expressions* [Frie02] schreibt Jeffrey Friedl im Kapitel über Java RegEx-Bibliotheken:

> „Sun's regex package, java.util.regex, comes standard with Java as of Version 1.4. It provides powerful and innovative functionality with an uncluttered (if somewhat simplistic) class interface [...] It has fairly good Unicode support, clear documentation, and good efficiency. [...] It is the all-around fastest of the engines listed here.“ [ebd].

Die Existenz ausgereifter integrierter Entwicklungsumgebungen (wie z.B. die quelloffene und kostenlose IDE Eclipse 3.0 [wwwEclipse]) und das deutlich durchgängigere objektorientierte Design der Sprache machen damit Java zu einer interessanten Alternative zur Implementierung vor allem komplexerer automatisierter Textverarbeitungsaufgaben.

Trotz der Tatsache eine interpretierte (und nicht kompilierte) Programmiersprache zu sein, steht perl im Ruf, zeiteffizient selbst bei der Verarbeitung großer (Text)-Datenmengen zu sein. Daher werden im Folgenden die Ergebnisse eines Geschwindigkeitsvergleiches zwischen Java und perl wiedergegeben. Bei

diesem Vergleich wurde der Schwerpunkt auf typische Aufgaben gelegt, wie sie bei Erstellung von Wortlisten aus größeren Textmengen auftreten: Suche tausender Dateien aus dem Dateisystem, Einlesen dieser Dateien, Löschung von HTML-Tags, Aufbrechen der verbleibenden Textzeilen in Worte und Analyse dieser Worte auf ihre jeweilige Häufigkeit.

Das Experiment wurde durchgeführt mit dem auf 60.748 HTML-Dateien (UTF-8 kodiert) verteilten *Roche Lexikon Medizin* [Roch03] mit einem Platzbedarf von ca. 73 MB (HTML-Version). An Hardware und Software kamen zum Einsatz: Ein PC mit Intel Pentium4-CPU (2,6 GHz), 512 MB RAM und dem Betriebssystem Microsoft Windows XP. Weiter wurden die Sprachen Activestate perl Version 5.8.0 (Build 806) und Java Version 1.4.2_06 verwendet.

Das Experiment wurde schrittweise in aufeinander aufbauenden Teilaufgaben durchgeführt, um ein klareres Verständnis dafür zu gewinnen, an welchen Stellen die beiden Programmiersprachen Vorteile oder Nachteile haben. Die dabei untersuchten Aufgaben waren:

1.) Sammeln der 60.748 HTML-Dateinamen aus dem Dateisystem
2.) Sammeln, Einlesen (und Verwerfen) aller Zeichen (ca. 73 MB)
3.) Sammeln, Einlesen aller Zeichen, Merken im Arbeitsspeicher
4.) Sammeln, Einlesen, Merken, dabei Löschen aller `<.*?>`-HTML-Tags
5.) Sammeln, Einlesen, Merken, Tag-Löschen, ′Split′ + ′Unique′ aller Worte

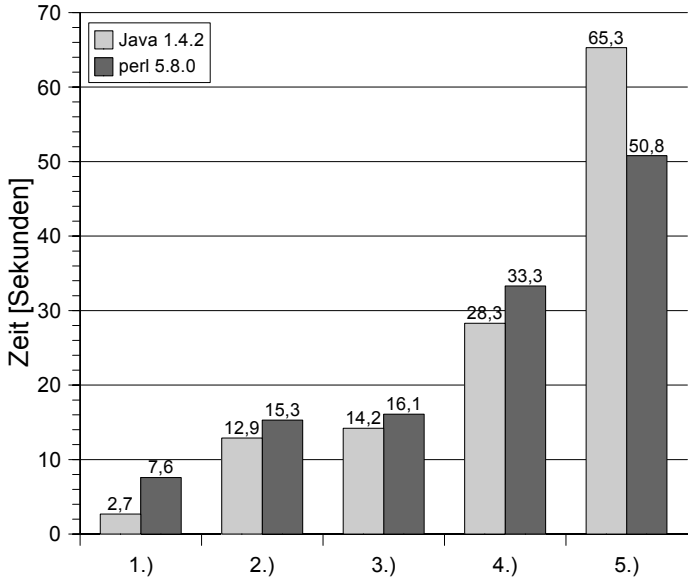


Abb. 7.1. Laufzeit-Vergleich zwischen Java und perl bei der Verarbeitung umfangreicherer HTML-Textmengen

Das Experiment zeigt, dass in der Regel Java die gestellten Aufgaben (marginal) schneller löst als perl. Erst bei der letzten Aufgabe schneidet Java schlechter ab. Hier werden nach der Tag-Bereinigung auch tatsächlich mittels einer Hash-Map die vorkommenden Wortklassen analysiert und dabei jeweils die Wortinstanzen gezählt. Es scheint also so, dass die Java-Hash-Maps nicht ganz so effektiv implementiert sind, wie die perl-Hash-Maps. Trotz der um fast 30% längeren Laufzeit bei Teilaufgabe 5.) wird Java als Programmiersprache für die in diesem Kapitel vorgestellten Ansätze gewählt. Die absolute Gesamtlaufzeit von gut einer Minute ist tolerierbar, und die bessere Wartbarkeit des Codes durch strengere OOP-Merkmale der Sprache sprechen für diese Wahl. Ebenso gilt dies für die einfache Möglichkeit, unter Java auch komplexe grafische Benutzeroberflächen zu erstellen, und es spricht die Existenz ausgereifter integrierter Java-Entwicklungsumgebungen für Java.

Das Experiment ergibt also, dass ein tolerierbarer, leicht erhöhter Zeitbedarf zugunsten besserer Sprachmerkmale akzeptiert wird.

7.3 Bildung von XML-Wortlisten

Die Basis für die zu untersuchenden Heuristiken ist eine aus verschiedenen Werken derselben Textdomäne gewonnene Wortliste (Ausnahme: Der Duden als Quelle für eine allgemeine deutsche Wortliste). Analysiert wurde der Textkorpus der in Tabelle 7.1 mit ihren Kenngrößen aufgetragenen Werke. Dabei wurde natürlich auch der Textkorpus des zu korrigierenden Werkes selber herangezogen. Denn ein Wort, welches in diesem Text nur „lang genug“ ist und „oft genug“ verwendet wird, kann auch hier als korrektes Wort angenommen werden und daher als Korrekturvorschlag für seltene und daher eventuell falsche Worte dienen.

Tab. 7.1. Verwendete Werke zum Aufbau des Wörterbuches mit Kenngrößen wie durchschnittliche Wortlänge bzgl. Wortinstanzen und Wortklassen

Werk	***Wortinstanzen***	***⌀ |W|***	***Wortklassen***	***⌀ |W|***
Reuter [Reut04b]	720.419	7,8	86.781	12,5
Pschyrembel [Psch02]	857.606	8,2	122.631	12,6
Roche [Roch03]	1.955.526	8,6	176.691	12,7
Duden [Dud96]	113.460	10,2	108.894	10,3

Die Einträge der aus diesen Werken erzeugten Gesamt-Wortliste (im Folgenden *Wörterbuch* genannt) werden mit Meta-Informationen angereichert, die in der

Korrekturphase z.B. Aufschluss darüber geben, wie oft das Wort in wieviel unterschiedlichen Werken verwendet wurde. Da das als statisch angesehene Wörterbuch auf unterschiedlichen Rechnern möglichst direkt aus dem Arbeitsspeicher heraus genutzt werden soll, wird von der Verwendung einer Datenbank abgesehen und als Speicherformat XML verwendet.

Die Worte werden nach dem Entwurfsmuster „Schablonenmethode“ (*template method* vgl. [GHJV96]) von Spezialisierungen einer abstrakten Klasse namens `Harvester` aus den Rohtexten der Werke extrahiert (Paket: `info2.wordharvester.*`). Dabei werden nur Worte mit mindestens drei Zeichen in die Liste aufgenommen (wobei natürlich verhältnismäßig wenige Worte wie z.B. `um` und `zu` verloren gehen). Es stehen zwar generalisierte `Harvester`-Subklassen für HTML und UTF-8-Rohtext zur Verfügung, allerdings müssen auch diese wiederum für jedes zu untersuchende Werk spezialisiert werden, da jedes Werk Eigenheiten bezüglich zu verarbeitender Dateinamen, Layout-Formatierung oder eingestreuter Kommentare besitzt.

Die Klasse `WordList`, welche eine Spezialisierung der Java-Standard-Klasse `HashMap` ist, nimmt die vom `Harvester` gefundenen Worte als Instanzen der Klasse `WordEntry` entgegen und aggregiert dabei die Häufigkeiten. `WordList` verfügt über die Möglichkeit, die gemerkten Worte (mit ihren Meta-Informationen) in eine XML-Datei konform zur XML-Dokument-Typ-Definition `wordlist.dtd` zu speichern. Die genaue Format-Beschreibung dieser DTD findet sich im Anhang E (ab Seite 221). Neben dem XML-Format besteht weiter die Möglichkeit, die Worte mit einer konfigurierbaren Auswahl von Meta-Informationen in Tab-separierte Textdateien im Format UTF-8 oder ISO-Latin-1 zu speichern. Aus XML-Wortlisten und Tab-separierten Textdateien können die Wortlisten zu einem späteren Zeitpunkt wieder geladen werden.

Für die Kodierung der XML-Wortlisten wurde das Format UTF-8 gewählt ([RFC3626], [wwwUTF8]). Dieses Format bildet Zeichen aus dem Unicode-Raum auf Zeichen unterschiedlicher Bytelänge ab. Die auf Seite 103 genannten Nachteile für die Speicherung von UTF-8-Zeichenketten in Volltextsuche-Indizes gelten bei der hier beschriebenen Speicherung in XML-Dateien nicht, da hier die Texte nur einmal beim Laden geparst werden müssen und dann im Arbeitsspeicher zur Verfügung stehen. Wahlfreier Zugriff ist hier nicht nötig und der Vorteil, beliebige Sonderzeichen aus den unterschiedlichsten Werken in einem einheitlichen, genormten Format abzulegen, überwiegt den Nachteil des geringfügig größeren Speicherbedarfes (verglichen mit 1-Byte Zeichensätzen).

Beispiel 7-1 zeigt einen Ausschnitt aus dem XML-Wörterbuch, welcher die gespeicherten globalen und wortlokalen Attribute zeigt. Für das Wörterbuch wurden die Wortschätze von *Roche*, *Pschyrembel*, *Duden* und *Reuter* zusammengefasst. Im selben Beispiel zeigt der Eintrag mit der laufenden Nummer 620 das

Wort „aberrierender“, welches in nur einem der vier Werke vorkam. Im Gegensatz dazu steht das darauf folgende Wort „abfall“, welches in allen vier Werken vorkam. Der letzte Worteintrag „abführende“ im Beispiel zeigt die zwei Byte (0xC3 0xBC=Ã¼), welche in der UTF-8-Kodierung für den deutschen Umlaut „ü“ stehen. An diesem Eintrag sieht man auch, wie aus der Häufigkeit des Wortes (freq="11") auf seine Qualität und damit auf seine Korrektheit geschlossen wird (qual="98"). In dem Attribut für die „angehobene Qualität“ (boostqual="100") wurde durch Berücksichtigung von Wortlänge, Worthäufigkeit und Anzahl der Referenzierungen in unterschiedlichen Werken der maximal mögliche Qualitätswert 100 gespeichert.

Beispiel 7-1. Ausschnitte aus dem XML-Wörterbuch (Latin-1-Ansicht)

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE wordlist SYSTEM "wordlist.dtd">
<wordlist count="343869"
          keyCaseIgnore="true"
          wordsToLower="true">
  <name>((Roche+Pschyrembel)+Duden)+Reuter</name>
  <nameshort>(((roch)+(pschy))+(dud))+(reut)</nameshort>
  <fieldlist count="8">
    <field>NUM</field>        <field>FREQ</field>
    <field>QUAL</field>       <field>BQUAL</field>
    <field>WORD</field>       <field>REFCOUNT</field>
    <field>REFNAMES</field>   <field>STEM</field>
  </fieldlist>
[...]
  <entry num="620" freq="1" qual="1" boostqual="1">
    <word>aberrierender</word>
    <stem>aberrier</stem>
    <references count="1">pschy</references>
  </entry>
[...]
  <entry num="655" freq="99" qual="100" boostqual="100">
    <word>abfall</word>
    <stem>abfall</stem>
    <references count="4">roch,pschy,dud,reut</references>
  </entry>
[...]
  <entry num="779" freq="11" qual="98" boostqual="100">
    <word>abfÃ¼hrende</word>
    <stem>abfuhr</stem>
    <references count="3">roch,pschy,reut</references>
  </entry>
[...]
</wordlist>
```

7.4 Operatoren für Wortlisten

Die für diese Arbeit entwickelte Java-Kommandozeilen-Applikation `MainWordListCmd` stellt auf gespeicherten Wortlisten Möglichkeiten zur Formatkonvertierung, Sortierung, Filterung und Anwendung von Mengenoperationen zur Verfügung. Dabei wird in folgenden Schritten vorgegangen:

1.) **Laden** der **1. Wortliste** (XML, UTF-8-Text, ISO-Latin-1-Text)
2.) optional: **Filterung** nach Wortattributen (Wortlänge, Anzahl, Qualität, regulärer Ausdruck als Filter, Referenz-Zähler, etc.)
3.) optional: **Laden** einer **2. Wortliste** und Anwenden eines **Operators** auf beide geladenen Wortlisten (add, intersect, subtract, spellcheck, ...)
4.) optional: **Sortieren** der Ergebnisliste (Wortlänge, Anzahl, Qualität, ...)
5.) **Speichern** der Ergebnisliste (XML, UTF-8, ISO-Latin-1, HTML-Tabelle)

Da hierzu umfangreiche, sich teilweise gegenseitig ausschließende Kommandozeilen-Optionen nötig sind, wurde ein externes, auf der LGPL-Lizenz beruhendes, quelloffenes Java-Paket zum Parsen von Kommandozeilen-Parametern benutzt [wwwJSAP]. Mit dieser Klassenbibliothek ist es einfach möglich, Kommandozeilen wie die aus Beispiel 7-2 zu parsen, die unterschiedlichen Fehlerfälle abzufangen und im Fehlerfall eine automatisch generierte Hilfestellung auszugeben.

Im Beispiel wird die Eingabe-Wortliste (`-i`) gefiltert nach Qualität (`-q`) und Häufigkeit (`-e`) und anschließend auf die 2. Wortliste (`--in2file`) addiert (`--operator ADD`). Danach wird eine alphanumerische Sortierung nach Worten vorgenommen (`-s`) und die gewählte Anzahl von Wortattributen (`-F`) in die neu erstellte Ausgabedatei (`-o`) mit gegebenem Namen (`-n`) gespeichert. Zusätzlich werden Statistiken in einer separaten Datei abgelegt (`--save-statistics`). Eine komplette Liste der möglichen Kommandozeilen-Schalter für das Programm `MainWordListCmd` kann Anhang F (Seite 223) entnommen werden.

Beispiel 7-2. Beispiel-Aufruf von `MainWordListCmd`

```
java MainWordListCmd
   -i "./00mydict/roche5_EntriesComplete.wlx"  -q 40  -e 4
   --operator ADD  --in2file "./00mydict/pschyrembel_Words.wlx"
   -s WORD -F NUM,FREQ,QUAL,BQUAL,WORD,REFCOUNT,REFNAMES,STEM
   -o "./00mydict/dict_(ro+py).wlx"  -n "Roche+Pschy"
   --save-statistics
```

Die durch das Programm zur Verfügung gestellten Operatoren auf Wortlisten werden im Folgenden genauer aufgeführt und ihre Auswirkungen auf die Wortattribute erläutert.

- **ADD**: Fügt die Worteinträge der 2. Wortliste der 1. Wortliste hinzu. Dabei werden bei den Worten der Ergebnisliste die Häufigkeiten addiert und die Referenz-Zähler und Qualitätswerte angepasst.
- **SUBTRACT**: Löscht die Wortklassen, die in der 2. Liste stehen in der 1. Liste komplett (es werden also nicht nur die Instanzzähler subtrahiert).
- **SUBTRACTSTEM**: Berechnet (falls nicht schon in den Eingabedateien enthalten) zu den Worten beider Listen die jeweiligen Wortstämme. Dabei kommt das in Abschnitt 7.5 beschriebene Stemming-Verfahren zum Einsatz. Anschließend wird eine Wortklassen-Subtraktion bezüglich der Wortstämme vorgenommen (vgl. Operator SUBTRACT).
  Der Operator SUBTRACTSTEM kann eingesetzt werden, um aus einer Liste zu korrigierender Worte diejenigen zu löschen, die denselben Wortstamm haben, wie die als richtig angenommenen Worte eines Wörterbuches. Im Gegensatz zum normalen Subtrahieren wird durch das Stemming z.B. auch das Wort „`lungenvenen`“ subtrahiert (als richtig erkannt), wenn im Wörterbuch nur das Wort „`lungenvene`“ enthalten ist. So bleiben in der Ergebnisliste potenziell falsch geschriebene Worte übrig.
- **INTERSECT**: Bildet die Schnittmenge der beiden Wortliste. Dabei werden (wie beim Operator ADD) bei den Wortklassen der Ergebnisliste die Häufigkeiten addiert und die Referenz-Zähler und Qualitätswerte angepasst.
- **SPELLCHECK**: Die 1. Wortliste wird als zu korrigierende Liste angesehen und die 2. Wortliste als Wörterbuch korrekten Worten. Bei diesem Operator wird mithilfe der Edit-Distanz zu seltenen Worten aus der 1. Liste ermittelt, welches häufige Wort (oder Worte) aus dem Wörterbuch diesem Wort am ähnlichsten ist. Die Ergebnisse werden in das `WordEntry`-Attribut `wordsCorrected` und `freqBestCorrected` gespeichert und von der im Abschnitt 7.7 vorgestellten grafischen Feedback-Benutzeroberfläche verwendet.
  Dabei müssen fragliche Worte, zu denen kein Korrekturvorschlag gefunden wurde, als richtig geschrieben angenommen werden. Dies geschieht im Wesentlichen, um den menschlichen Korrektor nicht mit zu vielen potenziell falsch geschriebenen Worten zu konfrontieren. Die Entscheidung für dieses Vorgehen basiert auf der Annahme, dass die wichtigen Fachworte in anderen Werken derselben Domäne (oder im gleichen Werk) auch häufiger korrekt vorkommen müssten.
- **SPELLCHECKMORPH**: Arbeitet analog zum vorher beschriebenen Operator, bis auf die Tatsache, dass nun das WPM-Verfahren aus Kapitel 5 verwendet wird, um zu einem potenziell falsch geschriebenen Wort passende Korrekturvorschläge aus dem Wörterbuch zu finden.

Das durch die Addition der vier Werke zustande gekommene Wörterbuch umfasst 343.869 Wortklassen (durchschnittliche Wortlänge: 12,5 Zeichen) mit insgesamt 3.647.011 Wortinstanzen (durchschnittliche Wortlänge: 8,4 Zeichen). So wurde im Durchschnitt jedes Wort gut zehnmal in den analysierten Werken verwendet. Die XML-Datei für das Gesamt-Wörterbuch benötigt knapp 60 MB Speicherplatz.

7.5 Wortstamm-Bildung

Bei der Korrektur von Worten mit Hilfe eines Wörterbuches ist zu berücksichtigen, dass ein simpler Vergleich von Zeichenketten oft nicht ausreicht, um ein Wort als falsch oder richtig zu klassifizieren. In den meisten indo-europäischen Sprachen, wozu auch das Deutsche gehört, werden bei der so genannten Flexion Nomina dekliniert (Kasus, Numerus, Genus), Verben konjugiert (Person, Zeit, ...) und Adjektive und Adverbien werden gesteigert. Dies geschieht meist durch die Verwendung von Affixen und hier vor allem durch äußere Flexion, z.B. durch Anhängen von Suffixen (ein `Junge`, mehrere `Jungen`), aber auch innere Flexion mit Veränderung des Stammes ist möglich (ein `Baum`, mehrere `Bäume`).

7.5.1 Porters Snowball-Stemming-System

Übliche Rechtschreibkorrektur-Programme, wie beispielsweise das quelloffene `ispell`[wwwIspell], besitzen daher für jede zu korrigierende Sprache ein Lexikon (z.B. Deutsch nach neuer Rechtschreibung [wwwIGer98]), welches zu den Grundformen der Worte jeweils Regeln zur korrekten Flexion beinhaltet. Aus bereits genannten Gründen versagen solche universellen Korrektur-Lexika jedoch bei der Analyse von Texten mit vielen wissenschaftlichen Fachworten oft. Daher soll also auf ein Flexionslexikon verzichtet werden. Stattdessen sollen die Worte aus umfangreichem Textmaterial zur Korrektur eines anderen Textes verwendet werden. Um nun aber zu erkennen, dass z.B. das einmalig verwendete Wort `muskelschwach` richtig ist, weil es im Wörterbuch das korrekte Wort `muskelschwächende` gibt, ist ein Mechanismus notwendig, der diesen Zusammenhang zwischen solchen Worten herstellt: das so genannte *Stemming*.

> „Als **Lexem** (oder: **Wortstamm**) wird in der Sprachwissenschaft die ungebeugte Grundform eines Wortes oder auch der Wortstamm bezeichnet.“ [Wikipedia.de, „Lexem“]

Ziel des Stemmings (Wortstamm-Bildung) im Information Retrieval ist jedoch weniger, zu einem Wort den sprachwissenschaftlich korrekten Wortstamm zu finden – dies ist ohne entsprechend aufbereitete Lexika algorithmisch wohl nicht möglich. Vielmehr besteht das Ziel darin, durch Anwendung geeigneter Regeln alle Worte, die linguistisch betrachtet denselben Stamm haben, auf einen gemeinsamen, eindeutigen Stamm-String abzubilden.

Algorithmische Stemmer, also solche, die nur mit Regeln und nicht mit einem zugeschalteten Lexikon arbeiten, haben dabei abhängig vom gegebenen Wort manchmal das Problem des „*Over-stemming*“ bzw. „*Under-stemming*“. Beim Over-stemming wird das Wort zu stark vereinfacht und so gerät das Wort in eine evtl. zu große Wortstamm-Klasse gemeinsam mit Worten, die aus linguistischer Sicht einen anderen Wortstamm hätten (z.B. `Fastenkur`→`fastenkur`, `Fastenkurs`→`fastenkur`). Beim Under-stemming passiert das Gegenteil: Das Regelsystem vereinfacht zu wenig und das Wort bleibt in einer (zu kleinen) Wortstamm-Klasse hängen und wird nicht mit Worten zusammen gebracht, die eigentlich denselben linguistischen Wortstamm hätten (`Abstraktion`→`abstraktion`, `abstrakt` →`abstrakt`).

Ein bekannter und leistungsfähiger, regelbasierter Stemmer für das Englische ist der so genannte *Porter-Stemmer* [Port80]. Dieser wurde von seinem Entwickler mittlerweile in Form von *Snowball* als Sprache und Interpreter für Stemming-Regelsysteme verallgemeinert [wwwSnowb]. Ähnlich wie das Unix-Programm `flex` durch textbasierte Regeldateien Parser-Quellcode erzeugt, so erzeugt Snowball aus Textdateien mit sprachabhängigen Stemming-Regeln Programme, die für Worte dieser Sprache Wortstämme berechnen können. Auch für die deutsche Sprache gibt es einen solchen Regelsatz, und Snowball kann daraus ein Stemmer-Programm in Ansi-C- oder Java-Quellcode generieren.

Aufgrund der Effizienz der generierten Programme geht auch mit der Java-Version das automatisierte Stemming langer Wortlisten sehr schnell: Zu den 108.894 Wortklassen des *Duden* werden mit dem deutschen Snowball-Java-Stemmer auf einem 2,6 GHz PC in unter einer Sekunde sämtliche Wortstämme berechnet.

7.5.2 StringBuffer und OutOfMemoryException

Dem Standard-Snowball-Paket liegt zu Demonstrationszwecken die Java-Applikation `TestApp` bei, die die berechneten Wortstämme sofort in eine Datei schreibt (und diese daher nicht im Arbeitsspeicher hält). Mit `TestApp` wurde auch das obige kleine Laufzeit-Experiment auf dem Duden durchgeführt. Um so erstaunlicher war, dass eine interne Verwendung des Java-Stemmers im bestehenden `WordListCmd`-Projekt reproduzierbar zu einer `OutOfMemoryException`

führte – selbst bei kleineren Wortlisten mit nur einigen tausend Worten. Solche Speicher-Fehler kommen bei der Software-Entwicklung zeitweise vor und sind daher normalerweise nicht berichtenswert. In diesem Fall jedoch wurde durch die anschließende Ursachen-Forschung ein seit mehreren Jahren existierender Fehler in den Java-Quellen des offiziellen Porter-Stemmers aufgedeckt. Dieser Fehler wiederum basierte auf einer Java-Besonderheit beim Umgang mit Zeichenketten, die insgesamt den Fall doch interessant macht – da theoretisch beliebige andere textverarbeitende Java-Programme hiervon betroffen sein können. Es folgt daher eine kurze Beschreibung des Problems mit seiner Lösung.

Möchte man in Java effiziente Zeichenketten-Operationen durchführen, ist man oftmals darauf angewiesen, nicht die Standard-Klasse `String`, sondern die wesentlich speichernäher arbeitende Klasse `StringBuffer` zu benutzen. Denn addiert man z.B. zwei normale `String`-Objekte zur Konkatenation, wird jeweils ein neues Objekt angelegt und die alten Objekte werden freigegeben und vom so genannten Garbage-Collector später gelöscht. Dies ist bei häufiger Verwendung sehr ineffizient. Ein `StringBuffer`-Objekt hat hier effizientere Möglichkeiten, Zeichenketten durch schnelles Kopieren der Inhalte anzuhängen oder den bestehenden Pufferinhalt zu modifizieren. Diese Modifikationen sind weniger aufwändig, da der `StringBuffer` dynamisch bei Bedarf wächst, bzw. beim Verkürzen des Strings seine Größe nicht ändert sondern nur die Variable für das String-Ende verändert (der Puffer behält seine Größe). Pufferlänge und Stringlänge müssen also bei einem `StringBuffer` nicht übereinstimmen.

Diese Effizienz ist auch der Grund, warum der Snowball-Java-Stemmer intern vielfach mit `StringBuffern` arbeitet. Speziell der Ergebnis-Wortstamm steht am Ende des Stemmers in einem solchen `StringBuffer`-Objekt und muss durch dieses an die aufrufende Methode zurückgegeben werden.

Um im Rest des umgebenden Programms die normalen (meist für `String`-Objekte gemachten) Ein/Ausgabe-Methoden zu benutzen, gibt es Standard-Methoden, `StringBuffer` in `Strings` umzuwandeln und umgekehrt. Und auch hier wurde von Sun die Effizienz in den Vordergrund gestellt: Bei der Umwandlung eines `StringBuffers sb` in einen `String s`, z.B. mit der Anweisung `String s = sb.toString()` wird nämlich der Puffer nicht kopiert, sondern `String` und `StringBuffer` teilen sich von da an durch einfache Verzeigerung einen gemeinsamen Puffer für die gespeicherten Zeichen. Dies geht vergleichsweise schnell und erst wenn der `Stringbuffer` verändert wird, bekommt der `String` per 1:1-Kopie seinen eigenen Speicher zugeteilt.

Und genau in dieser „*lazy-copy*“ genannten Strategie liegt die Gefahr für den Entwickler: Da in einem `StringBuffer` evtl. am Ende ungenutzter Speicher liegt, hat jetzt auch der String nach der 1:1-Kopie ungenutzten Speicher am Ende seines Puffers. Und wenn nun ein `StringBuffer` mehrfach wiederverwen-

det wird, so belegt er zu jedem Zeitpunkt Speicher für die bisher längste jemals in ihm gespeicherte Zeichenkette – und für alle aus ihm erzeugten String-Objekte gilt dies daher genauso.

Es gibt zwei wirksame Gegenmittel, massive Speicher-Verschwendung durch diese Besonderheit von Java-`StringBuffern` zu verhindern:

1.) `StringBuffer`-Objekte nicht mehrfach verwenden und stattdessen vor jeder Neu-Benutzung auch ein neues `StringBuffer`-Objekt anzulegen. Hierbei verhindert man zwar mehrfach akkumulierten leeren Speicher im `String`-Objekt, aber nicht die Tatsache, dass auch bei einmaliger Benutzung des `StringBuffers` der Puffer größer sein könnte als der darin liegende String (z.B. weil der Inhalt verkürzt wurde).

```
sb = new StringBuffer();
```

2.) Bei der Übergabe des Inhaltes an einen String nicht die lazy-copy-Methode `toString()` verwenden, sondern nur eine Kopie der wirklich benutzten Zeichen erzwingen.

