



Ähnlichkeitsbasierte Vorverarbeitung von Webseiten

Studienarbeit im Fach Informatik

*Sommersemester 2005
Friedrich-Alexander-Universität Erlangen-Nürnberg*

vorgelegt von Titus A. Miloi

(Matrikelnr. 1934340)

am 15.09.2005

Betreuer: Sebastian Schmidt

Abstract

Im Rahmen dieser Arbeit wurde ein neues Verfahren der Relevanzbewertung von Webseiten-Inhalten erarbeitet und implementiert. Ein Großteil der im Internet befindlichen Dokumente – z.B. Firmen-Websites, Online-Zeitungen und -Zeitschriften oder Internet-Shops – weisen innerhalb des Netzpades, in dem sie sich befinden – ein Netzpfad sei dabei der Teil der URL bis zum am weitesten rechts befindlichen Slash, also die URL ohne den Dateinamen der Seite –, weitgehend gleiche oder sehr ähnliche Strukturen auf, z.B. gemeinsame Teilbereiche wie Navigationsleisten, Menüs, Werbung, Copyright-Hinweise, etc. Diese Bereiche tragen unwesentlich oder gar nicht zum eigentlichen Inhalt der jeweiligen Seite bei. Deshalb ist es für viele Anwendungen, wie z.B. Suchmaschinen, von Interesse, welche Teile einer Webseite eben dieses Relevanzkriterium erfüllen. Die hier implementierten Algorithmen sollen also anhand von Ähnlichkeitsanalysen zwischen Seiten des gleichen Netzpades Teilbereiche in Webseiten gewichten und anschließend, abhängig von der vorgenommenen Gewichtung, die als irrelevant eingestuften Bereiche herausfiltern, so, dass im Ergebnis nur der für relevant befundene Inhalt der Seite übrig bleibt.

Inhaltsverzeichnis

1 Einleitung.....	4
2 Vergleichbare Arbeiten.....	6
3 Ähnlichkeitsbasierte Vorverarbeitung von Webseiten.....	8
3.1 Auffinden sinnvoller Einheiten in einer Webseite.....	9
3.2 Ermittlung der erforderlichen Vergleichs-Seiten.....	12
3.3 Die Levenshtein-Distanz als Ähnlichkeitsmaß.....	14
3.4 Wortzählung als Ähnlichkeitsmaß.....	18
3.5 Gegenüberstellung der Laufzeiten.....	21
4 Die Java-Klassenbibliothek: org.sbfilter.....	24
4.1 Kurzreferenz.....	24
4.2 Die wichtigsten Klassen.....	27
4.3 Grabben und Parsen von Webseiten.....	29
4.4 Interne Darstellung des DOM-Baums.....	29
4.5 Implementierung der Analyseverfahren.....	31
5 Auswertung der Ergebnisse.....	33
6 Zusammenfassung und Ausblick.....	41
Literaturverzeichnis.....	43
Eidesstattliche Versicherung.....	44

1 Einleitung

Durch die unüberschaubar große und ständig wachsende Informationsvielfalt, die das Internet heute bietet, werden die Anforderungen größer, die an Web Information Retrieval Systeme gestellt werden. Internet-Suchmaschinen werden fortlaufend verbessert, neue Crawler, Suchalgorithmen und Relevanz-Bewertungsverfahren werden implementiert, um die Internet-Suche effizienter und exakter zu machen und die menschlichen Erwartungen, die an die Suchergebnisse gestellt werden, so gut wie möglich zu erfüllen.

Diese Arbeit behandelt einen neuen Ansatz der Relevanzbewertung von Webseiten-Inhalten. Ein Großteil der im Internet befindlichen Dokumente – z.B. Firmen-Websites, Online-Zeitungen und -Zeitschriften oder Internet-Shops – weisen innerhalb des Netzpfades, in dem sie sich befinden – ein Netzpfad sei dabei der Teil der URL bis zum am weitesten rechts befindlichen Slash, also die URL ohne den Dateinamen der Seite –, weitgehend gleiche oder sehr ähnliche Strukturen auf, z.B. gemeinsame Teilbereiche wie Navigationsleisten, Menüs, Werbung, Copyright-Hinweise, etc. Diese Bereiche tragen unwesentlich oder gar nicht zum eigentlichen Inhalt der jeweiligen Seite bei. Deshalb ist es für viele Anwendungen, wie z.B. Suchmaschinen, von Interesse, welche Teile einer Webseite eben dieses Relevanzkriterium erfüllen.

Im Rahmen dieser Arbeit wurde eine Java-Klassenbibliothek implementiert, die es ermöglicht, eine gegebene Webseite zu untersuchen, auf ihr verlinkte Seiten im gleichen Netzpfad weiter zu verfolgen, und durch die Analyse der untersuchten Seiten Relevanzmaße für einzelne Teile der gegebenen Seite zu berechnen. Für die Analyse wurden zwei voneinander unabhängige Vergleichsverfahren implementiert, deren Algorithmen im weiteren Verlauf dieser Arbeit näher betrachtet werden sollen. Die Genauigkeit der Ergebnisse wurde mit einem Korpus von 50 stichprobenartig ausgewählten Webseiten berechnet, die manuell (und intuitiv) von Kommilitonen auf

die Relevanz ihrer einzelnen Seitenteile hin untersucht und bewertet wurden. Der Vergleich der menschlich ausgewerteten Seiten mit den Ergebnissen der beiden implementierten Algorithmen sollte ein zuverlässiges Maß für die Genauigkeit derselben liefern.

Es sei noch ausdrücklich auf den Unterschied zwischen „Webseite“ und „Website“ hingewiesen, letzteres die Gesamtheit aller inhaltlich zusammenhängenden, miteinander verlinkten und sich im selben Netzpfad befindlichen Webseiten bezeichnend. Es wird im Folgenden stattdessen auf den Begriff „Webpräsenz“ ausgewichen, um Verwechslungen zu vermeiden.

Im nächsten Abschnitt werden kurz vergleichbare Arbeiten vorgestellt, oder solche, die mit Verfahren arbeiten, die mit den hier verwendeten Ähnlichkeiten aufweisen. Im 3. Kapitel wird die Idee und das Implementierungskonzept der vergleichsbasierten Relevanzbewertung von Webseiten-Inhalten vorgestellt. Kapitel 4 geht näher auf die Implementierung der Java-Klassenbibliothek sowie auf die Strategien bei der Umsetzung der Vergleichsverfahren ein. Im 5. Kapitel wird die Untersuchung der Genauigkeit der Ergebnisse vorgestellt. Der abschließende 6. Teil gibt einen Ausblick auf mögliche Weiterentwicklungen und Verbesserungen sowie auf interessante Anwendungsgebiete für dieses Verfahren.

2 Vergleichbare Arbeiten

Es gibt bereits eine Reihe von Arbeiten, die sich mit der Vorverarbeitung von Webseiten bzw. der Relevanzbewertung von Seitenabschnitten auseinandersetzen. Die Analyse der Struktur von Webseiten mit Hilfe von Tag-Typen bzw. deren Zerlegung laut Document Object Model [1], einem Modell zur objektorientierten Darstellung von HTML/XHTML-Seiten-Inhalten (nähere Erläuterungen folgen später), haben sich bereits etliche wissenschaftliche Arbeiten zu Nutze gemacht. So wird beim 'Vision Based Page Segmentation' [3] Verfahren von Deng Cai et al. die DOM-Zerlegung von Seiten dazu verwendet, in einer Browser-Darstellung visuell voneinander getrennte Seitenbereiche zu identifizieren, um sie nach ihrer Position im Browserfenster zu gewichten: je zentraler ein Bereich positioniert ist, desto höher wird er gewichtet. Dies ist eine andere Herangehensweise für die gleiche Problemstellung, die auch in dieser Arbeit behandelt wird.