```
s = new String(sb);
```

Die Snowball-Entwickler entschieden sich für die erste der beiden Lösungsmöglichkeiten in der Methode `getCurrent()` der Klasse `SnowballProgram` im Paket `net.sf.snowball`. Es wird geraten, vorhandene Java-Snowball-Stemmer auf den neuen, deutlich robusteren Quellcode (vom 7. Juni 2004) zu aktualisieren.

7.5.3 Experimente zum Stemming

Mit dem so verbesserten Quellcode des Snowball-Java-Stemmers war es nun erstmals möglich, alle Worte des *Reuter*-Werkes per Stemming mit dem Programm `WordListCmd` zu analysieren. Tabelle 7.2 zeigt einige Fälle von korrekter Gruppenbildung – aber auch einige Problemfälle, in denen das oben angesprochene over-stemming bzw. under-stemming aufgetreten ist. Zum Beispiel passt das Verb `äußern` weniger zu den anderen Worten des Stammes `auss`. Und den Worten `abblättern`, `abblätterung` und `abblätternd` konnte trotz gemeinsamer linguistischer Wurzel kein gemeinsamer Wortstamm zugeordnet werden.

Am Wortstamm `haufig` in der Tabelle sieht man weiter, dass ein algorithmisch generierter Wortstamm mit dem sprachwissenschaftlichen Wortstamm nicht unbedingt identisch sein muss, denn dieser müsste eigentlich `häufig` lauten. Trotzdem werden unter dem algorithmisch berechneten Wortstamm `haufig` korrekt Worte wie `häufiger` und `häufigkeiten` zusammengefasst. In den Klammern hinter den Worten befindet sich die Anzahl der Wortinstanzen zu der jeweiligen Wortklasse im *Reuter*-Werk.

Tab. 7.2. Worte mit ihrem berechneten Wortstamm aus [Reut04]

Stamm	#	*Worte aus [Reut04]*
stark	18	starker(71), starkes(37), stärkeren(8), stärker(34), stark (181), stärken(1), stärkste(4), stärkere(5), stärkstes(2), stärkerer(8), stärksten(10), starkem(32), starken(51), stärke(31), starke(64), stärkerem(4), stärkeres(1), stärkster(1)
haufig	14	häufiger(132), häufigkeiten(1), häufigsten(304), häufigkeit (53), häufigere(5), häufiges(15), häufig(368), häufige(76), häufigstes(7), häufigste(236), häufigster(28), häufigem(1), häufigeren(8), häufigen(15)
auss	12	äußern(3), äußerliche(2), äußerlich(102), äußerlichen(6), äußere(256), äußeren(303), außer(34), äußerem(10), äußerlicher(1), außen(297), äußeres(33), äußerer(58)
abblatt	1	abblättern(2)
abblatter	1	abblätterung(2)
abblatternd	1	abblätternd(1)

Im vorhergehenden Abschnitt wurden die Operatoren SUBTRACT und SUBTRACTSTEMMED auf Wortlisten vorgestellt. Es wird nun untersucht, inwiefern sich die Menge potenziell falscher Worte verringern lässt, wenn man nicht nur exakt im Wörterbuch vorkommende Worte im Text als richtig ansieht, sondern auch Worte, die denselben Stamm haben wie Wörterbuch-Worte. Aufgrund der Probleme der automatischen Stemmer kann es hierbei natürlich vorkommen, dass falsche Worte als richtig erkannt werden und umgekehrt. Aber es soll bei dieser Stamm-Subtraktion primär darum gehen, die manuelle Arbeit für den Autor bzw. Lektor durch starkes Ausdünnen der Fehler-Kandidaten zu vereinfachen. Daher wird diese Unschärfe bewusst in Kauf genommen.

Tab. 7.3. Ermittlung potenziell falscher Worte durch Subtraktion unter Berücksichtigung des Wortstammes: (*Reuter*) - (*Roche* + *Pschyrembel* + *Duden*)

	#Wortklassen	**# Wortinstanzen**
Reuter	86.781	720.419
Roche+Pschyrembel+Duden	318.606	2.926.592
SUBTRACT	25.170	36.347
SUBTRACTSTEMMED	19.268	26.819

Gegenüber der normalen Subtraktion konnten also gut 23% mehr Worte mit Wortstamm-Subtraktion aus dem *Reuter*-Wortschatz als „vermutlich korrekt“ entfernt werden. Es verbleiben 19.268 Wortklassen, die noch überprüft werden müssen, da sie weder exakt noch mit identischem Wortstamm im Wörterbuch vorkommen. Diese Zahl ist noch zu hoch, um sie vom Autor oder einem Lektor überprüfen zu lassen. Betrachtet man die Häufigkeit dieser übrig gebliebenen Worte, so kommen nur 1.864 Worte öfter als dreimal im *Reuter* vor (und könnten damit evtl. ebenfalls als „vermutlich korrekt“ angesehen werden). So kann die Häufigkeitszählung hier leider keinen nennenswerten Beitrag zur weiteren Reduktion der zu prüfenden Wortklassen liefern. Trotzdem werden (in Absprache mit dem Autor der Werkes) nur Worte mit Häufigkeit 1 oder 2 als möglicherweise falsch untersucht.

Für weitere Korrekturrunden soll die Menge der zu prüfenden Worte noch einmal verkleinert werden. Im folgenden Abschnitt wird daher ein Verfahren vorgestellt, welches dies ermöglicht: die so genannte *Dekomposition*.

7.6 Dekomposition

Anders als das Englische ist Deutsch eine Sprache, die vergleichsweise stark zur Bildung von *Komposita* neigt (z.B. `Kostendämpfungsgesetz`). Bei der Bildung eines solchen zusammengesetzten Wortes werden (üblicherweise) lexikalische Morpheme zu einem neuen Wort aneinander gereiht, wobei das letzte Teilwort die Hauptbedeutung, die Wortart und auch die Flexionsklasse bestimmt.

Komposita erschweren insofern eine automatisierte Rechtschreibkorrektur, dass es hier vorkommen kann, dass zwar alle Teilworte eines Kompositums im Wörterbuch enthalten sind, nicht aber deren Komposition – so wird das vielleicht richtige zusammengesetzte Wort eventuell trotzdem als „falsch“ klassifiziert.

Zusätzlich erschwert wird das Problem der Komposita durch den Umstand, dass zwischen den Teilworten an den „Nahtstellen“ oftmals so genannte *Fugenlaute* auftreten (`Klasse`+`Zimmer`=`Klassen`**`n`**`zimmer`) oder Buchstaben ausgelassen werden (`Schul`**`e`**+`Haus`=`Schulhaus`).

Weiter ist zu bedenken, dass es Worte geben kann, die man auf mehr als eine Art in gültige Teilworte zerlegen kann. In Beispiel 7-3 sind einige solche Problem-Komposita aufgeführt. Diesem Problem wird man jedoch nur begegnen können, indem der Textkorpus, aus dem die Worte kommen, daraufhin untersucht wird, ob die Fragmente auch einzeln oft genug vorkommen.

Beispiel 7-3. Worte mit mehreren gültigen Zerlegungen

Wort	*eher falsch*	*eher richtig*
brauereileiter	brauer|eileiter	brauerei|leiter
obstbaumesse	obstbaum|esse	obstbau|messe
silberuhr	silbe|ruhr	silber|uhr

Im folgenden Abschnitt wird also ein Algorithmus entwickelt, der mithilfe eines Wörterbuches versucht, ein gegebenes Wort, welches nicht im Wörterbuch zu finden ist, in gültige Teilworte zu zerlegen. Dabei werden die Teilworte jeweils aus dem Wörterbuch entnommen und Fugenlaute oder ausgelassene Laute an den Nahtstellen berücksichtigt. Wird eine gültige Zerlegung in Wörterbuch-Worte gefunden, so soll auch das Kompositum als „gültiges Wort“ angesehen werden.

7.6.1 Konzeption der Wortzerlegung

Zunächst werden einige Vorüberlegungen angestellt, die das Design des zu erstellenden Algorithmus beeinflussen werden.

- **Wortarten**: Kompositionen können nicht nur zwischen Substantiven vorkommen. Kompositionen sind auch zwischen folgenden Wortarten möglich (jeweils mit einem Beispiel):

Subst. + Subst.	Lebensversicherung, Nahtstelle
Adj. + Subst.	Querbalken, Hochhaus
Verb + Subst.	Waschmaschine, Trinkhalm
Subst. + Adj.	alkoholfrei, haushoch

  Da jedoch bei dieser Arbeit die Wörterbuch-Worte aus den Fließtexten der Werke gewonnen werden, ist eine Klassifizierung nach Wortart nicht bekannt. Der Algorithmus kann sich also nicht auf eine linguistische Klassifizierung der Teilworte abstützen und so (un-)gültige Kompositionen erkennen.

- **Affixe**: Oft werden im Deutschen Worte auch durch Voranstellen von Präfixen oder Anhängen von Suffixen in ihrer Semantik verändert. Diese Präfixe und Suffixe können meist universell eingesetzt werden. Zum Beispiel das Präfix „haupt-“: haupt|mann, haupt|tor, haupt|sache, haupt|grund, haupt|schlagader oder das Präfix „ur-“: ur|alt, ur|suppe, ur|ahn, ur|beginn. Aber auch Konkatenationen von Präfixen sind möglich: haupt|ur|sache.
  Mit den Suffixen verhält es sich ähnlich: Auch diese können durch Anhängen die Semantik des Wortes verändern, z.B. das Suffix „-bar“: wunderbar, schneidbar, bezwingbar oder „-sam“: einsam, betriebsam, mitteilsam.
  Daher muss der Algorithmus über eine möglichst umfangreiche Liste gültiger

Präfixe und Suffixe verfügen und beim Zerlegen der Worte diese bei Bedarf hinzuziehen, da die Präfixe und Suffixe vermutlich nur in Ausnahmefällen auch einzeln im Wörterbuch auftauchen werden. Dem Algorithmus wurden über 90 Präfixe und über 40 Suffixe bekannt gemacht.

- **Fugenlaute**: Auch eine Liste gültiger Fugenlaute muss dem Algorithmus bekannt sein (z.B. zur Dekomposition von `Lunge[n]flügel`, `Verband[s]kasten`, `Krankheit[s]erreger`, `Maus[e]falle`, `Ente[n]ei`). Für das Deutsche wurden ermittelt: `-e-`, `-en-`, `-ens-`, `-er-`, `-es-`, `-n-`, `-ns-` und `-s-`. Löschung an Nahtstellen kann vorkommen bezüglich der Zeichenketten: `-e-` (`Schulhaus`), `-en-` (`Gehhilfe`) und `-n-` (`Wanderdüne`). Trotz sorgfältiger Recherche bezüglich dieser beiden Listen kann hier wohl noch nicht mit Sicherheit von einer Vollständigkeit ausgegangen werden. Dazu ist Sprache eine zu lebendige und vielfältige Thematik. Die Standardfälle sollten jedoch bereits mit diesen kurzen Listen abgedeckt sein.
- **Verbotsworte**: Es gibt einige Worte, die im Deutschen üblicherweise nicht Bestandteil eines Kompositums sind. Trotzdem können Sie aber im Fließtext auftauchen und damit wären sie auch Bestandteil des Lexikons. Zu dieser Kategorie von Worten gehören die Artikel: `der`, `die`, `das`, `den`, `dem`, `des`. Der Algorithmus verfügt also über eine (erweiterbare) Liste von verbotenen Teilworten. Probleme könnten hier beim Wechsel der Textdomäne auftauchen: In nicht-medizinischem Kontext wird das Substantiv „`Die`“ z.B. bei der Chip-Produktion gebraucht. Hier könnten also Worte wie „`Diefertigung`“ oder „`Chipdie`“ gültige Komposita sein. Die Verbotsliste muss also je nach Anwendungszweck vorsichtig überprüft werden.
- **Bindestriche**: Wird in einem Wort ein Bindestrich gefunden, so wird ein Schnitt an dieser Stelle erzwungen, wie z.B. bei den Worten: `ästhetisch-plastische`, `nicht-toxinbildender`, `gasbrand-erreger`.
- **Backtracking**: Wo auch immer im Kompositum die Schnittstellen während des Programmlaufes gesetzt werden – es kann sich später herausstellen, dass bereits gesetzte Schnitte unvorteilhaft waren, weil dann bei den sich ergebenden Rest-Zeichenketten keine gültigen Schnittstellen mehr gefunden werden können. Daher wird der Algorithmus per *Backtracking*-Strategie implementiert, damit einmal gemachte Entscheidungen kontrolliert zurückgenommen werden können und stattdessen eine andere Art der Auftrennung untersucht werden kann.

Eingabe für den Algorithmus soll ein zu zerlegendes Wort und ein Fragment-Wörterbuch mit gültigen Worten sein. Optional kann der Algorithmus noch ein Wörterbuch für Worte aus der Textdomäne bekommen. Dabei müssen Fragment-Wörterbuch und Domänen-Wörterbuch nicht zwingend identisch sein. Der Algorithmus soll dann eine möglichst plausible Zerlegung des Wortes finden

(falls das Wort nicht identisch zu einem Wort aus dem Wörterbuch ist). Bei dieser Zerlegung sind drei grundsätzliche Strategien denkbar:

1.) **Non-Greedy**: Ab dem ersten Zeichen wird das Wort Zeichen für Zeichen analysiert und die erste Schnittstelle von vorne, die auf ihrer linken Seite ein gültiges Wort hat, wird akzeptiert und anschließend das restliche Wort zerlegt. Dabei entstehen im vorderen Bereich eher kurze Fragmente, was sich aber gegen Wortende als vorteilhaft erweisen kann.

2.) **Greedy**: Es wird (ab dem ersten Zeichen) versucht, möglichst lange Zeichenketten zu finden, die gültige Worte aus dem Wörterbuch sind. Hierbei entstehen tendenziell vorne lange Worte und hinten kurze Worte.

3.) **Multiple-Splitter**: Dieser Ansatz sucht zunächst alle möglichen gültigen Zerlegungen und versucht dann, über eine geeignete Bewertungsfunktion die plausibelste aller möglichen gültigen Zerlegungen zurückzuliefern. Dabei können zum Beispiel die folgenden Kenngrößen in die Bewertung einer Zerlegung einfließen:
    - Anzahl Fragmente insgesamt (egal welchen Typs)
    - Anzahl Wörterbuch-Fragmente (ohne Präfixe, Suffixe, Fugenlaute)
    - Durchschnittliche Länge der verwendeten Wörterbuch-Fragmente

Erst der Multiple-Splitter würde es erlauben, aufgrund der Bewertung der ermittelten „besten“ Zerlegung diese dann auch final zu akzeptieren oder diese trotzdem zu verwerfen. Die beiden anderen Ansätze müssten jedes Wort, zu dem irgendeine gültige Zerlegung gefunden wird, als korrektes Wort akzeptieren.

7.6.2 Dekomposition im Praxis-Test

In Absprache mit dem Autor wurden Begriffe aus dem *Reuter*-Textkorpus ab einer absoluten Häufigkeit von drei als „korrekt“ angesehen. Nach Anwendung des Operators SUBTRACTSTEMMED bezüglich des kombinierten Wörterbuches aus Roche, Pschyrembel und Duden mit dem `WordListCmd`-Programm blieben 17.040 fragliche Worte übrig. Als Wörterbuch wurde zur Dekomposition dieser fraglichen Worte nur der Duden verwendet. Der Duden sollte die üblichen Wortfragmente eines Kompositums in der benötigten Grundform enthalten. Tabelle 7.4 zeigt, wieviele der fraglichen 17.404 Worte mit welcher Methode als vermutlich gültiges Kompositum erkannt werden konnten. Außerdem ist die durchschnittliche Fragment-Länge in Zeichen angegeben. Es wird bereits deutlich, dass der Multiple-Splitter-Ansatz die anderen beiden Ansätze bezüglich durchschnittlicher Fragment-Länge dominiert.

Tab. 7.4. Erkannte Komposita mit durchschnittlicher Fragment-Länge

Methode	*# Komposita*	*∅ |Fragment|*
Non-Greedy	7.853	4,1
Greedy	7.901	5,3
Multiple-Splitter	7.819	5,6

Das Diagramm in Abbildung 7.2 zeigt, wie oft welche Fragment-Länge bei welcher Methode in den erkannten Komposita vorkam. Hier wird deutlich, warum der Multiple-Splitter die höchste durchschnittliche Fragment-Länge hat: Er verwendet die Fragmente mit Längen 1, 2, 3 und 4 deutlich seltener als die beiden anderen Ansätze, und im Ausgleich dazu verwendet er Fragmente mit mehr als als 5 Zeichen öfter.

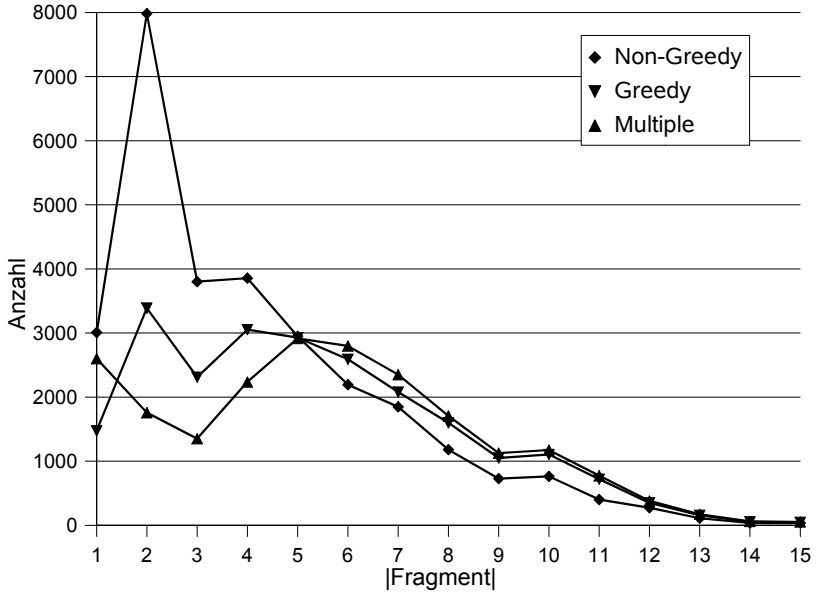


Abb. 7.2. Häufigkeiten erzeugter Fragment-Längen der drei Ansätze

In Abbildung 7.2 wurde die Abszisse ab einer Länge von 15 Zeichen abgeschnitten. Vereinzelt gab es jedoch noch Komposita mit längeren Fragmenten, z.B.

`|farbe|[n]|unterscheidungsvermögen|` (max. Fragm.-Länge=23)
`|laser|oberflächenbehandlung|` (max. Fragm.-Länge=21)
`|hirn|entwicklungsstörung|` (max. Fragm.-Länge=19)

Sowohl in Abbildung 7.2 als auch in Tabelle 7.4 zeigt sich der Greedy-Ansatz wiederum deutlich besser als der Non-Greedy-Ansatz. Dies beruht auf der Tatsache, dass der Non-Greedy-Ansatz immer die erstmögliche Trennstelle nimmt und dann weitersucht. Dies führt im Durchschnitt zu deutlich kürzeren Fragmenten.

Bei der Sortierung von unterschiedlichen Trennungsvarianten beim Multiple-Splitter wurde nach zahlreichen Experimenten das folgende Sortierkriterium als das leistungsfähigste – bezogen auf die semantische Korrektheit der Trennstellen – ermittelt. Die weitere Optimierung des Dekomponierers auf semantisch korrekte Trennung ist Bestandteil des Ausblicks.

Die verwendete Sortier-Vorschrift beim Multiple-Splitter war: Beim Vergleich zweier möglicher Zerlegungen gilt als bessere Zerlegung diejenige mit:

1.) geringerer Anzahl Fragmente, sonst: (also bei Gleichheit)
2.) durchschnittlich mehr Buchstaben in Wörterbuch-Fragmenten, sonst:
3.) höherer Anzahl Wörterbuch-Fragmente, sonst:
4.) gleiche Qualität wird angenommen

Tab. 7.5. Beispiele für qualitativ unterschiedlich gute Zerlegungen von fraglichen Worten

W	*Non-Greedy*	*Greedy*	*Multiple*
apophysenabriss	apophyse|[n]|abriss|	|apophyse|nabe|riss|	**|apophyse|[n]|abriss|** ~~|apophyse|nabe|riss|~~ |apophyse|na|brise|s |apophyse|[n]|ab|riss| ~~|apophyse|na|brie|ss|~~ |apophyse|na|brie|[s]s
arterienanomalie	|art|[e]|rien|anomalie|	|arterie|nano|mali|e	**|arterie|[n]|anomalie|** ~~|arterie|nano|mali|e~~ |art|[e]|rien|anomalie| ~~|arterie|na|no|mali|e~~ ~~|arterie|na|[n]|om|ali|e~~ ~~|arterie|[n]|an|om|ali|e~~ ~~|art|[e]|rien|an|om|ali|e~~
arteriensklerose	|art|[e]|rien|sklerose|	|arterie|[ns]|klee|rose|	**|arterie|[n]|sklerose|** ~~|arterie|[ns]|klee|rose|~~ |art|[e]|rien|sklerose| ~~|arterie|[n][s]|klee|rose|~~ ~~|art|[e]|rien|[s]|klee|rose|~~
geschlechtsausprägung	^ge^|schlecht|[s]| ausprägung|	|geschlecht|saus|prägung|	**|geschlecht|[s]|ausprägung|** |geschlecht|saus|prägung| ^ge^|schlecht|[s]|ausprägung| |geschlecht|sau|[s]|prägung| |geschlecht|[s]|aus|prägung| |ge|schlecht|[s]|ausprägung| ^ge^|schlecht|saus|prägung| |ge|schlecht|saus|prägung| |geschlecht|[s]|au|[s]|prägung|

Tabelle 7.5 zeigt bezüglich einiger ausgewählter Beispiele, welche Zerlegung die drei Ansätze vorschlagen. In der Spalte „Multiple“ sind die Vorschläge absteigend nach Qualität sortiert – es wird also jeweils die oberste Zerlegung zurückgeliefert. Die durchgestrichenen Vorschläge werden ausgeschlossen, weil nicht alle ihrer Wörterbuch-Fragmente auch als freistehende Worte im Domänen-Wörterbuch vorkommen. In der Tabelle werden normale Wörterbuch-Fragmente in ′|′ eingefasst, Präfixe in ′^′, Suffixe in ′$′ und Fugenlaute in [Klammern].

Das Fragment ′rien′, welches überraschenderweise bei der Zerlegung von arterienanomalie und arteriensklerose verwendet wird, steht im Duden als Teilwort des Eintrages „rien ne va plus“ (franz. „nichts geht mehr“).

Schon an den wenigen, exemplarischen Zerlegungen aus Tabelle 7.5 zeigt sich, dass der Mutiple-Ansatz oft die semantisch korrekte Zerlegung zurückliefert. Es wurde nun untersucht, wie oft dieser Ansatz besser ist als der nächstbessere, also der Greedy-Ansatz. Dazu wurde ermittelt, dass in 524 Fällen der Multiple-Splitter eine andere Zerlegung liefert, als der Greedy-Splitter. Diese Fälle wurden der Reihe nach manuell kontrolliert, und dabei wurde ermittelt, dass davon in 259 Fällen (49%) bessere bzw. optimale Splittings, in 12 Fällen (2%) schlechtere und in 253 Fällen (49%) nach-wie-vor schlechte Splittings enthalten waren. Tabelle 7.6 gibt exemplarisch einige Beispiele für die drei Klassen, wobei in der Spalte ′?′ eine Bewertung bezüglich der semantisch besseren oder schlechteren Zerlegung steht.

Tab. 7.6. Beispielhafte Bewertung der Qualität von unterschiedlichen Zerlegungen

Wort	***Greedy***	***Multiple***	***?***
ablösemittel	|ablöse|mittel|	^ab^|lösemittel|	+
autoreinfektion	|autor|[e]|infektion|	|auto|reinfektion|	+
eibefruchtung	|eibe|frucht|ung	|ei|befruchtung|	+
flüssigkeitsanteils	|flüssigkeit|sante|il|s	|flüssigkeit|san|$teils$	O
gallenblasenleiden	|gallenblase|[n]|leiden|	|gallen|blasenleiden|	—

Mit dem Multiple-Splitter-Ansatz kann die Menge der 17.040 fraglichen Worte aus dem *Reuter*-Textkorpus um fast 45% reduziert werden. Zusammenfassend wird hier nun die Reihenfolge und die Effizienz der angewendeten Filter noch einmal aufgeführt.

1.) Anzahl Wortklassen im Original-Textkorpus: 86.781 (100%)
2.) nach Wortstamm-Subtraktion: 19.268 (minus 78%)
3.) Betrachtung von Klassen mit weniger als 3 Instanzen: 17.404 (minus 10%)
4.) nach Test auf Dekomponierbarkeit: 9.585 (minus 45%)

Es soll nicht verschwiegen werden, dass beim Test auf Dekomponierbarkeit zwei mögliche Sorten von Fehlern auftreten: Es können Worte nicht dekomponiert werden, die trotzdem richtig geschrieben sind, und es können Worte erfolgreich dekomponiert werden, die eigentlich falsch geschrieben sind. Für beide Fälle werden hier einige Beispiele gegeben.

Beispiel 7-4. Falsch positive und falsch negative Bewertung einiger Worte bei Filterung durch Dekomposition mit dem Multiple-Splitter
Nicht zerlegbar (obwohl richtig geschrieben):
`körbchenförmige`, `hyperserotoninämie`, `translationssymmetrie`, `lipaseausscheidung`, `elektrohauttest`, `zellfragmentierung`
Erfolgreich zerlegbar (obwohl falsch geschrieben):
`abtranport (^ab^|tran|port|)`, `betandteil (^be^|tand|teil|)`, `beuteilung (|beut|eile|$ung$)`, `lebertransplantaton (|leber|transplantat|on|)`

Die hier vorgestellten Ergebnisse der Dekomposition im Praxis-Test halfen wesentlich bei der Entwicklung und Verbesserung der zugrunde liegenden Algorithmen. Wie die in Kapitel 7 vorgestellten Techniken bei der tatsächlichen Fehlerkorrektur des Werkes *Springer Lexikon Medizin* von Dr. Peter Reuter eingesetzt wurden und wie die Ergebnisse dort den Algorithmus zur Dekomposition weiter verbessern konnten wird im Abschnitt 9.4 des Kapitels „Anwendungsfälle“ detailliert beschrieben.

7.7 Feedback-GUI für das Lektorat

Die unterschiedlichen bisher vorgestellten Filter können also die Liste der fraglichen Worte jeweils immer weiter verkleinern. Trotzdem ist das Ergebnis eines jeden Filter-Schrittes immer wieder auch eine Wortliste. Am Schluss der Kette muss der Autor bzw. Lektor stehen, der die einzelnen fraglichen Worte in falsche und richtige Worte einteilt und so entsprechendes Feedback gibt, welche Worte wirklich zu korrigieren sind. Da jedoch diesem Personenkreis nicht zugemutet werden kann, direkt auf den XML-Wortlisten zu arbeiten, wurde eine grafische Oberfläche (Graphical User Interface, GUI) entwickelt, mit der diese Arbeit komfortabel und schnell durchgeführt werden kann. Dabei wurden die folgenden Design-Anforderungen berücksichtigt:

- **Plattformunabhängigkeit**: Speziell auf lange Sicht kann nicht vorhergesagt werden, mit welcher PC-Plattform ein Autor arbeitet: Auch Macintosh und Linux etablieren sich neben der zur Zeit wohl verbreitetsten Plattform Windows.

- **Installation**: Das Programm muss einfach zu installieren sein, und vom Entwickler bereit gestellte neuere Versionen müssen einfach (am besten automatisch) verteilt werden.
- **Benutzerfreundlichkeit**: Das Programm muss einfach und robust zu benutzen sein. Sowohl Benutzer, die primär mit der Maus arbeiten wollen, als auch Benutzer, die die reine Tastaturbedienung vorziehen, sollen unterstützt werden. Dabei sollen beide Bedienungsarten durchgängig unterstützt werden, sodass unnötige (und zeitraubende) Wechsel zwischen Tastatur und Maus verhindert werden.
- **Funktionalität**: Das Programm kann XML-Wortlisten konform zur `wordlist.dtd` (siehe Anhang E) einlesen und anzeigen. Der Benutzer kann über Checkboxen markieren, ob ein als fehlerhaft vermutetes Wort wirklich falsch ist.
- **Kontext**: Vielfach ist eine Beurteilung, ob ein Wort falsch geschrieben ist, ohne Kontext nicht möglich. In den XML-Wortlisten muss also zu jedem Wort mindestens ein Verwendungskontext gespeichert sein, den das Programm zusammen mit dem Wort und den eventuellen Korrekturvorschlägen anzeigt.

Auf Basis der obigen Anforderungen wurde die Java-Swing-Applikation *Correc d'Or* entwickelt, über die im Folgenden ein kurzer Überblick gegeben wird.

Da sich durch die Forderung nach Plattformunabhängigkeit Java als Entwicklungssprache aufdrängt, kann zur einfachen Distribution von Updates auf die *Java Web Start Technologie* [wwwWebstart] mit dem darunter liegenden *Java Network Launch Protocol* (JNLP) [Zuko02] zurückgegriffen werden. Bei dieser Technik werden zu einer (Java)-Applikation gehörende Komponenten (Klassen, jar-Archive, Startklasse, Icons, sonstige Ressourcen) in einer XML-Datei beschrieben. Diese XML-Datei und alle referenzierten Komponenten liegen auf einem WWW-Server. Zur Verdeutlichung ist im Folgenden die zur *Correc d'Or* Applikation gehörende Webstart-XML-Datei aufgeführt.

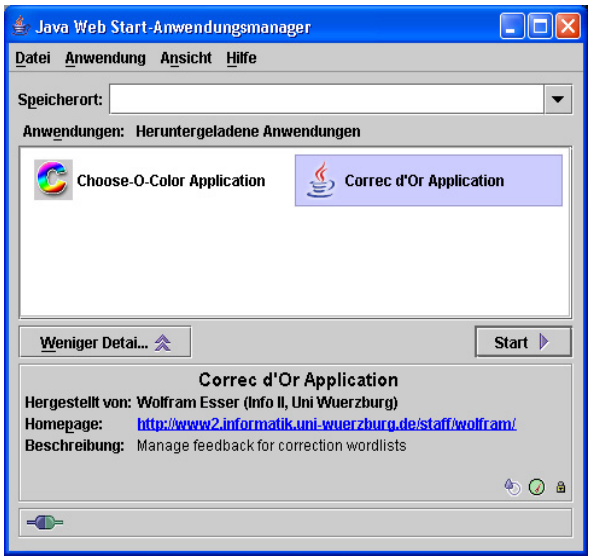


Abb. 7.3. Der Web Start Anwendungsmanager

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp
    spec="1.0+"
    codebase="http://www2.informatik.uni-wuerzburg.de/CorrecDOr/"
    href="CorrecDOr.jnlp">
  <application-desc main-class="MainCorrecDOr" />
  <information>
    <title>Correc d'Or Application</title>
    <vendor>Wolfram Esser (Info II, Uni Wuerzburg)</vendor>
    <homepage href="http://www2.informatik.uni-wuerzburg.de/" />
    <description>Correc d'Or Application</description>
    <description kind="short">Wordlist Feedback</description>
    <offline-allowed/>
  </information>
  <security>
    <all-permissions/>
  </security>
  <resources>
    <j2se version="1.4+"/>
    <jar href="CorrecDOr.jar"/>
    <jar href="forms-1.0.3.jar"/>
  </resources>
</jnlp>
```

Der Web Start Client ist Bestandteil der Java-Laufzeitumgebung (JRE), sodass jeder Benutzer einer aktuellen JRE diese Technik nutzen kann. Der Benutzer bekommt die WWW-Adresse der oben beschriebenen XML-Datei, welche von seinem Web Start Client heruntergeladen wird.

Das Herunterladen der benötigten Komponenten übernimmt der Web Start Client und trägt die Applikation bei „Heruntergeladene Anwendungen“ ein (vgl. Abb. 7.3). Bei jedem (Neu-)-Start der Applikation überprüft der Web Start Client (falls eine Internetverbindung besteht), ob auf dem Server evtl. neuere Komponenten zur Applikation vorhanden sind. Bei Bedarf werden vor dem Start die aktualisierten Komponenten nachgeladen. Dies bedeutet für Entwickler und Benutzer, dass die Verwaltung von Updates komfortabel und automatisch im Hintergrund stattfindet. Für den Entwickler bedeutet dies aber auch eine gesteigerte Verantwortung dahingehend, dass auf dem Server jeweils eine beim Benutzer auch wirklich lauffähige Version vorhanden sein muss.

Nach erfolgreicher Installation und durchgeführtem Start von *Correc d'Or* kann eine XML-Wortliste geladen und korrigiert werden (vgl. Abb. 7.4). Neben dem als falsch vermuteten Wort steht der erste im Werk gefundene Kontext dieses Wortes und darunter ein Korrekturvorschlag.

Das Programm kann komplett mit der Maus oder auch komplett mit der Tastatur bedient werden. Mit der Leertaste kann z.B. die „Correct“-Markierung zum

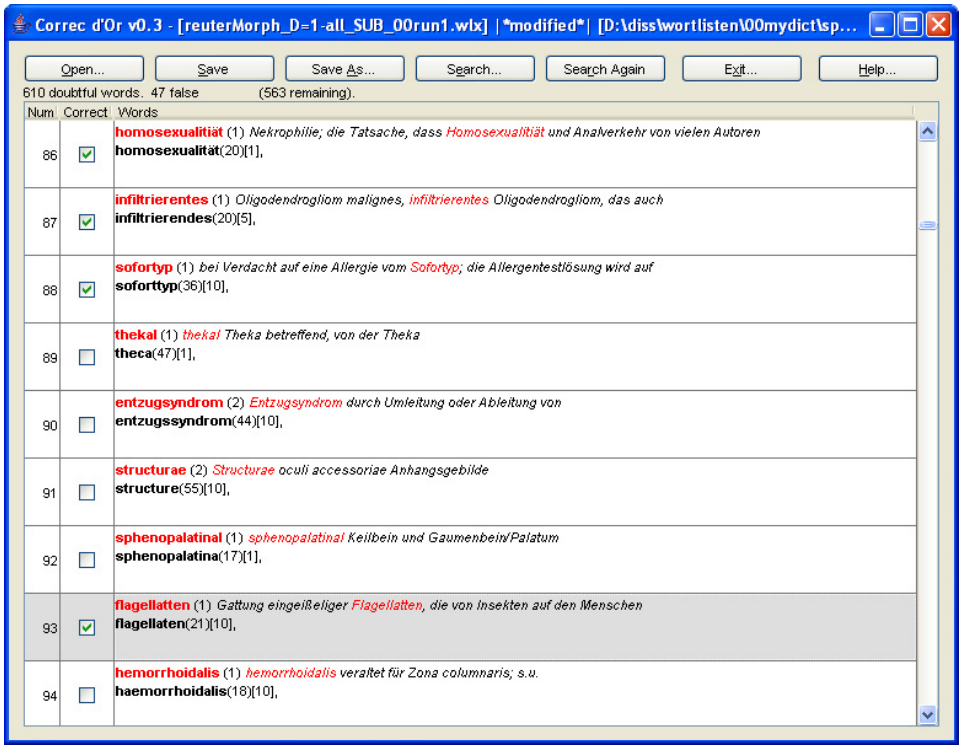


Abb. 7.4. Die Java-Applikation *Correc d'Or* mit markierten falschen Worten

aktuellen Einträg umgeschaltet werden. Mit dem Knopf „Search...“ kann nach einem bestimmten Wort in der Liste gesucht werden. Im momentanen Implementierungsstadium fehlt dem Programm jedoch noch die Funktionalität, beliebige Korrekturen an den gezeigten Worten durchzuführen, falls der präsentierte Korrekturvorschlag (wie in einigen Ausnahmefällen) nicht der gewünschten Korrektur entspricht. Da aber im konkreten Anwendungsfall *Springer Lexikon Medizin* die gefundenen Fehler nicht automatisch, sondern manuell im Textkorpus korrigiert wurden, wurde diese Funktionalität nicht benötigt.

7.8 Unscharfe Schlüssel-Vergleiche

Mit den in Kapitel 7 vorgestellten Techniken wurden mit dem Autor des Werkes *Springer Lexikon Medizin* zwei Korrekturrunden durchgeführt. Zunächst wurde eine Runde mit einfachem Stemming (simples Abschneiden von bis zu 3 Buchstaben am Ende) und der Edit-Distanz zum Auffinden von Korrekturvorschlägen durchgeführt.

Dabei stellte sich der Bedarf nach einigen Verbesserungen des Systems zur Ermittlung potenziell falsch geschriebener Worte heraus. Eine Veränderung lag in der Einführung eines verbesserten Stemmers, welcher bereits in Abschnitt 7.5

beschrieben wird. Eine weitere Verbesserung war die Einführung „*unscharfer Schlüssel-Vergleiche*“ (*fuzzy keys*), welche hier beschrieben wird.

Das zugrunde liegende Problem ist aus Beispiel 7-5 ersichtlich: Zu einigen Worten gibt es mehr als eine gültige Schreibweise. Während ein Werkautor die eine Schreibweise eines Wortes bevorzugt, verwendet der andere Autor (und damit z.B. das Wörterbuch) eventuell eine andere gültige Schreibweise. Wenn nun bei der Ermittlung potenziell falsch geschriebener Worte mit exakten String-Vergleichen gearbeitet wird, so werden speziell im medizinischen Umfeld oft Worte als falsch klassifiziert, die eine korrekte aber unübliche Schreibweise verwenden.

Beispiel 7-5. Unterschiedliche korrekte Schreibweisen von Fachworten

abduzensparese	abducensparese
antioxidans	antioxydans, anti-oxydans
balkannephritis	balkan-nephritis
blepharoconjunctivitis	blepharokonjunktivitis
photographie	fotografie
potentiell	potenziell

Wie bereits in Abschnitt 7.3 (S. 123ff) beschrieben benutzt die Klasse `WordList` zum Verwalten der Wortlisten die Funktionalität der Superklasse `HashMap`. Sämtliche Mengenoperationen zum schnellen Auffinden von Listeneinträgen stützen sich auf diese `HashMap`. Eine `HashMap` ordnet jeweils einem Schlüssel-String eine Instanz von `WordEntry` aus der Liste zu. Der Schlüssel eines Wortes berechnet sich dabei beim Einlesen (oder Neu-Anlegen) eindeutig aus diesem Wort. So könnte z.B. im Wörterbuch das Schlüssel-Wert-Paar: PHOTOGRAPHIE→`photographie` und im zu korrigierenden Werk das Paar FOTOGRAFIE→`fotografie` existieren. Bei exaktem Schlüsselvergleich würde das Wort `fotografie` daher als potenziell falsch klassifiziert werden.

Bei der Verwendung des (optional wählbaren) unscharfen Schlüsselvergleichs werden nun bei der Generierung der internen Schlüssel einige Vereinheitlichungen der Zeichenketten durchgeführt: `k`→`c`, `z`→`c`, `ph`→`f`, `-`→ε usw. Mit den so vereinheitlichten `HashMap`-Schlüsseln werden dann die Listenoperationen durchgeführt. Dadurch werden unterschiedliche Schreibweisen wie diejenigen aus Beispiel 7-5 ignoriert.

Nach diesen Verbesserungen wurde erneut mit dem Autor des Werkes *Springer Lexikon Medizin* eine Korrekturrunde durchgeführt. Neben den beschriebenen unscharfen Schlüssel-Vergleichen wurde das Snowball-Stemming und das WPM-Verfahren eingesetzt, um dem Autor unnötige Korrekturarbeit zu ersparen.

Bereits in der ersten Korrekturrunde auf Basis von einfachem Stemming und Verwendung der Edit-Distanz zur Ermittlung der Korrekturvorschläge konnten 311 fehlerhafte Wortklassen aus dem vorher bereits mehrfach korrekturgelesenen Werk eliminiert werden. In der zweiten Korrekturrunde, welche auf Basis von

Snowball-Stemming, unscharfen Schlüssel-Vergleichen und Nutzung des WPM-Verfahrens durchgeführt wurde, konnten insgesamt 315 (davon 91 neue) fehlerhafte Wortklassen gefunden und korrigiert werden. Die zur Entlastung des Autors durchgeführte, gleichzeitige Veränderung mehrerer Experiment-Parameter in der zweiten Korrekturrunde verhindert zwar einen fairen Vergleich der beiden Strategien, da die verschiedenen Einflussfaktoren nur schwierig separiert werden können – trotzdem werden die Ergebnisse der beiden Korrekturrunden im Abschnitt 9.4 des Kapitels „Anwendungsfälle“ detaillierter aufgeführt und einander gegenübergestellt.

Kapitel 8

Qualitätsverbesserung bei der Software-Entwicklung

Die Techniken, die in den vorangegangenen Kapiteln beschrieben wurden, konnten in zahlreichen (meist kommerziellen) Anwendungsfällen (vgl. Kapitel 9) ihre Praxistauglichkeit unter Beweis stellen. Für die konkrete Umsetzung der Anwendungsfälle mussten zahlreiche Softwarewerkzeuge entwickelt und angepasst werden. Wurde im vorherigen Kapitel beschrieben, wie die Fehlerraten (und damit die Qualität) der Werktexte verbessert werden können, so werden nun die Techniken beschrieben, die die Qualität auf Seite der Software-Entwickler verbessern bzw. sichern.

Das bei der Software-Entwicklung dieser Arbeit zur Anwendung gekommene Vorgehensmodell gleicht demjenigen des *Agile Software Development* (ASD) [wwwASD]. Von seiner Radikalität her noch unterhalb des Extreme Programming (XP) [wwwXP] angesiedelt stellt ASD doch bereits die regelmäßige Kommunikation mit dem Kunden und die häufige Konstruktion lauffähiger (evtl. noch prototypischer) Programmversionen in den Vordergrund. Diese lauffähigen Zwischenversionen sollen die Diskussion mit dem Kunden bezüglich zukünftiger Entwicklungsschwerpunkte unterstützen und so den durch „Eigenschaften gesteuerten“ (feature-driven) Entwicklungsprozess ermöglichen. Das Manifest, welches die Begründer des agilen Vorgehensmodells in einem gemeinsamen Workshop im Februar 2001 festlegten, lautet:

> „**Individuals and interactions** over processes and tools
> **Working software** over comprehensive documentation
> **Customer collaboration** over contract negotiation
> **Responding to change** over following a plan“
> (*Manifesto for Agile Software Development* [wwwASDMani])

Die 17 Autoren des ASD-Manifests, die ihre Gruppe „*Agile Alliance*“ nannten, waren führende Repräsentanten der verschiedenen existierenden agilen Vorgehensmodelle für Software-Entwicklungsprozesse. Unter anderem waren Vertreter des *Extreme Programming*, *SCRUM*, *DSDM*, *Adaptive Software Development*, *Crystal*, *Feature-Driven Development* und des *Pragmatic Programming* anwesend. Sie einte das Bedürfnis, ein gemeinsames Vorgehensmodell zu definieren, welches eine Alternative zum etablierten, so genannten „schwergewichtigen und dokumentenzentrierten Software-Entwicklungsprozess“ darstellt und die Gemeinsamkeiten ihrer unterschiedlichen Ansätze vereint [ebd.]. Die Bedeutung des agilen Vorgehensmodells für die Praxis wird auch durch die Tatsache unterstrichen, dass mittlerweile auch standardisierte Vorgehensmodelle (wie z.B. „*V-Modell XT*“) den agilen Prozess als mögliche Projektdurchführungsstrategie vorsehen (siehe [wwwVMXT], [Hag05]).

Das am 4. Februar 2005 offiziell von Bundesinnenminister Otto Schily vorgestellte „*V-Modell XT*“ ist der Nachfolger des für IT-Projekte des Bundes vorgeschriebenen Vorgehensmodells „*V-Modell 97*“ und bemerkt zur Agilität:

> „Die Agile Systementwicklung (AN) basiert auf der Erkenntnis, dass es oft nicht möglich ist, die Anforderungen an ein System vorab zu definieren. Außerdem stellt sie sicher, dass nichts spezifiziert wird, was sich als nicht realisierbar herausstellt.“ [VMXT04], Kapitel 3, S. 59

Von den in der Agile Alliance zusammengefassten agilen Vorgehensmodellen hat sicherlich das Extreme Programming in jüngerer Vergangenheit die größte Aufmerksamkeit erfahren. So wurden auch bei der Entwicklung der Software, die in dieser Arbeit beschrieben wird, Anregungen – vor allem die Verwendung von Unit-Tests – aus dem XP-Vorgehensmodell entliehen.

Um den Rahmen der vorliegenden Arbeit nicht zu sprengen wird bei den in diesem Kapitel beschriebenen Vorgehensweisen auf vertiefte theoretische Betrachtungen und umfangreichere Experimente verzichtet. Es sollen vielmehr die in der praktischen Umsetzung der Anwendungsfälle verwendeten Maßnahmen zur Software-Qualitätsverbesserung zusammengetragen werden, um eine Basis für ähnlich gelagerte Projekte und Startpunkt für weitere Forschungen zu bieten.

8.1 Konsistenzprüfungen

In allen in dieser Arbeit betrachteten Anwendungsfällen liegen die Eingabedaten der digitalen Werke in vielen unterschiedlichen Formaten vor. Die folgende Liste gibt – nach Dokumenttyp gegliedert – einen Überblick über diese Formatvielfalt.

- **Eingangstexte**: Extensible Markup Language (`XML`), Standard Generalized Markup Language (`SGML`), Microsoft Word (`DOC`), StarOffice/OpenOffice (`SXW`), Hypertext Markup Language (`HTML`), Portable Document Format (`PDF`)
- **Pixel-Grafiken**: Tagged Image File Format (`TIFF` als CMYK oder RGB), Portable Network Graphics (`PNG`), Joint Picture Experts Group (`JPEG`), Truevision Advanced Raster Graphics Array (`TGA`), Kodak Photo CD (`PCD`)
- **sonstige Abbildungen**: Encapsulated Postscript (`EPS`), Scalable Vector Graphics (`SVG`), Isis Draw (chem. Strukturformeln) (`SKC`), 3D Molecules (`MOL`)
- **Audio**: Microsoft Wave (`WAV`), MPEG-1 Audio Layer 3 (`MP3`)
- **Video**: Apple Quicktime (`MOV`), Motion Picture Experts Group (`MPEG-1`), Microsoft Audio Video Interlaced (`AVI`), Real Media Video (`RM`)

Nach zahlreichen voneinander abhängigen Konvertierungsschritten soll aus diesen heterogenen Eingangsdateiformaten ein homogenes Werk werden. Da die Werke in der Regel enzyklopädischen und daher umfangreichen Charakter haben sind bei der Entstehung jeweils zahlreiche Dateien der unterschiedlichen Formate involviert. Hier ist also eine möglichst vollständige Automatisierung der Konvertierungsschritte zwingend erforderlich. Da aber die Produktion der Eingangsdaten und die Konvertierung in die finale Präsentationsform kein sequenzieller sondern ein spiralförmiger Prozess mit zahlreichen Iterationsstufen ist, können sich in jeder Iteration auch Fehler einschleichen: Beispielsweise ist referenziertes Bildmaterial (noch) nicht vorhanden, verweisen Hyperlinks auf (noch) nicht existente Objekte, werden Sonderzeichen falsch kodiert oder fehlen Feldbezeichner in den Texten für die Volltextsuche.

Daher ist es notwendig, dass während der maschinellen Verarbeitung der Daten ständig Kenngrößen ermittelt werden, die über die Korrektheit und Konsistenz der Daten im Gesamtwerk Auskunft geben. Zu diesem Zweck werden in den Generierungsprozess zahlreiche Konsistenzprüfungen integriert.

Zum jetzigen Zeitpunkt müssen die von den Prüfskripten generierten Warnungen und Fehlermeldungen noch in regelmäßigen Abständen manuell überprüft werden. Eine automatisch generierte E-Mail im Fall schwerwiegender Fehler erfordert jedoch nur minimale Änderung an den existierenden Skripten. So gibt die folgende Liste einen Eindruck über die von den meistenteils in perl geschriebenen Prüfskripten und ihre überwachten Kenngrößen.

- `check_sonderzeichen.pl`: Ermittelt in den bereits nach HTML konvertierten Texten die verwendeten Sonderzeichen (unabhängig von der Art der Schreibweise (ö = `&ouml;` = `&#246;` = `&#x00F6;`) und prüft diese auf Bekanntheit in der `chartab` (vgl. Abschnitt 6.1 und Anhang C). Dieses Skript überprüft weiter, ob die Summe aller verwendeten Zeichen (nach Wandlung in

Kleinbuchstaben) nicht die Grenze von 255 Zeichen sprengt, welche das Maximum für das 8-bit Alphabet der Volltextsuche darstellt.

- `check_suchreihe.pl`: Die Generierung des Index für die Volltextsuche ist ein verhältnismäßig langwieriger Prozess, und schon früh im Gesamt-Generierungsprozess steht fest, welche Dateien in diesen Index aufgenommen werden sollen (und welche nicht). Die zu indizierenden (HTML)-Dateien stehen in einer Liste, und daher bietet sich eine frühe Überprüfung auf Existenz aller dieser Dateien im Dateisystem an. Fehlt auch nur eine dieser Dateien, kann prinzipiell der gesamte Generierungslauf abgebrochen werden: Die Daten sind nicht konsistent.
- `check_empty_fields.pl`: Ermittelt in den bereits nach HTML konvertierten Texten, ob Feldgrenzen-Kennungen existieren, die leer oder fehlerhaft formatiert sind. Da die Feldgrenzen in einem manuellen Prozess in die Ausgangsdateien eingepflegt werden müssen, besteht hier die Möglichkeit fehlerhafter Codierung und damit die Notwendigkeit, eine automatisierte Analyse der Korrektheit zu betreiben.
- `check_images.pl`: Da das interne Präsentationsformat der distributierten Texte HTML ist, werden Bilder über das `<img>`-Tag eingebunden. Das Skript überprüft, ob alle referenzierten Bilddateien wirklich vorhanden sind und löscht im Gegenzug Bilddateien, die nicht (mehr) verwendet werden, aus den für die Distribution erzeugten Verzeichnissen.
- `check_links.pl`: Ermittelt in den bereits nach HTML konvertierten Texten, auf welche Dateien `<A HREF>`-Hyperlink-Tags zeigen. Dabei werden vom `HREF`-Attribut auch eventuell mit „#“ angehängte Anker abgetrennt. Es wird anschließend untersucht, ob die referenzierten Dateien existieren und ob in diesen Dateien die benötigten Ankerziele vorhanden sind.

Diese automatisch überwachten Merkmale sollen nur Diskussionsgrundlage für ähnlich gelagerte Projekte bilden. Mehr Prüfskripte sind denkbar und wünschenswert, wenn ihre routinemäßige Ausführung den Generierungsprozess nicht unnötig verlangsamt.

Prototypisch wurde außerdem ein Skript entwickelt, welches Kenngrößen wie Anzahl verwendeter Text- und Bilddateien, Anzahl Hyperlinks in den Dokumenten, Speicherplatzbedarf aller Text- und Bilddateien, durchschnittliche Anzahl Bytes pro Datei, Anzahl Dateien für die Volltextsuche, Speicherplatzbedarf für den Volltextsuche-Index usw. über jeden Generierungslauf hinweg mitprotokolliert. Die Kenngrößen werden darauf in ein jeweils automatisch aktualisiertes Kurvendiagramm in einer Arbeitsmappe der Tabellenkalkulation Excel eingetragen. Mit dieser Grafik werden eventuell vorhandene signifikante Schwankungen in den Kenngrößen-Kurven schnell sichtbar. Solche Schwankungen müssten

dann entweder durch bekannte Änderungen an den Daten oder Skripten oder durch Fehler in Daten oder Skripten erklärbar sein.

8.2 Testbed für die Volltextsuche

Die fehlertolerante Volltextsuche ist eine der komplexeren Funktionen der in dieser Arbeit vorgestellten Anwendungsfälle. In der Vorverarbeitungsstufe werden u.a. HTML-Texte bereinigt, Sonderzeichen-Alphabete aufgebaut, der q-Gramm-Index erzeugt und komprimiert und Feldgrenzen gespeichert. Zur Laufzeit muss u.a. das Benutzer-Suchmuster analysiert werden, eventuell Varianten des Suchmusters durch das WPM-Verfahren generiert werden und die q-Gramm-Dateien nach Trefferstellen durchsucht werden.

Alle diese Prozess-Abschnitte sind ständigen Weiterentwicklungen und damit prinzipiell auch fehlerträchtigen Veränderungen unterworfen. Um nun zu bestimmten Zeitpunkten feststellen zu können, ob die Suchfunktion (noch) korrekt arbeitet, wurde das so genannte *Testbed* entwickelt.

Das Testbed ist ein externes perl-Programm, welches eine Eingabeliste von unterschiedlichen Suchmustern, dazugehörigen Suchoptionen und erwarteten Ergebnissen verwendet. Die erwarteten Ergebnisse sind in Form von „Anzahl Treffer gesamt“ und „Anzahl Einträge mit Treffern“ gespeichert. Konkrete Trefferpositionen (`FileID` und `Offset`) werden nicht vorgegeben, denn dies würde selbst bei kleinen Änderungen am Werktext jeweils größeren Wartungsaufwand für die Testbed-Testfälle bedeuten.

Das Skript wertet die Eingabeliste zeilenweise aus und startet zu jeder Zeile die (externe) Kommandozeilen-Volltextsuche mit dem entsprechenden Suchmuster und den dazugehörigen Suchoptionen. Die von der Suche ermittelten Trefferpositionen werden verworfen, doch die „Anzahl Treffer gesamt“ und die „Anzahl Einträge mit Treffern“ und bei fehlertoleranten Suchen die „Anzahl der Treffer-Morphs“ wird mit den erwarteten Werten aus der Eingabeliste verglichen. Ergeben sich hier Abweichungen, so hat sich eventuell ein Fehler in den Quelltext der Index-Generierung oder der Volltextsuche eingeschlichen oder es muss die Eingabeliste angepasst werden, da sich die zugrunde liegenden Texte geändert haben.

Weiter misst das Skript mittels des perl-Moduls `Perl::Time::HiRes` die Laufzeit der einzelnen Suchen, die Gesamtzeit für das Testbed und die durchschnittliche Dauer einer Suche. Es ist daher in begrenztem Umfang auch zu vergleichenden Zeitmessungen unterschiedlicher Implementierungsansätze geeignet.

Beispiel 8-1. Testfälle aus dem Testbed

Format einer Zeile: [Treffer,Einträge,Morphs]%% @@[Optionen]@@[Suchmuster]
Dabei bedeutet: '---' keine Treffer und '+++' zu viele Treffer.

```
     ---         %% @@@@dieser satz kein verb.
     +++         %% @@@@sch
    3015,3006, 1%% @@@@definition der droge
     224,  90, 6%% @@-t 2@@antioxidans
```

Beim Testen von Software wird üblicherweise zwischen so genanntem „*Black-Box-Testing*“ und „*White-Box-Testing*“ unterschieden.

Beim Black-Box-Testing hat der Entwickler der Testfälle üblicherweise keinerlei Kenntnis über Interna der zu testenden Software. Rein aufgrund einer bestimmten Eingabe wird eine vorgegebene Ausgabe des Programms erwartet. Der Test schlägt fehl, wenn eine andere (oder keine) Ausgabe vom Programm produziert wird. Der Vorteil dieser Testmethode ist, dass der Tester nicht mit dem Software-Entwickler identisch sein muss und daher unvoreingenommen Testfälle entwickeln kann. Nachteil dieser Methode ist aber auch, dass eventuell bestimmte Codeabschnitte durch die vorgegebenen Testfälle nicht überprüft werden, weil die benötigten Bedingungen zum Betreten dieser Codeabschnitte bei den gegebenen Testfällen niemals eintreten.

Beim White-Box-Testing werden zum Design der Testfälle Kenntnisse über die internen Abläufe einer Software angewendet. Bewusst wird versucht, die Software in Grenzbereichen oder mit anderweitig schwierigen Eingabedaten zu testen. Nicht bestandene White-Box-Tests können daher genauere Hinweise auf die interne Ursache eines Fehlers geben als es das Black-Box-Testing vermag.

Da das Suche-Testbed von den Entwicklern des Volltextsuche-Systems entwickelt wird und dabei Suchmuster und Suchoptionen geprüft werden, die jeweils bestimmte Schwierigkeiten in sich bergen, ist hier also von White-Box-Testing zu sprechen. Da hierbei jeweils das ganze Suchsystem getestet wird, handelt es sich um einen so genannten Systemtest, der im Gegensatz zu Modul- oder Unit-Tests steht, welche im nächsten Abschnitt angesprochen werden.

8.3 JUnit-Tests und Code-Coverage-Prüfungen

Bei der Implementierung der Java-Programme zur Fehlerratenverbesserung und hierbei speziell bei der Entwicklung des Programms `MainWordListCmd` mit seinen zugehörigen Klassen wurde Gebrauch von so genannten *Unit-Tests* mit dem Java-Test-Framework *JUnit* gemacht.

Beim Unit-Testing werden bei Verwendung einer objektorientierten Programmiersprache alle nicht-trivialen Methoden einer Klasse einzeln und unabhängig voneinander mit Testfällen geprüft. Dieses Vorgehen benötigt also Einblick in den Quellcode, und meist entwickelt der Programmierer selbst die passenden Unit-Tests zu seiner Klasse.

Als Technik zur Software-Validierung im Prinzip schon länger bekannt, hat in jüngerer Vergangenheit wohl vor allem die zunehmende Bekanntheit des Softwareprozess-Vorgehensmodells *Extreme Programming* [wwwXP] zu verstärkter Nutzung von Unit-Tests beigetragen. Auch die Existenz von Unit-Test-Frameworks für zahlreiche Programmiersprachen (vgl. [wwwXPUT]) von Java über C/C++ und perl bis hin zu C# und VisualBasic hat zunehmende Akzeptanz bei Software-Entwicklern bewirkt. Das für Java üblicherweise benutze Framework heißt *JUnit* und ist kostenlos im Internet erhältlich [wwwJUnit]. Über diese kleine aber sehr nützliche Klassensammlung schreibt Martin Fowler, der Autor von [FS02]:

> „Never in the field of software development was so much owed by so many to so few lines of code.“ [1] (siehe [wwwJUnit])

Zur Erzeugung von Testfällen mit JUnit leitet der Programmierer, um die Methoden einer Klasse zu testen, für diese Klasse einen Klassentest von der im Framework enthaltenen Klasse `TestCase` ab (vgl. Abb. 8.1).

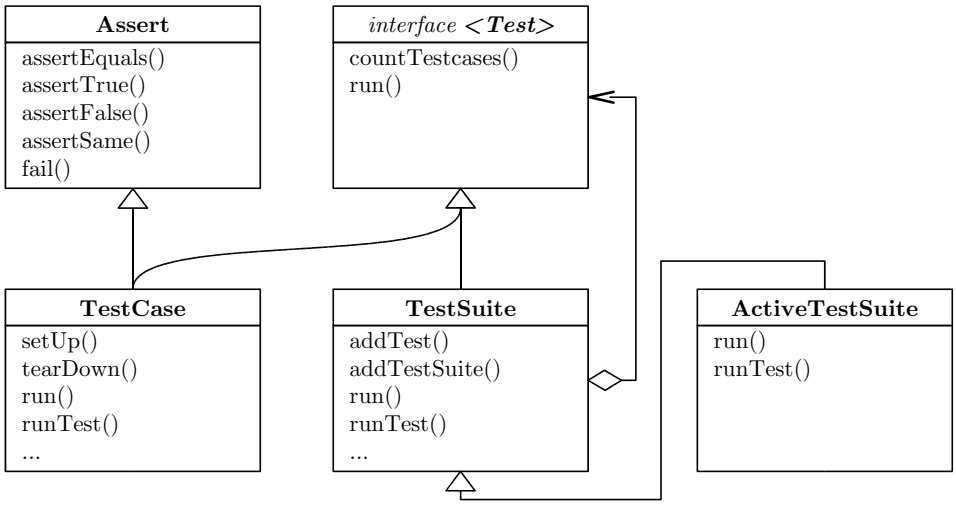


Abb. 8.1. Die wichtigsten Klassen und Schnittstellen des JUnit-Frameworks

1 Dieses Zitat scheint an den berühmten Ausspruch von Winston S. Churchill angelehnt, welchen dieser am 20. August 1940 in einer Rede vor dem britischen Unterhaus machte. In dieser Rede sagte er zur Rolle der Royal Air Force in der Luftschlacht um England: „*Never in the field of human conflict was so much owed by so many to so few.*“ (vgl. [Chur54], S. 410)

In einer `TestSuite` werden `TestCases` (und optional andere `TestSuites`) mit der Methode `addTest()` aggregiert und dann mit `run()` zur Ausführung gebracht. Diese Aggregation ist möglich, da sowohl `TestCase` als auch `TestSuite` das Interface `Test` implementieren. Innerhalb eines Testfalles werden per Java-Reflection-API alle Methoden ausgeführt, deren Methodenname mit „`test`“ beginnt. So können ohne viel Aufwand, während die zu testende Klasse weiterentwickelt wird, parallel die Tests angepasst und erweitert werden.

Innerhalb einer Test-Methode werden die zu überprüfenden Klassenmethoden mit Testwerten aufgerufen, deren Rückgabewerte oder Seiteneffekte bekannt sind. Anschließend wird überprüft, ob die gewünschten Resultate eingetreten sind. Dazu bieten sich die in der JUnit-Klasse `Assert` definierten Methoden `assertEquals()`, `assertTrue()` usw. an. Schlägt ein Test fehl, bricht das Framework den Testvorgang kontrolliert ab und weist mit einer entsprechenden Fehlermeldung auf den konkreten Testfall hin. Entweder muss der Testfall oder die überprüfte Methode korrigiert werden.

JUnit wird zusammen mit einer grafischen Oberfläche geliefert, die Aggregationen von `TestSuites` ausführen kann. Da mit zunehmender Anzahl von Testfällen der Testlauf gewisse Zeit in Anspruch nehmen kann, wird mit einem Fortschrittsbalken und einem Verlaufsprotokoll innerhalb eines `JTree`-Objektes ständig darüber informiert, welcher Subtest gerade läuft. Solange alle Tests be-

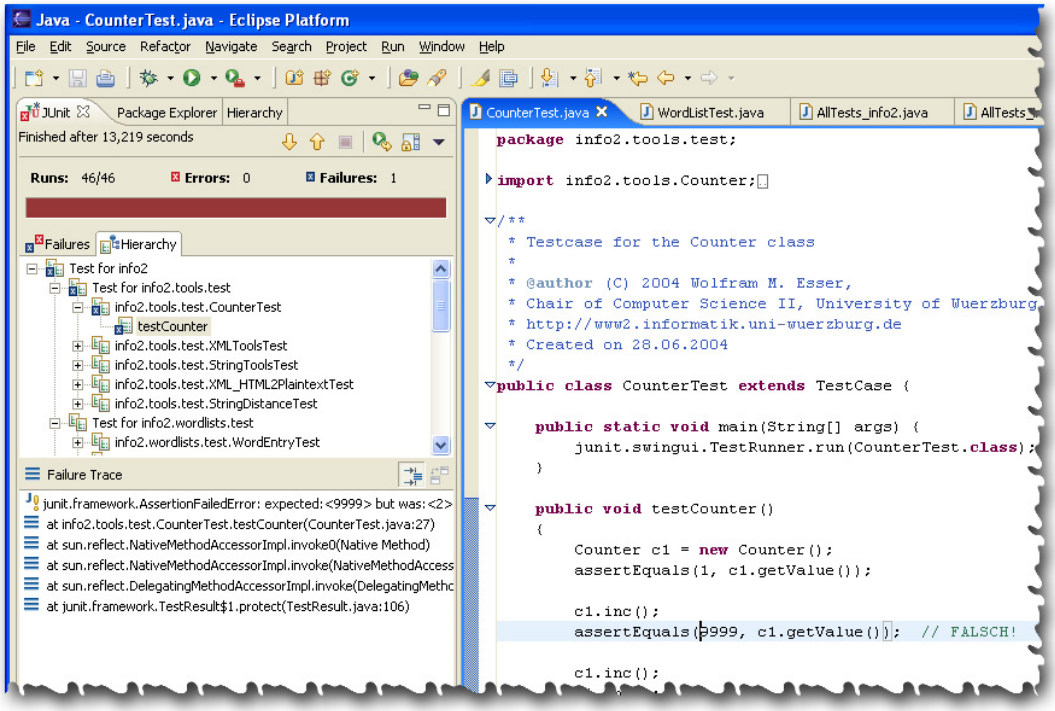


Abb. 8.2. Das JUnit-Plugin der Eclipse IDE mit einem nicht bestandenen Testfall

standen werden, ist der Fortschrittsbalken grün. Schlägt auch nur ein Testfall fehl, wird der Balken rot. Von diesem Umstand her rührt auch das Motto der zu JUnit gehörenden WWW-Seite: „Keep the bar green to keep the code clean“. Das JUnit-Framework wird von zahlreichen integrierten Java-Entwicklungsumgebungen direkt mitgeliefert oder ist als Plugin nachrüstbar (vgl. Abb. 8.2).

In Abbildung 8.2 ist zu erkennen, wie der Testfall zur Klasse `Counter` beim Überprüfen der Methoden `inc()` und `getValue()` einen Fehler produziert hat. Unten links in der Abbildung zeigt die so genannte *Failure Trace*, welches Testergebnis erwartet wurde <9999> und welches von der zu testenden Methode `getValue()` geliefert wurde <2>. Per Doppelklick gelangt man zur fraglichen Zeile im Testfall (rechts im Bild). Im Beispiel wurde hier absichtlich mit dem falschen erwarteten Ergebnis 9999 der Test zum Abbruch gebracht.

Innerhalb der Gruppe der „Extreme Programmer“ hat sich eine Strömung von Entwicklern gebildet, die so genanntes *Extreme Testing* betreibt. Hierbei wird das Schreiben von Unit-Tests soweit getrieben, dass *zuerst* die Unit-Tests implementiert werden (die zu diesem Zeitpunkt natürlich alle fehlschlagen und damit ′rot′ sind) und *dann erst* wird nach und nach die eigentliche Programm-Funktionalität dazu implementiert – mit dem Ziel, die Tests ′grün′ zu bekommen. Neben der Tatsache, dass man durch diese interessante Vorgehensweise immer überprüfbaren Code hat, wird auch bewirkt, dass nicht zu viel und nicht zu wenig Programm-Funktionalität implementiert wird: Ist der Test ′grün′, ist das Programm fertig – sofern die Tests die geforderte Funktionalität abdeckten.

Neben einer abschließenden Validierung der Software sind Unit-Tests (und mit ihnen JUnit) vor allem auch dafür gedacht, kleinere und größere Umbauten an der Software-Architektur (*Refactoring*) zu unterstützen bzw. überhaupt erst zu ermöglichen. In der Entwicklung eines Software-Projektes kann es Zeitpunkte geben, zu denen eine bereits gemachte Design-Entscheidung sich als suboptimal herausstellt. Diese Design-Entscheidung zu ändern, kann aber negative Auswirkungen auf die Funktionalität anderer Stellen im Code haben. Daher sind Entwickler – speziell in umfangreichen Projekten – an solchen Punkten tendenziell eher konservativ und behalten daher eventuell ein schlechteres aber dafür funktionierendes Design bei.

Sind aber alle (nicht-trivialen) Methoden der am Projekt beteiligten Klassen über Unit-Test in ihrer Funktionalität abgesichert, können selbst radikalere Änderungen am Design durchgeführt werden. Nach Anpassen der Testfälle der veränderten Klassen/Methoden kann sofort und automatisch überprüft werden, ob alle anderen Klassen unter den veränderten Rahmenbedingungen noch korrekt arbeiten können. Die Autoren Erich Gamma und Kent Beck schreiben in ihrem JUnit-Artikel [BG98] über Refactoring in Code, der mit JUnit überwacht wird:

> „You will be able to refactor much more aggressively once you have the tests. [..] You will be surprised at how much ground you can cover in a couple of hours if you aren't worrying every second about what you might be breaking." [ebd.]

Ein generelles Problem beim Entwickeln und Durchführen von Tests ist, dass man Kenntnis darüber benötigt, ob es noch ungetestete Codeabschnitte gibt. Die Testfälle sollten schließlich so konstruiert sein, dass jeder nicht-triviale Codeabschnitt mindestens von einem Test überdeckt wird. Dies kann natürlich beim White-Box-Testing (und auch bei Verwendung von Uni-Tests) durch Analyse des Quellcodes und der Bedingungen, die zu seiner Ausführung benötigt werden, geschehen. Aber schon bei etwas umfangreicheren Projekten ist dieses Vorgehen vergleichsweise aufwändig. Eine einfache Lösung bieten so genannte *Code-Coverage-Werkzeuge*.

Laufen die Tests unter ihrer Kontrolle ab, so protokollieren solche Werkzeuge, welche Codeblöcke zur Ausführung gekommen sind und welche nicht. Dabei kann diese Analyse sowohl für automatische Tests (z.B. Unit-Tests) als auch für manuelle Tests (z.B. während Tests mit Endnutzern) von großem Nutzen sein.

Der Entwickler Vladimir Roubtsov pflegt mit dem Projekt EMMA ein solches Code-Coverage-Tool für Java, und dieses ist mit Quellcode kostenlos über die Projekt-Homepage im WWW zu beziehen [wwwEMMA]. EMMA instrumentalisiert vor dem Start des zu testenden Programms den Bytecode der compilierten `class`-Dateien mit Messpunkten. Dies wird an den Grenzen von Codeblöcken durchgeführt, deren interne Anweisungen nicht Ziel von Sprüngen sind und aus denen nicht herausgesprungen wird.

Nach der Instrumentalisierung wird das zu testende Programm (oder die JUnit-Testsuite) unter der Kontrolle von EMMA gestartet. Wird während des Programmlaufes ein EMMA-Messpunkt passiert, so wird diese Tatsache protokolliert. Da `class`-Dateien üblicherweise Markierungen enthalten, welche Bytecode-Abschnitte von welchen Zeilen der dazugehörigen Java-Quellcode-Datei erzeugt

COVERAGE BREAKDOWN BY CLASS AND METHOD

name	class, %	method, %	block, %	line, %
class StopWatch	100% (1/1)	44% (4/9)	67% (136/204)	62% (26,1/42)
StopWatch (): void		0% (0/1)	0% (0/12)	0% (0/5)
getDeltaMillis (): long		0% (0/1)	0% (0/11)	0% (0/3)
getName (): String		0% (0/1)	0% (0/3)	0% (0/1)
main (String []): void		0% (0/1)	0% (0/9)	0% (0/3)
setName (String): void		0% (0/1)	0% (0/4)	0% (0/2)
getDeltaTime (): String		100% (1/1)	77% (96/125)	86% (18/21)
StopWatch (String): void		100% (1/1)	100% (15/15)	100% (6/6)
stop (): long		100% (1/1)	100% (9/9)	100% (2/2)

Abb. 8.3. Statistik zu einem Unit-Test mit dem Code-Coverage Werkzeug EMMA

wurden, kann EMMA von den passierten Messpunkten auf die abgedeckten Java-Quellcode-Zeilen zurückschließen. Bei Beendigung der Applikation (oder der Testsuite) generiert EMMA Reports in unterschiedlichen Formaten (Text, HTML, XML). In den HTML-Reports wird (falls vorhanden) der Java-Quellcode integriert, wobei die Zeilen farbig hinterlegt werden (**rot**=niemals ausgeführt, **grün**=mindestens einmal ausgeführt und **gelb**=teilweise ausgeführt). Ein Eindruck dieser farbigen Quellcode-Hinterlegung mittels einer Abbildung kann in dieser nicht-farbig gedruckten Ausarbeitung leider nicht erfolgen. In den ebenfalls von EMMA generierten statistischen Übersichten wird auf Paket- oder Klassenniveau die prozentuale Überdeckung bezüglich Klassen, Methoden, Bytecode-Blöcken und Java-Programmzeilen aufgeführt (siehe Abb. 8.3).

JUnit und EMMA ergänzen sich hervorragend bei der Erstellung und Durchführung von Unit-Tests für Java-Software-Projekte. Werden die JUnit-Testsuites unter der Kontrolle von EMMA durchgeführt, erhält man schnell einen Überblick, welche Klassen, Methoden oder Teile von Methoden noch von keinem Testfall abgedeckt werden. So war es beispielsweise möglich, in der für diese Arbeit erstellten Applikation `MainWordListCmd` in zwei Methoden kleinere Codefragmente zu entdecken, die unabhängig von den Eingabewerten der jeweiligen Methode niemals angesprungen werden konnten. Dieser „tote Code“ konnte also bedenkenlos entfernt und so die Lesbarkeit des Quellcodes erhöht werden.

8.4 Automatisierte Erstellung von 3D-Büchern

Bei der digitalen Aufbereitung der handschriftlichen Würzburger Bischofs-Chronik [Frie1582] von Magister Lorenz Fries (1489-1550) als Multimedia-DVD für Windows-PCs [Frie04] sind – verglichen mit den anderen Anwendungsfällen – einige neue Herausforderungen zu bewältigen. Neben der Darstellung der handschriftlichen Texte in moderner Fassung und der Anzeige von zoombaren Faksimile-Scans der Chronik-Seiten soll auch die Möglichkeit geboten werden, alle auf der DVD enthaltenen Seiten am PC virtuell in einem 3D-Buch durchblättern zu können. Von den insgesamt knapp 1200 Seiten der Chronik sind die 330 interessantesten auf der DVD enthalten.

Für das virtuelle Durchblättern muss also auf Seiten der Software-Entwickler ein Weg gefunden werden, aus den zweidimensionalen Seitenscans ein interaktives, virtuelles 3D-Buch zu erzeugen. Das gesuchte Verfahren soll fehlerresistent, reproduzierbar und homogen dem Endnutzer das Erlebnis des Durchblätterns aller enthaltenen Buchseiten vermitteln. Das hier vorgestellte Verfahren leistet dieses mittels Fehlerraten-Minimierung durch Automatisierung.

Die Möglichkeit, mittels 3D-beschleunigter Hardware ein virtuelles Buchmodell in Echtzeit darzustellen (z.B. mittels OpenGL oder DirectX) muss verworfen werden, weil sonst die minimalen Hardware-Anforderungen des Produktes zu hoch wären. Die Alternative, das Blättern in der Original-Handschrift einfach mit einer Videokamera abzufilmen und diese Filme geeignet zu präsentieren, schien ebenfalls nicht praktikabel: Zu ungleich wäre das Timing der Blättervorgänge, und für die Bewegung der Buchseiten benötigte Halteseile, Drähte oder sichtbare menschliche Hände würden das interaktive Erlebnis stören. Außerdem ist ein so altes und einzigartiges Buch natürlich mit größter Sorgfalt zu behandeln: Z.B. müssen Leser beim Umblättern Baumwollhandschuhe tragen, und die Dauer der Lichtexposition einzelner Seiten muss streng kontrolliert werden. Dieser Umstand verbietet jegliches Anbringen von Blätter-Apparaturen oder wiederholtes Abfilmen von Blättervorgängen. Es muss also ein Weg gefunden werden, aus den sowieso für die DVD angefertigten 2D-Scans das virtuelle 3D-Buch zu erstellen.

Im folgenden Abschnitt wird daher die Technik vorgestellt, mit der die über 330 Scans von Handschriften-Seiten automatisch in ein interaktives 3D-Buch kon-

Abb. 8.4. Aus vier 2D-Scans erzeugtes 3D-Buch der Fries-Chronik (Folio 101v ff, [Frie1582])

vertiert werden. Hierzu werden die zweidimensionalen Scans als flexible Texturen auf das Drahtgittermodell eines virtuellen 3D-Buches gelegt. Das Videomaterial für das 3D-Buch wird mit dem kostengünstigen aber mächtigen Animationsprogramm *Animation:Master 2004* (A:M 2004) der Firma Hash, Inc. [wwwHash] erstellt und jeweils nach Anpassen der vier benötigten Texturen der Blättervorgang einer Buchseite in einen Film gerendert. Die dabei entstehenden 170 Blätterfilme werden von einem Java-Programm angezeigt, welches es erlaubt, mit der Maus das Buch durch Anheben und Bewegen der Seiten virtuell durchzublättern. Die Software muss abhängig von den Mausbewegungen die Blätterfilme vorwärts oder rückwärts abspielen. Soll die nächste (vorherige) Seite geblättert werden, so muss das nächste (vorherige) Video entsprechend geladen und angezeigt werden. Das Vorgehen lässt sich leicht auf beliebige andere Werke übertragen, zu denen aus flachen Seitenscans ein virtuelles, interaktives 3D-Buch mit Blättermöglichkeit erstellt werden soll.

Bei dem gesamten Vorgang wird großer Wert auf möglichst weitgehende Automatisierbarkeit gelegt, um menschliche Fehler in der Serienfertigung der Blätterfilme auszuschließen. Folgende Schritte fassen das Vorgehen kurz zusammen – sie werden im weiteren Verlauf dieses Abschnitts genauer erklärt.

1.) Scannen der Buchseiten in hoher Auflösung (600 dpi)
2.) Zurechtschneiden aller Scans auf gleiche Größe und Position
3.) Retuschieren eventuell fehlender Seitenränder
4.) Konstruieren eines (statischen) 3D-Buchmodells mit A:M 2004
5.) Platzieren der vier Texturen für einen Blättervorgang im Modell
6.) Animieren des Blättervorganges einer Seite im 3D-Buchmodell
7.) Autom. Erzeugen der Blätterreihenfolge aus den Buchabschnitten
8.) Autom. Texturen-Austausch jeweils mit Rendern der Einzelbilder
9.) Autom. Skalieren und Komprimieren der Einzelbilder in Quicktime-Filme

Während die Schritte 1.) - 6.) einmalige manuelle Arbeiten erfordern, lassen sich die Schritte 7.) - 8.) voll automatisieren und sind damit robust gegen Fehler, wodurch ein gleichbleibendes Qualitätsniveau der entstehenden Animationen gewährleistet werden kann.

8.4.1 Vorverarbeitung der Scans

Die zwei Bände der *Fries-Chronik* bestehen aus 1188 handbeschriebenen Seiten im Großformat (42,5 cm x 29,5 cm) und werden im Rechenzentrum der Universität Würzburg in einem Raum mit überwachtem Klima eingescannt. Die Seiten werden für zukünftige Projekte zwar zunächst alle eingescannt – für die Multi-

media-DVD werden daraus aber nur ca. 330 der für heutige Leser interessantesten Seiten ausgewählt. Um die Bindung der Bücher während des Scans nicht zu gefährden und trotzdem perfekte Ergebnisse zu erzielen, wird eine elektrische Buchwippe OT 90 der Firma Zeutschel verwendet (vgl. Abb. 8.5).

Mit dieser Wippe ist es möglich, das Buch nur soweit aufzuklappen und zu fixieren, dass der Scan jeweils einer Seite möglich ist. Die Buchwippe drückt das Buch durch Fußschalter gesteuert und mit regelbarem Anpressdruck von unten gegen eine Scheibe aus Spezialglas. Von oben wird mit dem an einem Galgen befestigten Kigamo Scanback 6000XP die jeweils aktuelle Seite abgescannt. Dabei erfolgt die Ausleuchtung des Buches mit zwei Kaltlicht-Lampen, welche ein kontrollierbares, konstantes Licht auf die Seiten werfen.

Für jede Seite entsteht ein 24-bit Truecolor-Scan mit einer Auflösung von 600x600dpi und ca. 200MB Platzbedarf (260 GB für alle Scans). In dieser Auflösung sind auch kleinste Details der Handschrift (vgl. Abb. 8.6) und der zahlreichen farbigen Miniatur-Zeichnungen noch zu erkennen. Als Dateiformat wird das (verlustlose) TIFF-Format benutzt. Die Verluste, die das JPEG-Format – selbst bei höchster Qualitätseinstellung mit sich bringt, schienen angesichts der beabsichtigten Verwendung in mehreren Folgeprojekten und der damit verbundenen Langzeit-Archivierung nicht tolerabel. Die Gesamtmenge aller Scans belegt den Speicherplatz von über 50 single-sided, single-layer DVD-Medien.

Abb. 8.5. Zeutschel Buchwippe OT 90 (u.) und Galgen mit Kigamo Scanback 6000XP (o.)

Für die zu erstellende Multimedia-DVD werden die Scans jedoch zunächst auf 200 dpi heruntergerechnet, damit die Datenmengen auf heutigen Rechnern verarbeitet und angezeigt werden können. Für eine qualitativ hochwertige Anzeige an einem PC-Monitor ist diese Auflösung mehr als ausreichend, und die 600 dpi-Auflösung wird für spätere Faksimile-Drucke archiviert.

Beim Scannen muss das Buch jeweils manuell umgeblättert und neu unter der Glasscheibe positioniert werden. Dabei kommt es zwangsläufig zu minimalen Positionsverschiebungen und leichten Rotationen des Seiteninhaltes auf den da-

bei entstehenden Scans. Da der Scan einer einzelnen Seite durch die zahlreichen benötigten Handgriffe und bedingt durch die hohe Auflösung um die 6-8 Minuten benötigt, erstreckt sich der Scanvorgang des Gesamtwerkes über mehrere Monate. Durch diesen relativ langen Zeitraum kann es durch unterschiedliche Initialisierung der Scansoftware zu minimalen Schwankungen in der Pixelmasse der Scans kommen. Die Scans müssen also zunächst manuell mit einer Bildbearbeitungssoftware alle auf eine einheitliche Breite und Höhe und auf eine einheitliche Positionierung der Seiten innerhalb der Scans gebracht werden.

Durch die kleinen Drehungen des Buches unter der Glasscheibe der Buchwippe kann es außerdem vorkommen, dass auf einigen Scans ein schmaler Keil vom schwarzen Hintergrund oder vom sichtbaren Papierstapel der restlichen Seiten zu sehen ist (vgl. Abb. 8.6). Diese Artefakte müssen vor der Generierung der 3D-Filme manuell beseitigt werden. Dazu wird mit digitaler Fotoretusche-Software wie z.B. Adobe Photoshop der Papier-Hintergrund in jedem dieser Problem-Scans bis zu den Rändern ausgedehnt.

Abb. 8.6. Detailvergrößerung eines 600dpi-Scans mit Papierstapel (fol. 30v, [Frie1582])

Die bei diesem Prozess entstehenden Scans haben alle eine Auflösung von 200 dpi, sind in Breite und Höhe pixelgenau identisch und haben keinerlei störende Ränder außerhalb des gescannten Papiers. Sie können nun als austauschbare Texturen für das zu erstellende 3D-Modell dienen.

8.4.2 Design und Animation des 3D-Modells

Das bewegliche 3D-Modell des Buches wurde, wie eingangs erwähnt, mit der Software Animation:Master 2004 (A:M) erstellt. A:M ist ein vollständiges Animationspaket bestehend aus *Modeller* (zum Erstellen von Drahtgittermodellen), *Rigger* (Steuerung der Drahtgitterpunkte durch „Knochen“ und „Muskeln“), *Animator* (wiederverwendbare, addierbare, zeitlich interpolierte Bewegungsabläufe), *Choreographer* (setzt Licht, Kamera, Objekte, Bewegungspfade zu Szenen zusammen) und *Renderer* (Generierung der Einzelbilder einer Animation).

Interessant für Software-Entwickler ist, dass das Format, welches A:M zur Speicherung seiner Objekt- und Projektdateien benutzt, eine menschenlesbare Textdatei ist. So ist es möglich, skriptgesteuert Veränderungen an diesen Dateien vorzunehmen, welche von A:M dann entsprechend in veränderten Render-Ergebnissen umgesetzt werden.

Zunächst wird in A:M das Drahtgittermodell eines flachen Buches entworfen (vgl. Abb. 8.7, links). Das Buch darf noch nicht seine endgültige (natürliche) Krümmung erhalten, da sonst die Projektion der flachen Seitenscans auf die gekrümmten Oberflächen zu unnatürlichen Verzerrungen führen würde. Erst nachdem alle Texturen auf das (flache) Buch gelegt worden sind, darf das Buch seine natürliche Krümmung erhalten, wobei die Texturen – wie eine aufgezogene Gummihaut – ebenfalls diese Krümmung nachvollziehen (vgl. Abb. 8.7, rechts). Damit die Krümmung steuerbar und nicht-destruktiv ist, wird sie separat von der Gitterstruktur als so genannte *An-Aus-Pose* in A:M gespeichert. Posen lassen sich regeln, ein- und ausschalten, überlagern und jederzeit getrennt vom Modell verändern. In der Figuren-Animation werden Posen z.B. für verschiedene wiederverwendbare Gesichtsausdrücke einer Figur gespeichert (rechter Mundwinkel oben, linke Augenbraue nach unten usw.).

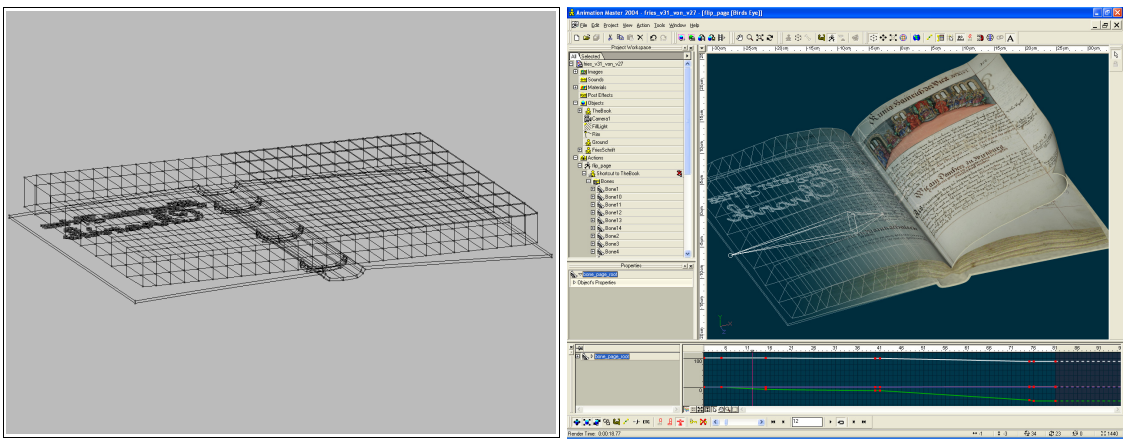


Abb. 8.7. Das flache Drahtgittermodell des 3D-Buches (links) wird erst nach Aufbringen der Seitenscan-Texturen naturalistisch gekrümmt (rechts).

Das Buchmodell besteht aus einem linken und einem rechten Papierstapel und einer beweglichen (sehr dünnen!) Seite dazwischen. Die Seite liegt zunächst flach auf dem rechten Papierstapel und kann dann nach links umgeblättert werden. Das Modell muss also nachher vier Seitenscans als Texturen aufnehmen, um diese eine Seite zu blättern: 1.) linker Papierstapel, 2.) bewegliche Seite vorne, 3.) bewegliche Seite hinten und 4.) rechter Papierstapel. Um nun mit diesem einen Modell mit einer beweglichen Seite ein ganzes Buch durchblättern zu können, werden einfach aus der Liste aller Seitenscans immer vier aufeinander folgende Scans auf die Textur-Positionen 1.) bis 4.) platziert und ein Film generiert, der die Scans auf Textur-Position 2.) + 3.) vom rechten Papierstapel auf den linken Stapel blättert. Für den nächsten Film müssen die Texturen von Position 3.) + 4.) um zwei Positionen nach vorne auf Position 1.) + 2.) gerückt werden. Die frei werdenden Positionen 3.) + 4.) werden mit den nächsten Bildern aus der Liste aller Seitenscans belegt und der nächste Blätterfilm gerendert.

Die bewegliche Seite des Buchmodells bekommt ein Skelett, das es erlaubt, die Seite flexibel umzublättern, wobei der so genannte *Wurzel-Knochen* dafür sorgt, dass die Seite fest im Buchbund verankert bleibt. Vom Buchbund beginnend wird eine Abhängigkeitshierarchie von 14 Knochen gebildet, die jeweils eine Spalte von Gitterpunkten kontrollieren. Da A:M *inverse Kinematik* beherrscht, kann man bereits jetzt innerhalb von A:M die Seite über den äußersten Knochen am Seitenrand anfassen und in Echtzeit umblättern, wobei sich Translation und Rotation von den äußeren zu den inneren Knochen propagieren.

Um eine flexible Seitenecke zu erhalten, an der per Maus der virtuelle Blättervorgang eingeleitet werden kann, wird wiederum eine Pose verwendet. Diesmal jedoch keine „An-Aus-Pose“, sondern eine „Prozent-Pose“. Bei einer Prozent-Pose kann eine beliebige Zahl von Gitterpunkten und Knochen über einen Pro-

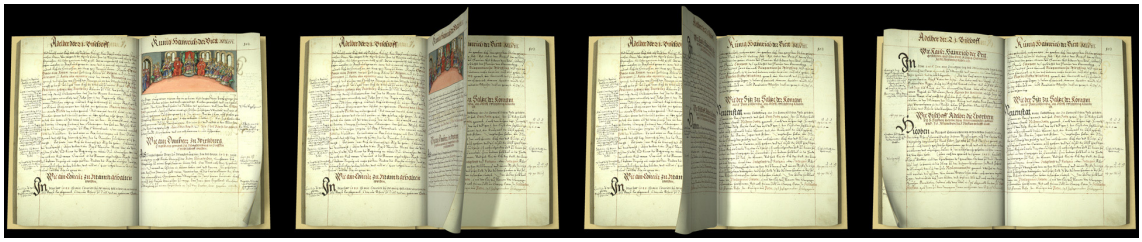


Abb. 8.8. Eine virtuelle Chronik-Seite wird umgeblättert: Frame Nr. 7, 35, 49, 70 von insgesamt 81 Frames (Folio 101v ff. [Frie1582])

zent-Schieberegler gesteuert werden. Dabei legt der Animator nur zu einigen Prozent-Stützstellen veränderte Positionen der Gitterpunkte und Knochen ab, und die Software interpoliert die Zwischen-Schritte. So werden die Gitterpunkte der Seitenecke bei „+100%“ maximal nach oben und bei „-100%“ maximal nach

unten verbogen und trotzdem kann die Seitenecke annähernd stufenlos gebogen werden.

Nun ist das Buchmodell komfortabel animierbar, und es wird ein zeitlicher Bewegungsablauf festgelegt. In einem Zeitfenster von 5,4 Sekunden wird die Seite umgeblättert. Zunächst wird via Prozent-Posen-Regler die Seitenecke rechts hoch gebogen (Frame Nr. 7 in Abb. 8.8), dann die gesamte Seite mithilfe der Skelett-Knochen angehoben und über die Buchmitte geblättert (Frame Nr. 35 und 49). Kurz vor Ende des Vorganges wird die Seitenecke wieder mit dem Prozent-Posen-Regler hoch gebogen, allerdings diesmal in der anderen Richtung (Frame Nr. 70), um das Festhalten beim Absenken der Seite zu simulieren.

Das fertig animierte Buch wird nun noch in eine so genannte *Choreographie* gebracht. Dort wird es auf einen schwarzen Hintergrund gelegt und mit drei Scheinwerfern bestrahlt. Obwohl dies physikalisch nicht möglich ist, werfen zwei der drei Scheinwerfer keine Schatten – diese dienen nur dazu, dass keine sichtbare Seite des Buches in der Dunkelheit verschwindet. Eine Kamera nimmt die Szenerie senkrecht von oben auf, und der dritte Scheinwerfer, der etwas links von der Kamera platziert ist, sorgt für einen realistischen Schattenwurf der angehobenen Seite.

8.4.3 Rendern und Kompression der Filme

Aus den ca. 330 Buchscans für die Multimedia-DVD müssen nun 170 Blätterfilme gerendert werden. Da das Rendern eines Blättervorgangs aus 81 Einzelbildern zu je 1024x768 Pixeln besteht, ist der Vorgang relativ zeitaufwändig: Ein Pentium-4-Prozessor mit 3,0 GHz benötigt für die Berechnung einer Blätteranimation ca. 45 Minuten und für die Berechnung aller Animationen über 5 Tage. Bevor eine Blätteranimation gerechnet wird, müssen die richtigen vier Seitenscans als Texturen in die Projektdatei eingebettet werden. Da die Projektdatei reiner ASCII-Text ist, kann dieser Texturen-Austausch mittels eines externen Hilfsprogramms geschehen. Das Einbetten der richtigen Scanseiten und der anschließende Render-Vorgang ist zeitaufwändig und fehleranfällig. Daher wurde dieser Prozess mit einem perl-Skript automatisiert.

Für perl gibt es ein kostenloses Modul namens `Win32::GuiTest`, welches eigentlich zum automatisierten Testen von grafischen Windows-Anwendungen entwickelt wurde. Entsprechend kann man mithilfe dieses Moduls von einem perl-Skript aus an ein geöffnetes Programm-Fenster Tastenfolgen (CTRL+C, ALT+X), Mausklicks an bestimmte X-Y-Koordinaten oder Mausklicks an bestimmte Windows-Kontrollelemente (Buttons, Menüs, Rollbalken etc.) schicken. So ist es möglich, laufende grafische Anwendungen, die keine Skript-Automatisierung beherrschen, von einem perl-Skript aus fernzusteuern.