Die Levenshtein-Distanz (siehe Kap. 3.3) als Metrik für die Ähnlichkeit von Webseiten wurde bereits vereinzelt für unterschiedliche Problemstellungen aufgegriffen, z.B. für die Extraktion von Informationen aus ähnlichen Seitenstrukturen in 'Mining Data Records in Web Pages' [6] von Bing Liu et al., wurde bisher aber zum Zweck der Relevanzbewertung von Seitenbereichen nicht angewandt. Ein anderes Ähnlichkeitsmaß wird bei William W. Cohen [5], 'Recognizing Structure in Web Pages Using Similarity Queries', verwendet: die von Salton [7] vorgeschlagene Verwendung eines themengebundenen Thesaurus mit gewichteten Begriffen, wobei eine Zeichenkette als ein Vektor v der Länge n von reellen Zahlen dargestellt wird, n die Anzahl der Wörter in der Zeichenkette ist und v_i die Gewichtung des Wortes i laut Thesaurus darstellt. Für solche Vektoren kann dann eine gängige Metrik (z.B. der euklidische Abstand) als Ähnlichkeitsmaß benutzt werden. Auf diese Weise ist es nicht nur möglich, Zeichenketten miteinander zu vergleichen, sondern vielmehr noch, diese in

Themengruppen einzuteilen und so inhaltlich zuzuordnen. Diese Metrik in Verbindung mit einem erweiterbaren, lernfähigen Thesaurus bietet sich hervorragend als eine Erweiterung der vorliegenden Arbeit an; vorstellbar wäre damit z.B. das automatische Auffinden thematisch verwandter Inhalte bereits gefilterter Webseiten. Dieser Ansatz wird jedoch nicht weiter ausgeführt, da dies den Rahmen dieser Arbeit sprengen würde. Dennoch wurde vor allem für solche Fälle, aber auch für die Entwicklung neuer Ideen und Algorithmen, die in dieser Arbeit implementierte Klassenbibliothek erweiterbar und ausbaufähig entworfen, so dass weitere Entwicklungen mit möglichst wenig Implementierungsaufwand realisiert werden können.

3 Ähnlichkeitsbasierte Vorverarbeitung von Webseiten

Die Ähnlichkeit von Webseiten innerhalb einer Webpräsenz ist ein Kriterium, das in bisherigen Untersuchungen der Relevanz von einzelnen Seitenbereichen innerhalb einer Seite und bei der Vorverarbeitung von Webseiten wenig beachtet wurde. Dennoch kann eine Untersuchung dieser Ähnlichkeit Aufschluss über die Relevanz der erwähnten Seitenbereiche geben. Der Gedanke dahinter ist, dass innerhalb einer Webpräsenz sich wiederholende oder nur schwach variierende Seitenabschnitte weniger zum relevanten Inhalt der untersuchten Seite beitragen als Abschnitte, die sich selten wiederholen oder gar einzigartig sind. Ein sehr anschauliches Beispiel für Abschnitte, die sich auf allen Seiten einer Webpräsenz in gleicher oder leicht veränderter Form wiederfinden lassen, sind Navigationselemente. Diese würde man auch intuitiv sicherlich nicht zum Inhalt einer Webseite zählen, da sie wegen ihres unveränderten Auftretens auf allen Webseiten offensichtlich inhaltlich nicht mit Seitenabschnitten zusammenhängen können, die auf der untersuchten Seite einzigartig sind. Die Abbildungen 3/1 und 3/2 zeigen zwei Seiten mit unterschiedlichem Inhalt, die jedoch zur gleichen Webpräsenz gehören, und veranschaulichen die Idee, sich wiederholende Seitenabschnitte innerhalb einer Webpräsenz als für den Seiteninhalt irrelevant einzustufen.



Abb. 3/1 Ein Artikel von www.stiftung-warentest.de

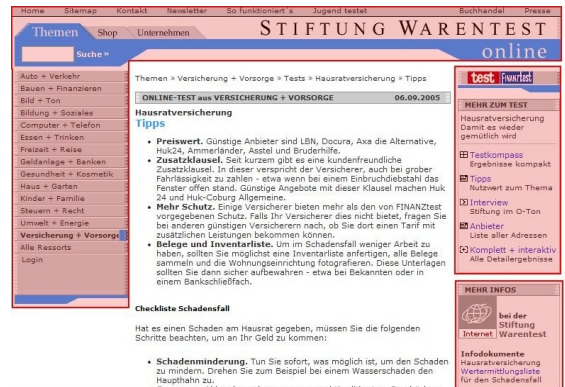


Abb. 3/2 Eine weitere Seite der Gleichen Webpräsenz wie Abb. 3/1

Die markierten Bereiche sind nahezu identisch, während die relevanten Bereiche – also die Artikel selbst – sich bei den beiden ausgesuchten Seiten offensichtlich deutlich unterscheiden.

Ein Verfahren, das eine solche Gewichtung von Webseiten-Abschnitten automatisiert vornehmen kann, wäre in vielerlei Hinsicht von Nutzen, sei es für die gezielte Extraktion von Informationen von Webseiten oder aber für die Vorverarbeitung von Webseiten für die Suchmaschinen-Indizierung, um genauere Suchergebnisse zu erzielen.

3.1 Auffinden sinnvoller Einheiten in einer Webseite

Das Aufteilen einer Webseite in die oben erwähnten Seitenabschnitte, die für die Ähnlichkeitsuntersuchung herangezogen und gewichtet werden sollen – im Folgenden Content-Einheiten oder Einheiten genannt – kann auf unterschiedliche Weise erfolgen. Als hilfreiches Konstrukt für diesen Zweck erweist sich das bereits erwähnte Document Object Model [1] des w3 Konsortiums. Beim Document Object Model (DOM) handelt

es sich um ein Modell zur objektorientierten Darstellung von HTML/XHTML-Seiten-Inhalten. Abbildung 3.1/1 zeigt den sog. DOM-Baum einer HTML-Seite, ein azyklischer, gerichteter Graph (Kantenrichtung implizit von oben nach unten), dessen Knoten die Tags der Seite und dessen Kanten das Verschachtelungsverhältnis der Tags zueinander darstellen.

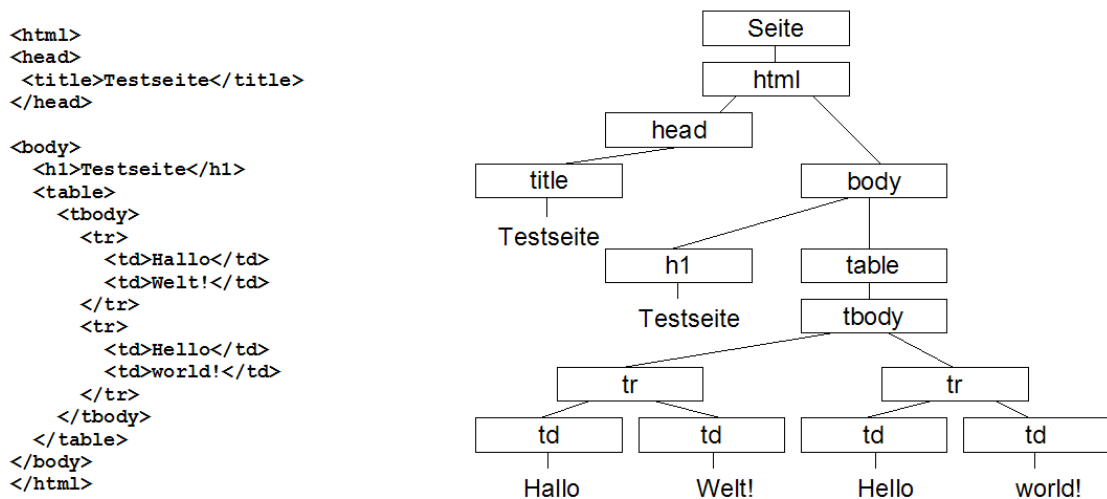


Abb. 3.1/1 Beispiel-DOM-Baum (Tag-Attribute nicht dargestellt)

Jede HTML/XHTML-Seite lässt sich (bijektiv) auf einen DOM-Baum abbilden.

Da die Tag-Attribute nicht zum Inhalt sondern zur Darstellung der Seite beitragen, sind sie für unsere Zwecke irrelevant und können verworfen werden. Auch Java-Scripts, Meta-Tags, Kommentare, u.Ä. können entfernt werden (die Bijektivität geht hierbei natürlich verloren). Relevant ist ausschließlich der <body>-Tag und die darin enthaltenen Tags.

Bei einer solchen vereinfachten Darstellung von Webseiten kann somit der body-Tag einer Seite als größtmögliche Content-Einheit, die einzelnen Baumknoten, die Text enthalten, als kleinste Einheiten betrachtet werden. Dass ersteres nicht sinnvoll ist, leuchtet ein, da die undifferenzierte Gewichtung des gesamten Seiteninhaltes nicht in

unserem Sinne ist. Das Heranziehen aller Text enthaltenden Knoten als kleinste Content-Einheiten würde intuitiv in einigen Fällen funktionieren, ist jedoch auch selten sinnvoll, da bestimmte Tags wie z.B. `
`, ``, `<a>`, etc. inhaltlich zusammenhängenden Text in mehrere Knoten unterteilen würde, die dann getrennt untersucht und gewichtet würden, was dazu führte, dass z.B. einzelne Teile eines relevanten Abschnittes möglicherweise geringer gewichtet und ausgeblendet würden. Es ergibt sich also die Notwendigkeit, nur ganz bestimmte Tags bzw. Knoten als Content-Einheiten aus einem DOM-Baum zu extrahieren bzw. die nicht relevanten Knoten auszublenden. Da kein Inhalt verloren gehen darf, dürfen keine Text enthaltenden Knoten ausgeblendet werden. Dies kann erreicht werden, indem Textknoten als Sonderfälle relevanter Knoten markiert werden und rekursiv alle nicht relevanten Knoten aus dem Baum entfernt werden, nachdem zuvor deren Kindknoten dem jeweiligen Elternknoten zugewiesen wurden.

Eine inhaltliche Unterteilung von Webseiten geschieht oft durch Tabellenzellen, Absätze `<p>`, Überschriften oder Abschnitts-Markierungen `<div>`. Daher betrachten wir die folgenden Tags als Content-Einheiten: `body`, `table`, `tbody`, `td`, `tr`, `div`, `p`, `form`, `h1-h3`. Abweichungen hiervon wären auch denkbar, jedoch hat sich diese Auswahl in der Praxis als sinnvoll erwiesen.

Der Algorithmus, der aus einem nicht attributierten DOM-Baum, also einem DOM-Baum ohne Tag-Attribute, sinnvolle Content-Einheiten extrahiert, sieht in Pseudo-Code wie folgt aus:

```
relevant = (body, table, tbody, td, tr, div, p, form, h);

foreach node in domTree do
  if not node in relevant do
    replace node with node.children;
  end if;
```

`end foreach;`

Man beachte, dass die Reihenfolge der Knoten von Bedeutung ist; deshalb ist es wichtig, die Kindknoten des zu entfernenden Knotens genau an dessen Position einzusetzen.

Die folgende Abbildung zeigt einen veränderten DOM-Baum, dessen Knoten nun nur die relevanten Tags darstellen:

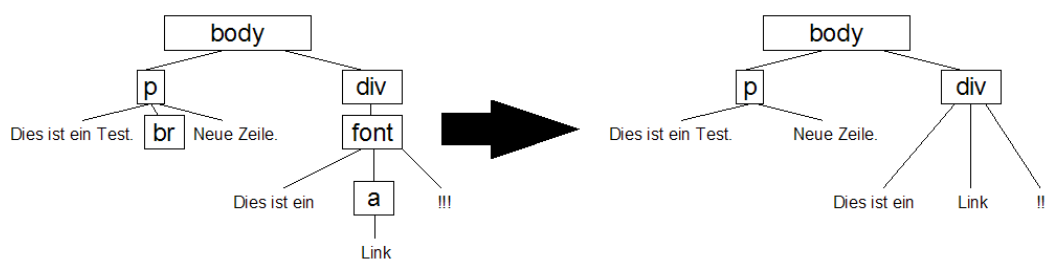


Abb. 3.1/2 Links: Ausschnitt eines DOM-Baums; Rechts: vereinfacht

Zur weiteren Vereinfachung kann man nun noch den Inhalt unmittelbar benachbarter Textknoten konkatenieren (ggf. mit Leerzeichen dazwischen) und zu einem einzigen Textknoten zusammenfassen; diese Operation vereinfacht die Struktur des Baumes und beeinträchtigt nicht die für spätere Ähnlichkeitsanalysen erforderliche, darin enthaltene Text-Information.

3.2 Ermittlung der erforderlichen Vergleichs-Seiten

Wie bereits erwähnt, werden zur Ähnlichkeitsanalyse der aus einer Zielseite gewonnenen Content-Einheiten, Webseiten derselben Webpräsenz benötigt – Vergleichs-Seiten, bei denen man von einer ähnlichen Struktur wie die der Hauptseite

ausgehen kann. Zwar stimmen nicht immer die Seitenstrukturen von Webseiten der gleichen Webpräsenz überein, doch stellt das für unser Verfahren kein echtes Problem dar: eine geringere strukturelle Übereinstimmung führt zu höheren Relevanzen für potentiell irrelevante Einheiten und damit zu ungenaueren Ergebnissen, jedoch niemals zu Fehlern, sprich, zu fälschlicherweise zu niedrig gewichteten Einheiten, und damit möglicherweise zur fehlerhaften Ausblendung derselben. Demnach ist die Verwendung von Seiten der gleichen Webpräsenz eine sinnvolle Vorgehensweise.

Die Gewinnung dieser Vergleichs-Seiten stellt sich bei der Betrachtung der Hauptseite als wenig aufwändig heraus: man extrahiere vor dem Verwerfen der Tag-Attribute das href-Attribut jedes <a>-Knotens im DOM-Baum der Hauptseite und ermittle daraus die absoluten Adressen, auf die diese Links verweisen (ggf. ist eine Änderung des Base-Pfades durch einen <base>-Tag zu beachten). Dann verwerfe man diejenigen Adressen, deren Pfad (ohne Dokumentname) nicht mit dem der Adresse der Hauptseite übereinstimmt. Diese Vorgehensweise kann auch auf <frame>-Tags angewandt werden.

Hier die Link-Extraktion in Pseudo-Code:

```
set base = mainPage.path;
foreach node in domTree do
  if node is baseTag
    set base = node.attribute("target");
  else if node is aTag
    load_page makeAbsolute(base, node.attribute("href"));
  else if node is frameTag
    load_page makeAbsolute(base, node.attribute("src"));
  end if;
end foreach;
```

Das target-Attribut des <base>-Tags kann neben dem base-Pfad die Schlüsselwörter `_blank`, `_top`, `_parent` oder `_self` enthalten – diese Fälle müssen gesondert behandelt

werden. Um einen größeren Vorrat an Vergleichsseiten zu erhalten, kann die Link-Extraktion auch rekursiv auf die bereits so gewonnenen Seiten angewandt werden, bis z.B. eine bestimmte Anzahl von Seiten gewonnen oder eine vorgegebene Rekursionstiefe erreicht wird.