Stehen genug A:M-Lizenzen zur Verfügung, kann das für die Render-Automatisierung entwickelte Skript im Prinzip die Aufgabe auf maximal so viele Rechner parallelisieren, wie Animationen zu rendern sind. Im vorliegenden Fall hätte man also theoretisch mit 170 3,0 GHz-Rechnern alle 170 Animationen in 45 Minuten fertig gerechnet. Da die Seitenscans jedoch als statisch angesehen werden und nach einigen Proben auch das Timing der Blätteranimation realistisch wirkt, kann die Zeit für einen kompletten Render-Vorgang einmalig investiert werden: Nach gut 5 x 24 Stunden hat der durch das perl-Skript ferngesteuerte Animation:Master 170 Animationen zu je 81 Frames gerendert. Die Frames mit der Auflösung von 1024x768 Pixeln liegen im Format „Truevision Advanced Raster Graphics Array“ (Targa = TGA) vor und belegen über 20 GB Plattenplatz. Die Zwischenspeicherung in solchen verlustfrei komprimierten Einzelbildern ist insofern sinnvoll, da man Bildmanipulationen wie z.B. die Skalierung auf eine niedrigere Auflösung oder die Verminderung der Framerate leicht und unter Beibehaltung der maximal möglichen Qualität durchführen kann.

Für die Multimedia-DVD sollten die Blätteranimationen für unterschiedlich ausgestattete PCs in zwei Auflösungen vorgehalten werden: 800x600 und 1024x768 Pixel. Für die DVD müssen die Animationen aus Platzgründen und um eine flüssige Wiedergabe zu ermöglichen jedoch in ein Videoformat komprimiert werden. Nach Untersuchungen, ob das *Java Media Framework* (JMF) oder *Apples Quicktime* bessere Videowiedergabe von einer Java-Applikation aus ermöglichen, fiel die Entscheidung zugunsten von Quicktime aus.

Innerhalb von Quicktime wird der *Sorensen v3 Codec* benutzt, da dieser einen guten Kompromiss zwischen geringem Speicherbedarf der Videos und flüssiger Wiedergabe darstellt. Da der Film vorwärts und rückwärts synchron zur Mausbewegung des Benutzers abgespielt werden muss, ist für die flüssige Wiedergabe notwendig, dass nur Vollbilder (I-Frames) in den Videostrom kodiert werden. Auf prädizierte Bilder (B-/P-Frames) sollte aus diesem Grund verzichtet werden.

Es muss nun ein Weg gefunden werden, aus den 170x81 TGA-Einzelbildern 170 Quicktime MOV-Filme zu generieren. Da dies eine zeitaufwändige und fehleranfällige Arbeit ist, soll auch hier wieder automatisiert werden. Leider beherrscht selbst die lizensierte Pro-Version der Quicktime-Software keinen Import von Einzelbildern. Daher kommen zwei kostenlose Programme zum Einsatz, die auf die automatisierte Verarbeitung von Videos spezialisiert sind: *AviSynth* [wwwAviSyn] und *VirtualDub* [wwwVirtDub].

AviSynth klinkt sich in das systemweite Video-Codecsystem von Windows ein und nimmt als Input-Format Textdateien entgegen, die in einer einfachen Skriptsprache Videos und Operationen auf diesen Videos beschreiben. Auf der Ausgabeseite ist AviSynth ein so genannter *Frameserver*, und fast jedes Pro-

gramm, welches unter Windows Videos einlesen kann, kann einen solchen Frameserver als Eingabequelle wählen.

Das folgende kurze AviSynth-Skript generiert aus einer Sammlung von 81 Einzelbildern einen Videostrom, der zur Laufzeit von AviSynth auf die Größe von 800x600 Pixeln zurecht skaliert wird (vgl. Bsp. 8-2).

Beispiel 8-2. AviSynth erzeugt aus 81 Einzelbildern einen Videostrom mit einer Framerate von 15 fps und der Auflösung von 800x600 Pixeln

```
ImageReader("o:\3d_book\fries_001\fries_001_%1d.tga",0,80,15,true)
BicubicResize(800,600)
FlipVertical()
```

Da AviSynth ein reiner Frameserver ist, benötigt man ein Hilfsprogramm wie z.B. VirtualDub, mit welchem man den erzeugten Videostrom in eine Datei umlenken kann. Seinem Ursprung in der Apple-Macintosh-Welt hat Quicktime wohl das Fehlen einer Möglichkeit zur Kommunikation mit Windows-Frameservern zu verdanken – es gibt leider keinen direkten Weg von AviSynth zu Quicktime. Daher muss der Umweg über VirtualDub genommen werden.

VirtualDub lässt sich mit so genannten Job-Dateien automatisieren, und so wurden temporäre unkomprimierte Windows-Videodateien (AVI) erzeugt, die sich problemlos in die Pro-Version der Apple Quicktime-Applikation importieren lassen. Die folgende Job-Datei, von der hier nur die wichtigsten Zeilen wiedergegeben werden, verbindet VirtualDub mit dem AviSynth-Frameserver (über die AviSynth-Datei `c:\temp\fries_temp.avs`) und schreibt den Videostrom in eine unkomprimierte AVI-Datei.

Beispiel 8-3. Job-Datei für VirtualDub wandelt AVS in AVI

```
VirtualDub.Open("c:\temp\fries_temp.avs","",0);
VirtualDub.video.SetCompression();
 [...]
VirtualDub.SaveAVI("o:\3d_book\mov\001.avi");
VirtualDub.Close();
```

Mit wenigen Tastatur-Befehlen, die automatisiert mittels perl-Skript und dem Modul `Win32::GuiTest` an die Quicktime-Applikation gesendet werden, lässt sich die so generierte AVI-Datei in Quicktime importieren und als Quicktime-Movie (MOV) mit dem Sorensen-Codec komprimiert wieder exportieren. Die unkomprimierte AVI-Datei kann jeweils nach der Konvertierung gelöscht werden.

Die 170x81 TGA-Einzelbilder belegen, wie oben erwähnt, über 20 GB Plattenplatz. Die 170 MOV-Videos belegen nach der Kompression in der Auflösung 800x600 ca. 930 MB und in der Auflösung 1024x768 ca. 1,6 GB. Der Speicherplatz für beide Auflösungen lässt auf einem einseitigen, einschichtigen DVD-Me-

dium also noch genug Platz für andere Applikationsdaten. Die Konvertierung der TGA-Bilder in die 340 MOV-Dateien benötigt auf einem 3,0 GHz PC ca. 8 Stunden.

Abbildung 8.9 gibt einen zusammenfassenden Überblick, wie die verschiedenen Programme und Skripte miteinander interagieren, um automatisch die Animationen zu rendern und die Render-Ergebnisse in das benötigte Zielformat Quicktime-MOV zu konvertieren. Perl kommt hier mehrfach zum Einsatz, um einerseits temporäre, textbasierte Steuerdateien für die jeweiligen Programme zu erzeugen oder anzupassen (`*.PRJ`, `*.AVS`, `*.JOB`) und andererseits, um Programme mittels Tastatur- oder Mausevents fernzusteuern (*Animation:Master*, *Quicktime*).

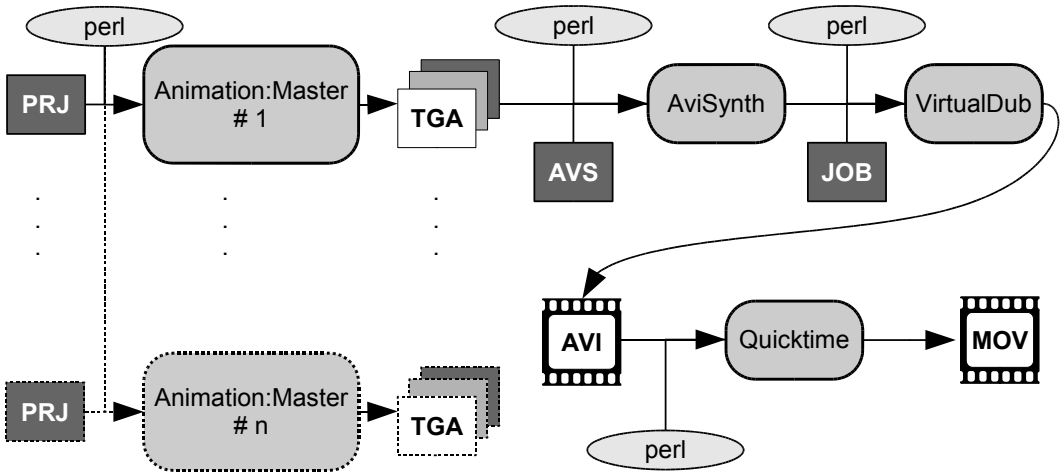


Abb. 8.9. Datenfluss bei automatischer Generierung von 3D-Blätterfilmen

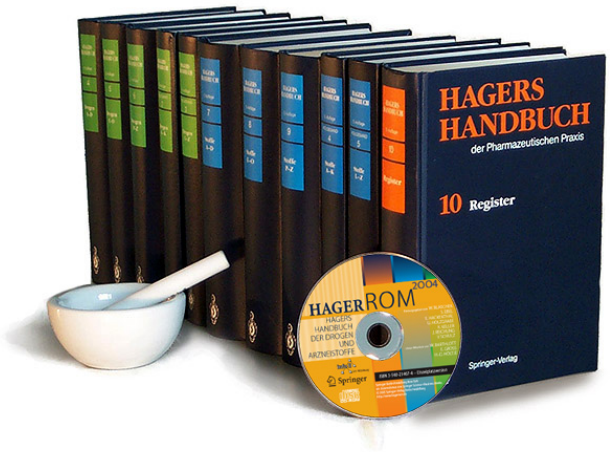
Eine oder mehrere Instanzen von Animation:Master generieren aus den Projektdateien mit den jeweils passend eingebetteten Seitenscans die Animation in TGA-Einzelbilder. Diese werden mit dem Frameserver AviSynth skaliert und zu einem virtuellen Videostrom zusammengefasst. Dieser Videostrom wird von VirtualDub in eine unkomprimierte Videodatei im Format AVI gespeichert. Diese AVI-Datei wird mit der ferngesteuerten Quicktime-Applikation in einen Film im Format MOV mit dem Codec Sorensen v3 komprimiert.

Kapitel 9

Anwendungsfälle

In den vorangegangenen Kapiteln wurden Konzepte und Vorgehensweisen beschrieben, die prinzipiell auf beliebige digitale Textwerke angewendet werden können. In diesem Kapitel sollen nun die (kommerziellen) Anwendungsfälle vorgestellt werden, mit denen die Verfahren in der Praxis erprobt wurden. Eine kurze Übersicht über die Anwendungsfälle wurde bereits in Abschnitt 1.2 gegeben. In diesem Kapitel werden die Anwendungsfälle nun jedoch detaillierter beschrieben und mit Bildschirmfotos, zusätzlichen Statistiken und einigen weiterführenden Experimenten vorgestellt. So unterstreicht dieses Kapitel die universelle Einsetzbarkeit der in dieser Arbeit vorgestellten Vorgehensweisen.

9.1 HagerROM



Die digitale pharmazeutische Enzyklopädie *HagerROM* [BEH+04] entstand aus den Quellen zum Druckwerk der 12-bändigen Enzyklopädie „*Hagers Handbuch der Pharmazeutischen Praxis*“ [Bru+00], welche ca. 12.000 Buchseiten umfasst und im Springer-Verlag erschienen ist. Die erste Version der *HagerROM* wurde 2001 veröffentlicht und seitdem erscheinen Folgeversionen in annähernd jährlichem Rhythmus. Zusätzlich zu den rechts in der Abbildung dargestellten Bänden sind in *HagerROM 2004* weitere Register-Inhalte, der

Wortschatz, das medizinische Deutsch⇔Englisch-Wörterbuch von Dr. Reuter und eine verlinkte Liste von Fertig-Arzneimitteln integriert.

Tab. 9.1. Merkmale des Anwendungsfalles „Hager“

Merkmal	*Wert*
Domäne	Pharmazie, Chemie, Biologie, Medizin
Primärsprache	Deutsch
Distributionsmedium	CD-ROM (Einzelplatz, Mehrplatz)
Umfang HTML-Text[1)]	175,9 MB
Umfang bereinigter Rohtext[2)]	62,5 MB
Anzahl Dateien[2)]	18.200
Umfang aller Volltextsuche-Indizes[3)]	484,7 MB

[1)] bzgl. aller Dateien des Werkes [2)] nur bzgl. Dateien der Volltextsuche [3)] inkl. Kontexter-Dateien

Es folgen einige kommentierte Bildschirmfotos der *HagerROM 2004*.

Abbildung 9.1 zeigt das Hauptfenster der überwiegend in Java implementierten *HagerROM 2004*. Mit dem ausklappbaren Navigationsbaum kann ein hierar-

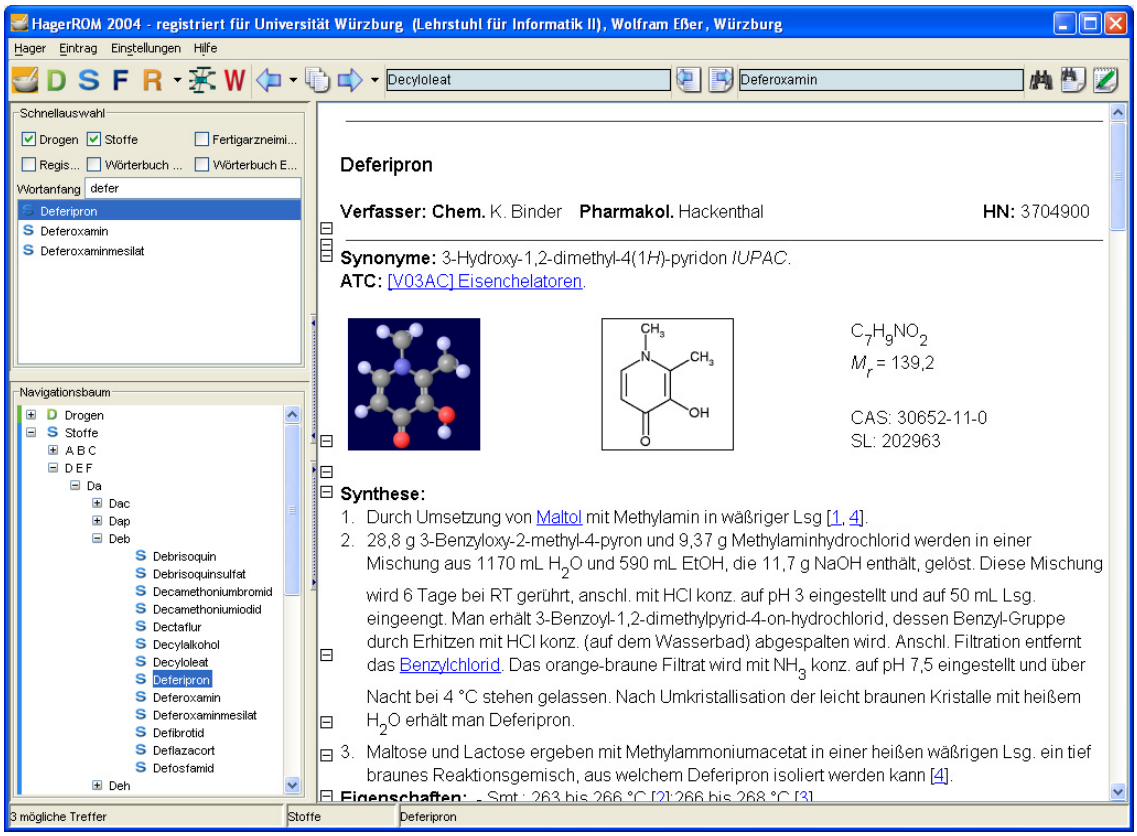


Abb. 9.1. Das Hauptfenster der *HagerROM 2004* mit Navigationsbaum und Textfenster

chischer Abstieg zum gewünschten Eintrag erfolgen. Über dem Navigationsbaum kann in der Schnellauswahl eine Filterung der Bauminhalte nach Wortanfängen der Eintragstitel durchgeführt werden. Gibt man hier z.B. „`echi`“ als Präfix ein, so werden u.a. alle Einträge zur Heilpflanze `Echinacea` angezeigt. Im rechten Textfenster wird der aktuelle Eintrag dargestellt. Hier kann man persönliche Notizen eingeben oder mit einem leuchtenden Markierstift Textstellen hervorheben. Ein Klick auf die kleinen Vorschaubilder von Abbildungen, 3D-Molekülen und chemischen Strukturformeln startet einen Bildbetrachter mit einer detaillierten zoombaren Ansicht der entsprechenden Grafik.

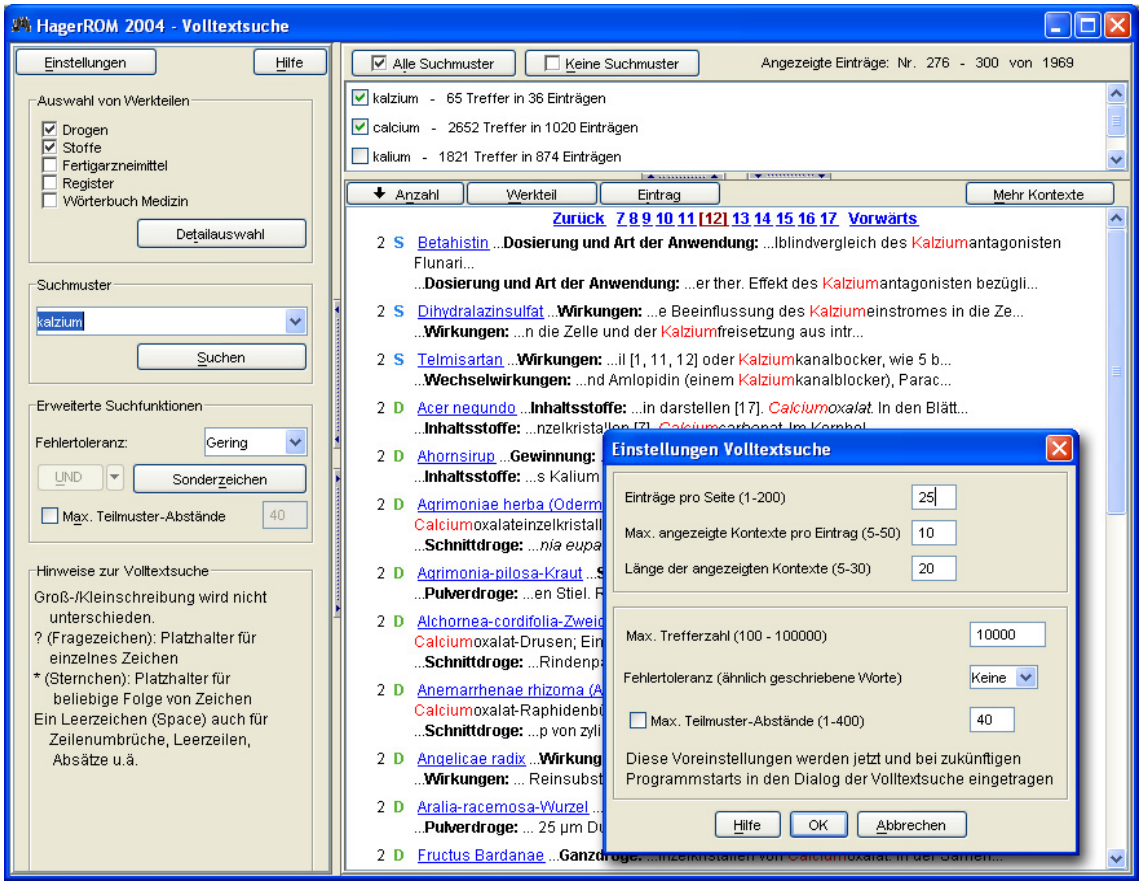


Abb. 9.2. Präsentation der Ergebnisse einer fehlertoleranten Volltextsuche nach „`kalzium`“

In Abbildung 9.2 ist das Ergebnis einer fehlertoleranten Volltextsuche (mit geringster Toleranzstufe) in den Werkteilen „Drogen“ und „Stoffe“ wiedergegeben. Es wurden nicht nur Treffer zum Original-Suchmuster „`kalzium`“ sondern u.a. Treffer für die Morphs „`calcium`“ und „`kalium`“ gefunden. Der nicht gewünschte Morph „`kalium`“ wurde im oberen rechten Fensterbereich abgewählt, so dass seine Treffer aus der unten rechts stehenden Trefferliste herausgenommen wurden. In der Trefferliste bedeuten die Zahlen der ersten Spalte die Anzahl der Treffer in dem entsprechenden Eintrag. In der zweiten Spalte steht ein Buchstabe für

den Werkteil (hier: „**D**rogen“ und „**S**toffe“) gefolgt von der anklickbaren Überschrift des Eintrags. Dahinter stehen zu jedem Treffereintrag einige Treffer-Kontexte. Zu jedem Kontext wird die Überschrift des Abschnitts fett gesetzt, in dem sich der jeweilige Treffer befindet. Vor dem Volltextsuche-Dialog befindet sich der Dialog für die Volltextsuche-Einstellungen. Hier kann z.B. der Grenzwert für die maximal gewünschte Trefferzahl oder die maximale Länge der Kontext-Fragmente an den persönlichen Bedarf angepasst werden.

Unter den in dieser Arbeit vorgestellten Anwendungsfällen sticht *HagerROM* vor allem durch den Umfang der Werktexte hervor. Mit über 62 MB Platzbedarf für den bereinigten Text ist es das umfangreichste aller hier betrachteten Werke. Daher werden im Folgenden die Ergebnisse einiger Messungen zur Geschwindigkeit der exakten und fehlertoleranten Volltextsuche auf diesem Werk wiedergegeben.

Alle Messungen erfolgten auf einem PC mit Intel Pentium-4-Prozessor mit 2,6 GHz, 512 MB Arbeitsspeicher von der lokalen ATA-100-Festplatte unter dem Betriebssystem Windows XP. Die Volltextsuche wird als Kommandozeilen-Version benutzt und für jede Suchanfrage jeweils neu gestartet.

Zunächst wird der Zeitbedarf für die exakte Suche gemessen. Hierbei werden zufällig 1000 Suchmuster P mit einer zufälligen Länge $5 \leq |P| \leq 25$ aus den bereinigten Dateien ermittelt. Für Experiment 1 (links) in Abb. 9.3 werden die Suchmuster exakt so gesucht, wie sie im Text stehen und liefern daher immer mindestens einen Treffer. Für Experiment 2 (rechts) werden solange an zufälligen Stellen im Muster benachbarte Buchstaben vertauscht, bis im Werk keine Treffer mehr vorkommen. So soll ermittelt werden, wieviel schneller die exakte q-Gramm-Suche nicht existente Suchmuster mit „0 Treffern“ als solche erkennt.

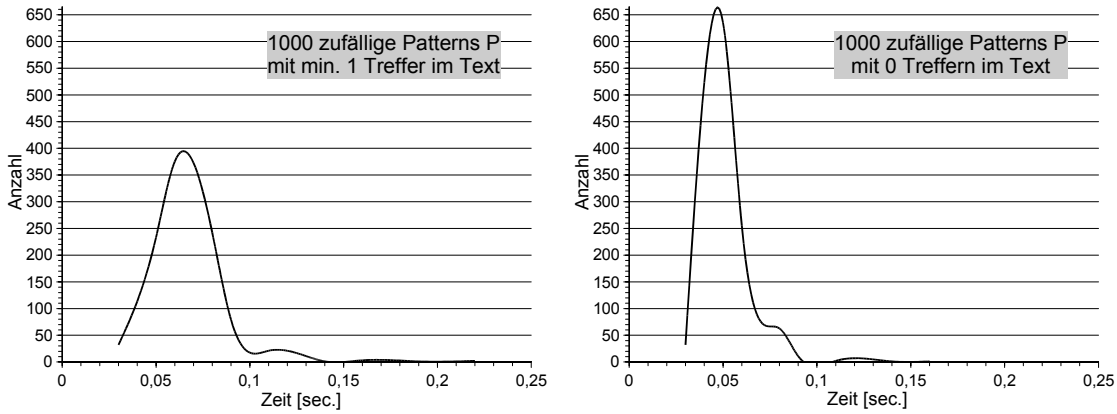


Abb. 9.3. Laufzeit-Experimente zur exakten Volltextsuche mit Patterns aus dem Text. Exp. 1 (li.) sucht Patterns unverändert. Exp. 2 (re.) vertauscht Buchstaben für „0 Treffer“.

Für die Kurven von Abbildung 9.3 wurden die jeweils 1000 ermittelten Laufzeiten in Clustern mit einer Genauigkeit von zwei Nachkommastellen gezählt. So

terminiert die Volltextsuche bei Experiment 1 in 375 von 1000 Fällen bereits nach 0,06 Sekunden, wodurch die Kurve an dieser Stelle ihr Maximum hat. In Experiment 2 befindet sich das Maximum der Kurve bei 0,05 Sekunden, welches der Zeitbedarf für 635 von 1000 Volltextsuchen ist.

Aus der Abbildung ist zu erkennen, dass die q-Gramm-Suche im Mittel etwas schneller ist, wenn das Pattern *nicht* Bestandteil des Textes ist. Dieser Umstand ist besonders interessant, da es durch das optional vorgeschaltete WPM-Verfahren vorkommen kann, dass Morphs den Filter-Graphen passieren, die trotzdem nicht Bestandteil des Textes sind. Diese 0-Treffer-Morphs müssen jedoch zunächst mit der Volltextsuche gesucht werden, um sie abschließend verwerfen zu können. Das Experiment belegt, dass dies sehr effizient geschieht.

Tabelle 9.2 zeigt zu den beiden in Experiment 1 und Experiment 2 ermittelten Messreihen den minimalen, maximalen und mittleren Zeitbedarf einer exakten q-Gramm-Volltextsuche.

Tab. 9.2. Zeitbedarf für exakte q-Gramm-Volltextsuchen mit und ohne Treffer im Text

	Exp. 1	***Exp. 2***
min. Zeitbedarf (Exp. 1 z.B. `äureac`)	0,031 sec.	0,031 sec.
max. Zeitbedarf (Exp. 1 z.B. `ac, dem man genauso phen`)	0,219 sec.	0,156 sec.
mittlerer Zeitbedarf	0,055 sec.	0,044 sec.

Die Zeitmessung der exakten Suche zeigt, dass diese selbst im schlimmsten hier überprüften Fall auf einem üblichen PC deutlich unter einer Sekunde abläuft.

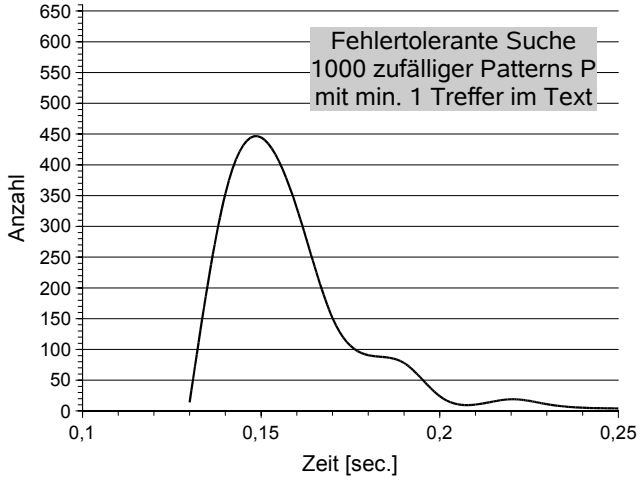


Abb. 9.4. Experiment 3 zur Laufzeit der Fehlertoleranz

Für die optionale Fehlertoleranz ist hier aufgrund des erhöhten Rechenaufwandes mit einer Steigerung des Zeitbedarfs zu rechnen. Analog zu Experiment 1

werden 1000 Suchmuster diesmal jedoch mit geringster Fehlertoleranz gesucht und die benötigte Zeit gemessen, welche in Abbildung 9.4 grafisch dargestellt ist.

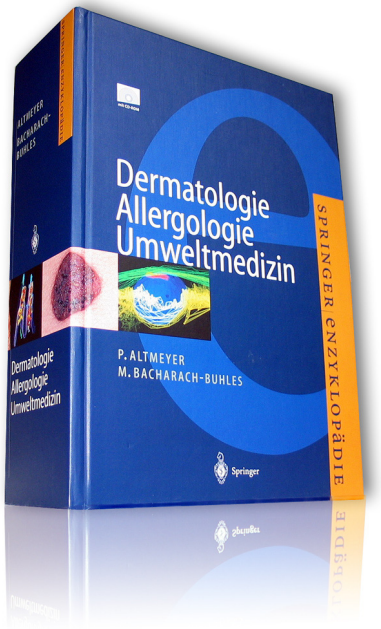
Tabelle 9.3 gibt die durchschnittlichen Laufzeiten bei geringer, mittlerer und hoher Fehlertoleranz (FT) numerisch wieder.

Tab. 9.3. Mittlerer Zeitbedarf für 1000 fehlertolerante Volltextsuchen mit Treffern im Text

	Exp. 3
mittlerer Zeitbedarf (geringe FT)	0,162 sec.
mittlerer Zeitbedarf (mittlere FT)	0,409 sec.
mittlerer Zeitbedarf (hohe FT)	2,009 sec.

Ein Vergleich von Experiment 1 und Experiment 3 zeigt, dass im Mittel eine fehlertolerante Suche zwischen dreimal so viel Zeit (bei geringer Fehlertoleranz) und 36 mal soviel Zeit (bei hoher Fehlertoleranz) benötigt wie eine exakte Suche mit Treffern im Text. Trotz vorgeschaltetem WPM-Verfahren liegt aber die durchschnittliche Laufzeit bei den 1000 zufälligen Suchmustern selbst auf höchster Toleranzstufe bei tolerierbaren 2 Sekunden.

9.2 Springer Enzyklopädie Dermatologie



Das von P. Altmeyer und M. Bacharach-Buhles im Jahr 2002 erstmals im Springer-Verlag veröffentlichte Werk „*Springer Enzyklopädie: Dermatologie, Allergologie und Umweltmedizin*“ [AB02] enthält über 13.000 Stichworte, wobei ca. 2.500 Haupteinträge ausführlicher auf Themen der Dermatologie, Venerologie, Allergologie, Umweltmedizin sowie auf ästhetische und korrektive Verfahren eingehen.

Nach Veröffentlichung des Buches, dem eine CD-ROM mit einer PDF-Version des Werkes beilag, wurde im März 2003 eine aus den digitalen Quellen aufbereitete Version über den Web-Auftritt des Pharmaunternehmens Galderma online gestellt. Der Zugang ist für Ärzte, Medizin-Studenten und Apotheker kostenlos, jedoch ist aufgrund der Bestim-

mungen des Heilmittelwerbegesetzes dieser Bereich der Galderma-Webseite nur über ein kostenloses DocCheck-Konto (`http://www.doccheck.de`) möglich.

Tab. 9.4. Merkmale des Anwendungsfalles „*Altmeyer*“

Merkmal	*Wert*
Domäne	Medizin, Dermatologie, Allergologie
Primärsprache	Deutsch
Distributionsmedium	Online (`http://www.galderma.de`)
Umfang HTML-Text[1)]	43,7 MB
Umfang bereinigter Rohtext[2)]	8,2 MB
Anzahl Dateien[2)]	18.912
Umfang aller Volltextsuche-Indizes[3)]	92,4 MB

[1)] bzgl. aller Dateien des Werkes [2)] nur bzgl. Dateien der Volltextsuche [3)] inkl. Kontexter-Dateien

Es folgen einige kommentierte Abbildungen zur Online-Version des Werkes.

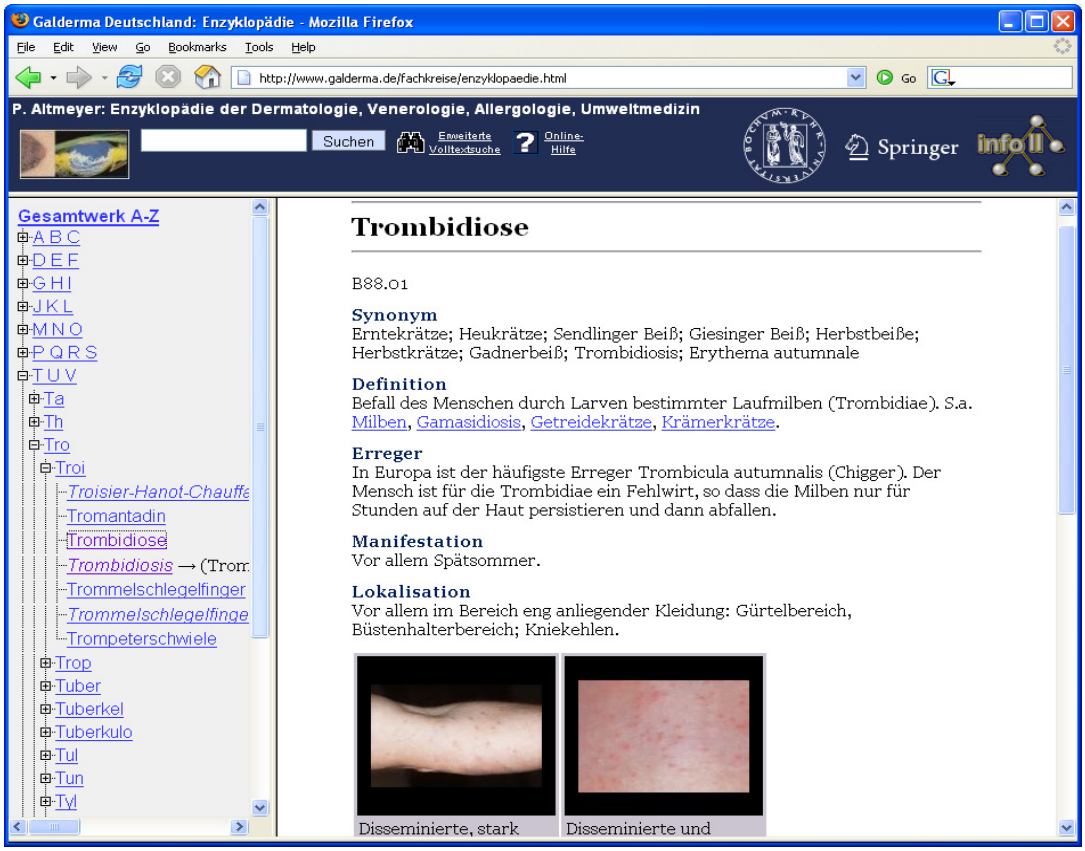


Abb. 9.5. Ein Eintrag aus der Online-Version des *Altmeyer*-Werkes

In Abbildung 9.5 ist der Eintrag „Trombidiose“ innerhalb eines WWW-Browsers abgebildet. Der Navigationsbaum links wird über ein XML-gesteuertes PHP-

Skript realisiert. Die Seiten sind Standard-HTML-Seiten, welche die eingebetteten Bilder mittels eines PHP-Skriptes zoombar machen, wie Abb. 9.6 zeigt.

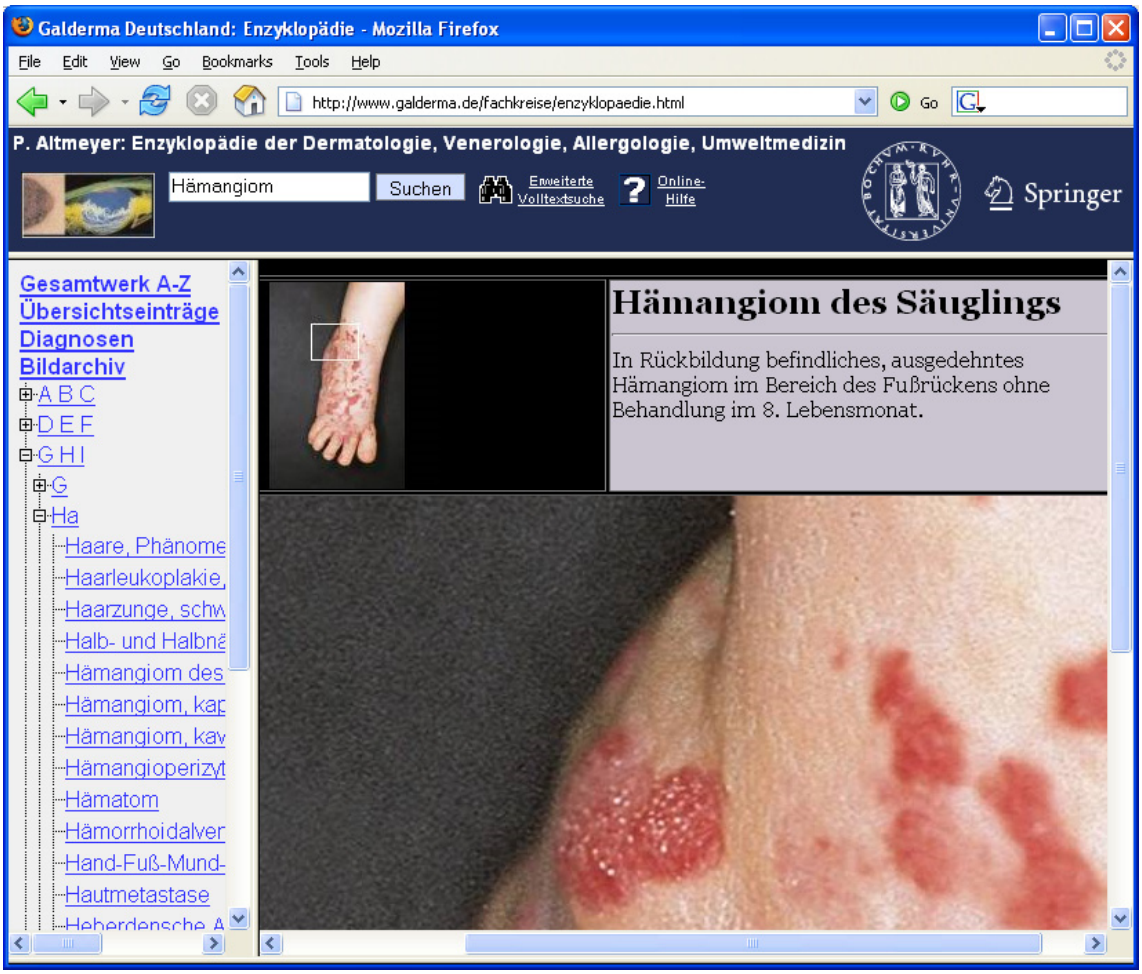


Abb. 9.6. Interaktives Zoomen von Online-Bildern mittels PHP

Über das Suche-Feld oberhalb von Navigationsbaum und Eintragsanzeige kann eine exakte Standard-Suche durchgeführt werden. Um Parameter wie z.B. Fehlertoleranz einstellen zu können, muss der Benutzer in der Titelzeile auf „Erweiterte Volltextsuche“ klicken und erhält dann einen Dialog, der der Java-Version (vgl. vorheriger Abschnitt) ähnlich ist.

Abbildung 9.7 zeigt die Ergebnisse einer fehlertoleranten Volltextsuche nach dem Begriff „`hämangiom`“ in der WWW-Version des *Altmeyer*-Werkes. Über den Treffern wird analog zur lokalen *HagerROM*-Version der Volltextsuche eine Liste mit allen gefundenen Morphs angezeigt, damit der Benutzer Treffer von eventuell nicht erwünschten Morphs aus der Trefferliste entfernen kann. Zum ersten Treffer eines jeden Eintrags wird ein Kontext-Fragment angezeigt, in welchem das Treffermuster optisch hervorgehoben wird.