3.3 Die Levenshtein-Distanz als Ähnlichkeitsmaß

Wir wollen als nächstes versuchen, für jede Content-Einheit unserer Hauptseite zu ermitteln, in wie vielen Vergleichs-Seiten sich diese wiederfinden lässt. Die Relevanz unserer Einheit errechnet sich dann indirekt proportional aus der Anzahl der Funde. Da in den meisten Fällen damit zu rechnen ist, dass sich bestimmte Einheiten nicht in völlig unveränderter Form auf unseren Vergleichs-Seiten wiederholen, sondern leichte Variationen durchaus üblich sind – z.B. können in einem Navigationsmenü auf jeder Seite der jeweils entsprechende Menüpunkt hervorgehoben werden oder Menü-Unterpunkte hinzugefügt werden – würde eine auf einer einfachen Gleichheits-Überprüfung basierende Suche keine guten Ergebnisse liefern. Wir benötigen daher ein Maß für die Ähnlichkeit zweier Einheiten.

Die Levenshtein-Distanz (LD) zweier Zeichenketten beschreibt die minimale Anzahl der Änderungen – Zeicheneinfügungen, -löschungen oder -ersetzungen –, die nötig sind, um eine Zeichenkette in die andere zu überführen. Somit wäre diese eine brauchbare Größe zur Berechnung eines Ähnlichkeitsmaßes für den in Content-Einheiten enthaltenen Text. Um nun ein normiertes Maß aus der LD zu erhalten, muss noch die Wertemenge des Levenshtein-Algorithmus $[0, \infty)$ auf das Intervall $[0, 1]$ abgebildet werden. Hierfür muss die ermittelte Distanz durch die maximal mögliche Distanz der beiden Eingabe-Zeichenketten geteilt werden.

Die minimale Distanz zweier Zeichenketten beträgt immer 0 – in diesem Falle sind die beiden Eingabe-Zeichenketten gleich. Die maximale Distanz ist abhängig von den Eingabe-Zeichenketten und ist gleich mit der Länge der längeren Zeichenkette; zum Verständnis: sie ergibt sich aus der maximalen Anzahl der Ersetzungen, die nötig sind, um aus der kürzeren Zeichenkette die ebenso lange Teil-Zeichenkette der längeren Zeichenkette zu bilden plus die Anzahl der Einfügungen, die danach noch nötig sind, um die gesamte längere Zeichenkette zu erhalten (siehe Abb. 3.3/1).

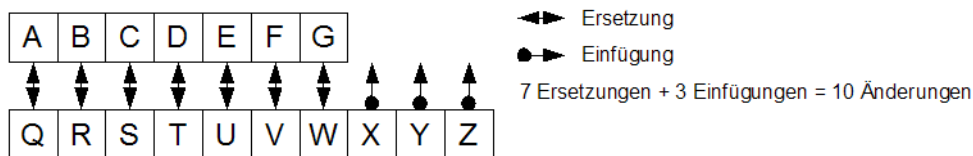


Abb. 3.3/1 Der maximale Levenshtein-Abstand

Wir definieren nun die Funktion $lev(s_1, s_2)$ zur Berechnung der Levenshtein-Distanz zweier Zeichenketten s_1 und s_2 :

$$c(i, j) = \begin{cases} 1 & \text{falls } s_{1i} \neq s_{2j}; \\ 0 & \text{sonst;} \end{cases}$$

$$M \in \mathbb{N}_0^{(m+1) \times (n+1)};$$

$$M_{x,1} = x - 1; \quad M_{1,y} = y - 1; \quad x = 1..m + 1; \quad y = 1..n + 1;$$

$$M_{u,v} = \min(M_{u-1,v} + 1, M_{u,v-1} + 1, M_{u-1,v-1} + c(u-1, v-1));$$

$$u = 2..m + 1; \quad v = 2..n + 1;$$

$$lev(s_1, s_2) = M_{m+1, n+1}$$

Dabei seien m und n die Längen der Zeichenketten s_1 bzw. s_2 , s_{ab} das b -te Zeichen der

Zeichenkette s_a und $\min(a, b, c)$ die Minimumsfunktion für drei ganzzahlige Werte.

Betrachten wir die obige Definition genauer: die Funktion c gibt an, ob zwei Zeichen s_{1x} und s_{2y} der Zeichenketten s_1 und s_2 gleich sind. Der erste Zeilen- und Spaltenvektor der Matrix M wird mit ganzen Zahlen von 0 bis m bzw. von 0 bis n initialisiert. Alle anderen Zellen von M werden nun rekursiv definiert, und zwar so, dass $M_{u,v}$ immer die kleinstmögliche Anzahl von Veränderungen beinhaltet, die benötigt werden, um die beiden Zeichenketten bis zur Position $u-1$ bzw. $v-1$ anzugleichen, denn:

- $M_{u-1,v}+1$ gibt die Anzahl der benötigten Veränderungen bis zum $u-2$ ten Zeichen von s_1 an plus einer Veränderung – also die Anzahl der benötigten Veränderungen beim Entfernen des $u-1$ ten Zeichens aus s_1 .
- $M_{u,v-1}+1$ gibt die Anzahl der benötigten Veränderungen bis zum $v-2$ ten Zeichen von s_2 an plus einer Veränderung – also die Anzahl der benötigten Veränderungen beim Entfernen des $v-1$ ten Zeichens aus s_2 oder anders gesagt, beim Einfügen eines Zeichens in s_1 an der Position $u-1$.
- $M_{u-1,v-1}+c(u-1,v-1)$ gibt die Anzahl der bisher benötigten Veränderungen an plus einer Veränderung, falls die Zeichen von s_1 und s_2 an der gegebenen Position nicht gleich sind, also falls eine Ersetzung erfolgen muss.

Das Minimum dieser drei Werte wird in die Matrix M an die Position u,v eingetragen. In der rechten, unteren Zelle der Matrix M , also an Position $m+1,n+1$ erhalten wir nun die minimale Anzahl der Veränderungen, die benötigt werden, um s_1 und s_2 anzugleichen.

Das gewünschte Ähnlichkeitsmaß für zwei Zeichenketten errechnet sich nun daraus durch Normierung – also Projektion des Wertebereichs der LD auf $[0, 1]$:

$$nlev(s_1, s_2) = \frac{lev(s_1, s_2)}{\max(len(s_1), len(s_2))}; nlev(s_1, s_2) \in [0, 1]$$

$len(x)$ sei dabei die Anzahl der in der Zeichenkette x enthaltenen Zeichen und $max(a, b)$ die Maximums-Funktion.

Durch Konkatination aller in einer Einheit enthaltenen Zeichenketten erhalten wir die Zeichenketten, die wir zur Berechnung der normierten LD heranziehen können.

Wir betrachten nun eine einzelne Einheit e in unserer Hauptseite h und wollen deren Relevanz berechnen. Hierfür ermitteln wir für jede Vergleichs-Seite $v_i, i=1..n$, die normierte LD der darin enthaltenen Einheit, die unserer Einheit am ähnlichsten ist – also das Minimum der normierten LD zwischen e und allen Einheiten in v_i . Aus den erhaltenen Minima aller v_1 bis v_n bilden wir schließlich den Durchschnitt und invertieren. Der so erhaltene Wert stellt eine brauchbare Relevanz-Gewichtung für unsere Einheit dar. Zusammengefasst:

$$rel_e = 1 - \frac{\sum_{v=1}^n \min(nlev(s_e, t_{v1}), nlev(s_e, t_{v2}), nlev(s_e, t_{v3}), \dots, nlev(s_e, t_{vm}))}{n}$$

rel_e sei die Relevanz der Einheit e der Hauptseite h , s_e die konkatenierte Zeichenkette der Einheit e , t_{vi} die konkatenierte Zeichenkette der Einheit i der Vergleichs-Seite v , m_v die Anzahl der Einheiten in der Vergleichs-Seite v und n die Anzahl aller Vergleichs-Seiten.

Der Algorithmus zur Berechnung der Relevanzen für alle Einheiten der Hauptseite sieht in Pseudo-Code wie folgt aus:

```

read in mainPage;
numberOfRefpagesReadIn = 0;

do

```

```

read in refPage;
foreach node in mainPage do
  node.relevance = 0;
  minSimilarity = 1;
  foreach refNode in refPage do
    minSimilarity =
      min(minSimilarity, nlev(node.text, refNode.text));
  end foreach;
  node.relevance += (1 - minSimilarity);
end for;
increase numberOfRefpagesReadIn by 1;
while not all needed pages read in;

foreach node in mainPage do
  node.relevance = node.relevance / numberOfRefpagesReadIn;
end foreach;

```

Der Pseudo-Code der normierten Levenshtein-Distanz-Funktion `nlev()` soll an dieser Stelle nicht näher erläutert werden, da die formale Definition den Algorithmus bereits hinreichend beschreibt.

3.4 Wortzählung als Ähnlichkeitsmaß

Ein anderes Maß zur Berechnung der Relevanz einer Content-Einheit, das wir genauer betrachten möchten, ist die Anzahl der Vergleichs-Seiten, in denen sich jedes einzelne „Wort“ einer Einheit wiederfinden lässt. Mit „Wort“ ist in diesem Fall ein Teilstück der konkatenierten Zeichenkette einer Einheit gemeint, das von Leerzeichen oder vom Zeichenkettenanfang bzw. -ende eingeschlossen ist und selbst keine Leerzeichen enthält

(welche Zeichen als „Leerzeichen“ betrachtet werden können, wird im Implementierungsteil erwähnt; hier soll das „Leerzeichen“ im intuitiven Sinne aufgefasst werden).

Die Idee hinter diesem Ansatz ist, dass eine Wortzählung eine wesentlich billigere Operation ist als die Ermittlung der Levenshtein-Distanzen für jedes Einheiten-Paar – die Laufzeit der Ähnlichkeitsanalyse also wesentlich verkürzt – und dennoch eine annähernde Ergebnisgenauigkeit liefern kann wie das LA-Verfahren. Dass die Praxis diese Vermutung mehr als bestätigt, wird die Auswertung der Ergebnisse zeigen.

Betrachten wir nun wiederum eine einzelne Einheit e unserer Hauptseite h und deren konkatenierte Zeichenkette s_e . Wir zerlegen die Zeichenkette s_e in die Wörter w_{e1} bis w_{en} wie oben beschrieben. Als nächstes definieren wir eine Funktion $c(w, s)$, die angibt, ob das Wort w in der Zeichenkette s enthalten ist:

$$c(w, s) = \begin{cases} 1 & \text{falls } w \in s; \\ 0 & \text{sonst;} \end{cases}$$

Die Relevanz der Einheit e lässt sich wie folgt berechnen:

$$rel_e = 1 - \frac{\sum_{i=1}^n \sum_{j=1}^m c(w_{ei}, s_{vj})}{n \cdot m}$$

Dabei sei m die Anzahl aller Vergleichs-Seiten, $v_i, i=1..m$ eine Vergleichs-Seite und s_v der Text-Anteil der gesamten Vergleichs-Seite v .

Wir zählen also für jedes Wort einer Einheit die Vergleichs-Seiten, in denen es vorkommt, teilen dies durch die Gesamtzahl aller Vergleichs-Seiten, um ein normiertes

Maß mit Wertebereich $[0, 1]$ zu erhalten, und bilden den Durchschnitt dieses Maßes über alle Wörter der Einheit. Schließlich wird invertiert, um aus dem Ähnlichkeitsmaß ein Relevanzmaß zu machen.

Durch die Suche der Wörter einer Einheit der Hauptseite in der gesamten Vergleichs-Seite entfällt in der Praxis die Notwendigkeit, die Vergleichs-Seiten als DOM-Bäume darzustellen – eine Volltext-Suche innerhalb des Text-Anteils der Vergleichs-Seiten genügt. Bei der Implementierung werden wir zudem die immer noch relativ teure Volltext-Suche durch deutlich billigere String-Vergleiche zwischen einzelnen Wörtern ersetzen, indem wir die Vergleichsseiten vor der Analyse in Wörter parsen. Dadurch kann die Laufzeit dieses Verfahrens nochmal reduziert werden.

Es folgt der dazu gehörige Algorithmus in Pseudo-Code:

```
read in mainPage;
numberOfRefpagesReadIn = 0;

do
  read in refPage;
  foreach node in mainPage do
    foreach word in node do
      if word in refpage.text do
        increase node.relevance by 1;
      end if;
    end foreach;
  end foreach;
  increase numberOfRefpagesReadIn by 1;
while not all needed pages read in;

foreach node in mainPage do
  node.relevance = node.relevance /
    #(node.words) * numberOfRefpagesReadIn;
```

`end foreach;`

3.5 Gegenüberstellung der Laufzeiten

Als Kriterien für die Bewertung der beiden Algorithmen werden die Laufzeit und die Ergebnisgenauigkeit derselben betrachtet werden – letzteres wird in Kapitel 5 anhand konkreter Zahlen untersucht und soll hier nicht weiter erörtert werden.

Bei der Laufzeit der beiden Algorithmen lassen sich deutliche Unterschiede erkennen. Wir betrachten zunächst den Pseudo-Code des WA: er benötigt in der innersten Schleife, wie bereits erwähnt, eine Wort-Suche, die in unserer Implementierung aus einer Gleichheitsüberprüfung von Zeichenketten in einer Schleife besteht. Diese besitzt das Laufzeitverhalten $O(z_w \cdot w_s)$, wobei z_w die mittlere Anzahl der Zeichen in einem Wort und w_s die mittlere Anzahl der Wörter einer Seite sei. Das ergibt also $O(z_s)$, z_s sei dabei die mittlere Anzahl der Zeichen in einer Seite.

Da die innerste Schleife von der Anzahl der Wörter in einer Einheit, und die zweite Schleife von innen von der Anzahl der Einheiten in der Hauptseite abhängt, kann man abschätzen, dass beide Schleifen zusammen $w_e \cdot e_s$, also w_s mal durchlaufen. Man verstehe die Symbole analog zur obigen Verwendung, d.h. die mittlere Anzahl des mit dem Symbolbuchstaben angegebenen Objekts (z Zeichen, w Wörter, e Einheiten, s Seiten) pro Objekt, das durch den Indexbuchstaben angegeben wird.

Die äußerste Schleife hängt von der gesamten Anzahl s der Vergleichs-Seiten ab. Also ergibt sich für den gesamten Algorithmus (ohne das Zerlegen der Zeichenketten in Wörter) das Laufzeitverhalten:

$$O(z_s \cdot w_s \cdot s) = O\left(z_s \cdot \frac{z_s}{z_w} \cdot s\right) = O\left(\frac{z_s^2 \cdot s}{z_w}\right)$$

Die Laufzeit, die für das Parsen der Zeichenketten in Wörter benötigt wird, ist linear und fällt nur ein mal pro betrachteter Vergleichs-Seite an, fällt somit also nicht ins Gewicht.

Um das Laufzeitverhalten des LA abzuschätzen, betrachten wir zunächst die Laufzeit, die für die Berechnung der LD zweier Zeichenketten anfällt. In der einfachsten Implementierung, die hier betrachtet werden soll, wird für die Zeichenketten s_1 und s_2 eine $(n+1) \times (m+1)$ -Matrix M benötigt; dabei sei n die Länge von s_1 und m die von s_2 . Der erste Zeilenvektor wird, wie in Abschnitt 3.3 beschrieben, mit $0..n$ initialisiert, der erste Spaltenvektor mit $0..m$. Die übrigen Zellen berechnen sich wie folgt:

$$c(i, j) = \begin{cases} 1 & \text{falls } s_{1i} \neq s_{2j}; \\ 0 & \text{sonst;} \end{cases}$$

$$M_{u,v} = \min(M_{u-1,v} + 1, M_{u,v-1} + 1, M_{u-1,v-1} + c(u-1, v-1));$$

$$u = 2..m+1; v = 2..n+1;$$

Demnach müssen also für jede Zelle der Matrix außer dem ersten Zeilen- und Spaltenvektor ein Zeichenvergleich, eine Minimumsermittlung, drei Leseoperationen, sieben Additionen und eine Schreiboperation durchgeführt werden. Wir erhalten also ein Laufzeitverhalten von $\Theta(m \cdot n) = \Theta(z_c^2)$.