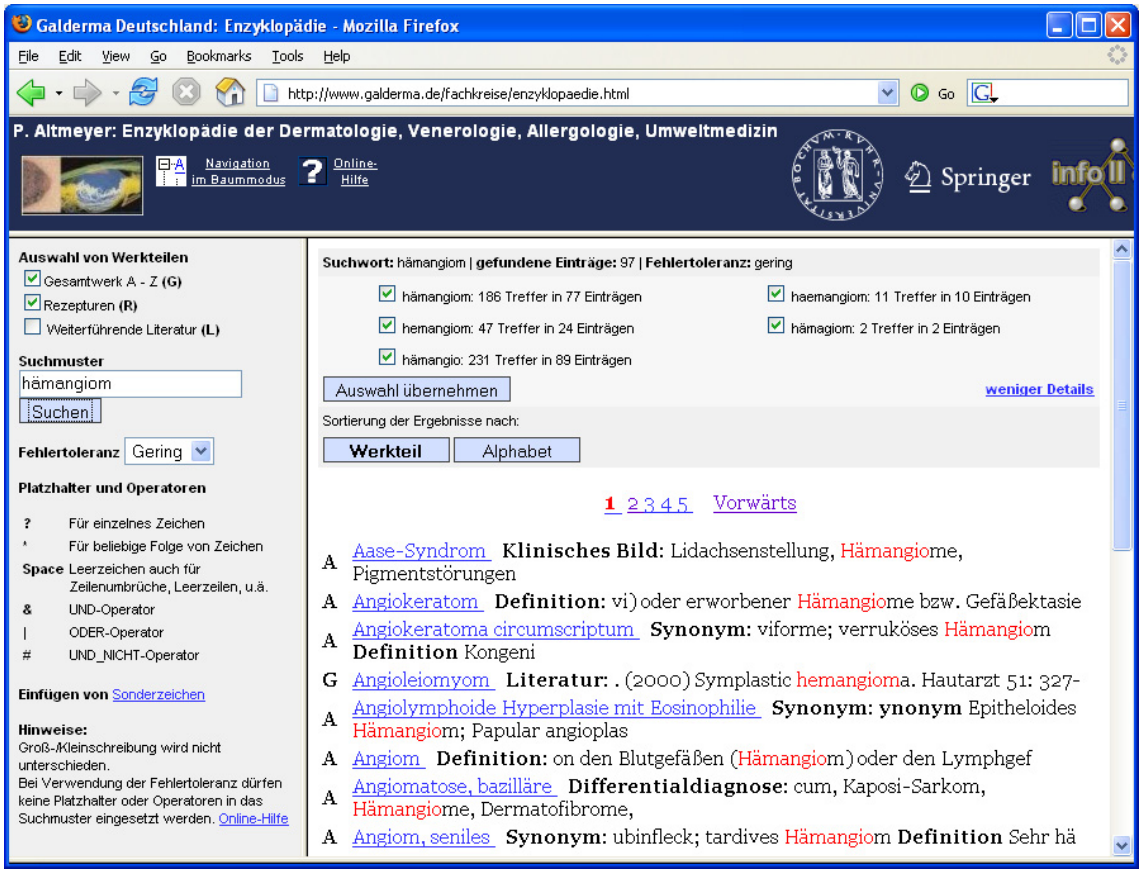


Abb. 9.7. Ergebnisse einer fehlertoleranten Online-Volltextsuche

Wie bereits vorher in dieser Arbeit beschrieben, ist ein interessantes Merkmal an der Online-Version dieses Werkes die Tatsache, dass über die Log-Dateien der WWW-Server die Suchzugriffe der Benutzer (anonym) mitprotokolliert werden. Im Folgenden werden daher einige Statistiken wiedergegeben, die auf den Protokollen basieren, die seit Start des Online-Angebotes (1. März 2003) bis zum 20. Dezember 2004 aufgezeichnet wurden. Es wurden nur die Suchanfragen ausgewertet und gezählt, nicht jedoch sonstige Zugriffe auf Texte oder Bilder. Aus den Statistiken lassen sich also keine absoluten Zugriffszahlen auf den Server herleiten. Aus den Protokollen der Suchzugriffe wurden vor der Durchführung der im Folgenden dargestellten Messungen zunächst alle Anfragen aus den IP-Adressenbereichen der Test- und Entwicklungsrechner entfernt.

Im Zeitfenster von knapp 22 Monaten wurden insgesamt 53.081 Volltextsuchen an 659 verschiedenen Tagen durch die Benutzer durchgeführt. Dies ergibt durchschnittlich 80 Suchaufträge pro Tag. Verglichen mit den Zeitmessungen aus dem vorhergehenden Abschnitt lässt sich also sagen, dass das System noch nicht ausgelastet ist und noch Reserven hat. Selbst bei einem großzügig dimen-

sionierten angenommenen Zeitbedarf von 1 Sekunde pro Suche könnte der Server mehrere Tausend (exakte) Suchen in einem 8-Stunden-Fenster durchführen.

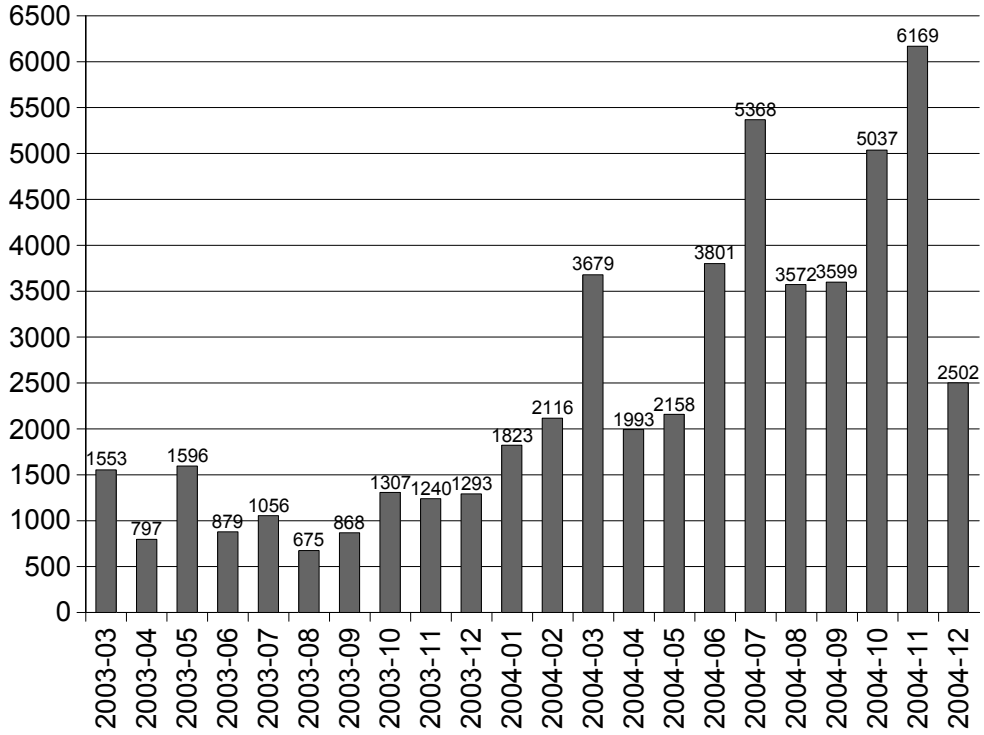


Abb. 9.8. Anzahl Aufrufe der Volltextsuche je Monat

Wie Abbildung 9.8 zeigt, erfreut sich das Online-Angebot in den letzten Monaten zunehmender Beliebtheit unter den Nutzern. Der Monat 2004-12 ist nicht komplett erfasst, da der Stichtag der Auswertung in diesen Monat fiel. Abbildung 9.9 zeigt die Anzahl der Suchaufrufe verteilt über 24 Tagesstunden.

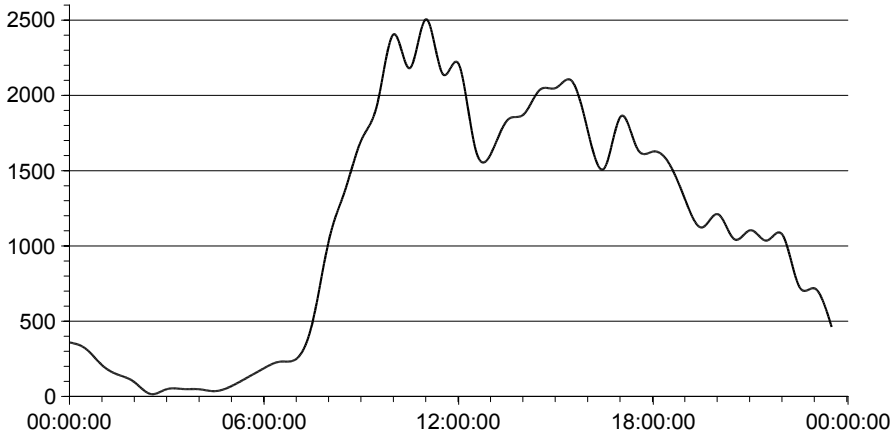


Abb. 9.9. Anzahl Aufrufe der Volltextsuche über Tageszeiten

Durch die Auswertung der Suchmuster der Benutzer konnten ebenfalls interessante Erkenntnisse gewonnen werden. So wurde im Schnitt jedes Suchmuster fast vier mal gesucht: 13.496 verschiedene Suchmuster finden sich in den Protokolldateien zu den 53.081 insgesamt durchgeführten Suchen, welche eine mittlere Länge von 11,6 Zeichen haben (vgl. Abb. 9.10). Die Hitliste der beliebtesten

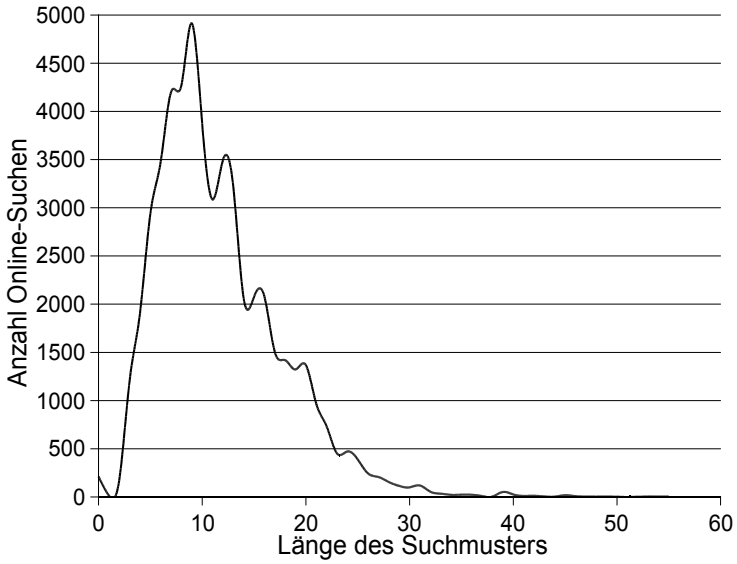


Abb. 9.10. Verteilung der Suchmuster-Längen bei 53.081 WWW-Suchen

Suchbegriffe ist in Tabelle 9.5 wiedergegeben. Die hohen Zahlen relativ kurzer – und damit unspezifischer – Begriffe legt den Verdacht nahe, dass die Benutzer vielfach das relativ junge Online-Angebot zunächst mit einer Test-Suche prüfen wollen. Hierfür verwenden sie Begriffe, die in einem Text der entsprechenden Domäne mit hoher Wahrscheinlichkeit vorkommen.

Tab. 9.5. Top-10 der meistgesuchten Begriffe im Online-*Altmeyer*

Pos.	***Anzahl***	***Suchmuster***
1.	530	psoriasis
2.	373	melanom
3.	346	lichen ruber
4.	303	akne
5.	212	*(das leere Suchmuster)*
6.	206	granuloma anulare
7.	171	ekzem
8.	163	erythema nodosum
9.	162	basaliom
10.	154	pityriasis

Von den 13.496 unterschiedlichen Musterklassen, die mittels WWW-Suche gesucht wurden, konnten 5.691 (42%) nicht im Werktext gefunden werden. Von diesen wiederum enthielten 3.104 (55% von 5.691) mindestens ein Leerzeichen. Beispiele für solche Suchmuster sind z.B. „`area migrans bei pocken`“, „`atopisches ekzem gehör`“ oder „`lentiginose mundschleimhaut`“. Dabei existieren sehr wohl Einträge, auf denen alle Teilworte der Suchmuster vorkommen – aber eben nicht in dieser Reihenfolge und in größerem Abstand.

Die gezeigten Suchmuster lassen daher vermuten, dass die Anwender hier in Anlehnung an WWW-Suchmaschinen wie z.B. Google das Leerzeichen als „boolesche UND-Verknüpfung“ der umgebenden Worte meinen und nicht als exakte Textphrase, wie sie von der *Altmeyer* WWW-Suche verstanden werden. Bei der Interpretation von Suchmustern durch den Suchalgorithmus sollte also in Erwägung gezogen werden, dass Anwender eventuell ohne Lektüre der vorhandenen Hilfetexte ihre Muster konform zu etablierten WWW-Suchmaschinen eingeben und durch diese Fehlbedienung die Qualität ihrer Suchergebnisse verringern. Für eine Umsetzung dieser Erkenntnis in eine veränderte Interpretation der Suchmuster muss jedoch momentan noch auf den Abschnitt „Ausblick“ verwiesen werden.

Alle 5.691 Suchmuster-Klassen, die keine Treffer im Werk liefern, werden in einem Experiment jeweils mit den drei Toleranzstufen gering, mittel und hoch gesucht. Falls eines dieser Suchmuster mindestens ein Leerzeichen enthält, wird es außerdem umgewandelt, indem alle Leerzeichen zu „|“ (UND) bzw. „&“ (ODER) verändert werden. So werden die Leerzeichen also quasi als UND/ODER-Operator interpretiert. Tabelle 9.6 zeigt, in wie vielen Fällen die Suchmuster dann Treffer liefern können, wobei natürlich keine Aussage darüber gemacht werden kann, ob die Benutzer auch tatsächlich die so im Experiment gefundenen Einträge suchten. Die Zahlen geben trotzdem eine Vorstellung vom möglichen Potenzial solcher veränderter 0-Treffer-Muster.

Tab. 9.6. Erfolgreiche „Korrektur“ von Benutzer-Suchmustern ohne Treffer

Suchmethode	*#*	*%*
Kein Erfolg (exaktes Benutzermuster)	5.691	100%
Erfolg durch UND	1.472	26%
Erfolg durch ODER	2.691	47%
Erfolg durch FT=*gering*	1.973	35%
Erfolg durch FT=*mittel*	2.813	49%
Erfolg durch FT=*hoch*	3.118	55%

Mithilfe der Protokolldateien war es auch möglich zu ermitteln, mit welchen Parametern die WWW-Suchen durchgeführt wurden. Standard-Einstellung ist die exakte, nicht-fehlertolerante Suche in allen Werkteilen. Tabelle 9.7 zeigt eine Anzahl von insgesamt 274 fehlertoleranten Suchen (aus allen 53.081 Suchanfragen). Interessant ist das Verhalten, dass bei Verwendung der Toleranz zu den extremen Toleranzstufen tendiert wird: Entweder es wird mit minimaler oder mit maximaler Toleranz gesucht.

Tab. 9.7. Anzahl und Verteilung toleranter Suchen durch Online-Nutzer

Benutzte Toleranzstufe	#	%
FT=*gering*	114	42%
FT=*mittel*	47	17%
FT=*hoch*	113	41%
Summe	274	100%

Da die Benutzer zur Aktivierung der Fehlertoleranz durch Umschalten der entsprechenden Option selber aktiv werden müssen, ist es nicht verwunderlich, dass dieses Leistungsmerkmal in der überwiegenden Mehrzahl der Fälle nicht verwendet wird. Trotzdem zeigt Tabelle 9.6 anhand der korrigierbaren Suchmuster ohne Treffer, dass die Benutzer diese Funktion öfter nutzen sollten.

Es ist daher ebenfalls Bestandteil des Ausblicks, dass die WWW-Suche, bevor sie bei einer exakten Suche das Ergebnis „0-Treffer“ präsentiert, den Begriff fehlertolerant suchen sollte. Falls der Begriff Leerzeichen enthält, kann zusätzlich versucht werden, mit eingefügten „UND/ODER“-Operatoren Treffer zu finden.

9.3 Parasitology Research & Encyclopedia

Das englisch-sprachige Kombinationswerk bestehend aus dem wissenschaftlichen Journal „*Parasitology Research*“ [CM04] und der „*Encyclopedic Reference of Parasitology*” [Meh01] wurde durch zahlreiche Querverweise der beiden integrierten Werke gebildet. Die Werke ergänzen sich insofern, dass die Enzyklopädie das gesicherte Wissen auf dem Gebiet der Parasitologie abdeckt und das Journal den jeweils aktuellsten Stand der Forschung darstellt. Im Verlauf dieser Arbeit wird vielfach für das

Gesamtwerk die Kurzbezeichnung „*Parasitology*“ verwendet. Obwohl Journal und Enzyklopädie kostenpflichtige Produkte des Springer-Verlags sind, hat dieser entschieden, während einer Pilotphase das Kombinationswerk seit Oktober 2004 zum kostenlosen Testen online zu stellen. Danach soll es nur noch den Subskriptionskunden des Journals bei Springer-Link [wwwSprLnk] zur Verfügung stehen. Springer-Link ist der Online-Informationsdienst für wissenschaftliche Bücher und Zeitschriften des Verlages.

Tab. 9.8. Merkmale des Anwendungsfalles „*Parasitology*“

Merkmal	***Wert***
Domäne	Parasitologie, Medizin
Primärsprache	Englisch
Distributionsmedium	Online (`http://www.springerlink.com`)
Umfang HTML-Text[1)]	171,0 MB
Umfang bereinigter Rohtext[2)]	36,5 MB
Anzahl Dateien[2)]	15.268
Umfang aller Volltextsuche-Indizes[3)]	266,2 MB

[1)] bzgl. aller Dateien des Werkes [2)] nur bzgl. Dateien der Volltextsuche [3)] inkl. Kontexter-Dateien

Es folgen kommentierte Bildschirmfotos des Anwendungsfalles *Parasitology*.

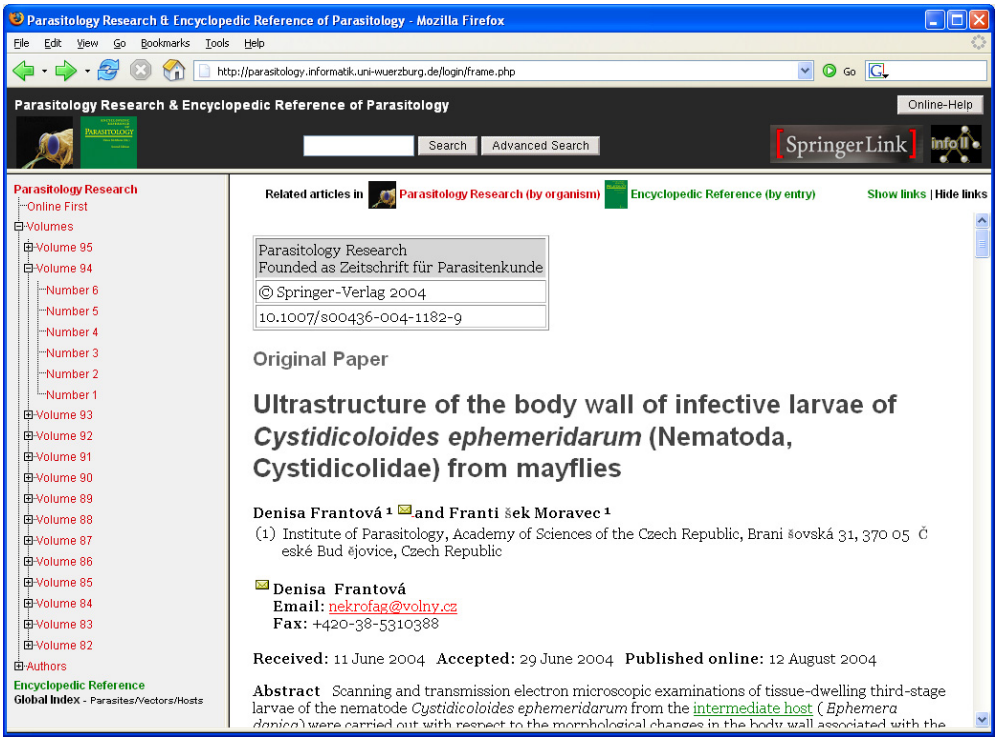


Abb. 9.11. Ein Eintrag aus dem Online-Werk *Parasitology*

In Abbildung 9.11 ist ein Journal-Eintrag aus *Parasitology* zu sehen. Dieser verfügt in der Kopfzeile und im Text des Artikels über (grün eingefärbte) Querverweise, die in die Enzyklopädie zeigen.

Wie im Verlauf der Arbeit bereits beschrieben wurde, konnte auch in diesem Anwendungsfall die fehlertolerante Volltextsuche mit dem WPM-Verfahren zum Einsatz kommen. Da beide Bestandteile des vorliegenden Werkes (Journal und Enzyklopädie) jedoch in englischer Sprache verfasst sind, mussten die Submorph-Regeln erst an diese Sprache angepasst werden, wie in Abschnitt 5.9 (Seite 86) beschrieben wurde. Abbildung 9.12 zeigt das Suchergebnis einer fehlertoleranten Suche, bei der englische Submorph-Regeln zum Einsatz kommen.

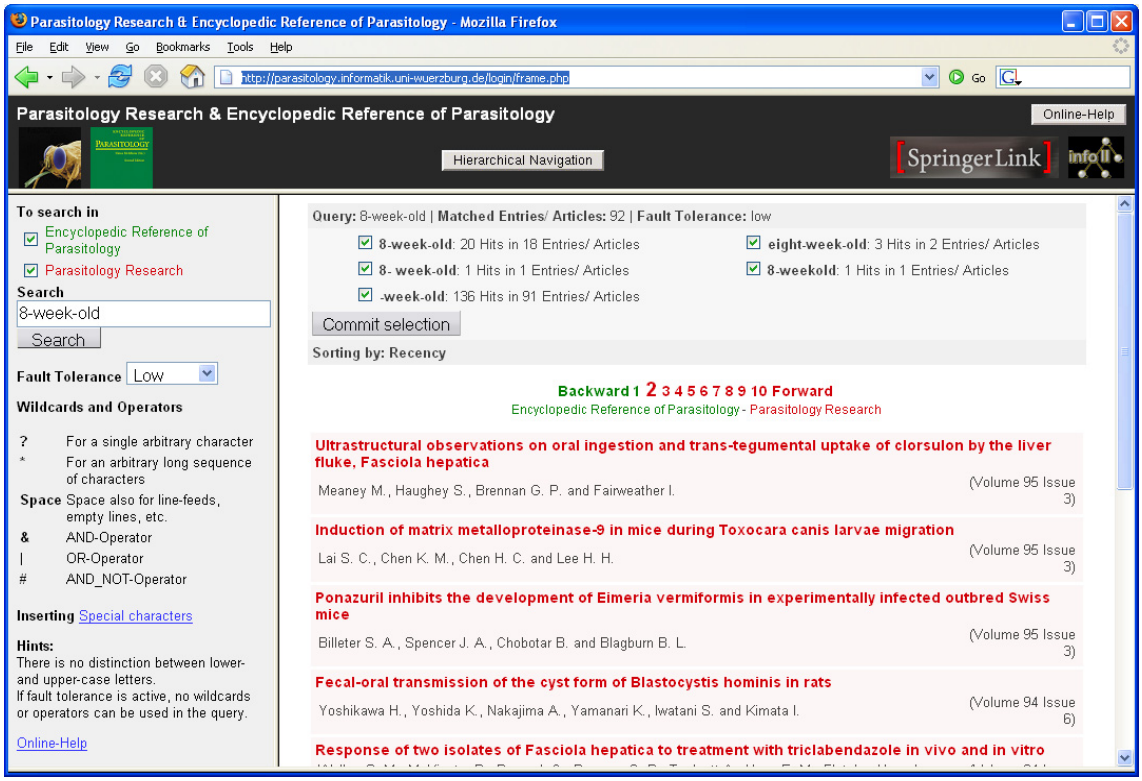


Abb. 9.12. Fehlertolerante Volltextsuche auf englisch-sprachigem Werk

Zur Bildung der englischen Submorph-Regeln werden die Ergebnisse des Artikels *Hou tu pranownse Inglish* von ROSENFELDER [wwwRos00] verwendet. Um zu ermitteln, inwiefern die neue Regelmenge auf dem englischen Textkorpus erfolgreicher arbeiten kann, wird erneut das Verfahren zur Performanz-Messung verwendet, welches bereits in Abschnitt 5.6 (Seite 71) vorgestellt wurde.

Mit diesem Verfahren werden aus den über 75.000 Wortklassen des Textkorpus über 36.000 Wortklassen extrahiert. Die so ausgewählten Worte werden dann mit den drei bekannten Stufen der Fehlertoleranz gesucht und es wird gezählt, wieviele Morphs generiert werden können, die auch wieder Bestandteil des engli-

schen Werktextes sind. Die zahlenmäßige Entwicklung der „gültigen Ziel-Morphs“ (vgl. ebenfalls Abschn. 5.6) über die drei Toleranzstufen zeigt Abbildung 9.13 – jeweils für die deutsche und die englische Submorph-Regelmenge.

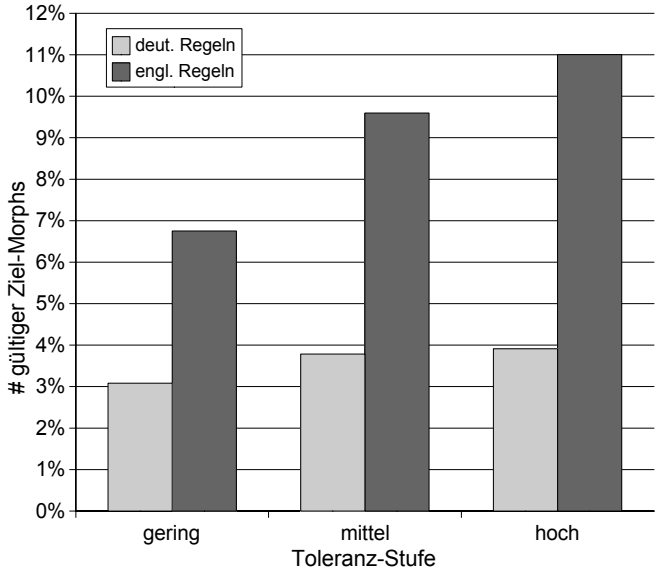


Abb. 9.13. Verbesserte WPM-Performanz auf englischen Texten durch angepasste englische Submorph-Regeln

Um einen Eindruck von den bei diesem Experiment gefundenen gültigen Ziel-Morphs zu vermitteln, werden in Beispiel 9-1 einige Suchmuster mit ihren jeweiligen Morphs aufgeführt. In den Klammern hinter den Worten werden wieder die Auftretenshäufigkeiten im Textkorpus von *Parasitology* angegeben.

Beispiel 9-1. Englische Suchmuster Treffer-Morphs aus *Parasitology*

1-electron (1)	one-electron (1)
8-week-old (4)	eight-week-old (1)
absorbence (1)	absorbency (2), absorbance (80)
acknowledgements (359)	acknowledgments (4) acknowlegements (1)
antagonized (5)	antagonised (2)
anthelminthic (12)	anthelmintic (394)
autoimmune (16)	autoimune (1)
analogous (31)	analogues (41)
behaviour (305)	behavior (181)
biotinilated (31)	biotinylated (65)
braziliensis (65)	brasiliensis (115)
characterized (364)	characterised (81)
cholodkowsky (2)	cholodkawsky (1)
favorable (29)	favourable (29)
goudenough (3)	goodenough (4)
hematophagous (12)	hematophageous (1)
lefkovits (1)	lefkowitz (2)

signaling (46)	signalling (28)
spraque-dawley (1)	sprague-dawley (8)
synergistic (74)	synergestic (1)
tuebingen (15)	tubingen (56)
wetherall (1)	whetherall (1)
two-dimensional (21)	2-dimensional (1)
unacceptable (1)	unacceptably (2)
unlabelled (1)	unlabeled (15)
visualize (100)	visualise (17)

Es soll aber auch nicht verschwiegen werden, dass auch durch die neuen englischen Submorph-Regeln teilweise Morphs gebildet werden, die vermutlich eher unerwünscht wären, wenn ein Anwender das zugehörige Suchmuster eingegeben hätte. Beispiel 9-2 zeigt solche Negativ-Beispiele.

Beispiel 9-2. Vermutlich unerwünschte englische Treffer-Morphs

productive (247)	protective (462)
protozool (164)	protocol (164)
thickness (87)	sickness (93)
visibility (3)	feasibility (17)
weighting (17)	waiting (11)
wetherall (1)	federal (87)

Zu jedem einzelnen dieser unerwünschten Morphs können Beispiele gefunden werden, in denen die an der Bildung beteiligten Submorph-Regeln hilfreich und daher in der Regelmenge zu belassen sind.

9.4 Springer Lexikon Medizin

Ein weiterer hier vorgestellter Anwendungsfall ist das *Springer Lexikon Medizin* [Reut04] von Dr. Peter Reuter, welches im März 2004 als Druck-Erstauflage veröffentlicht wurde. Das Werk umfasst auf mehr als 2.300 Seiten Erklärungen zu rund 80.000 medizinischen Stichworten, über 2.800 Abbildungen und zu zahlreichen Themen ausführliche mehrseitige Essays. Auch für dieses Werk wurde im Anschluss an die Veröffentlichung des Buches eine digitale Version erstellt. Das digitale Nachschlagewerk

Springer Lexikon Medizin – Die DVD erschien im Herbst 2004 als Multimedia DVD für Windows-PCs [Reut04b].

Tab. 9.9. Merkmale des Anwendungsfalles „*Reuter*“

Merkmal	*Wert*
Domäne	Medizin
Primärsprache	Deutsch
Distributionsmedium	DVD-ROM
Umfang HTML-Text[1)]	106,7 MB
Umfang bereinigter Rohtext[2)]	18,3 MB
Anzahl Dateien[2)]	18.700
Umfang aller Volltextsuche-Indizes[3)]	212,7 MB

[1)] bzgl. aller Dateien des Werkes [2)] nur bzgl. Dateien der Volltextsuche [3)] inkl. Kontexter-Dateien

In Abbildung 9.14 ist ein Eintrag aus dem Lexikon zu sehen, der durch die Eingabe des Eintragstitel-Präfixes in die Schnellauswahl geöffnet wurde.

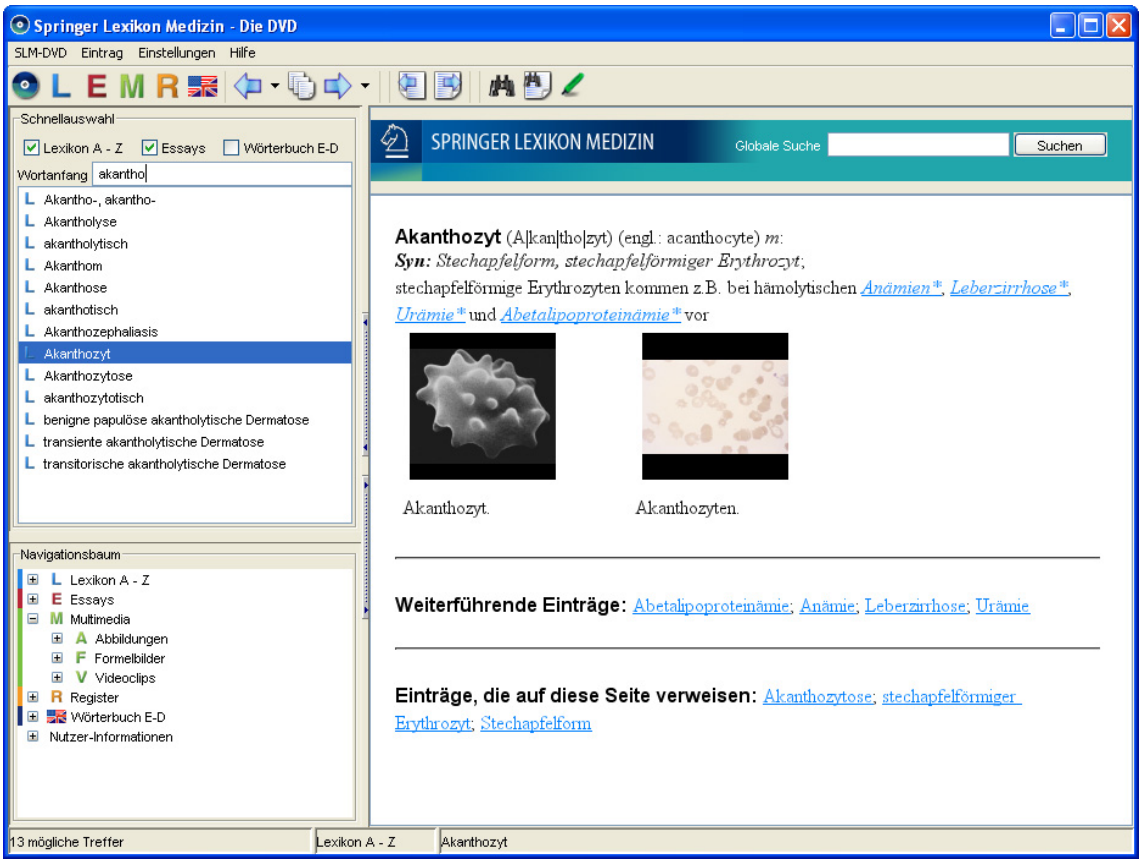


Abb. 9.14. Bildschirmfoto des *Springer Lexikon Medizin*

Auch dieses Produkt enthält die bereits bekannte, WPM-basierte, fehlertolerante Volltextsuche. Allerdings wird im weiteren Verlauf dieses Abschnitts auf eine andere Besonderheit des Werkes eingegangen: Wie bereits in Kapitel 7 beschrieben, konnten mit Hilfe von Wortlisten aus derselben Domäne Verbesserungen der Fehlerrate am Textkorpus des *Reuter*-Werkes durchgeführt werden. Zu dieser Thematik folgen nun einige detailliertere Beschreibungen und Auswertungen der zwei mit dem Werkautor durchgeführten Korrekturrunden.

9.4.1 Erste Korrekturrunde: Edit-Distanz

Für die erste Korrekturrunde wird aus den Wortklassen der Texte von Roche [Roch03], Pschyrembel [Psch02] und Duden [Dud96] die Vereinigungsmenge mit Hilfe des Programms `MainWordListCmd` gebildet. Wie stark die in dem so gebildeten Wörterbuch vorkommenden Worte den ca. 87.000 Worte umfassenden Wortschatz des *Reuter*-Werkes überdecken zeigt Abbildung 9.15.

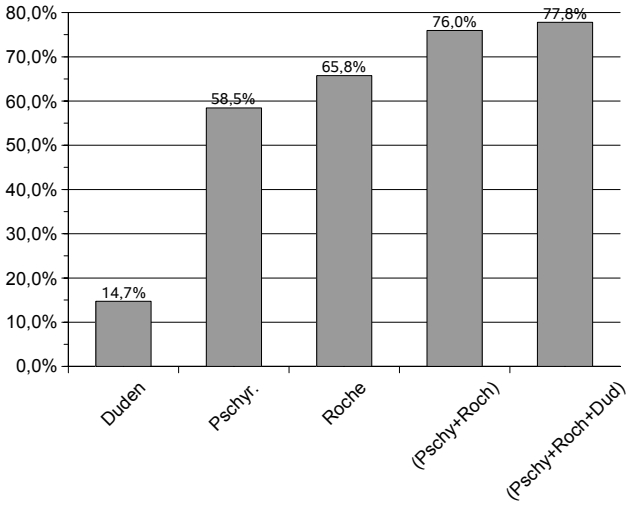


Abb. 9.15. Schnittmengen der Wortklassen des *Reuter*-Werkes mit den (teilweise addierten) Wortklassen des Werkes [...]

Die verbleibenden fraglichen Worte mussten jedoch vor einer manuellen Durchsicht weiter reduziert werden. Für die erste der beiden durchgeführten Korrekturrunden geschah dies durch die folgenden drei Maßnahmen:

1.) Wortklassen mit 3 oder mehr Instanzen werden als korrekt angenommen

2.) Wortklassen, die durch Entfernung von 1-3 Buchstaben am Wortende identisch mit einem Wort aus dem Wörterbuch sind (in dem ebenfalls bis zu 3 Zeichen am Ende entfernt sein dürfen) werden als korrekt angesehen. Diese Maßnahme soll ein einfaches Stemming simulieren.

3.) Nur Worte aus dem *Reuter*-Werk, zu denen mit einer Edit-Distanz ≤ 3 ein Korrekturvorschlag aus dem Wörterbuch gefunden werden kann, werden als „potenziell falsch“ in die Korrekturliste aufgenommen.

Durch dieses Vorgehen konnte dem Autor eine relativ kurze Liste mit ca. 5.500 fraglichen Worten zur manuellen Durchsicht präsentiert werden. Die Worte waren in drei Teillisten abhängig von der Edit-Distanz zum besten Korrekturvorschlag gruppiert, wie Tabelle 9.10 zeigt. Innerhalb der Listen wurde eine aufsteigende Sortierung gemäß dem Quotienten aus „Häufigkeit *Reuter*-Wort“ und „Häufigkeit Korrekturvorschlag“ vorgenommen. So gelangen die Worte, deren Korrektur wahrscheinlich nötiger ist, eher an den Anfang der Liste. Tabelle 9.10 zeigt die nach der manuellen Durchsicht der Liste ermittelten echten Wortfehler im Werk an, die anschließend vom Autor im Werktext korrigiert wurden.

Tab. 9.10. Ausbeute echt fehlerhafter Worte nach Durchsicht der Korrekturvorschläge

Edit-Distanz & einfaches Stemming:	***ED=1***	***ED=2***	***ED=3***	***Summe***
Fragliche Wortklassen (Listenlänge)	685	1.611	3.217	5.513
richtige Wortklassen (nach Dr. Reuter)	448	1.553	3.201	5.202
falsche Wortklassen (nach Dr. Reuter)	237	58	16	311
Prozent fehlerhafte Wortklassen	34,6%	3,6%	0,5%	5,6%

Mit dieser Technik konnten also 311 fehlerhafte Wortklassen in einem bereits mehrfach lektorierten wissenschaftlichen Werk gefunden werden. Da einige dieser Wortklassen zwei Instanzen im Textkorpus des Werkes haben, wurden insgesamt 375 falsche Worte im Werk korrigiert. Die in Kapitel 7 vorgestellten Techniken haben sich also als praxistauglich erwiesen, wobei sicherlich Raum für weitere Verbesserungen besteht. Zum jetzigen Zeitpunkt muss diese Art der Fehlerkorrektur noch als prototypisch angesehen werden – zu lang sind die zu überprüfenden Wortlisten und zu gering ist die tatsächliche Ausbeute falscher Worte, auch wenn die absolute Zahl von 375 gefundenen Fehlern die investierte Arbeit sicherlich rechtfertigt.

Tabelle 9.10 zeigt, dass die Liste der Korrekturvorschläge mit ED=3 erwartungsgemäß sehr lang ist, dafür aber im Gegenzug die prozentuale Ausbeute fehlerhafter Worte hier mit 0,5 Prozent nur äußerst niedrig liegt. Für eventuell folgende Korrekturrunden mit der Edit-Distanz wird daher empfohlen, nur die ersten beiden Listen (ED=1, ED=2) zur manuellen Durchsicht zu benutzen und dabei auf 0,5% nicht erkannte Fehler zu verzichten, bei einer gleichzeitig um über 60% reduzierten Arbeitslast für den Lektor. Werden alle drei Teillisten betrachtet, so ist im Schnitt jedes 20. Wort (5,6%) ein echter Fehler. Ignoriert man jedoch die Liste mit ED=3, so steigt innerhalb aller zu prüfenden Worte der Anteil echt falscher Worte auf 12,9% – dann ist jedes 8. Wort fehlerhaft.

Die in Beispiel 9-3 gezeigten Worte zeigen einige Fehler, die in der ersten Korrekturrunde gefunden werden konnten.

Beispiel 9-3. In Runde 1 gefundene Wortfehler im *Reuter*-Textkorpus

falsch	*richtig*	*ED*
abdomialen (1)	abdominalen (46)	1
abolutes (2)	absolutes (21)	1
achillessehenreflex (2)	achillessehnenreflex (15)	1
adams-stoke-anfall (1)	adams-stokes-anfall (10)	1
adenokarinom (1)	denokarzinom (144)	1
angiontensin (1)	angiotensin (330)	1
ebstein-barr-virus (1)	epstein-barr-virus (91)	1
fehlbilung (1)	fehlbildung (492)	1
hämogloginkonzentration (1)	hämoglobinkonzentration (35)	1
lympfgefäßen (2)	lymphgefäßen (112)	1
systemerkankungen (1)	systemerkrankungen(68)	1
alkolkonsum (2)	alkoholkonsum (54)	2
anitphlogistika (1)	antiphlogistika (246)	2
beckendlage (1)	beckenendlage(85)	2
symphatikus (2)	sympathikus (240)	2
allgemeinempfindens (1)	allgemeinbefindens (39)	3

Der folgende Abschnitt behandelt die zweite Korrekturrunde, bei der gleichzeitig einige Randparameter geändert werden mussten, um die erforderliche manuelle Arbeit für den Lektor möglichst weiter zu reduzieren.

9.4.2 Zweite Korrekturrunde: WPM

In der zweiten Korrekturrunde mit dem Autor des *Springer Lexikon Medizin* wurden die Listen der zu kontrollierenden Worte durch folgende Maßnahmen weiter verkürzt.

1.) Benutzung der so genannten „*fuzzy keys*“, wodurch z.B. das Wort „fotografie“ auch als korrekt eingestuft wird, wenn im Wörterbuch nur das Wort „photographie“ enthalten ist (vgl. Abschn. 7.8, S. 143).

2.) Benutzung des *Porter-Stemmers*, sodass z.B. das Wort „muskelschwächend“ als korrekt erkannt wird, wenn im Wörterbuch nur das Wort „muskelschwach“ enthalten ist (vgl. Abschn. 7.5, S. 128).