Die innerste Schleife des Levenshtein-basierten Ähnlichkeitsanalyse-Algorithmus durchläuft alle Einheiten einer Vergleichs-Seite; die darüber liegende Schleife durchläuft wiederum alle Einheiten der Hauptseite. Die äußerste Schleife durchläuft –

wie auch beim Wortzählungs-Algorithmus – alle Vergleichs-Seiten. So erhalten wir also ein Laufzeitverhalten für den Gesamten Algorithmus von:

$$\Theta(z_e^2 \cdot e_s \cdot e_s \cdot s) = \Theta(z_e^2 \cdot e_s^2 \cdot s) = \Theta(z_s^2 \cdot s)$$

Der WA ist also im Worst Case um den Faktor $\frac{1}{z_w}$ besser als der Levenshtein-Algorithmus und sollte im Average Case zudem noch deutlich darunter liegen. Hinzu kommt, dass es bei der Implementierung des Levenshtein-Algorithmus in der innersten Schleife de facto mehrerer zeitintensiver Operationen bedarf (wie z.B. das allozieren von Speicher für die Matrix), als bei der Wortzählung. Wir werden bei der Implementierung feststellen, dass dies bei Webseiten durchschnittlicher Größe bereits schwer ins Gewicht fällt.

Dennoch gibt es auch beim LA verschiedene Ansätze, um ihn effizienter zu machen. Eine effizientere Implementierung des Algorithmus zur LD-Berechnung sollte deutliche Laufzeiteinsparungen mit sich bringen. Eine weitere, sehr einfach umzusetzende Methode ist, nur für die Einheiten die LD zu berechnen, deren DOM-Knoten aus den gleichen Tags hervorgehen – z.B. <tr>-Knoten nur mit anderen <tr>-Knoten zu vergleichen – da man Ähnlichkeiten zwischen den Inhalten ohnehin unterschiedlicher Tags von vornherein ausschließt. Damit würde man für den LA das Laufzeitverhalten $O(z_s^2 \cdot s)$ erhalten und käme dem WA deutlich näher.

4 Die Java-Klassenbibliothek: org.sbfilter

Die oben beschriebenen Analyseverfahren wurden sowohl aus Gründen der Portierbarkeit als auch wegen des sauberen, gut lesbaren objektorientierten Codes, den man darin schreiben kann, vollständig in Java implementiert. Der gesamte Quellcode der Klassenbibliothek kann jederzeit als CVS Snapshot aus dem Internet von folgender Adresse geladen werden:

```
CVSPATH=:pserver:fauki@admin.miloi.de:4450/cvsroot
Passwort: vieW#8SB@
```

In den folgenden Abschnitten wird eine Kurzanleitung zur Verwendung der Klassenbibliothek per Kommandozeile und als API für eigene Java-Projekte gegeben. Anschließend werden die wichtigsten in der Bibliothek enthaltenen Klassen kurz vorgestellt, sowie die Implementierungsstrategien erklärt, die im Rahmen dieses Projekts angewandt wurden.

4.1 Kurzreferenz

Da kein großer Funktionsumfang benötigt wird, fällt die Verwendung der Klassenbibliothek relativ einfach aus. Die Hauptklasse heißt `org.sbfilter.Filter` und lässt sich unter Linux von der Kommandozeile wie folgt aufrufen:

```
# java -classpath "htmlparser.jar:." org/sbfilter/Filter [Parameter]
```

bzw.

```
> java -classpath "htmlparser.jar;." org/sbfilter/Filter [Parameter]
```

unter Windows. Voraussetzung ist, dass eine Java SE ≥ 1.4 installiert ist, man sich im Projektverzeichnis SBFilter/ befindet, das Projekt bereits mit dem Aufruf von 'ant' kompiliert wurde und die Klassenbibliothek htmlparser.jar [3] sich ebenfalls im Projektverzeichnis befindet. Für die beiden erwähnten Betriebssysteme stehen auch die zwei vorgefertigten Skripten SBFilter und SBFilter.bat bereit, um die Ausführung zu vereinfachen. Für die folgenden Beispiele wird der Einfachheit halber nur das Linux-Skript für Programmaufrufe verwendet.

SBFilter erhält per Kommandozeilenparameter die URL der Seite, die analysiert bzw. gefiltert werden soll, und optional eine Ausgabedatei, in die die gefilterte Seite geschrieben werden soll, sowie Optionen. Mit dem Parameter -h kann eine Liste der möglichen Optionen und deren Erklärungen ausgegeben werden, weshalb hier nur die wichtigsten Optionen mit einigen Beispielen weiter ausgeführt werden.

Der Aufruf

```
# ./SBFilter http://www.stiftung-warentest.de result.html
```

liest die gegebene Seite als Hauptseite ein, parst ihren HTML-Quelltext, extrahiert die darin enthaltenen Links und lädt die Seiten, auf welche diese verweisen, als Vergleichsseiten ein. Dann wird die Hauptseite in einen DOM-Baum übersetzt, der Wortzählungs-Algorithmus wird ausgeführt, wobei die DOM-Knoten mit Relevanz-Gewichtungen attribuiert werden. Schließlich werden alle Knoten (einschließlich HTML-Tags) in die Ausgabedatei result.html geschrieben, deren Relevanz größer oder gleich 50% ist. HTML-Tags von Knoten, die wiederum Knoten beinhalten, deren Relevanz $\geq 50\%$ ist, werden ebenfalls ausgegeben, um die HTML-Struktur der Ausgabe-Datei möglichst weitgehend zu erhalten.

Folgende Kommandozeilenoptionen seien noch kurz erwähnt: mit `-l <Prozent>` kann der Grenzwert für die Relevanz der Knoten, die angezeigt werden sollen, verändert werden. Standardmäßig ist er – wie im Beispiel oben – 50%. Mit `-l 0.0` werden alle Knoten angezeigt. Mit der Option `-f <Filtername>` kann man den Algorithmus wählen, der beim Filtern angewandt wird. Mögliche Filternamen sind: `ntpc` (Node To Page Comparison, Wortzählungs-Algorithmus, dieser Filter wird standardmäßig verwendet) und `lev` (Levenshtein). Die Option `-t` erzeugt ggf. fehlende HTML-Tags in der Ausgabedatei, um sie in HTML-Browsern besser darstellbar zu machen. Eine weitere nützliche Option ist `-b`; dabei werden in der Ausgabedatei zusätzlich die Relevanzen der jeweiligen Einheiten in Prozent angezeigt.

Zur schnellen Veranschaulichung der Verwendung von `SBFilter` als API im eigenen Java-Code soll dieses Beispiel dienen:

```
Filter myFilter = new Filter(  
    new CacheBot("cache/site1", "site1.idx"),  
    new LevProcessor());  
myFilter.setMinRelevance(70.0);  
myFilter.quietMode(true);  
myFilter.filter(new URI("http://www.stiftung-warentest.de"),  
    "result.html");
```

Der `CacheBot` erhält als Parameter im Konstruktor Dateipfad und Index-Datei, die zum lokalen Cachen der eingelesenen HTML-Seiten verwendet werden sollen. Die Index-Datei dient dabei als ein Verzeichnis, in dem die richtigen URLs und die Namen der dazugehörigen lokalen Dateien gespeichert werden und befindet sich zusammen mit den gecachten Daten im angegebenen Dateipfad. Benötigt man keinen Cache, kann man dem Konstruktor der Klasse `Filter` als ersten Parameter auch `null` übergeben. Der zweite Parameter dieses Konstruktors ist eine Instanz des gewünschten Filter-Algorithmus. Es sind die Klassen `NTPCProcessor` und `LevProcessor` implementiert. Die

Klassenbibliothek wurde jedoch so konzipiert, dass ein Benutzer jederzeit auch eigene Filter implementieren kann, ohne den Quellcode der Bibliothek selbst verändern zu müssen. Dazu folgt mehr im nächsten Kapitel.

Weitere Informationen über die Verwendung der Klassenbibliothek gibt es im Projektverzeichnis unter docs/javadoc/index.html (zuvor muss im Projektverzeichnis 'ant' aufgerufen worden sein, um das Projekt und die javadocs zu erzeugen).

4.2 Die wichtigsten Klassen

Das folgende Diagramm veranschaulicht grob die Architektur der Klassenbibliothek.

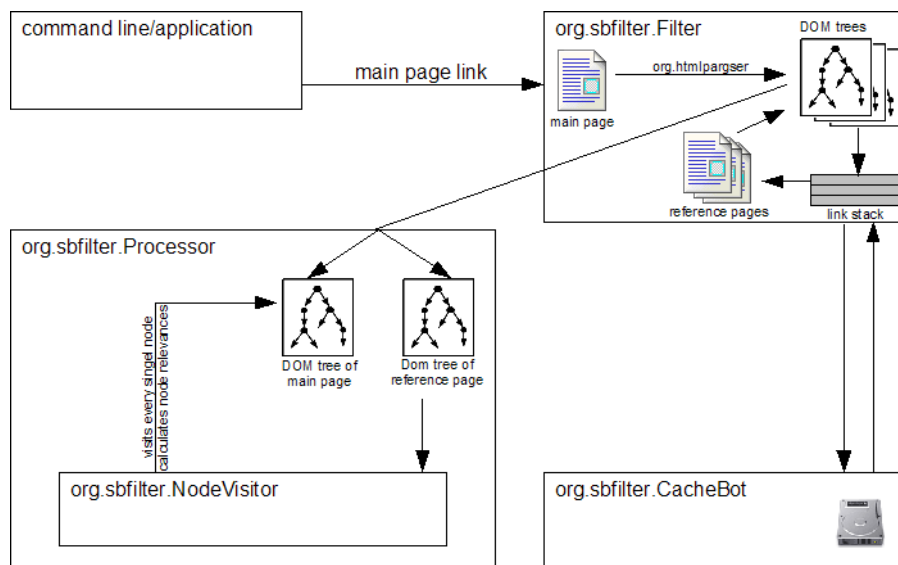


Abb. 4.2/1 Die Architektur der Klassenbibliothek org.sbfiler

Die Hauptklasse org.sbfiler.Filter lädt die gegebene Webseite, parst sie mit Hilfe der externen Klassenbibliothek org.htmlparser [3], extrahiert die darin enthaltenen Links,

die nicht aus dem Netzpfad der gegebenen Seite heraus führen, und lädt bzw. parst diese ebenfalls. Gleichzeitig werden die geladenen Seiten ab dem `<body>`-Tag in die DOM-Darstellung überführt; Tag-Attribute werden verworfen. Enthält eine Seite mehr als einen `<body>`-Tag, wird eine 'Invalid Page'-Warnung ausgegeben und es wird nur der erste `<body>`-Tag in einen DOM-Baum übersetzt.

Ggf. werden die Links im Link-Stack erst an eine Instanz der Klasse `org.sbfilter.CacheBot` übergeben. Diese speichert die Seiten erst lokal ab, falls sie noch nicht im Cache vorhanden sind, und liefert die URIs auf die entsprechenden lokalen Dateien zurück, welche die richtigen Seitenadressen ersetzt, so dass die Seiten nun nur noch lokal geladen werden müssen.

Für jede neu geladene und in die DOM-Darstellung überführte Vergleichs-Seite wird eine Instanz einer vom Interface `org.sbfilter.Processor` abgeleiteten Klasse (`org.sbfilter.NTPCProcessor` oder `org.sbfilter.LevProcessor`) aufgerufen, und der DOM-Baum der Hauptseite zusammen mit dem der Vergleichs-Seite werden ihr übergeben. Ein `org.sbfilter.NodeVisitor` besucht schließlich jeden Knoten des DOM-Baumes der Hauptseite und führt das entsprechende Gewichtungs-Verfahren aus, wobei die erforderlichen Daten für jeden Knoten als Attribute im jeweiligen Knoten abgespeichert werden. In einem letzten Schritt besucht der `NodeVisitor` jeden Knoten erneut, um aus den abgespeicherten Attributen die endgültige Relevanz für den jeweiligen Knoten zu berechnen.

In den meisten Fällen bietet es sich an, die beiden Interfaces `org.sbfilter.Processor` und `org.sbfilter.NodeVisitor` in einer Klasse zu vereinen, um alle mit dem gewünschten Algorithmus zusammenhängenden Methoden übersichtlich innerhalb einer einzigen Klasse zu implementieren.

4.3 Grabben und Parsen von Webseiten

Das Grabben von Webseiten wurde effizient in der Klasse `org.sbfilter.CacheBot` implementiert. Jeder Link, der zum Link-Stack hinzugefügt wird, wird gleichzeitig in eine Warteschlange des CacheBots eingereiht und wartet darauf, gecached zu werden. Für jeden Link in der Warteschlange wird wiederum ein eigener Faden zum cachen der entsprechenden Seite gestartet – insgesamt können so bis zu 30 Seiten gleichzeitig multithreaded geladen werden. Dadurch wird der „Flaschenhals“ vermieden, der durch die im Internet bekanntlich langen Antwortzeiten der Webserver entstehen kann. Sobald eine Seite geladen wurde, wird sie im Link-Stack als vorhanden markiert und kann nun aus dem Cache geladen und bearbeitet werden.

Da das Parsen von HTML-Quelltext – vor allem durch die im Internet kaum eingehaltenen w3c-Spezifikationen der Markup-Sprache – eine sehr empfindliche, fehleranfällige und aufwändige Angelegenheit ist und somit die Implementierung eines HTML-Parsers den Rahmen dieser Arbeit gesprengt hätte, wurde für diese Aufgabe die open-source Java-Klassenbibliothek `org.htmlparser` verwendet [3]. Sie bietet ein zwar nicht völlig fehlerfreies, aber dennoch sehr ausgereiftes und robustes Paket zum fehlertoleranten Parsen und Analysieren von HTML- und XHTML-Seiten.

4.4 Interne Darstellung des DOM-Baums

Die interne Repräsentation des DOM-Baums einer Webseite wird mit der abstrakten Klasse `org.sbfilter.DomNode` und den davon abgeleiteten Klassen `org.sbfilter.TagDomNode` und `org.sbfilter.TextDomNode` bewerkstelligt. Die zum Parsen verwendete Klassenbibliothek `org.htmlparser` erstellt zwar aus geparsen

HTML/XHTML-Seiten automatisch einen DOM-Baum unter Verwendung eigener Klassen, doch ist dieser für unsere Zwecke nicht geeignet, da es nicht die Möglichkeit gibt, in dessen Knoten benutzerdefinierte Attribute abzuspeichern, die für die Ähnlichkeitsanalyse benötigt werden. Außerdem benötigen wir für die Implementierung der Analyseverfahren bestimmte Methoden innerhalb der Baumknoten-Klassen, die uns in der Baum-Darstellung des HTML-Parsers nicht zur Verfügung stehen.

Bei der Übersetzung der DOM-Baum-Darstellung des HTML-Parsers in unsere Baum-Darstellung werden alle Knoten, die reinen Text enthalten in `TextDomNodes` übersetzt; alle Tag-Knoten werden in `TagDomNodes` übersetzt, wobei lediglich der Typ der Tags gespeichert wird (`table`, `body`, etc.) und die Tag-Attribute verworfen werden.