3.) Wieder sollen nur Worte als „potenziell falsch“ aufgelistet werden, zu denen auch ein Korrekturvorschlag im Wörterbuch gefunden wird. Allerdings sollen die Korrekturvorschläge diesmal nicht mit der Edit-Distanz sondern mit dem *WPM-Verfahren* gefunden werden (vgl. Kap 5, S. 45).

Tabelle 9.11 gibt eine Übersicht über die in der 2. Korrekturrunde gefundenen Fehler. Das in dieser Runde angewandte Verfahren ermittelte auch zahlreiche Wortklassen, die bereits in Runde 1 aufgetaucht sind und dort nach Durchsicht der Listen bereits als falsch oder richtig klassifiziert wurden. Diese Doubletten werden vor der manuellen Durchsicht in Runde 2 temporär entfernt und anschließend für die statistischen Betrachtungen wieder eingefügt.

Tab. 9.11. Ausbeute echt fehlerhafter Worte nach Durchsicht der Korrekturvorschläge auf den WPM-Toleranzstufen gering, mittel und hoch.

WPM, fuzzy keys & Porter-Stemmer:	***gering***	***mittel***	***hoch***	***Summe***
Fragliche Wortklassen (Anfangslistenlänge)	898	702	182	1.782
davon richtige Worte (aus Runde 1)	206	219	78	503
davon fehlerhafte Worte (aus Runde 1)	82	139	3	224
Summe Worte schon von Dr. Reuter gesehen	288	358	81	727
Rest Worte in Runde 2 zu überprüfen	610	344	101	1.055
Fehler in den verbleibenden Wortklassen	45	46	0	91
Summe gefundener Fehler insgesamt	127	185	3	315
Prozent fehlerhafte Wortklassen	14,1%	26,4%	1,6%	17,7%

Durch Verwendung der in Runde 2 beschriebenen Techniken kann also die Ausbeute der gefundenen Fehler auf 17,7% gesteigert werden. Insgesamt konnten 91 weitere fehlerhafte Wortklassen gefunden werden, die in der ersten Runde nicht ermittelt werden konnten.

Auch in Runde 2 fällt auf, dass die Teilliste mit dem größten Abstand (hier: Toleranzstufe=hoch) die geringste Ausbeute an falschen Worten liefert. Würde aufgrund dieser Erkenntnis diese Teilliste nicht bearbeitet werden, so würde sich die Ausbeute auf knapp 20% steigern (jedes 5. Wort ist ein falsches Wort).

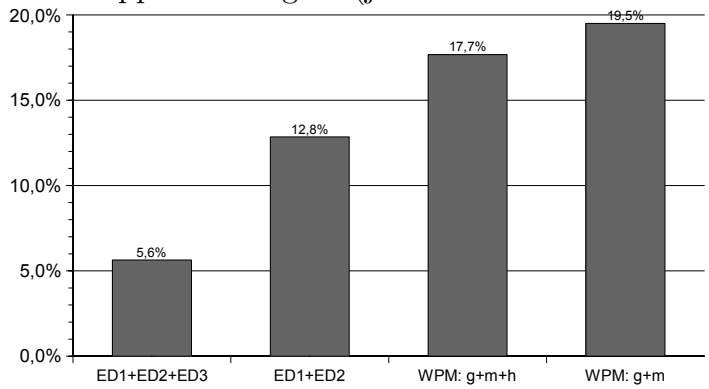


Abb. 9.16. Zunehmende Ausbeute falscher Worte in Runde 1 (Edit-Distanz) und Runde 2 (Weighted Pattern Morphing)

Aus den gemachten Beobachtungen ergibt sich die Schlussfolgerung, dass zukünftige Korrekturrunden – auch bei anderen Werken – mit den in Runde 2 angewendeten Techniken durchgeführt werden sollten. Allerdings kann auf die Teilliste, deren Korrekturvorschläge mit „hoher Fehlertoleranz“ gebildet wurden, verzichtet werden.

9.4.3 Verbesserungen am Dekomponierer

Der in Abschnitt 7.5 (Seite 128) vorgestellte Algorithmus zur Trennung deutscher Komposita soll Worte erkennen, die nicht im Wörterbuch stehen, dabei jedoch nur aus einer Verkettung von gültigen Wörterbuch-Worten bestehen – und daher als richtig angenommen werden müssen. Der Dekomponierer soll also für kürzere Listen von fraglichen Worten sorgen und damit den Aufwand für die manuelle Durchsicht dieser Listen verkleinern.

Der Dekomponierer wird erst in zukünftigen Korrekturrunden eingesetzt – eventuell auf anderen Werktexten. Trotzdem sind die bisher gemachten experimentellen Ergebnisse für diese Arbeit interessant und berichtenswert.

Mit den in den vorhergehenden Abschnitten beschriebenen Korrekturrunden existieren umfangreiche Listen von in einem konkreten Werk vorkommenden falschen und richtigen Worten. Es wird nun untersucht, welchen Prozentsatz von Worten aus diesen Listen der Dekomponierer in ihre Teilworte zerlegen kann und welche nicht.

Dabei sind vor allem zwei Fragestellungen interessant: Wie stark lassen sich die Listen fraglicher Worte verkürzen, um die manuelle Arbeit für den Lektor zu verringern (*gewünschte Zerlegungen*). Und wieviele in Runde 1 erkannte echte Fehlerworte lassen sich fälschlicherweise ebenfalls zerlegen und tauchen aus diesem Grund gar nicht in den Korrekturlisten auf (*falsche Zerlegung*)? Anzustreben ist dabei, dass es prozentual möglichst viele gewünschte und möglichst wenig falsche Zerlegungen gibt.

Als Basis für die Experimentreihe diente der existierende „multiple“ Dekomponierer, der aus allen möglichen Zerlegungen die „beste“ auswählt. Die folgenden Parameter des Dekomponierers wurden daraufhin untersucht, wie sie den prozentualen Abstand zwischen der Anzahl richtiger und falscher Zerlegungen möglichst vergrößern können.

- **Multi-Basis**: Der unveränderte Basis-Algorithmus
- **Dict Q=60**: In das Wörterbuch werden nur Worte mit einer minimalen „angehobenen Qualität“ Q aufgenommen. So soll verhindert werden, dass Worte deren Korrektheit fraglich ist, als Zerlegungsfragmente verwendet werden.

Der angehobene Qualitätswert berechnet sich aus Worthäufigkeit, Wortlänge und Anzahl der Verwendungen in unterschiedlichen Werken. Eine Experimentreihe, die nur diesen einen Parameter verändert, erzielte die besten Ergebnisse (d.h. möglichst viele gewünschte und dabei wenig falsche Zerlegungen), wenn nur Worte mit Q=60 in das Wörterbuch aufgenommen wurden.

- **⌀|NF|>4,5**: Wie in der Beschreibung des Dekomponierers gezeigt wurde, sind falsche Zerlegungen oft nur möglich, indem viele kurze Fragmente aneinander gereiht werden (z.B. `|arterie|[n]|ann|om|ali|$e$`). Nun wird zu jeder möglichen Dekomposition die durchschnittliche Länge der aus dem Wörterbuch stammenden Fragmente ermittelt. Eine Experimentreihe, die nur diesen einen Parameter verändert, erzielte die besten Ergebnisse, wenn die Wörterbuch-Fragmente einer Zerlegung im Mittel länger als 4,5 Zeichen sind.
- **CorpusRatio=0,3**: Wie ebenfalls in der Beschreibung des Dekomponierers gezeigt wurde, sind falsche Zerlegungen oft nur möglich, indem Fragmente aus dem Wörterbuch verwendet werden, die eigentlich der Textdomäne fremd sind. So ist z.B. die Zerlegung von `fernsehersatzteil` als `|fernseher|satzteil|` in einem Linguistik-Lehrbuch evtl. sinnvoll, während sie in einem Text über Elektrotechnik besser `|fernseh|ersatzteil|` lauten sollte. In einer Experimentreihe zu diesem Parameter wurden die besten Ergebnisse erzielt, wenn 30%-40% der Fragmente auch im Korpus des Werkes vorkommen.
- **#(|NF|=4)>1**: Werden die so genannten *normalen Fragmente* aus dem Wörterbuch gezählt, welche im Gegensatz zu Präfix-, Suffix- oder Fugen-Fragmenten stehen, dann ergab eine Messreihe über Anzahl und Länge dieser normalen Fragmente, dass es vorteilhaft ist, nur Zerlegungen zu akzeptieren, die wenigstens zwei Wörterbuch-Fragmente mit einer Länge von vier Zeichen haben.
- **DE-p LAT-p/s**: Die in den Algorithmus einkompilierten Präfix- und Suffixlisten machen in vielen Fällen korrekte Zerlegungen erst möglich, die nur mit Wörterbuch-Fragmenten alleine nicht erzeugbar wären (z.B. `verstimmbarkeit=^ver^|stimme|$bar$$keit$`). Allerdings sind Fragmente dieser Listen auch möglicherweise an falschen Zerlegungen beteiligt (z.B. `angiödems=^an^|gi|ödem|$s$`, `auseichend=^aus^|eichen|$d$`). Daher wurde in einem Experiment ermittelt, ob Zufügen oder Weglassen von deutschen oder lateinischen Präfixen oder Suffixen bessere Ergebnisse bei der Zerlegung liefern. Die besten Ergebnisse wurden bei den Worten aus der *Reuter*-Korrekturrunde 1 erzielt, wenn für das Deutsche nur die Präfixlisten und für das Lateinische Präfix- und Suffixlisten verwendet werden.

Abbildung 9.17 zeigt, wie die jeweils optimierten Parameter den prozentualen Abstand zwischen falsch und richtig getrennten Worten optimieren können. Als

Grundlage für diese Messungen wurden die Wortlisten aus der Korrekturrunde 1 mit ED=1, ED=2 und ED=3 verwendet.

Der Multi-Splitter hat in der Basisversion ein kontraproduktives Verhalten: Er kann zwar 93,3% der fraglichen Worte splitten – und damit als „vermutlich korrekt“ klassifizieren – dabei verschwinden aber auch 97,7% der fehlerhaften Worte aus den Listen. Man hätte also ein besseres Ergebnis, wenn man den Splitter gar nicht verwenden würde, sondern stattdessen einfach zufällige 93,3% der fraglichen Worte löschen würde. Am besten zeigt sich der Dekomponierer, wenn man die Präfix- und Suffixlisten wie beschrieben auswählt. Dann erhöht sich der Abstand zwischen falschen und richtigen Dekompositionen auf 23,7%.

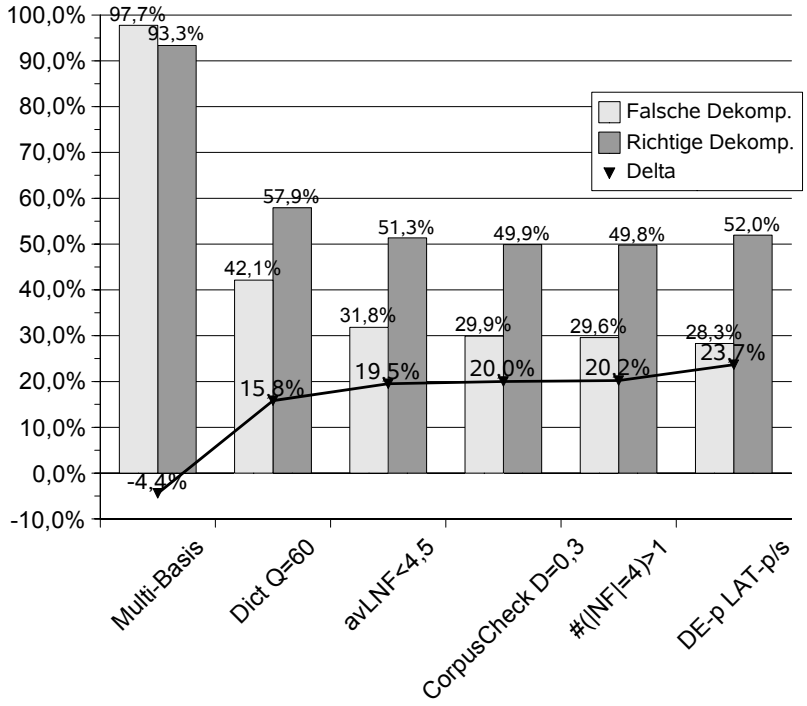


Abb. 9.17. Verbesserung des Multi-Dekomponierers mittels Veränderung der Filter-Parameter für erlaubte Zerlegungen (*Runde 1, ED=1,2,3*)

Eine der Schlussfolgerungen aus Korrekturrunde 1 war jedoch, die Liste mit Edit-Distanz=3 nicht mehr zu verwenden, da hier die Ausbeute der echt falschen Worte im Vergleich mit zur damit verbundenen Mehrarbeit zu gering war. Interessant ist das Ergebnis der in diesem Abschnitt untersuchten Parameter-Verbesserungen des Dekomponierers, wenn man nun also nur die Wortlisten mit Edit-Distanz=1 und Edit-Distanz=2 berücksichtigt. Dann ergibt sich plötzlich ein ganz anderes Bild (vgl. Abb. 9.18), und es liegt der Schluss nahe, dass der Multi-Splitter in seiner Basisversion unverändert belassen werden sollte. Denn hier hat dieser mit einem Prozent-Abstand von 62,4% die besten Ergebnisse.

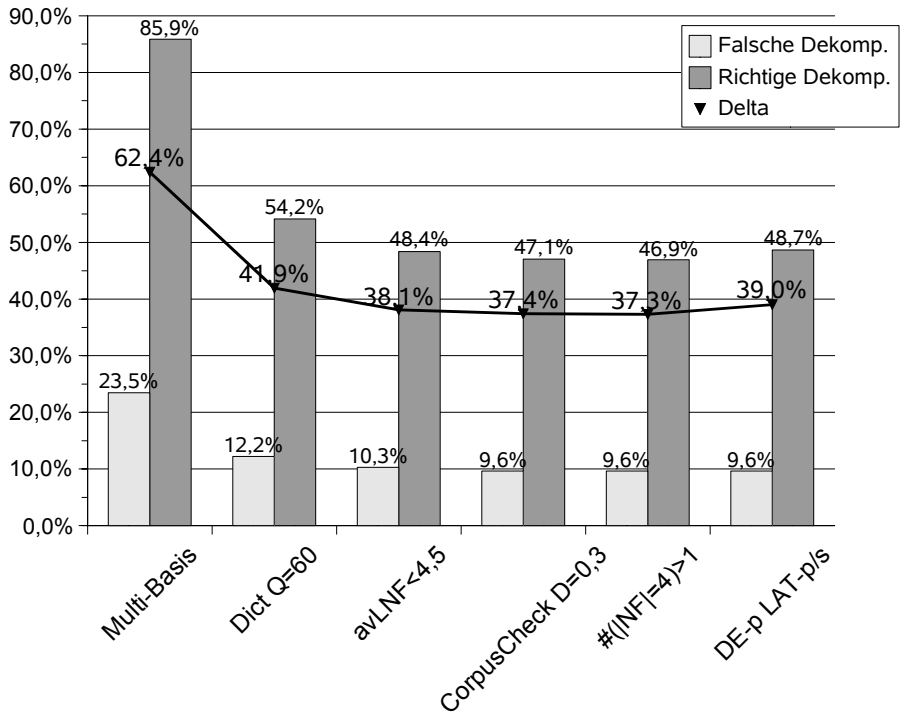


Abb. 9.18. Verbesserung des Multi-Dekomponierers mittels Veränderung der Filter-Parameter für erlaubte Zerlegungen (*Runde 1, ED=1,2*)

Diese recht widersprüchlichen Ergebnisse bezüglich Leistungsfähigkeit des Dekomponierers machen deutlich, dass hier je nach Einsatzzweck sehr vorsichtig zwischen Arbeitserleichterung für den Lektor und dabei übersehenen fehlerhaften Wortklassen abgewogen werden muss. Trotzdem scheint der Ansatz des Dekomponierers viel versprechend, und es ist Bestandteil des Ausblicks dieser Arbeit, hier konsistentere Qualität zu erreichen.

9.5 Fries-Chronik

Vorlage für den Anwendungsfall „*Fries-Chronik*“ [Frie04] ist die ursprünglich von Magister Lorenz Fries (1489-1550) verfasste, und 1574-1582 dann als Prachthandschrift für den Würzburger Fürstbischof Julius Echter (1545-1617) angefertigte Chronik über die Geschichte der Würzburger Fürstbischöfe [Frie1582]. Bei diesem Anwen-

dungsfall ist nicht nur die Umsetzung der Multimedia-Anteile der DVD für Windows PCs interessant (siehe Abschnitt 8.4 über die Generierung des virtuellen 3D-Buches), sondern auch die enthaltenen mittelalterlichen Textanteile stellen eine besondere Herausforderung dar.

Tab. 9.12. Merkmale des Anwendungsfalles „*Fries-Chronik*“

Merkmal	***Wert***
Domäne	Geschichte
Primärsprachen	Frühneuhochdeutsch und Deutsch
Distributionsmedium	DVD-ROM
Umfang HTML-Text[1)]	3,9 MB
Umfang bereinigter Rohtext[2)]	1,6 MB
Anzahl Dateien[2)]	835
Umfang aller Volltextsuche-Indizes[3)]	16,4 MB

[1)] bzgl. aller Dateien des Werkes [2)] nur bzgl. Dateien der Volltextsuche [3)] inkl. Kontexter-Dateien

Es folgt ein Bildschirmfoto des Anwendungsfalles *Fries-Chronik*. Für Abbildungen zum 3D-Buch sei auf den Abschnitt 8.4 dieser Arbeit verwiesen.

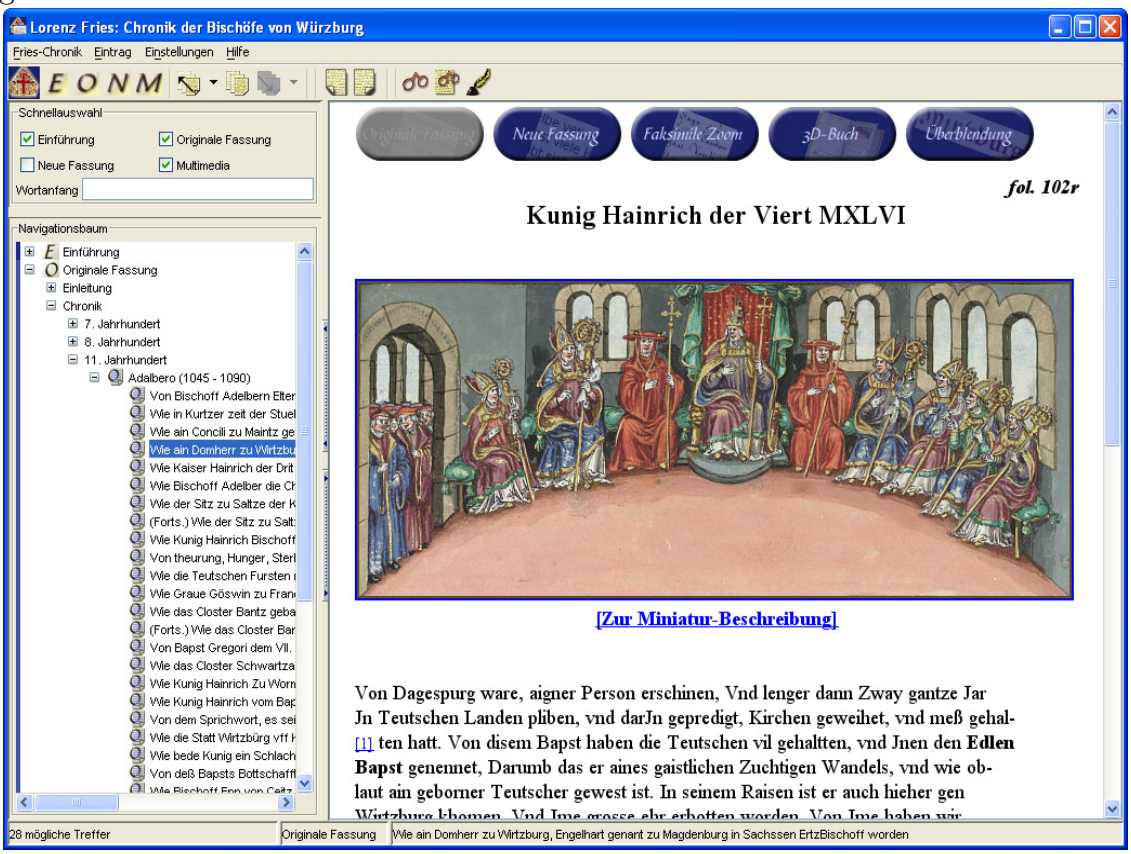


Abb. 9.19. Buchstabengetreue Transkription der Handschriften-Texte der Fries-Chronik

Die mittelalterlichen Texte der Chronik sind in *frühneuhochdeutscher* Sprache abgefasst und werden für die DVD buchstabengetreu von der Handschrift in eine digitale Textform überführt. Das Frühneuhochdeutsche ist seit ca. 1350 der Nachfolger des Mittelhochdeutschen und der Vorläufer des ab ca. 1600/1700 bis heute verwendeten Neuhochdeutschen, welches synonym auch oft als „Hochdeutsch“ bezeichnet wird.

Da die Multimedia-DVD nicht nur für Geschichtswissenschaftler konzipiert wurde, sondern auch für Schüler, Studenten und interessierte Laien, kann nicht davon ausgegangen werden, dass diese Zielgruppe die originalen Texte in einer heute teilweise nur schwer verständlichen Sprache ohne Probleme lesen kann. Daher wird eine „Übersetzung“ der frühneuhochdeutschen Texte in heutiges Hochdeutsch ebenfalls in die DVD integriert.

> „**Wie Kaiser Hainrich der Drit gestorben, vnd sein sun, Kunig Hainrich, In der Regirung bliben ist.** Jn dem 1056. jare, am funften tag des Weinmonds, starb Kaiser Hainrich der Dritt obgenant, Zu Wurmbs, vnd warde sein Toder Leichnam gen Speyr gefurt, Vnd daselbst begraben. Bey der Leich waren gegenwerttig Jn aigner Persone, Bapst Victor der Ander, der Patriarch von Agley, Bischoff Gebhart von Regenspurg deß Kaysers Vatters bruder, vnd sonst Vil Fursten.“
>
> [Frie04], Folio 102v, *Originale Fassung*

> „**Vom Tod Kaiser Heinrichs III. und dessen Nachfolger Heinrich IV.** Am 5. Oktober 1056 starb zu Worms Kaiser Heinrich III. Sein Leichnam wurde zu Speyer in Gegenwart des Papstes Viktor II., des Patriarchen von Aquileia und des Bischofs Gebhard von Regensburg sowie mehrerer weltlicher Fürsten beerdigt.“
>
> [Frie04], Folio 102v, *Neue Fassung*

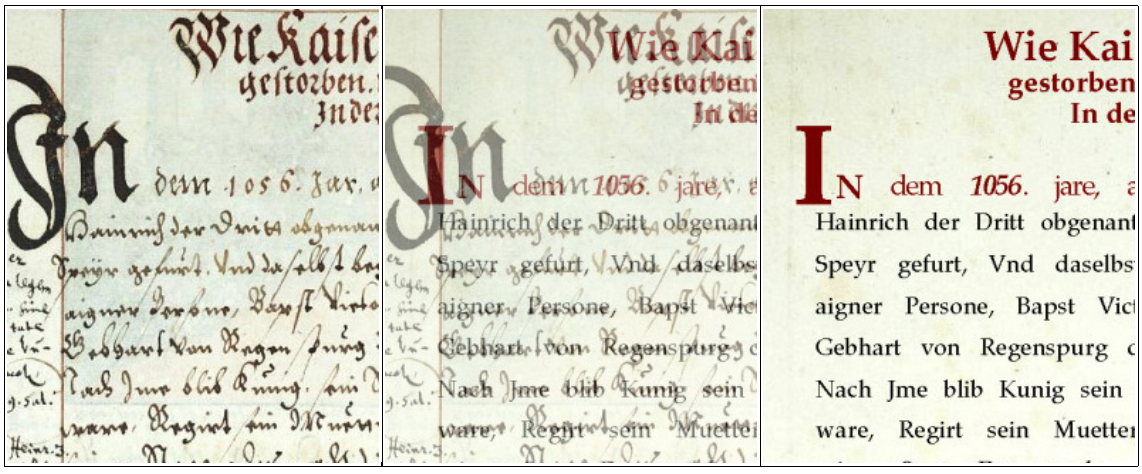


Abb. 9.20. Weiche Überblendung der Original-Handschrift mit der Transkription

Wie Abbildung 9.20 illustriert, werden die Texte nicht nur als reine Lesetexte, sondern außerdem auch als Pixel-Grafiken integriert. Da sich die Positionierung der buchstabengetreuen Transkription auf der Seitenfläche dabei möglichst an der Handschrift orientiert, kann mit dem entsprechenden Programmmodul eine durch den Benutzer steuerbare, weiche Überblendung dieser beiden Textansichten erfolgen. So kann der Benutzer dabei unterstützt werden, das Lesen der mittelalterlichen Handschrift zu erlernen und nur im Bedarfsfall in eine für ihn besser lesbare Antiqua-Schriftart überblenden.

Neben den ins Neuhochdeutsch übertragenen Texten sind auch die frühneuhochdeutschen Texte mit der Volltextsuche durchsuchbar. Hierbei treten zwei der bereits beschriebenen Gründe, die für eine fehlertolerante Suche in einem digitalen Text sprechen, noch einmal besonders hervor.

1.) **Inkonsistenzen und Fehler im Werk**: Wie das oben aufgeführte Zitat aus [Frie04] (Folio 102v, *Originale Fassung*) exemplarisch belegt, gab es im Mittelalter noch keine einheitliche Rechtschreibung (vgl. 3. Zeile des Zitats: *Kaiser* mit der letzten Zeile: *Kaysers*). Es wird deutlich, dass selbst ein und derselbe Autor mitunter im Verlauf eines Textes mehrfach die Schreibweise eines Begriffs wechselt – selbst innerhalb eines Absatzes.

2.) **Unsicherheit des Benutzers bzgl. korrekter Schreibweise**: Um die Benutzer trotz der für sie eher ungewohnten Sprache bei der Suche nach Begriffen im Werk zu unterstützen, wird auch versucht, von der heutigen Schreibweise zu den möglichen frühneuhochdeutschen Schreibweisen zu gelangen. Dies kann jedoch aufgrund des beschriebenen kompletten Fehlens einheitlicher Rechtschreibregeln nicht immer gelingen.

Als Quelle der für diesen Anwendungsfall neu zu erstellenden Submorph-Regeln zur fehlertoleranten Suche in frühneuhochdeutschen Texten wurden aus dem Buch „*Kleines frühneuhochdeutsches Wörterbuch*“ [Bauf96] von Christa Baufeld die einleitenden Kapitel verwendet ([ebd.] Seite XXff).

Im Frühneuhochdeutschen werden die in Tabelle 9.13 gruppierten Einzelbuchstaben üblicherweise nicht in ihrer Verwendung unterschieden. Es werden daher Submorph-Regeln eingeführt, die die Einzelbuchstaben innerhalb dieser Gruppen jeweils mit dem kleinsten Strafgewicht ineinander umwandeln.

Tab. 9.13. Synonyme Einzelbuchstaben-Verwendung im Frühneuhochdeutschen (s. [Bauf96])

b, p	c, k, z	d, t
f, konsonantisches v	i, j, y	u, v, w

Neben diesen Einzelbuchstaben sind in der oben angegebenen Quelle auch zahlreiche Buchstabenverbindungen angegeben, die im Frühneuhochdeutschen austauschbar verwendet wurden. Tabelle 9.14 zeigt die wichtigsten Gruppen.

Tab. 9.14. Synonyme Buchstabengruppen-Verwendung im Frühneuhochdeutschen (s. [Bauf96])

au, aw, ou, ow	d, t, dt	ei, ey, ai, ay
eu, ew, äw, öu, öw	g, k, gk	k, c, ck, g, gk
m, mb	pf, ppf, pph	qu, kw
s, sch	tw, qu, zw	u, uu, v, vv, b
x, cks, chs	z, cz, tz, zc, czc, tzc, ...	

Die hieraus neu generierten Submorph-Regeln werden den bestehenden normalen deutschen Regeln hinzugefügt, welche bereits Regeln z.B. für die Verdoppelung von Buchstaben (m⇔mm) enthalten oder für das Einfügen oder Löschen von Dehnungs-h (e⇔eh). Mit den so erweiterten deutschen Submorph-Regeln können anschließend Wortvarianten gefunden werden, wie sie in Beispiel 9-4 aufgeführt sind.

Beispiel 9-4. Wortvarianten in der *Fries-Chronik* (Originale Fassung)

papst (12) pabst (12), babst (53)
würzburg (7) würzburgk (1), wurtzburg (194), wirtzburg (791), wirzburg (1)
kaiser (368) khaiser (2), kayser (48), keiser (5)
heinrich (35) hainrich (314), heinrici (3), heinrico (4), heinricu (13), heinricv (3)
deutscher (1) teütscher (1), teutscher (13)
domherr (66) dombher (10), domher (1)
bayern (7) bayrn (28)

So bietet das WPM-Verfahren nicht nur die Möglichkeit, aktuelle wissenschaftliche Texte zu erschließen, sondern speziell auch für nicht mehr gebräuchliche Sprachen und Dialekte können Überführungsregeln gefunden werden, die es heutigen Benutzern möglich machen, diese Texte als Informationsquelle besser nutzen zu können.

Kapitel 10

Zusammenfassung und Ausblick

Im nun folgenden abschließenden Kapitel werden zunächst die Ergebnisse dieser Arbeit zusammengefasst und die vorgestellten Verfahren noch einmal kurz erläutert. Im Anschluss daran findet sich eine Zusammenstellung möglicher weiterführender Ideen, die auf den Inhalt dieser Arbeit aufbauen.

10.1 Ergebnisse der Arbeit

In der vorliegenden Arbeit wurde das Konzept und die praktische Umsetzung einer fehlertoleranten Volltextsuche vorgestellt, welche die unscharfe Recherche nach Suchmustern in umfangreichen, digitalen, enzyklopädischen Werken ermöglichen. Das dabei zur Anwendung kommende neue Verfahren, welches durch Gewichte gesteuert das ursprüngliche Benutzer-Suchmuster in seiner Gestalt verändert (Weighted Pattern Morphing, WPM) und anschließend mit einer nachgeschalteten exakten Volltextsuche sucht, konnte in zahlreichen kommerziellen Anwendungsfällen seine Praxistauglichkeit beweisen. Im Folgenden werden die Merkmale des WPM-Verfahrens noch einmal kurz zusammengefasst.

- **Optionalität.** Das fehlertolerante Modul ist als optionales, zuschaltbares Merkmal der Volltextsuche realisiert. Ist die Fehlertoleranz ausgeschaltet, wird die exakte Suche in ihrer Laufzeit oder ihrem Platzbedarf für den Index nicht negativ beeinflusst. Ist sie dagegen eingeschaltet, kann der Benutzer mit der Toleranzfunktion zu seinem Suchmuster synonyme, ähnlich geschriebene und ähnlich klingende Begriffe finden.
- **Gewichtete Submorph-Regeln.** Die Fehlertoleranz wird u.a. durch eine Menge so genannter *Submorph-Regeln* realisiert. In einer Submorph-Regel werden Ersetzungen von kurzen String-Fragmenten mit einem zugehörigen

Strafgewicht kombiniert, wobei Quell- und Ziel-Strings dieser Regeln dabei (fast) beliebig lang sein können. Die Menge der Submorph-Regeln, ihre Quell- und Ziel-Strings und ihre jeweiligen Gewichte sind veränderbar und damit an unterschiedliche Werke, Sprachen und Textdomänen anpassbar. So können z.B. Submorph-Regeln für werkspezifische Abkürzungen oder Synonyme definiert werden, oder die Regelmenge kann z.B. die Veränderungen der Schreibweise durch die deutsche Rechtschreibreform berücksichtigen. Durch die Gewichte der Regeln kann gesteuert werden, welche String-Ersetzungen häufiger/seltener angewendet werden sollen, da sie das Suchmuster mehr/weniger verändern. Das Paar bestehend aus String und zugehöriger Gewichtssumme heißt *Morph*.

- **Multiple Regelanwendung.** Durch rekursive Implementierung des WPM-Verfahrens werden bei Bedarf mehrere Submorph-Regeln auf das Original-Suchmuster angewendet, und es wird so ermöglicht, auch Treffer zu Suchmustern zu finden, die an mehreren Stellen korrigiert werden müssen.

- **Werkspezifität.** Durch das *Morph-Feedback* genannte Verfahren ist es möglich, die Regelgewichte an ein spezielles Werk anzupassen. Dabei haben Experimente mit mehreren tausend Suchmustern von Online-Benutzern gezeigt, dass man diese Justage auch nur mithilfe des Textkorpus selber durchführen kann und trotzdem die Performanz für die Benutzer signifikant verbessert wird. Dies ist vor allem vor dem Hintergrund interessant, dass in der Entwicklungsphase eines (fehlertoleranten) Volltextsuche-Systems üblicherweise keine wirklichen Suchmuster von Endnutzern zur Verfügung stehen, mit denen man das System testen und optimieren kann.

- **Mehrsprachigkeit.** Durch die Integration der fehlertoleranten Volltextsuche in sehr unterschiedliche Anwendungsfälle konnte gezeigt werden, dass sich das WPM-Verfahren mit vertretbarem Aufwand nicht nur an unterschiedliche Textdomänen sondern auch an verschiedene Werksprachen anpassen lässt. Bisher wurde das WPM-Verfahren auf (neu-)hochdeutschen, frühneuhochdeutschen und englischen Texten erfolgreich eingesetzt.

- **Austauschbares Backend.** Die Fehlertoleranz ist als vorgeschaltetes Modul vor eine exakte, schnelle q-Gramm-Volltextsuche realisiert, welche die Rolle des Suche-Backends einnimmt. Das WPM-Verfahren macht jedoch keine Annahmen über die Art des Suche-Backends: Als Eingabe benötigt der WPM-Algorithmus das Suchmuster des Endnutzers und als Ausgabe produziert der Algorithmus eine Menge von Strings als Varianten des Original-Suchmusters. Die Strings sind aufsteigend nach Gewichtssumme ihres zugehörigen Morphs sortiert, und die Zeichenkette des Original-Suchmusters, welche dem Start-Morph mit einer Gewichtssumme von Null entspricht, ist ebenfalls in dieser Liste von Rückgabe-Strings enthalten. Da das Rückgabe-Format des WPM-

Algorithmus lediglich eine Liste von Zeichenketten ist, über die dann eine sequenzielle Suche durchgeführt werden muss, ist das Suche-Backend austauschbar. Denkbar ist hier statt einer exakten q-Gramm Volltextsuche z.B. eine (relationale) Datenbank oder eine Suchmaschine für das WWW, welche so fehlertolerant gemacht werden könnten.

- **Vergleich mit Edit-Distanz.** Sowohl die zahlreichen in der Arbeit aufgeführten Beispiele aus realen Werktexten, die 0-Treffer-Suchmuster von Online-Benutzern und die aufgezeigten Problemfälle als auch die durchgeführten Messungen von Precision und Recall zeigen die Fähigkeit des WPM-Verfahrens, qualitativ hochwertigere fehlertolerante Ergebnisse zu liefern, als es die üblicherweise in fehlertoleranten Textsuchen verwendete Edit-Distanz vermag. Neben dem Verzicht auf eine erzwungene Symmetrie bei der Anwendung der Stringveränderungen bietet das WPM-Verfahren außerdem die Möglichkeit, über Quell- und Ziel-Strings und ihre zugehörigen Gewichte das Verfahren sehr feinkörnig an ein spezielles Werk, seine Domäne und seine Sprache anzupassen.
- **Skalierbarkeit.** Der Endnutzer kann über die grafische Oberfläche den gewünschten Grad der Fehlertoleranz von *keine* über *gering*, *mittel* und *hoch* festlegen. Über die Stufen der Fehlertoleranz werden interne Parametersätze ausgewählt, die im WPM-Algorithmus steuern, 1.) wieviele Regelanwendungen maximal bei der Erzeugung eines Morphs beteiligt sein dürfen, 2.) wieviele Strafpunkte ein Morph maximal aufsummieren darf und 3.) wieviele der Morphs mit den wenigsten Strafpunkten tatsächlich mit der nachgeschalteten exakten Volltextsuche gesucht werden.
- **Lokal und im WWW.** Die Anwendungsfälle haben gezeigt, dass das WPM-Verfahren nicht nur bei lokaler Suche im Rahmen von CD/DVD-ROM basierten Projekten erfolgreich ist, sondern auch bei Projekten, die über das WWW online zugänglich gemacht werden.
- **Filterbarkeit.** Es gibt Worte, die in ihrer Bedeutung weit auseinander liegen, jedoch Schreibweise und sogar Klang können sich dabei trotzdem stark ähneln. Oft reicht das Löschen, Einfügen oder Ersetzen eines einzelnen Buchstabens oder das Vertauschen zweier benachbarter Buchstaben, um die Schreibweise solcher Worte ineinander zu überführen. Es ist daher Bestandteil der Benutzer-Schnittstelle der in dieser Arbeit vorgestellten fehlertoleranten Suche, dem Endnutzer die Möglichkeit zu geben, Treffer-Morphs, die er nicht für relevant hält, aus der finalen Liste der Treffer herauszufiltern und damit das Ergebnis der Suche genauer an seine Fragestellung anzupassen.

Das WPM-Verfahren kann jedoch nicht nur für den Endnutzer bei der fehlertoleranten Recherche in umfangreichen, wissenschaftlichen Texten behilflich sein. Auch im redaktionellen Umfeld, vor der Veröffentlichung eines Werkes, kann das Verfahren eingesetzt werden, z.B. um Fehler und inkonsistente Schreibweisen im Textkorpus zu finden, die dann nach Überprüfung durch den Autor oder Lektor korrigiert werden. Das in dieser Arbeit vorgestellte Verfahren zum Auffinden potenziell falsch geschriebener Worte in umfangreichen Texten besteht aus folgenden Bausteinen und Ideen.

- **Werk korrigiert Werk.** Standardisierte Wörterbücher in üblichen Rechtschreibkorrektur-Systemen sind i.d.R. ungeeignet, umfangreiche wissenschaftliche Texte zu korrigieren. Daher wurde ein Konzept realisiert, mit dem Worte aus umfangreichen Texten dazu beitragen können, in einem anderen Werk derselben Domäne Tippfehler oder inkonsistente Schreibweisen zu entdecken.
- **Harvester.** Das Modul, welches die Worte aus den Werktexten herausfiltert, ist der so genannte Harvester. Dabei kann das Werk über beliebig viele Dateien verteilt sein, in den Formaten HTML, XML oder Rohtext und in den Zeichencodierungen ISO-Latin-1 oder UTF-8 vorliegen. Worttrenner können flexibel definiert werden und die gefundenen Worte werden mit einem Verweis auf ihr Quellwerk und zusammen mit der gezählten Häufigkeit in einer XML-Datei gespeichert.
- **Operationen auf XML-Wortlisten.** Es wurde ein Programm entwickelt, welches auf den oben erwähnten XML-Wortlisten Operationen zur Filterung, Sortierung, Addition, Subtraktion usw. zur Verfügung stellt. Damit ist es komfortabel und effizient möglich, potenziell falsche Worte (z.B. aufgrund ihrer Länge bzw. Anzahl und aufgrund der Nicht-Existenz im Wörterbuch) aus einem Werk zu ermitteln.
- **Porter-Stemmer.** Mit dem mittlerweile im Snowball-System aufgegangenen Porter-Stemmer wurde ein leistungsfähiger Algorithmus zur Zusammenfassung von Worten integriert, die mit hoher Wahrscheinlichkeit aus linguistischer Sicht denselben Wortstamm haben. Dies geschieht vor dem Hintergrund, dass im Wörterbuch oft nur anders gebeugte Wortformen als korrekte Worte vorhanden sind, die aber durch den Stemmer trotzdem zur Reduktion der zu überprüfenden fraglichen Worte beitragen können.
- **Dekomponierer.** Für die zu häufiger Verwendung von komponierten Worten neigende deutsche Sprache wurde ein konfigurierbarer Dekomponierer entwickelt, der sich bei der Wahl der Trennstellen auf ein Wörterbuch abstützt. Der Dekomponierer berücksichtigt die im Deutschen vorkommenden Fugenlaute und übliche Präfixe und Suffixe. Ist zu einem Wort mehr als eine Art der Auftrennung möglich, wählt der Algorithmus nach einem konfigurierbaren Bewertungsschema die plausibelste Dekomposition aus, wodurch in den

allermeisten Fällen auch eine semantisch korrekte Trennung erfolgt. Dadurch kann erreicht werden, dass Worte aus dem zu korrigierenden Text als gültige Komposition und damit als „richtig geschrieben“ klassifiziert werden, obwohl sie in ihrer Gesamtheit nicht im Wörterbuch enthalten sind.

- **WPM-Korrekturvorschläge.** Mit dem WPM-Verfahren werden zu Worten, die nicht im Wörterbuch vorkommen und deren korrekte Schreibweise daher fraglich ist, möglichst ähnliche Worte gesucht. Diese ähnlichen Worte werden in der XML-Wortliste als Korrekturvorschläge zum fraglichen Wort vermerkt und der Autor/Lektor eines Werkes kann dann überprüfen, ob das Wort tatsächlich falsch geschrieben wurde.
- **Correc d′Or Korrektur-Oberfläche.** Damit der Autor/Lektor die fraglichen Worte mit ihren Korrekturvorschlägen komfortabel überprüfen kann, wurde eine plattformunabhängige, leicht zu installierende Applikation entwickelt. Die Applikation zeigt zu jedem fraglichen Wort mindestens einen Kontext aus dem Werk an, und lässt sich wahlweise durchgängig per Tastatur oder Maus bedienen und erlaubt es so, selbst umfangreiche Listen fraglicher Worte effizient abzuarbeiten und dabei als falsch oder richtig zu markieren.

Aber nicht nur auf redaktioneller Seite muss die Qualität überwacht werden. Auch auf Seiten der Software-Entwickler ist es wichtig, speziell in großen Projekten mit zahlreichen Eingangsdateien unterschiedlicher Datenformate automatisch Kenngrößen zu ermitteln, die mögliche Fehler in Daten oder Programmen schnell sichtbar machen. Weitgehende Automatisierung komplexer Generierungsabläufe macht die Produktion eines digitalen Werkes dabei robust gegen Fehler und Inkonsistenzen, die durch manuelle Erzeugung von Zwischen- oder End-Daten entstehen würden. Auf Seiten der Software-Entwickler kamen bei der Realisierung der vorgestellten Anwendungsfälle u.a. folgende Maßnahmen zur Qualitätsverbesserung zum Einsatz.

- **Konsistenzprüfung.** Zahlreiche, überwiegend in perl entwickelte Skripte überwachen während der Generierung eine Vielzahl von Kenngrößen, die darüber Auskunft geben, ob sich das Werk in einem konsistenten Zustand befindet. Dabei wird z.B. überwacht, ob referenzierte Bilddateien fehlen oder ob dem System alle in den Werktexten enthaltenen Sonderzeichen bekannt sind.
- **Volltextsuche-Testbed.** Für die Volltextsuche als einer der komplexesten Bausteine der hier vorgestellten Anwendungsfälle existiert ein so genanntes Testbed, mit welchem per Black-Box-Testing überprüft werden kann, ob nach Änderungen an Algorithmen oder Datenbestand die Suche zu einer Reihe von Testfällen noch die erwartete Zahl von Treffern generiert.
- **JUnit-Tests und Code-Coverage.** Während der Entwicklung der Java-Programme zur Verarbeitung und Korrektur von XML-Wortlisten wurden

parallel umfangreiche JUnit-Tests entwickelt, die mittels White-Box-Testing auf Klassenebene nicht-triviale Methoden auf ihre Funktionalität überprüfen. So konnte während einiger notwendiger Refactoring-Phasen stets die Funktionalität sichergestellt werden. Durch Code-Coverage-Überprüfungen konnte ermittelt werden, welche Codebereiche noch nicht mit Testfällen abgedeckt waren und auch so genannter „toter“ Code (Code, der niemals durchlaufen werden kann) konnte so lokalisiert und entfernt werden.

- **Automatisierte 3D-Animationserstellung.** Bei der Übertragung einer mittelalterlichen Handschrift in eine Multimedia-DVD für PCs musste das Problem gelöst werden, das empfindliche Originalwerk als virtuelles 3D-Buch im PC erfahrbar zu machen. Dabei wurde eine Technik entwickelt, die Faksimile-Scans als Oberflächen-Texturen auf dem Drahtgitter eines animierten 3D-Buchmodells zu verwenden und so künstliche Animationen von Blättervorgängen zu rendern. Der zeitaufwändige Render-Vorgang wurde skriptgesteuert parallelisiert, und die komplexen Schritte zur Konvertierung der Render-Einzelbilder in Filme des benötigten Zielformates konnten soweit automatisiert werden, dass selbst eine große Anzahl von Blätterfilmen konsistent, fehlerfrei und komfortabel erzeugt werden konnte. Das Verfahren ist so flexibel, dass es auf andere Werke übertragen werden kann, die aufgrund ihres Wertes oder ihrer Seltenheit normalen Lesern als Original nicht ausgehändigt werden können. Hier kann ein virtuelles 3D-Buch in vielen Fällen einen kostengünstigen und trotzdem realitätsnahen Ersatz für ein kostbares Original bieten.

10.2 Mögliche Erweiterungen

Obwohl die im Rahmen dieser Arbeit vorgestellten Konzepte und ihre konkreten Realisierungen bereits u.a. in den in Kapitel 9 vorgestellten Anwendungsfällen erfolgreich zum Einsatz kommen, existieren für zukünftige Projekte Ideen zu möglichen Erweiterungen. Diese denkbaren Erweiterungen sollen im Folgenden kurz erläutert werden.

- **Endnutzer-Regeln.** Zur Zeit wird die Submorph-Regelmenge – vor allem aus Geschwindigkeitsgründen – fest in das Morpher-Modul der Volltextsuche einkompiliert und ist somit zur Laufzeit nicht veränderbar. Es wäre jedoch denkbar, dass der Endnutzer Einfluss auf die Regeln (Quell- und Ziel-Strings bzw. Gewichte) nimmt und so die Fehlertoleranz an seine persönlichen Bedürfnisse anpasst. Dabei wäre auch möglich, dass das System den Nutzer bei der Regeldefinition unterstützt. Zum Beispiel könnte der Benutzer jeweils Wortpaare eingeben, deren Bedeutung er als synonym ansieht, und das Sys-

tem könnte selbständig die benötigten Submorph-Regeln generieren, die die beiden Worte ineinander überführen.

- **Stemming-Morphs.** Die Vorteile einer Volltextsuche, die sich nicht auf einen Wortindex abstützt, sondern (z.B. mittels eines q-Gramm-Index) beliebige Textstellen finden kann, wurden in der Arbeit dargestellt. Jedoch kann es manchmal sinnvoll sein, Kenntnis über die in einem Text vorkommenden Worte zu haben. Speziell wenn der Benutzer Wortvarianten sucht, die erst dann zu einer umfassenden Treffermenge führen, wenn auf Suchmuster und Textworte ein Stemming angewendet wird (z.B. wenn das Suchmuster `muskelschwach` ist, im Text jedoch nur `muskelschwächend` vorhanden ist). Es wäre nun denkbar, in einem Vorverarbeitungsschritt automatisch (z.B. mit dem Porter-Stemmer) zu allen Worten aus dem Textkorpus die Wortstämme zusammen mit den zugehörigen Worten aus dem Text in einem geeigneten Index zu speichern. Ist nun in einem Suchmuster ein Wort enthalten, das einen in diesem Index existierenden Wortstamm besitzt, so könnten dynamisch Morph-Regeln angewendet werden, die dieses Wort durch im Text vorkommende Varianten mit demselben Wortstamm ersetzt.

- **Automatische Regelgenerierung.** Wie die Gewichte in einem automatisierten Prozess (Morph-Feedback) der Submorph-Regeln an einen Textkorpus angepasst werden können wurde in dieser Arbeit beschrieben. Das Festlegen von Quell- und Ziel-Strings dieser Regeln ist jedoch noch ein manueller Prozess, der Kenntnisse in der Sprache verlangt, in der das Werk abgefasst ist. Es könnte Bestandteil weitergehender Forschungsarbeit sein, auch Quell- und Ziel-Strings der Regeln automatisch zu erzeugen. Dabei könnte z.B. ausgenutzt werden, dass umfangreiche Texte (eventuell aus verschiedenen Quellen zur selben Domäne) eine gewisse Bandbreite an Synonymen und Wortvarianten beinhalten. Ist nun der mit einer geeigneten Kenngröße gemessene Abstand zwischen zwei Worten klein genug und bieten die Worte trotzdem noch ausreichend viele identische Zeichen, so können aus den nicht-identischen Zeichen Submorph-Regeln konstruiert werden. Eventuell muss dieser Prozess semi-automatisch unter Mithilfe eines (Fach)-Autors laufen, der die gefundenen, ähnlichen Wortvarianten daraufhin einordnet, ob sie wirklich gleiche Bedeutung haben.

- **Verhalten bei 0-Treffer-Mustern.** Wie die Auswertung der (anonymen) Protokolldateien der Online-Version des Werkes [AB02] gezeigt haben, liefert eine Vielzahl von Benutzer-Suchmustern bei der standardmäßig verwendeten *exakten* Suche zunächst keine Treffer im Werk – obwohl durchaus Varianten des Suchmusters im Text existiert hätten. Anstatt dem Benutzer in solchen Fällen die Suche zu terminieren und als Ergebnis seiner Anfrage „0 Treffer“ zu präsentieren, könnte man in solchen Fällen immer eine nachgeschaltete fehlertolerante Suche laufen lassen. Die Präsentation der Ergebnisse könnte

dann in der Form erfolgen, dass der Anwender darauf hingewiesen wird, dass sein Suchmuster bei exakter Suche zwar keine Treffer besitzt, aber (falls dies der Fall ist) Varianten seines Musters Treffer im Werk aufweisen, die nachfolgend angezeigt werden.

- **Hyper-Toleranz.** Selbst auf höchster Stufe der Fehlertoleranz gibt es Suchmuster, die nicht zu Treffern im Werk gemacht werden können. Dies wird u.a. an der Messung zu Precision und Recall deutlich, bei der die Edit-Distanz (unter starkem Abfall der Precision) mitunter noch Wortvarianten ermitteln kann, zu denen das WPM-Verfahren nicht mehr gelangt. Denkbar wäre, in einer zusätzlichen „Hyper"-Toleranzstufe die bisherige Taktik der festen Algorithmus-Parameter aufzuweichen. Stattdessen könnte man den Algorithmus solange Submorph-Regeln (auch beliebige Löschung, Ersetzung, Einfügung, Vertauschung von Zeichen) anwenden lassen, bis entweder mindestens ein Morph Treffer im Werk produziert, bis eine einstellbare Zeitschranke für die Suche überschritten wurde oder bis der Anwender den Vorgang abbricht. So könnte man im Erfolgsfall dem Anwender zumindest präsentieren, in welchem Abstand sich die ersten Treffer-Morphs zu seinem Suchmuster befinden, wenn die üblichen Stufen der Fehlertoleranz nicht mehr ausreichen.

- **Google-Suchmuster-Syntax.** Die Suchmuster aus den Protokollen der Online-Benutzer haben gezeigt, dass offensichtlich die vorhandene Bedienungsanleitung zur Online-Volltextsuche nicht immer gelesen wird und stattdessen Suchmuster in der Syntax bekannter WWW-Suchmaschinen eingegeben werden. Herausragend ist dabei die falsche Benutzung des Leerzeichens: Während WWW-Suchmaschinen (wie z.B. Google) dieses Zeichen als logische UND-Verknüpfung der Worte links und rechts des Leerzeichens verstehen, steht das Leerzeichen bei der q-Gramm-Suche für eine nicht-leere Folge von Whitespace (Leerzeichen, Tabulator, Zeilenumbrüche) im Text. So kann es vorkommen, dass 0-Treffer-Suchmuster durch einfache Ersetzung aller Leerzeichen zu '&'-Symbolen plötzlich Treffer produzieren, weil die Worte im Text nicht genau in der Form (Position und Reihenfolge) des Suchmusters auftauchen. Zumindest bei der Online-Version des in dieser Arbeit vorgestellten Suchsystems kann darüber nachgedacht werden, die Interpretation des Leerzeichens in Suchmustern dem Quasi-Standard anzupassen. Dabei könnte die Kern-Suche sogar unverändert gelassen werden – es könnte stattdessen in der Schicht oberhalb der Kern-Suche eine Übersetzung des Leerzeichens in den intern verwendeten UND-Operator '&' vorgenommen werden.

- **Verbesserungen am Dekomponierer.** Speziell im Deutschen ist eine automatische Korrektur von Wortfehlern mithilfe eines Wörterbuches kaum möglich, wenn nicht auch eine Funktion zur automatischen Dekomposition von zusammengesetzten Worten existiert. Die jetzige Form des Dekomponie-

rers leistet zwar bereits gute Dienste, aber Messungen an den von Dr. Reuter als „richtig“ oder „falsch“ klassifizierten Worte haben gezeigt, das hier wohl noch Raum für Verbesserungen besteht. Noch zu oft können korrekte Komposita nicht zerlegt werden, und gleichzeitig wird zu falsch geschriebenen Komposita eine mögliche (aber eben auch falsche) Zerlegung gefunden. Sicherlich wird ein automatischer Dekomponierer ohne wirkliches Textverständnis immer diese beiden Kategorien von Fehler machen – das Potenzial, das ein gut arbeitender Dekomponierer jedoch bei der Korrektur von Texten bietet, würde hier jedoch weitere Forschungsarbeit rechtfertigen.

- **Übertragbarkeit der Fehlerkorrektur.** Das Verfahren zum Auffinden von Tippfehlern und inkonsistenten Schreibweisen, welches am Textkorpus des Werkes [Reut04] praktisch erprobt wurde, muss noch darauf hin untersucht werden, inwieweit es sich auf andere Werktexte übertragen lässt. Im behandelten Anwendungsfall war hilfreich, dass die Wortlisten zu zwei ebenfalls umfangreichen Medizin-Lexika zur Korrektur des *Reuter*-Werkes herangezogen werden konnten. Nicht immer stehen solche umfangreichen Listen jedoch zur Verfügung. Bestandteil weiterer Forschungsarbeit könnte daher sein, die beschriebene Technik unter Benutzung kleinerer Wörterbücher auf anderen Werken zu erproben.
- **Übertragbarkeit 3D-Buch.** Während die automatisierten Schritte bei der Generierung der 3D-Blätteranimationen bereits jetzt problemlos für andere Werke funktionieren sollten, ist der manuell auszuführende Teil der Arbeit nicht so leicht übertragbar. Dazu gehört zum Beispiel die Tatsache, dass das 3D-Drahtgittermodell des virtuellen Buches das Seitenverhältnis der Original Fries-Chronik aufweisen muss, um die Texturen mit möglichst geringem Verschnitt darauf platzieren zu können. Für die Übertragung auf andere Werke ist hier mit Anpassungsarbeiten zu rechnen, die man jedoch auch automatisieren könnte. Es ist ein Programm denkbar, welches zu einer Menge von gegebenen Faksimile-Scans selbständig das Seitenverhältnis der Scans ermittelt, das Drahtgitter auf dieses Verhältnis anpasst und anschließend die (evtl. parallelisierte) Kette von Rendervorgängen durchführt. So könnte komfortabel zu einem beliebigen eingescannten Original-Buch ein entsprechendes virtuelles 3D-Buch erstellt werden.
- **Direkt-Zoom im 3D-Buch.** Selbst in der höchsten generierten Auflösung der 3D-Blätteranimationen sind viele Feinheiten der Handschrift und der Miniatur-Zeichnungen der Fries-Chronik nicht ausreichend zu erkennen, denn diese sind für eine bildschirmfüllende Gesamtansicht oft zu detailreich. Momentan ist es bereits möglich, im 3D-Buchbetrachter per Klick auf eine Seite einen externen 2D-Faksimile-Betrachter mit Zoom-Funktionalität für die ausgewählte Seite zu starten. Komfortabler für den Anwender wäre jedoch sicherlich, stattdessen mit einer so genannten Online-Lupe innerhalb der aufge-

schlagenen Seite des 3D-Buches beliebige Bereiche unter der momentanen Position des Mauszeigers zu vergrößern.

Anhang

A Submorph-Zeichenketten

Da die Gewichte der Submorph-Regeln entweder subjektiv manuell oder werkabhängig durch das Morph-Feedback-Verfahren festgelegt werden, scheint ihre Veröffentlichung an dieser Stelle nicht sinnvoll. Um trotzdem den Eindruck einer Submorph-Regelmenge zu vermitteln, ohne dabei den Rahmen der Arbeit zu sprengen, werden im Folgenden Quell- und Ziel-Strings für die deutsche Sprache mit lateinischen und griechischen Fachtermini aufgeführt.

Gruppenname	***Regeln***
A-Gruppe	a→aa, a→ah, a→ar, a→er, aa→a, aa→ah, aa→er, ah→a, ah→aa, ar→a, er→a, er→aa, er→ah
E/Ä-Gruppe	ae→e, ae→ä, ae→ee, ae→eh, ae→äh, ae→aeh, ai→ä, aeh→e, aeh→ä, aeh→ae, aeh→ee, aeh→eh, aeh→äh, e→ä, e→ae, e→ee, e→eh, e→äh, e→aeh, ee→e, ee→ä, ee→ae, ee→eh, ee→äh, ee→aeh, eh→e, eh→ä, eh→ae, eh→ee, eh→äh, eh→aeh, ä→e, ä→ae, ä→ai, ä→ee, ä→eh, ä→äh, ä→aeh, äh→e, äh→ä, äh→ae, äh→ee, äh→eh, äh→aeh
I/IE/Y-Gruppe	i→y, i→ie, i→ih, i→ieh, ie→i, ie→y, ie→ih, ie→ieh, ih→i, ih→y, ih→ie, ih→ieh, ieh→i, ieh→y, ieh→ie, ieh→ih, y→i, y→ü, y→ie, y→ih, y→ieh, ü→i, ü→y
Ö-Gruppe	oe→ö, oe→öh, oe→oeh, oeh→ö, oeh→öh, ö→oe, ö→öh, öh→ö

Gruppenname	***Regeln***
J/Ü/Y-Gruppe	j→y, ue→y, ue→ü, ue→ui, ue→üh, ue→ueh, ueh→y, ueh→ü, ueh→ue, ueh→üh, y→j, y→ue, y→üh, y→ueh, ü→ue, ü→ui, ü→üh, ü→ueh, üh→y, üh→ü, üh→ue, üh→ueh
O-Gruppe	o→oh, o→oo, oh→o, oh→oo, oo→o, oo→oh
U-Gruppe	u→uh, u→uu, uh→u, uh→uu, uu→u, uu→uh
AI/EI-Gruppe	ai→ei, ai→eih, ei→ai, ei→eih, eih→ei
EU/ÄU/OI-Gruppe	eu→oi, eu→äu, oi→eu, oi→äu, äu→eu, äu→oi
G/K/C/Q-Gruppe	c→g, c→k, c→cc, c→ch, c→ck, c→kk, cc→c, cc→k, cc→ck, cc→kk, ch→c, ch→k, ck→c, ck→g, ck→k, ck→cc, ck→kk, g→c, g→k, g→cc, g→ck, g→kk, k→c, k→g, k→cc, k→ch, k→ck, k→kk, kk→k, kk→cc, kk→ck
QU/KW-Gruppe	kw→qu
SCH/CH-Gruppe	ch→sch, sh→sch, sch→ch, sch→sh
B/P-Gruppe	b→p, p→b
D/T-Gruppe	d→t, d→dh, d→th, dh→d, dh→t, dh→th, t→d, t→th, th→d, th→t
Dehnungs-H (Reste)	p→ph, ph→p, r→rh, rh→r
G/J-Gruppe	g→j, j→g
S,SS,ß-Gruppe	c→ss, s→c, s→ß, s→ss, s→sz, ss→s, ss→ß, ss→sz, sz→s, sz→ß, sz→ss, ß→s, ß→ss, ß→sz
W,V,F,PH-Gruppe	f→v, f→ph, ph→f, ph→v, ph→w, v→f, v→w, v→ph, w→f, w→v, w→ph
X/KS/CS/GS-Gruppe	cs→x, cs→chs, chs→x, chs→ks, gs→x, gs→chs, ks→x, ks→chs, x→cs, x→gs, x→ks, x→chs,
Z/TZ/TS/C-Gruppe	c→z, c→ts, c→tz, ts→c, ts→z, ts→tz, tz→c, tz→z, tz→ts, tz→zt, z→c, z→ts, z→tz, zt→tz
Doppelbuchstaben	b→bb, bb→b, d→dd, dd→d, f→ff, ff→f, g→gg, gg→g, p→pp, pp→p, l→ll, ll→l, m→mm, mm→m, n→nn, nn→n, r→rr, rr→r, t→tt, tt→t

Gruppenname	*Regeln*
Zahlen und Zahlenwörter	0→null, 1→eins, 10→zehn, 11→elf, 12→zwölf, 12→zwoelf, 2→zwei, 3→drei, 4→vier, 5→fünf, 5→fuenf, 6→sechs, 7→sieben, 8→acht, 9→neun, acht→8, drei→3, elf→11, eins→1, fünf→5, fünf→fuenf, fuenf→5, fuenf→fünf, neun→9, null→0, sechs→6, sieben→7, vier→4, zehn→10, zwei→2, zwölf→12, zwölf→zwoelf, zwoelf→12, zwoelf→zwölf, 20→zwanzig, 30→dreißig, 40→vierzig, 50→fünfzig, 60→sechzig, 70→siebzig, 80→achtzig, 90→neunzig, achtzig→80, dreißig→30, fünfzig→50, neunzig→90, sechzig→60, siebzig→70, vierzig→40, zwanzig→20
Tausender-Punkt	0→0., 0.→0, 1→1., 1.→1, 2→2., 2.→2, 3→3., 3.→3, 4→4., 4.→4, 5→5., 5.→5, 6→6., 6.→6, 7→7., 7.→7, 8→8., 8.→8, 9→9., 9.→9

B Suchmuster für Precision-Recall

Im folgenden Abschnitt werden die Suchmuster mit ihren erwünschten Varianten aufgelistet, die für die Messung von Precision und Recall in Abschnitt 5.10 verwendet wurden. Es folgen zunächst die Muster aus dem Hager-Textkorpus mit ihren Varianten. Anschließend sind die 0-Treffer-Muster der Altmeyer-Online-Benutzer aufgeführt, die aus den Protokolldateien des Servers entnommen sind. Auch zu diesen Mustern sind die erwünschten Varianten aufgeführt.

Suchmuster	*erwünschte Varianten (HagerROM)*
12stündigem	zwölfstündigem
4zelligen	vierzelligen
50fach	fünfzigfache, fünfzigfach
abschwächt	abschwaecht, abschwächen, abgeschwächten, abgeschwächtem, abgeschwächter, abgeschwächt, abgeschwächte, abschwächende, abschwächung
absorption	absoprtion, absorptionsfl, uv-absorption, endabsorption, absorptionem, absorptionen, uv-absorptionen, ir-absorption, uv-absortion, uv-absorbtion
abysinnica	abyssinica, abyssinicae
aciclovir	acyclovir, aciclovirs

Suchmuster	*erwünschte Varianten (HagerROM)*
adenocarcinom	adeno-carcinoma, adenokarzinomen, adenokarzinom, adenocarcinoma, adenocarcinome, adenocarcinomen
adoleszenz	adolescents, adolescence, adolescentis, adolescenten, adolescentes, adolescent, adoleszenten
ägypt	aegypten, aegyptia, aegypti, egypt, egypte, aegyptica, d'egypte, d'egypt, egyptian, ostägypten, ägypten, ägyptens, ägypter
äquifacial	aequifacial, äquifazialer, äquifaziale, äquifacialer, äquifacialen, äquifaciale, äquifacialem
ätherisch	aetherisch, etherisch, aetherische, aetherischen, aetherischer, etherischen, etherischem, etherischer, etherische, ätherischem, ätherischen, ätherisches, ätherischer, ätherische
aggressivität	agressivität
akkomodation	accomodation, akommodation, akkommodation
alkoholfreie	alkoholfrei, alkoholfreier, alkoholfreies, alkoholfreiem, alkoholfreien
amenorrhoe	amennorhoe, amenorhoe, amenorrhöe, amenorrhö, amenorrhea, ammenorrhoe, anemorrhoe, amenorrhoen,
aminobenzoesäureester	aminobenzoesäurester
antiarrhythmika	antiarrhythmica, antiarrythmika, antiarrhythmikum, antiarrhythmic, antiarrhythmikums, anti-arrhythmia, anti-arrhythmic, antiarrhythmisch, antiarrhythmische, antiarrhythmischen, antiarrhythmischer, i-antiarrhythmikum, i-antiarrhythmika, iv-antiarrhythmika, ia-antirrhythmika
arteether	---
asthmatica	astmatica, athmatica, asthmatic, asthmaticus, asthmatics, asthmatiker, asthmatikern, asthmathicus, asthmatischen, asthmatischer, asthmatische
austrittsstellen	austrittstellen, ausstrittstellen, austrittstelle
azyklische	acyclisches, acyclischen, acyclische
beeinflussen	beeinflusen, beinflußen, beeinflussung, beeinflußten, beeinflussungen, beeinflußte, beeinflussenden, beeinflussendes, beeinflussende
behaviour	behavior, behavioral, behaviors, behavioural, behaviours
beierlein	beyerlein
cytokine	zytokine, cytokin, zytokin, zytokinen, cytokins, cytokininen, cytokinine, cytokinen, cytokines
ekzem	ekzme, eczema, ekzeme, ekzema, rißekzem, ekzems, ekzemen, ekzemtyp

Suchmuster	*erwünschte Varianten (HagerROM)*
hämorrhoiden	hämorroiden, haemorrhoiden, hämorrhoidalen, hémorrhoidal
hartschicht	---
herzrhythmusstörungen	herzrhythmusstoerungen, herzrhythmusströungen, herzrythmusstörungen, herzrhythmusstörung
homöopathisch	homöophatische, homoöpathischen, homöopatische, homöopatischen, homeopathic, homöopathie, homöopathika, homöopathischer, homöopathisches, homöopathischen, homöopathische
hypercalcämie	hypercalcaemie, hyperkalzämie, hypercalciämie, hypercalcaemia, hypercalcaemic, hypercalzämie, hypercalciämien, hypercalcemia, hypercalcämische, hypercalcämischen, hyperkalziämie, hyperkalziämien, hypercalcämien
indo-malayischer	indo-malaiischer, indomalayischer, indo-malaiischen, indo-malaiisches, indomalayische, indomalaiischen
kalzium	calcium, calciums, calcium2, calcium1, calciumanw, calciumant, calcium-di, calciumakt, calciumrkt, clacium, kalziums, calicum
kjeldahlkolben	kjeldahl-kolben, kjehldahlkolben, ml-kjeldahl-kolben, ml-kjeldahlkolben
lokation	location
morfin	morphin, morphini, morphine, morphins, morphinum, morphinarm, morphintyp, morphindos, morfina, dimorphin, diamorphin, apomorphin, apomorphins
nefrol	nephrol, nephrology
öfters	öfter, öfteren, öfterem
pityriasis	pytyriasis
plattform	---
quellende	quellend, quellen, aufquellen, aufquellenden, quellenden, quellendes
schattige	schattig, schattiert, schattiger, schattigen, schattigem, unschattiert,
schlacken	schlacke
soccotrina	socotrina, succotrina, socotorina, socotrine, socotrino, socotrin, sokotrina
vassiljev	vasiljev, vasilyev, vassilev, vasilev, vasiliev, vasilév, vassilieff
venkelfrucht	---
verhütung	verhüten, verhütet

Suchmuster	*erwünschte Varianten (HagerROM)*
verkahlen	verkahlten, verkahlenden, verkahlend, verkahlende, verkahlendem
vitamin-c-mangel	vitamin-c-mangels

Suchmuster	*erwünschte Varianten (Online-Altmeyer)*
acanthose	akanthose, acanthosis, akanthosis
adipopzyten	adipozyten
allopezie	alopezie, alopezien, zugalopezie, alopezieherd, alopecie, alopezieherde, alopecia
antiphsopholipid	antiphospholipid, antiphospholipid-ak, anti-phospholipid
arznzeimittelallergie	arzneimittelallergie, arzneimittelallergien, arzneimittel-allergie
azneimittelexanthem	arzneimittelexanthem, arzneimittelexanthemen, arzneimittelexantheme
basalliom	basaliom, basalioma, basaliome
berylliumkrankheit	beryllium-krankheit
blepharithis	blepharitis
calciumglukonat	calciumgluconat, kalziumgluconat
chlioquinol	clioquinol
contratubex	contractubex
decubitus	dekubitus
dermacolour	dermacolor
druck-depigmentierung	druckdepigmentierung
effluviuum	effluvium, effluviums
entsprechnd	entsprechend, entsprechenden, entsprechendem, entsprechende, dementsprechend, entsprechender, dementsprechende, entsprechendes, entsprechen, entspechende, entspricht
erythematosqamös	erythematosquamöse, erythematosquamösen, erythematosquamosa, erythemato-squamöse
etyhlenglykol	ethylenglykol
extrammammär	extramammärer, extramammärem, extramammary
fotodermatitis	photodermatitis
glokagonom	glukagonom, glucagonoma
haemhorroide	haemorrhoiden, hämorrhoiden, haemorrhoids
heparinnekrosen	heparin-nekrosen
hochgebierge	hochgebirge, hochgebirgen

Suchmuster	*erwünschte Varianten (Online-Altmeyer)*
hydroccortison	hydrocortison
ichthioseptal	ichthoseptal
insektengifttr	insektengift, insektengifts, insektengifte, insektengiften
karposi-sarkom	kaposi-sarkom, kaposi-sarkomen, kaposi-sarkoms, kaposi-sarkome, kaposisarkoms
koloidmilium	kolloidmilium, kolloidmiliums
kosemtisch	kosmetisch, kosmetische, kosmetischen, kosmetisches, kosmetischer
kummulativ	kumulativ, kumulativen, kumulative, cumulative, kumulation
levedovaskulitis	livedovaskulitis
leukplakia	leukoplakia, leucoplacia, leucoplakia, leukoplakien, leukoplakie
magencarcinom	magenkarzinom
mealnoerythrodermie	melanoerythrodermie, melanoerythrodermien
mirkosporie	mikrosporie, mikrosporid
mosaikwarzen	mosaik-warzen, mosaik-warze
mucocutanes	mukokutanes, mucocutanea, mucocutaneous, mukokutaner, mukokutanen, mukokutanem, mukokutane, muco-cutaneous
naevuszellneavus	naevuszellnaevus
neurrdermatitis	neurodermatitis
orthoklone	orthoclone
pathogent	pathogen, pathogenen, pathogenesa, pathogenie, pathogens, pathogenic, pathogene, pathogenese, pathogener, pathogenetic, pathogenität
phlebologiee	phlebologie, phlebologe, phlebology, phlebologisch, phlebologische, phlebologischer, phlebologischen
pigmentnävus	pigmentnaevus
plathelmite	plathelmithen, plathelmintes, plathelminthes
präkursorläsionen	precursor-läsionen
psioriasis	psoriasis, i-psoriasis, ii-psoriasis
reflektion	reflection, reflexionen, reflexion, reflektieren, reflektierte, reflektiert
rosaceatherapie	rosazeatherapie
schwangerschatf	schwangerschaft, schwangerschaften
spermatozooen	spermatozoen
streptokpkken	streptokokken, b-streptokokken, a-streptokokken, streptokkokken, streptokokkus
syringoakanthom	syringoacanthom

Suchmuster	*erwünschte Varianten (Online-Altmeyer)*
thrombophlebistis	thrombophlebitis, thrombophlebitische
tricoblastisch	trichoblastisches, trichoblastic
urtilkaria	urtikaria, urticaria, urtikariatyp, urticarias, urticarial, urtikara
verruccae	verrucae, verruca, verrucas, verrucatum, verrucca
vulvakarzionom	vulvakarzinom, vulvakarzinome, vulvakarzinoms
zytizerkose	zystizerkose

C Format der Chartab

Die Datei `chartabXX.csv` ist eine Textdatei (Format: ISO-Latin-1), welche TAB-separiert verschiedene Darstellungen der in einem Werktext enthaltenen Sonderzeichen beinhaltet. Da die `chartab` von zahlreichen Modulen des Generierungsprozesses eines Werkes verwendet wird, ist darauf zu achten, dass in dieser Datei alle vorkommenden Sonderzeichen auch wirklich vermerkt sind. Die erste Zeile der `chartab` beinhaltet die Spaltenüberschriften und wird beim Auslesen der Datensätze überlesen. Alle daran anschließenden Zeilen in der `chartab` enthalten zu einem Datensatz die in der folgenden Tabelle dargestellten Felder.

Da fast kein Zeichen gleichzeitig alle Felder belegen muss (bzw. kann), werden für zwei verschiedene Beispielzeichen ihre Feldinhalte in der Tabelle angegeben. Das erste Beispielzeichen ist das große lateinische A mit Circumflex (Â); das zweite Beispielzeichen ist das kleine griechische Alpha (α).

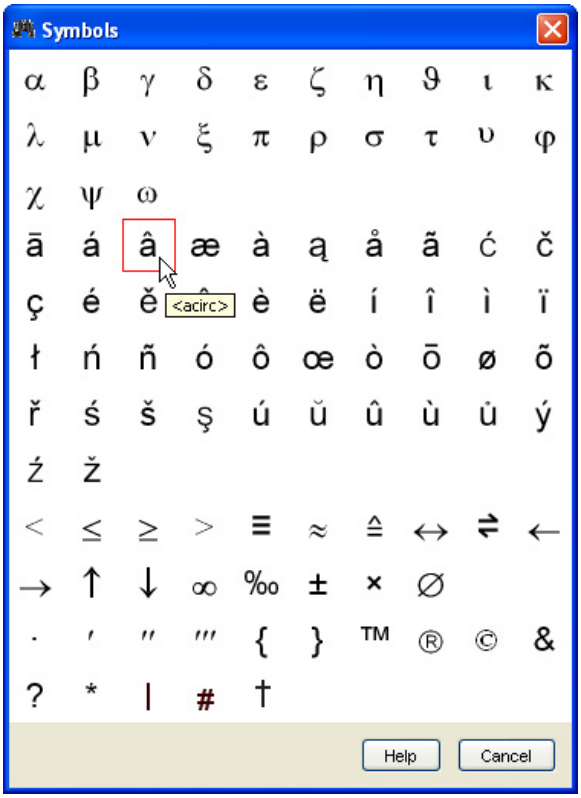
Da einige Felder der `chartab` die automatische Generierung des so genannten Sonderzeichen-Dialoges steuern, ist hinter der nun folgenden Tabelle ein Bildschirmfoto dieses Dialoges abgebildet.

Feldname	*Beispiel und Bedeutung*
`Nr`	1.) `5` 2.) `10` Laufende Nummer des Datensatzes zu Verwaltungszwecken. Die Nummer hat jedoch keinen Einfluss auf die Verwendung des Zeichens innerhalb der Konvertierungsroutinen.

Feldname	*Beispiel und Bedeutung*
`SGML`	1.) `&Acirc;` 2.) `&alpha;` Die „übliche“ SGML-Entity-Schreibweise ist das einzige Feld, welches zu einem Sonderzeichen eindeutig sein muss, es dient quasi als „primary key“ der Sonderzeichen-Datenbank. Ein Zeichen kann unter verschiedenen Entity-Schreibweisen aufgenommen werden (z.B. ist das kleine grichische Alpha auch als `&agr;` enthalten).
`HTML`	1.) `Â` 2.) `<font face="Symbol">a</font>` Die HTML-Schreibweise für Systeme, die nicht über Unicode-Unterstützung verfügen (z.B. ältere Windows-Versionen).
`HTML-Unicode`	1.) `&#194;` 2.) `&#945;` Die HTML-Schreibweise als dezimale Character-Reference.
`HTML-UnicodeHex`	1.) `&#x00c2;` 2.) `&#x03b1;` Die HTML-Schreibweise als hexadezimale Character-Reference mit führenden Nullen auf 4 Stellen aufgefüllt.
`Java-Unicode`	1.) `\u00c2` 2.) `\u03b1` Die Schreibweise des Zeichens für die Programmiersprache Java. So ist beispielsweise in diesen Java-String ein Alpha eingebettet (`String s = "Winkel \u03b1 war kleiner."`)
`Register`	1.) `A@Ad` 2.) `&a` Die Schreibweise des Zeichens im Format für das Satz-System Advent 3B2, welche vornehmlich im Registerband der *HagerROM* verwendet wurden.
`Fontname`	1.) 2.) `Symbol` Der benötigte Zeichensatz für Systeme ohne Unicode-Unterstützung (vgl. Feld „HTML“).

Feldname	*Beispiel und Bedeutung*
`8Bit-ISOLatin1`	1.) `Â` 2.) Die Darstellung des Zeichens mit einem Byte ISO-Latin-1. Der Buchstabe α ist nicht in diesem Characterset enthalten.
`8Bit-dezimal`	1.) `194` 2.) Der dezimale Wert des Zeichens im Characterset Latin-1.
`ToLower`	1.) `â` 2.) Schreibweise des Zeichens nach Wandlung des kompletten Textes in Kleinbuchstaben. Da das Beispiel-α bereits klein geschrieben ist, ist die 2. Beispiel-Zeile leer.
`ToUpper`	1.) 2.) `&#913;` Schreibweise des Zeichens nach Wandlung des kompletten Textes in Großbuchstaben. Da das Beispiel-Â bereits klein geschrieben ist, ist die 1. Beispiel-Zeile leer. In der zweiten Zeile befindet sich die Character-Reference auf ein großes griechisches Alpha.
`ToLatin`	1.) `A` 2.) Darstellung des lateinischen Zeichens, welches dem gegebenen Zeichen zugrunde liegt. Hier werden also zum Zeichen gehörige Diakritika (wie Punkte, Häkchen, Striche oder Kringel) entfernt, welche die Aussprache und Betonung markieren. Da dem griechischen Alpha kein lateinisches Zeichen zugrunde liegt, ist seine Beispiel-Zeile leer.
`SZDLG-Name`	1.) `acirc` 2.) `alpha` Der Name des Zeichens, den der Sonderzeichen-Dialog (siehe Abbildung nach dieser Tabelle) der Volltextsuche in das Benutzer-Suchmuster bei Bedarf (innerhalb von spitzen Klammern) einfügt. So kann der Benutzer z.B. mit `"winkel <alpha>"` nach dem Text `"winkel` α`"` suchen.

Feldname	*Beispiel und Bedeutung*
`SZDLG-Gruppe`	1.) `B` 2.) `G` Im Sonderzeichen-Dialog (siehe Abbildung) der Benutzeroberfläche werden die Sonderzeichen zu Gruppen zusammengefasst, um dem Benutzer die Suche nach einem bestimmten Zeichen zu erleichtern. Bisher gibt es vier solcher Gruppen: B=Buchstaben, G=Griechische Zeichen, M=Mathematische Symbole, S=Sonstige.
`SZDLG-Position`	1.) `2` 2.) `1` Im Sonderzeichen-Dialog (s. Abb.) der Benutzeroberfläche sollen die Zeichen innerhalb der Zeichengruppen sortiert werden. Diese Sortierung geschieht bezüglich des Feldes „SZDLG-Position“. So wird erreicht, dass z.B. im Dialog auf das Alpha das Beta und auf dieses das Gamma folgt usw..



Der Dialog zum einfachen Einfügen von Sonderzeichen in das Benutzer-Suchmuster

D Die Kommandozeilen-Suche

Im Folgenden werden die Schalter der Kommandozeilen-Version `CLISearch` (CLI = Command Line Interface) der fehlertoleranten Volltextsuche aufgeführt. Diese Version ist primär zu Test- und Entwicklungszwecken und nicht für den Endnutzer gedacht.

```
Aufruf: CLISearch.exe [ <optionen> ] <suchstring>

Opt.: -p <path>      Pfad zu den Suchdaten (Default: akt. Verz './')
                     Auch Umgebungsvariable SEARCH_DATA_PATH verwendbar
      -q             Quiet Mode (nur Zusammenfassung, keine Treffer ausgeben)
                     mehrfaches -q macht die Suche noch ruhiger.
      -b             ParseTree zeigen
      -l <int>       Zu durchsuchende FileId(s), erlaubt sind komma-getrennte
      -l <i-liste>   einzelne Ids und Bereiche, z.B. 1-300,500,700-750.
                     Es dürfen keine Leerzeichen enthalten sein!
                     Lokale Suche auf nur einer FileID ebenfalls möglich.
      -w <int>       Zu durchsuchende Werkteile. Als int-codierter Bit-
                     Vektor. z.B. -w 3 setzt 0. und 1. Werkteil
      -w <s-liste>   Zu durchsuchende Werkteile. Als String-Liste.
                     z.B. -w drogen oder -w "drogen,reuter d-e"
      -f <int>:<int> Feldsuche: 1.<int> ist Werkteilnummer 0-basiert
      -f <i-liste>   und dezimal. 2.<int> sind gewünschte Felder für Werkteil
                     Felder als int-codierter Bit-Vektor.
                     z.B. -f 1:3,7-4 durchsucht im 0.WT das 0. und 1. Feld
                     UND im 6.WT das 2. Feld.
      -t <int>       Suche mit Fehlertoleranz (Stufe <int>, gült. Werte: 0-2)
      -TP <int>      Freie Fehlertoleranz: MaxPenalty auf <int> setzen
      -TM <int>      Freie Fehlertoleranz: MaxMorph auf <int> setzen
      -TB <int>      Freie Fehlertoleranz: MaxBestN auf <int> setzen
      -TD <int>      Freie Fehlertoleranz: TrieDepth beschränken
      -TW <int>      Freie Fehlertoleranz: Trie Window-Delta
      -!             Suchstring NICHT nach lowercase wandeln. Wörtlich nehmen!
      -g             * liefert alle Kombinationen, nicht nur erste.
      -n <int>       max. Abstand zwischen Teilen von * und &
      -M <int>       Zu benutzender Hauptspeicher (in MB)

Suchreihenfolge für Datenpfad:
  1. "-p" Switch-Angabe
  2. Umgebungsvariable %SEARCH_DATA_PATH bzw. $SEARCH_DATA_PATH
  3. Verzeichnis >./<
  4. Verzeichnis >./search_data/<
Benötigt werden folgende Datenfiles:
  1.>header<  2.>data<  3.>fields_b.dat<  4.>namelistnena.txt|namelist.txt|
  namelistnena_v5.txt<  5.>symbolReplacements.txt<  6.>tischlerTrie<
```

Bevor die Volltextsuche arbeiten kann, muss sie ein Basisverzeichnis kennen, in welchem die benötigten Hilfs- und Indexdateien liegen. Die in diesem Verzeichnis mindestens benötigten Dateien sind aus den letzten beiden Zeilen der obigen Ausgabe ersichtlich. Bei der Suche nach diesem Verzeichnis geht `CLISearch` in

folgender Reihenfolge vor: 1.) wird untersucht, ob mit dem Schalter -p ein Verzeichnis angegeben wurde, in dem alle benötigten Dateien enthalten sind. Falls nicht, wird 2.) untersucht, ob eventuell eine Umgebungsvariable `% SEARCH_DATA_PATH` (Windows) bzw. `$SEARCH_DATA_PATH` (Linux) auf ein existierendes Verzeichnis verweist, in dem sich die benötigten Dateien befinden. Falls auch dies nicht der Fall ist, so wird versucht, ob sich die benötigten Dateien im aktuellen Verzeichnis (aus welchem `CLISearch` aufgerufen wurde) befinden. Ist auch dies nicht der Fall, so wird versucht, ob sich die Dateien im Unterverzeichnis `search_data/` zum aktuellen Verzeichnis befinde. Scheitert diese letzte Option, so bricht das Programm mit einer Fehlermeldung ab.

Im Folgenden werden nun die wichtigsten Anwendungsfälle der Volltextsuche an einigen Beispielen erklärt.

- `CLISearch -p /HOME/hager/raid/distribute/current wolfskraut`
  Der Datenpfad wird gesetzt und eine exakte Suche nach dem Begriff `wolfskraut` durchgeführt. Treffer im Text werden auf der Konsole ausgegeben.
- `CLISearch "toxizität von wolfskraut"`
  Suchmuster, die Leerzeichen enthalten müssen in Anführungszeichen stehen.
- `CLISearch "toxizität&wolfskraut" -n 250`
  Boolesche UND-Kombination der beiden Suchmuster-Teile. Durch den Schalter `-n` wird der maximale Abstand der Suchmuster-Teile auf 250 Zeichen begrenzt.
- `CLISearch "wolfskraut" -w drogen`
  Führt die exakte Suche nur im Werkteil mit dem Namen „Drogen“ durch.
- `CLISearch "kalzium" -t 0 -q`
  Hiermit wird eine tolerante Suche mit „geringer“ Fehlertoleranz durchgeführt. Die Ausgabe der konkreten Trefferpositionen wird mit der Option `-q` unterdrückt, sodass nur die Treffer-Morphs ausgegeben werden.

E Die Datei wordlist.dtd

Die Speicherung der Wortlisten zur Analyse und Korrektur von Werken geschiet im XML-Format konform zur folgenden Dokument Typ Definition (DTD) `wordlist.dtd`. Sollen die für diese Arbeit entwickelten Werkzeuge oder die für die Korrektur generierten Wortlisten weiterverwendet werden, so müssen sich Programmerweiterungen aus Gründen der Kompatibilität an dieses XML-Format halten.

```
<?xml version='1.0' encoding='US-ASCII'?>
<!--
 ********************************************************
 wordlist DTD v1.6
 ********************************************************
-->

<!ELEMENT wordlist (name, nameshort, fieldlist*, entry+)>
<!ATTLIST wordlist
 count           CDATA               #IMPLIED
 keyCaseIgnore   (true|false)        #IMPLIED
 wordsToLower    (true|false)        #REQUIRED
>
<!ELEMENT name (#PCDATA)>
<!ELEMENT nameshort (#PCDATA)>
<!ELEMENT fieldlist (field+)>
<!ATTLIST fieldlist
 count           CDATA               #REQUIRED
>
<!ELEMENT field (#PCDATA)>


<!ELEMENT entry (word, stem?, context?, references?,corrections?)>
<!ATTLIST entry
 num             CDATA          #IMPLIED
 freq            CDATA          #IMPLIED
 qual            CDATA          #IMPLIED
 boostqual       CDATA          #IMPLIED
 falsepositive   (true|false)   #IMPLIED
>
<!ELEMENT word          (#PCDATA)>
<!ELEMENT context       (#PCDATA)>
<!ELEMENT references    (#PCDATA)>
<!ATTLIST references
            count       CDATA   #REQUIRED
>
<!ELEMENT corrections   (#PCDATA)>
<!ATTLIST corrections
            bestfreq    CDATA   #IMPLIED
            bestdist    CDATA   #IMPLIED
>
<!ELEMENT stem          (#PCDATA)>
```

Eine korrekte Verwendung der obigen DTD durch eine XML-Wortliste ist in Beispiel 7-1 (auf Seite 125) zu sehen. Die Elemente `stem` (für den Wortstamm), `context` (für einen Beispiel-Kontext) und `corrections` (für Korrekturvorschläge) werden nur während der Fehlerkorrektur verwendet und sind daher in den Wörterbüchern nicht enthalten.

F Schalter für MainWordListCmd

```
Usage: java MainWordListCmd -h
 - or -
Usage: java MainWordListCmd -p <file-with-params>
 - or -
Usage: java MainWordListCmd [Options, see below]



[-h|--help]                              display this help-text
[-a|--save-statistics]                   save an additional statistics file
                                         (outfilename + _STAT.txt)

****** INPUT/OUTPUT-FORMAT ******
-i <infile>                              infile for WordList (txt, xml)
-o <outfile>                             outfile for WordList (txt, xml, html)

[-n <outlongname>]                       outfile's new long <name> (in XML)
[-f if1,if2,...,ifN ]                    infile's fields (NUM, QUAL, ...) -
                                         not for XML
[-F of1,of2,...,ofN ]                    outfile's fields (NUM, QUAL, ...)
[-c <in-charset>]                        infile's character set (ISO-8859-1,
                                         UTF-8, ...) - not for XML
[-C <out-charset>]                       outfile's character set
                                         (ISO-8859-1,UTF-8, ...)
[-k|--in-keynocase]                      infile has keywords case-INsensitive
                                         - not for XML
[-w|--in-wordlower]                      infile has words lowercased-not for XML
[--infuzzykeys]                          unify keys (del hyph.,k>c,ph>f,z>c,...)


********** SORTING *************
[-s <sort-field>]                        sort field (FREQ, QUAL, WORD, ...)
[-d|--sort-descending]                   sort order: descending (default is
                                         ascending)

********** FILTERS *************
[-q <filter-min-qual>]                   filter: min. word quality
[-Q <filter-max-qual>]                   filter: max. word quality
[-l <filter-min-leng>]                   filter: min. word length
[-L <filter-max-leng>]                   filter: max. word length
[-e <filter-min-freq>]                   filter: min. word frequency
[-E <filter-max-freq>]                   filter: max. word frequency
[-u <filter-min-refcount>]               filter: min. reference count
[-U <filter-max-refcount>]               filter: max. reference count
[-t <filter-min-dist>]                   filter: min. distance to corrections
[-T <filter-max-dist>]                   filter: max. distance to corrections
[--false-positive <false-positive>]      [true|false] filter: false-positive
                                         field
[-r <reg-exp>]                           filter: regular expression
[-I|--re-case-ins]                       filter: reg.exp. is case-INsensitive
[-M|--re-comp-mat]                       filter: reg.exp. must be complete match
```

```
********** OPERATORS *************
[--operator <operator>]                combine two lists with operator : ADD
                                       or SUBTRACT or INTERSECT or
                                       SUBTRACTSTEM or SPELLCHECK or
                                       SPELLCHECKMORPH
[--in2file <in2file>]                  2nd infile's filename, for operator
                                       (txt, xml)
[--in2charset <in2charset>]            2nd infile's character set
                                       (ISO-8859-1, UTF-8, ...) - not for XML
[--in2fields <in2fields>]              2nd infile's fields
[--in2arg2 <in2arg2>]                  2nd argument for operator. e.g.
                                       infile's language (german, english,
                                       ...) for operator SUBTRACTSTEM or
                                       [TOLERANCE|MP,MM,MB] for
                                       SPELLCHECKMORPH
[--in2keynocase]                       2nd infile has keywords
                                       case-INsensitive - not for XML
[--in2wordlower]                       2nd infile has words lowercased – not
                                       for XML
[--in2fuzzykeys]                       unify keys (del hyphens, k>c, ph>f,
                                       z>c, ...)
```

Die Schalter für `MainWordListCmd` sind in vier Gruppen aufgeteilt. In der ersten Gruppe befinden sich die Schalter, mit denen man das Eingabe- und Ausgabeformat der Wortlisten festlegen kann. Liegt die Eingabedatei im Textformat vor (jede Datensatz-Zeile TAB trennt), so muss mit den Schaltern `-c`, `-f`, `-k`, `-w` spezifiziert werden, wie diese Liste abgespeichert wurde (Zeichensatz, Felder usw.). Für eine Liste gültiger Feldnamen siehe unten stehende Tabelle. Ist die Eingabedatei im XML-Format konform zur `wordlist.dtd` (siehe Anhang E) gespeichert, so ermittelt das Programm selbständig die Werte für diese Parameter.

Unabhängig vom Ausgabeformat (Text, XML, HTML) kann mit den Schaltern `-C` und `-F` gesteuert werden, welcher Zeichensatz (UTF-8 oder ISO-8859-1) verwendet wird, und welche Felder in der Ausgabedatei gespeichert werden. Gültige Feldnamen sind der folgenden Tabelle zu entnehmen.

Feldname	***Bedeutung***
NUM	Laufende Nummer (abhängig von der gewählten Sortierung)
WORD	Das Wort
FREQ	Häufigkeit des Wortes, eventuell addiert aus mehreren Referenzwerken (siehe auch REFNAMES und REFCOUNT).
REFNAMES	Durch Kommata getrennte Liste von Wortlisten-Namen, in denen das Wort referenziert wurde. Diese Liste wird z.B. bei Verwendung des Operators ADD aktualisiert.

Feldname	***Bedeutung***
REFCOUNT	Anzahl unterschiedlicher Wortlisten, in denen das Wort referenziert wurde. Diese Zahl wird z.B. bei Verwendung des Operators ADD aktualisiert.
QUAL	Qualität des Wortes. Diese kann z.B. vom Harvester wärend des Sammelns der Worte extern gesetzt werden. Z.B. aufgrund der Häufigkeit oder aufgrund der Verlässlichkeit der Quelle des Wortes. Weil letztere Einflussgröße eher subjektiver Natur ist, wird der Wert QUAL nie automatisch berechnet. Der Wertebereich läuft von 0 (unbekannte Q.) über 1 (geringe Q.) bis zu 100 (höchste Q.).
BQUAL	Angehobene Qualität des Wortes. Unter Berücksichtigung von Wortlänge, Worthäufigkeit und Anzahl der unterschiedlichen Werke, in denen das Wort verwendet wurde (Feld: REFCOUNT) wird hier die Qualität des Wortes ggfs. gegenüber dem Feld QUAL leicht angehoben (jedoch nie über den maximalen Wert 100). Im Gegensatz zum QUAL-Wert wird dieser Wert vom Programm berechnet.
CONTEXT	Speicherung für ein oder mehrere Verwendungskontexte zu einem Wort. Bei Listen, die als Wörterbuch verwendet werden, sollte dieses Feld aus Geschwindigkeitsgründen leer bleiben. Es dient vor allem bei fraglichen Worten dazu, das Wort schneller als „falsch“ oder „richtig“ zu klassifizieren.
FALSEPOS	Boolescher Wert zur Speicherung, ob das Wort wirklich falsch ist, oder ob es richtig geschrieben ist.
WORDSCORR	Liste von möglichen Korrekturvorschlägen zu einem fraglichen Wort.
FREQBC	Häufigkeit des besten Korrekturvorschlages. Dieses Feld dient der optionalen Sortierung der fraglichen Worte. So können die fraglichen Worte, die selten vorkommen und deren Korrekturvorschlag dagegen verhältnismäßig oft vorkommt an den Anfang der Liste sortiert werden, da hierdurch erfahrungsgemäß die Fehlerdichte am Anfang der Liste steigt.
STEM	Speicherung des (automatisch generierten) Wortstammes zu einem Wort. Auch wenn der Stemming-Algorithmus nur wenig Zeit zur Berechnung benötigt, dient die Speicherung bereits berechneter Stämme der Beschleunigung zukünftiger Programmläufe.

Mit den Schaltern `-s` und `-d` kann die Liste vor der Ausgabe bezüglich eines Feldes aufsteigend oder absteigend sortiert werden. Z.B. kann hier eine alphabetische Sortierung der Wortliste nach Worten erfolgen, oder eine numerische Sortierung nach Häufigkeit oder Länge der jeweiligen Worte.

In der dritten Gruppe befinden sich die Schalter für die Filter. Mit diesen können obere und untere Schranken für numerische Felder festgelegt werden, die ein Wort erfüllen muss, damit es in die Ausgabe-Wortliste aufgenommen wird. Solche Grenzen können z.B. für die Felder Qualität, Länge, Häufigkeit oder Anzahl Referenzen festgelegt werden. Mit regulären Ausdrücken kann außerdem die Zeichenkette des Wortes gefiltert werden (z.B. Löschung von Wörtern, die mit Ziffern beginnen).

In der vierten Gruppe befinden sich die Schalter für die Operatoren. Hier dienen zunächst einige Schalter dazu, die zweite Liste (den 2. Operanden) zu spezifizieren (`--in2file, --in2charset, --in2fields, --in2keynocase, --in2wordlower, --in2fuzzykeys`). Mit dem Schalter `--operator` kann der benötigte Operand festgelegt werden, der die beiden beteiligten Eingabelisten verknüpfen soll, ehe diese in die Ausgabeliste geschrieben werden. Für eine kommentierte Übersicht der möglichen Operanden sei auf die an Beispiel 7-2 (Seite 126) anschließende Übersicht verwiesen.

Abschließend sei noch auf die Möglichkeit hingewiesen, dass die möglicherweise umfangreichen – und daher unübersichtlichen – Verkettungen von Kommandozeilen-Parametern sich auch alle in einer Textdatei (mit beliebigen Zeilenumbrüchen zwischen den Schaltern) speichern lassen. Diese Parameter-Datei kann dann mit nur einem Schalter (`-p`) an das Programm übergeben werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AB02] Altmeyer P., Bacharach-Buhles M.: *Springer Enzyklopädie Dermatologie, Allergologie, Umweltmedizin.* Springer-Verlag, Heidelberg, ISBN: 3-540-41361-8, 2002 http://www.galderma.de/anmeldung.enz.html

[ACTA03] Institut für Demoskopie Allensbach: *Allensbacher Computer- und Technik-Analyse.* Institut für Demoskopie Allensbach, 2003 http://www.acta-online.de/

[ASU88] Aho A., Sethi R., Ullman J.: *Compilerbau (Teil 1).* Addison-Wesley, ISBN: 3-89319-150-X, 1988

[Bauf96] Baufeld Chr.: *Kleines frühneuhochdeutsches Wörterbuch.* Max Niemeyer Verlag, Tübingen, ISBN: 3-484-10268-3, 1996

[BEH+01] Blaschek W., Ebel S., Hackenthal E., Holzgrabe U., Keller K., Reichling J. (Hrg.): *HagerROM 2001 - Hagers Handbuch der Drogen und Arzneistoffe. CD-ROM.* Springer-Verlag, Heidelberg, ISBN: 3-540-14868-X, http://www.hagerrom.de

[BEH+04] Blaschek W., Ebel S., Hackenthal E., Holzgrabe U., Keller K., Reichling J., Schulz V. (Hrg.): *HagerROM 2004 - Hagers Handbuch der Drogen und Arzneistoffe. CD-ROM.* Springer-Verlag, Heidelberg, ISBN: 3-540-21467-4, http://www.hagerrom.de

[Behl04] Behl J.: *Learning Objects Modeling für heterogene Informationsquellen der Audi AG.* Diplomarbeit Lehrstuhl für Informatik II, Universität Würzburg, 2004

[BG98] Beck K., Gamma E.: *Test Infected: Programmers Love Writing Tests.* In: Java Report Vol. 3 No 7. (37-50), July 1998 http://junit.sourceforge.net/doc/testinfected/testing.htm

[BP98] Brin S., Page L.: *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. In: Computer Networks and ISDN Systems Vol. 3 (107-117), 1998 http://www-db.stanford.edu/pub/papers/-google.pdf

[BR99] Baeza-Yates R., Ribeiro-Neto B.: *Modern Information Retrieval*. Addison Wesley, ISBN: 0-201-39829-X, 1999

[Bru+00] Bruchhausen F.v. et al. (Hrg.): *Hagers Handbuch der Pharmazeutischen Praxis. 10(+2) Bände. u. Folgebände.* Springer-Verlag, Heidelberg, ISBN: 3-540-52640-4, 1992-2000

[CH94] Clark J., Holton D.: *Graphentheorie*. Spektrum Akadademischer Verlag, ISBN: 3-8602-5331-X, 1994

[Chur54] Churchill W. S.: *Der Zweite Weltkrieg, 1. Auflage von 1960.* Alfred Scherz Verlag (Bern), Droemersche Verlagsanstalt Th. Knaur Nachf. (München, Zürich), ISBN: 3-502-19132-8 (aktuelle Auflage), 1954

[CM04] Chobotar B., Mehlhorn H. (Hrg.): *Parasitology Research.* Springer-Verlag, Heidelberg, ISBN: 0-932-0113, 2003 http://link.springer.de/link/service/journals/00436/

[Dies00] Diestel P.: *Graphentheorie (Elektronische Ausgabe).* Springer-Verlag, Heidelberg, ISBN: 3-5406-7656-2 (Buchausgabe), 2000 http://www.math.uni-hamburg.de/home/diestel/books/-graphentheorie/

[Dud96] Duden-Verlag (Hrsg.): *Duden - Die deutsche Rechtschreibung*. Duden-Verlag, ISBN: 3-411-04011-4, 1996

[EH04] Esser W., Höhn H.: *Supporting E-Learners With Fault-Tolerant Fulltext Retrieval*. In: Bernath U. et al: Supporting the Learner in Distance Education and E-Learning, Proceedings of the 3rd EDEN Research Workshop, Oldenburg (434-440), 2004 http://www.eden-online.org/papers/publications/toc-oburg.pdf

[Ess03] Esser W.: *Retrieval Tools und Fehlertoleranz für Lehr- und Lernplattformen*. In: Puppe, F. et al: Rechnergestützte Lehr- und Lernsysteme in der Medizin, Proceedings 7. Workshop der Arbeitsgemeinschaft der GMDS, Würzburg, Shaker-Verlag (24-35), 2003 http://www.shaker.de/Online-Gesamtkatalog/Details.idc?ISBN=3-8322-1361-9

[Ess03b] Esser W.: *Fault-Tolerant q-Gram Text-Retrieval in Electronic Libraries by Weighted Pattern Morphing.* In: Tochtermann, K. et al: Proceedings of I-KNOW'03, 3rd International Conference on Knowledge Management, Graz (435-441), 2003

[Ess04] Esser W.: *Fault-Tolerant Fulltext Information Retrieval in Digital Multilingual Encyclopedias with Weighted Pattern Morphing.* In: McDonald S., Tait J.: Advances in Information Retrieval, Proceedings of the 26th European Conference in IR Research, ECIR 2004, Sunderland, LNCS 2997, Springer-Verlag (338-352), 2004
http://springerlink.com/link.asp?id=u2y8wxlwft5nba94

[Ess04b] Esser W.: *Fault-tolerant Fulltext Search for Large Multilingual Scientific Text Corpora.* In: McKnight C. et al (Eds.): Journal of Digital Information (JoDI), Vol. 6 (1), Article No. 303 (1-9), 2004 http://jodi.ecs.soton.ac.uk/Articles/v06/i01/Esser/

[Fred60] Fredkin E.: *Trie Memory.* In: Communications of the ACM Vol. 3 (9), Association for Computing Machinery (ACM) (490-499), 1960

[Frie02] Friedl J.: *Mastering Regular Expressions (2nd Edition).* O'Reilly & Associates, Inc., ISBN: 0-596-00289-0, 2002

[Frie04] Fries L..: *Chronik der Bischöfe von Würzburg - Die Prachthandschrift des Fürstbischofs Julius Echter als Multimedia-DVD.* Universitätsbibliothek Würzburg, ISBN: 3-923959-32-X, 2004 http://fries.informatik.uni-wuerzburg.de/dvd-edition/

[Frie1582] Fries L.: *Histori, Namen, Geschlecht, Wesen, Thaten, gantz leben und sterben der gewesnen Bischoffen zu Wirtzburg und Hertzogen in Franken, Auch was bey einem Jeden in Zeit seiner Regirung, sunderlich gehandelt worden, ergangen vnd beschehen ist.* Handschrift Universitätsbibliothek Würzburg, M.ch.f.760 (sog. "Irmelshäuser Exemplar"), 1582

[FS02] Fowler M., Scott K.: *UML konzentriert, 2. Auflage.* Addison-Wesley, ISBN: 3-8273-1617-0, 2000

[Gad90] Gadd T.: *PHONIX: The algorithm.* In: Program: automated library and information system 24(3) (363-366), 1990

[Gerh02] Gerhardt C.: *Lehrbuch der Mathematik - Analysis I.* Verlag BoD GmbH, Norderstedt, ISBN: 3-8311-4256-4, 2002
http://www.math.uni-heidelberg.de/studinfo/gerhardt

[GHJV96] Gamma E., Helm R., Johnson R., Vlissides J.: *Entwurfsmuster.* Addison-Wesley, ISBN: 3-89319-950-0, 1996

[GKHE+03] Gaigl Z., Kramp K., Höhn H., Esser W., Albert J., Hamm H.: *The DEJAVU Project: Presentation of a Teledermatological Learning System.* In: European Confederation of Telemedical Organizations in Dermatology, ECTODERM 2003, Graz, Abstract in Experimental Dermatology, (11) 6 (611), 2003

[GR99] Golding A., Roth D.: *A Winnow-Based Approach to Context-Sensitive Spelling Correction.* In: Machine Learning (34), Kluwer Academic Publishers. (107-130), 1999

[Gri01] Grimm M.: *Random Access und Caching für q-Gramm-Suchverfahren.* Diplomarbeit Lehrstuhl für Informatik II, Universität Würzburg, 2001

[HA01b] Hodge V., Austin J.: *An Evaluation of Phonetic Spell Checkers.* Tech. Report YCS338 Dept. of CS, University of York, U.K., 2001

[Hag05] Hagen U. von: *Das neue V-Modell XT.* In: LDVZ-Nachrichten, 6. Jahrgang, Ausgabe 1/2005 (29-37), 2005
http://www.kbst.bund.de/Anlage306569/Das-neue-V-Modell.pdf

[Harr78] Harrison M.: *Introduction to Formal Language Theory.* Addison-Wesley, ISBN: 0-201-02955-3, 1978

[HB03] Hirst G., Budanitsky A.: *Correcting real-word spelling errors by restoring lexical cohesion.* In: Natural Language Engineering [to appear] (1-44), 2003
http://ftp.cs.toronto.edu/pub/gh/Hirst+Budanitsky-2001.pdf

[HMU02] Hopcroft J., Motwani R., Ullman J.: *Einführung in die Automatentheorie Formale Sprachen und Komplexitätstheorie.* Addison-Wesley, ISBN: 3-8273-7020-5, 2002

[Höh02] Höhn H.: *Multimediale, datenbankgestützte Lehr- und Lernplattformen.* Dissertation Bayerische Julius-Maximilians-Universität Würzburg, 2002

[IEC00] International Electrotechnical Commission (IEC): *When is a kilobyte a kibibyte? And an MB an MiB? (Standard IEC 60027-2 Ed. 2.0).* International Electrotechnical Commission (IEC), Nov. 2000
http://www.iec.ch/zone/si/si_bytes.htm

[Knut98] Knuth D.: *The Art of Computer Programming, Second Ed., Volume 3: Sorting and Searching.* Addison-Wesley, ISBN: 0-201-89685-0, 1998

[Koeh01] Koehler W.: *Web page change and persistence - A four-year longitudinal study.* In: Journal of the American Society for Information Science and Technology, Volume 53, Issue 2 (162-171), 2001 http://www3.interscience.wiley.com/cgi-bin/abstract/88510894/

[Krü04] Krüger G.: *Handbuch der Java-Programmierung (4. Auflage).* Addison-Wesley, ISBN: 3-8273-2201-4, 2004
http://www.javabuch.de/download.html

[Kuk92] Kukich K.: *Technique for automatically correcting words in text.* In: ACM Computing Surveys 24(4) (377-439), 1992

[Kuk92b] Kukich K.: *Spelling correction for the telecommunications network for the deaf.* In: Communications of the ACM 35, 5 (May) (80-90), 1992

[Lev65] Levenshtein V.: *Binary codes capable of correcting deletions, insertions, and reversals.* In: Problems in Information Transmission 1 (8-17), 1965

[McCr76] McCraight E.: *A space-economical suffix tree construction algorithm.* In: Journal of the ACM, Vol. 23, No. 2 (262-272), 1976

[Meh01] Mehlhorn H. (Hrg.): *Encyclopedic Reference of Parasitology.* Springer-Verlag, Heidelberg, ISBN: 3-5406-6239-1, 2001

[Mye94] Myers E.: *A sublinear algorithm for approximate keyword searching.* In: Algorithmica, 12(4/5) (345–374), 1994

[Nav01] Navarro G.: *A Guided Tour to Approximate String Matching.* In: ACM Computing Surveys, Vol. 33, No. 1 (31-88), 2001

[NB98] Navarro G., Baeza-Yates R.: *A Practical q-Gram Index for Text Retrieval Allowing Errors*. In: CLEI Electronic Journal, Vol. 1, No. 2 (1), 1998 http://www.clei.cl/cleiej/paper.php?id=32

[NBST01] Navarro G., Baeza-Yates R., Sutinen E., Tarhio J.: *Indexing Methods for Approximate String Matching*. In: IEEE Bulletin of the Technical Committee on Data Engineering, Vol. 24, No. 4 (19-27), 2001

[Phi00] Philips L..: *The Double Metaphone Search Algorithm*. In: C/C++ Users Journal (1), 2000 http://www.cuj.com

[Port80] Porter M.: *An algorithm for suffix stripping*. In: Program, 14 (3) (130-137), 1980 http://www.tartarus.org/~martin/PorterStemmer/

[Pre02] Pretto L.: *A Theoretical Analysis of Google's PageRank*. In: Proc. of the 9th International Symposium on String Processing and Information Retrieval (SPIRE), Lisbon, LNCS 2476, Springer-Verlag, Heidelberg (131-144), 2002

[Psch02] Pschyrembel, W. (Begr.): *Pschyrembel Klinisches Wörterbuch*. Verlag Walter de Gruyter, ISBN: 3-11-016523-6, 2002

[Reut04] Reuter P.: *Springer Lexikon Medizin*. Springer-Verlag, Heidelberg, ISBN: 3-540-20412-1, 2004

[Reut04b] Reuter P.: *Springer Lexikon Medizin - Die DVD*. Springer-Verlag, Heidelberg, ISBN: 3-540-21873-4, 2004

[RFC3626] The Internet Society: *RFC 3629 - UTF-8, a transformation format of ISO 10646*. (Stand: 2003) http://www.faqs.org/rfcs/rfc3629.html

[Ried98] Riedl M.: *Effiziente Volltextsuche mittels Q-Grammen*. Diplomarbeit Lehrstuhl für Informatik II, Universität Würzburg, 1998

[Roch03] Urban & Fischer (Hrsg.): *Roche Lexikon Medizin (Online-Version der 5. Auflage)*. Elsevier (Urban & Fischer), ISBN: 3-437-15150-9, 2003 http://www.tk-online.de/rochelexikon/

[Rus18] Russell R.: *INDEX (Soundex Patent)*. In: U.S. Patent No. 1,261,167 (1-4), 1918

[Scha03] Schaupp A.: *Nutzerorientierte Bewertungen und Präsentation von Volltextsuche-Ergebnissen.* Diplomarbeit Lehrstuhl für Informatik II, Universität Würzburg, 2003

[Schm94] Schmoeckel Chr.: *Lexikon und Differentialdiagnose der klinischen Dermatologie.* Thieme Verlag, ISBN: 3-13-112202-1, 1994

[Schm97] Schmoeckel Chr.: *CD Klinische Dermatologie (CD-ROM für Windows 3.x).* Thieme Interactive, ISBN: 3-13-108391-3, 1997

[Schr04] Schreckling D.: *Volltextsuche mit einfachen regulären Ausdrücken auf XML-Dateien.* Diplomarbeit Lehrstuhl für Informatik II, Universität Würzburg, 2004

[Spin03] Spinellis D.: *The Decay and Failures of Web References.* In: Communications of the ACM, 46(1) (71-77), 2003
http://www.dmst.aueb.gr/dds/pubs/jrnl/2003-CACM-URLcite/html/urlcite.pdf

[ST96] Sutinen E., Tarhio J.: *Filtration with q-Samples in Approximate String Matching.* In: Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96), Laguna Beach, LNCS 1075, Springer-Verlag, Heidelberg (50-63), 1996

[Ste94] Stephen G.: *String Searching Algorithms, Lecture Notes Series on Computing, Vol. 3.* World Scientific Publishing, ISBN: 981-02-1829-X, 1994

[Stro97] Stroustrup B.: *The C++ Programming Language.* Addison Wesley, ISBN: 0-201-88954-4, 1997

[Swa88] Swanson D.: *Historical note: Information retrieval and the future of an illusion.* In: Journal of the American Society for Information Science Vol. 39(2) (92-98), 1988

[Taka94] Takaoka T.: *Approximate pattern matching with samples.* In: Proceedings of the 5th International Symposium on Algorithms and Computation (ISAAC'94), Beijing, LNCS 834, Springer-Verlag, Heidelberg (234–242), 1994

[TW58] Taube M., Wooster H. A.: *Information Storage and Retrieval: Theory, Systems, and Devices.* Columbia University Press, New York, 1958

[TWWCP02] The Whole World Company Press: *PhraseBook for Writing Papers and Research*. The Whole World Company Press, ISBN: 1-903384-00-1, 2002
http://www.wholeworldcompany.com/englishforresearch/writing_help/british_american.htm

[Ukk85] Ukkonen E.: *Finding approximate patterns in strings*. In: Journal of Algorithms, 6 (132-137), 1985

[Ukk92] Ukkonen, E.: *Approximate string-matching with q-grams and maximal matches*. In: Theoretical Computer Science 92 (191-211), 1992

[Ukk93b] Ukkonen E.: *Approximate string-matching over suffix trees*. In: Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM'93), Padova, LNCS 684, Springer-Verlag, Heidelberg (228-242), 1993

[Ukk95] Ukkonen E.: *On-line construction of suffix-trees*. In: Algorithmica 14 (249-260), 1995

[Ull04] Ullenboom Chr.: *Java ist auch eine Insel (4. Auflage)*. Galileo Press, 2004
http://www.galileocomputing.de/openbook/javainsel4/

[VMXT04] Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung (KBSt): *V-Modell® XT (Version 1.0)*. Bundesrepublik Deutschland, 2004
http://www.kbst.bund.de/static/pdf/V_Modell_XT_v1_0.pdf

[WCS96] Wall C., Christiansen T., Schwartz R.: *Programming Perl*. O'Reilly & Associates, Inc., ISBN: 1-56592-149-6, 1996

[WF74] Wagner R., Fischer M.: *The String-to-String Correction Problem*. In: Journal of the ACM (JACM) Volume 21, Issue 1, Association for Computing Machinery (ACM) (168-173),

[WMB94] Witten I., Moffat A., Bell T.: *Managing Gigabytes*. Van Nostrand Reinhold, New York, ISBN: 0-442-01863-0, 1994

[wwwApache] Apache Software Foundation: *The Apache Webserver*. (Stand: Januar 2005) http://www.apache.org/

[wwwASD] Agile Alliance: *Homepage der Agile Alliance zu "Agile Software Development"*. (Stand: Februar 2005)
http://www.agilealliance.com/

[wwwASDMani] Beck K., Fowler M. et al.: *Manifesto for Agile Software Development.* (Stand: Februar 2005) http://www.agilemanifesto.org

[wwwAviSyn] Rudiak-Gold B.: *AviSynth: A Powerful Video FrameServer for Win32.* (Stand: Februar 2005) http://www.avisynth.org/

[wwwBellm] IEEE History Center: *Legacies: Richard Bellman (aus IEEE Awards Reception Brochure 1979).* (Stand: August 2004) http://www.ieee.org/organizations/history_center/legacies/bellman.html

[wwwBibel] Luther M. (Projekt Gutenberg): *Die Bibel nach der deutschen Übersetzung D. Martin Luthers (neu durchgesehen nach dem vom Deutschen Evangelischen Kirchenausschuß genehmigten Text, 1912).* (Stand: Aug. 2004) http://gutenberg.spiegel.de/luther/bibel/bibel.htm

[wwwEclipse] Eclipse Foundation: *The Eclipse Project Homepage.* (Stand: Februar 2005) http://www.eclipse.org

[wwwEMMA] EMMA Projekt: *EMMA: a free Java code coverage tool.* (Stand: Februar 2005) http://emma.sourceforge.net

[wwwEmn04] TNS EMNID: *(N)ONLINER Atlas - Deutschlands größte Studie zur Nutzung und Nicht-Nutzung des Internets.* (Stand: Juni 2004) http://www.nonliner-atlas.de/

[wwwHash] Hash, Inc.: *Animation:Master 2004 (Homepage der Firma Hash).* (Stand: Februar 2005) http://www.hash.com

[wwwIGer98] Jacke B.: *Deutsches Wörterbuch für Ispell und Aspell nach den neuen Rechtschreibregeln.* (Stand: Februar 2005) http://j3e.de/ispell/igerman98/

[wwwIPA] IPA: *The International Phonetic Association.* (Stand: Oktober 2004) http://www.arts.gla.ac.uk/IPA/

[wwwIspell] Geoff Kuenning: *International Ispell.* (Stand: Februar 2005) http://ficus-www.cs.ucla.edu/geoff/ispell.html

[wwwJSAP] Martian Software, Inc.: *JSAP: the Java-based Simple Argument Parser.* (Stand: Februar 2005) http://www.martiansoftware.com/jsap/

[wwwJUnit] Beck K.: *JUnit: Unit-Test-Framework for Java.* (Stand: Februar 2005) http://www.junit.org

[wwwLehmann] Lehmann Chr.: *Sprachen der Welt, Veranstaltungsskript zur gleichnamigen Vorlesung.* (Stand: Okt. 2004) http://www.uni-erfurt.de/sprachwissenschaft/personal/lehmann/CL_Lehr/Spr_Welt/SW_Index.html

[wwwPerl] The Perl Foundation: *The Perl Directory at Perl.org.* (Stand: August 2004) http://www.perl.org/

[wwwPgRnk] eFactory Internet-Agentur KG: *A Survey of Google's PageRank.* (Stand: Januar 2005) http://pr.efactory.de/

[wwwRegExp] The Open Group: *IEEE Standard 1003.1 - Chapter 9: Regular Expressions.* (Stand: Januar 2005) http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html

[wwwRos00] Rosenfelder M.: *Hou tu pranownse Inglish.* (Stand: 2003) http://www.zompist.com/spell.html

[wwwSnowb] Porter M.: *Snowball: A language for stemming algorithms.* (Stand: Februar 2005) http://snowball.tartarus.org/

[wwwSpie04] Spiegel Online: *ORTHOGRAFIE / ORTHOGRAPHIE - Hitzige Debatte um Rechtschreibreform.* (Stand: August 2004) http://www.spiegel.de/kultur/gesellschaft/0,1518,312035,00.html

[wwwSprLnk] Springer-Verlag: *Springer-Link - der wissenschaftliche Online-Informationsdienst des Springer-Verlages.* (Stand: Februar 2005) http://www.springerlink.com

[wwwTomcat] Apache Software Foundation: *The Apache Jakarta Tomcat Project.* (Stand: Januar 2005) http://jakarta.apache.org/tomcat/

[wwwUnicode] Unicode, Inc.: *Was ist Unicode?.* (Stand: Januar 2005) http://www.unicode.org/standard/translations/german.html

[wwwUTF8] Wikipedia Autoren: *Das Unicode Transformation Format UTF-8.* (Stand: Februar 2005) http://de.wikipedia.org/wiki/UTF-8

[wwwVirtDub] Lee A.: *VirtualDub - A Video Capture and Processing Utility for 32-bit Windows Platforms.* (Stand: Februar 2005) http://www.virtualdub.org

[wwwVMXT] Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung (KBSt): *Webseite zum V-Modell® XT (Version 1.0)*. (Stand: Februar 2005) http://www.v-modell-xt.de/

[wwwWebstart] Sun Microsystems, Inc.: *Desktop Java: Java Web Start Technology*. (Stand: Februar 2005) http://java.sun.com/products/javawebstart/

[wwwXP] Jeffries R.: *What is Extreme Programming?*. (Stand: August 2004) http://www.xprogramming.com/xpmag/whatisxp.htm

[wwwXPUT] Jeffries R.: *XProgramming: Software Downloads*. (Stand: Februar 2005) http://www.xprogramming.com/software.htm

[ZD96] Zobel J., Dart Ph.: *Phonetic String Matching: Lessons from Information Retrieval*. In: Proceedings of the 19th International Conference on Research and Development in Information Retrieval (SIGIR'96), Zurich, ACM Press (166-172), 1996

[Zuko02] Zukowski J.: *Deploying Software with JNLP and Java Web Start*. In: Sun Developer Network (1), 2002 java.sun.com/developer/technicalArticles/Programming/jnlp/

Bildnachweis

Den folgenden Personen und/oder Institutionen sei an dieser Stelle ganz herzlich für die überlassenen Rechte zur Reproduktion der im Folgenden genannten Abbildungen im Rahmen der Veröffentlichung der vorliegenden Arbeit gedankt.

Springer-Verlag (Dr. Thomas Mager)

- Veröffentlichung meines Fotos der Enzyklopädie-Bände „*Hagers Handbuch der Pharmazeutischen Praxis*“ (Kap.9.1, S.169)
- Veröffentlichung der von mir angefertigten Bildschirmfotos sämtlicher in dieser Arbeit besprochener Anwendungsfälle aus dem Springer-Verlag (siehe Kapitel 9 „Anwendungsfälle“).
- Veröffentlichung meines Fotos des Buches „*Springer Enzyklopädie: Dermatologie, Allergologie und Umweltmedizin*“ (Kap. 9.2, S. 174)
- Offizielles PR-Foto zur Markteinführung des Produktes *„Springer Lexikon Medizin – Die DVD“* (Kap. 9.4, S. 185)

Universitätsbibliothek Würzburg (Dr. Hans-Günter Schmidt)

- Faksimile Scans der Fries-Chronik (Abb.8.6, S.161; Abb.9.20, S.196)
- Fries-Seiten als Oberflächen-Texturen des virtuellen 3D-Buches (Abb.8.4, S.158; Abb.8.7, S.162; Abb.8.8, S.163)

Universitätsbibliothek Würzburg (Irmgard Götz-Kenner)

- Foto Original-Fries-Chronik mit DVD und Notebook (Kap. 9.5, S. 194)

Rechenzentrum der Universität Würzburg (Peter Ruff)

- Foto der Zeutschel Buchwippe (Abb.8.5, S.160)