Die von `org.sbfilter.DomNode` abgeleiteten Klassen müssen alle folgende Methoden besitzen:

- `acceptVisitor()` - nimmt eine `org.sbfilter.NodeVisitor` Klasse entgegen und führt darauf `visit()` aus. Anschließend wird das `acceptVisitor()` der Kinderknoten rekursiv aufgerufen. Diese „Besucher“-Technik erlaubt es, durch Implementierung unterschiedlicher, von `org.sbfilter.NodeVisitor` abgeleiteter Klassen, beliebige Operationen rekursiv auf allen Knoten eines DOM-Baumes auszuführen, ohne die Knoten des Baumes zu verändern oder gar zu reimplementieren. Als kleine aber nützliche Änderung des „Besucher“-Konzepts erweist sich noch eine Bitmaske, die angibt, welche Knoten besucht werden sollen. Dadurch besteht nicht mehr die Notwendigkeit, die in Kapitel 3.1 beschriebene Vereinfachung am DOM-Baum vorzunehmen, da nun alle relevanten Knoten im Baum markiert werden können, damit die nicht relevanten Knoten bei der Traversierung des Baumes übergangen werden. Auf diese Weise kann erreicht werden, dass ausschließlich Knoten, die Content-Einheiten repräsentieren, analysiert werden.
- `toString()`, `toStringWithRelevance()`, `toHTMLWithRelevance()` - geben den Text

des Knotens und aller Kinderknoten als ASCII-Zeichenkette (ohne Tags) bzw. als HTML-Code (mit Tags) aus. Die auf WithRelevance endenden Methoden blenden diejenigen Knoten aus, die unterhalb einer gegebenen Relevanzgrenze liegen. Auch diese Methoden können durch eine Bitmaske konditioniert werden.

Außerdem müssen die von `org.sbfilter.DomNode` abgeleiteten Klassen selbstverständlich alle Methoden bereitstellen, die für die Navigation durch den DOM-Baum bzw. zur Veränderung desselben erforderlich sind (z.B. `getParent()`, `getChildren()`, `addChild()`, etc.).

4.5 Implementierung der Analyseverfahren

Das in Abschnitt 4.4 kennengelernte Besucher-Konzept ermöglicht eine gleichermaßen saubere wie effiziente Implementierung unterschiedlicher Analyseverfahren und bietet dem Benutzer der SBFilter-Klassenbibliothek die Möglichkeit, selbst Analyseverfahren zu entwickeln, ohne den Code der Bibliothek verändern zu müssen, indem er die Interfaces `org.sbfilter.Processor` sowie `org.sbfilter.NodeVisitor` implementiert.

Im Rahmen dieser Klassenbibliothek wurden die beiden vorgestellten Analyseverfahren – die Levenshtein-Distanz-Analyse sowie der Wortzählungs-Algorithmus – implementiert und getestet.

Eine implementierte Filter-Klasse sollte von `org.sbfilter.Processor` und `org.sbfilter.NodeVisitor` abgeleitet sein und wird in den folgenden Schritten ausgeführt:

Dem Konstruktor wird beim Erzeugen der Filter-Klasse der Wurzelknoten der Hauptseite übergeben. Dabei werden gleichzeitig die Relevanz-Attribute aller Knoten in der Hauptseite auf den Wert 0 zurückgesetzt.

Der folgende Schritt wird für jede einzelne Vergleichsseite der Reihe nach ausgeführt, so dass es nicht nötig ist, alle Vergleichs-Seiten im Speicher zu behalten: die aktuelle, zu bearbeitende Vergleichs-Seite wird in einen DOM-Baum übersetzt. Dann wird die Methode `org.sbfilter.Processor.process()` mit dem Wurzelknoten der Vergleichs-Seite als Parameter aufgerufen. Die Methode `process()` ruft am Wurzelknoten der Hauptseite `acceptVisitor()` auf und übergibt als Parameter (und als `NodeVisitor`) sich selbst, so dass die `visit()`-Methode der Filter-Klasse auf alle oder auf durch eine Bitmaske bestimmte Knoten des Hauptseiten-DOM-Baumes ausgeführt wird.

Jede von `org.sbfilter.DomNode` abgeleitete Klasse besitzt die Methoden `getInfo()` und `setInfo()`, mit denen man ein beliebiges Objekt an jeden DOM-Knoten eines Baumes anhängen kann. So können z.B. Zwischenwerte oder andere temporäre Informationen pro Knoten abgespeichert werden.

In einem letzten Schritt wird `acceptVisitor()` erneut am Wurzelknoten der Hauptseite aufgerufen, um aus den temporären Daten, die mit `setInfo()` in jedem Knoten abgespeichert wurden, die endgültige Relevanz zu berechnen.

5 Auswertung der Ergebnisse

Zur Untersuchung der Ergebnisgenauigkeit der implementierten Verfahren wurden 50 Webseiten zufällig ausgesuchter Webpräsenzen von menschlicher Hand (von Kommilitonen und mir selbst) untersucht und gefiltert. Das Ergebnis wurde anschließend mit den Ergebnissen der implementierten Filter – Levenshtein und das Wortzählungsverfahren – verglichen.

Die manuelle Filterung fand wie folgt statt: Kommilitonen erhielten per Email jeweils ein Paket mit Kopien von aktuellen Webseiten aus dem Internet und sollten aus deren HTML-Text alles entfernen, was sie intuitiv als inhaltlich nicht relevant einstufen. Dazu war es erlaubt, den HTML-Quelltext von Hand zu bearbeiten, dabei jedoch möglichst die Tag- (DOM-)Struktur zu erhalten; oder aber einen WYSIWYG-Editor (What You See Is What You Get), also einen graphischen Webseiten-Editor zu verwenden, um die irrelevanten Bereiche zu entfernen. Eine gewisse Ungenauigkeit war bei einer solchen Vorgehensweise durchaus zu erwarten, sollte jedoch mit der im nächsten Abschnitt beschriebenen Auswertungsmethode kaum ins Gewicht fallen. Da die angefallene Menge von Dokumenten recht umfangreich war und sich nur vergleichsweise wenige Kommilitonen fanden, die sich (unentgeltlich) bereit erklärten, die damit verbundene Arbeit auszuführen, musste ich mich an der manuellen Filterung auch selbst in erheblichem Maße beteiligen: die ersten 10 Seiten wurden von mir selbst gefiltert. Dabei habe ich NVU 1.0 als HTML-Editor verwendet und versucht, möglichst unvoreingenommen zu arbeiten, d.h. mir möglicherweise durch meine Kenntnis der Algorithmen bekannte Filterungs-Ergebnisse nicht einfließen zu lassen, sondern rein intuitiv zu urteilen.

Für den Vergleich der manuell gefilterten Seiten mit den Ergebnissen der Klassenbibliothek wurde eine weitere Klasse, `Comparator.java`, implementiert. Diese macht sich den in `LevProcessor.java` implementierten Levenshtein-Algorithmus zu

Nutze, um damit zwei HTML-Dokumente zu vergleichen. Die Klasse kann auch von der Kommandozeile aus aufgerufen werden mit

```
> java -classpath "htmlparser.jar; ." org/sbfilter/Comparator p1 p2
```

Dabei sind p1 und p2 die HTML-Seiten, die miteinander verglichen werden sollen.

Die Klasse geht ähnlich wie der Levenshtein-Filter vor: beide Seiten werden erst geparkt, in DOM-Bäume umgewandelt und die Knoten, die Content-Einheiten repräsentieren, werden markiert. Für jede dieser Einheiten einer Seite wird nun das Minimum der normierten LD zu allen Einheiten der zweiten Seite ermittelt (also die normierte LD zwischen einem Knoten der ersten Seite und dem damit ähnlichsten Knoten der zweiten Seite) und im jeweiligen Knoten gespeichert. Anschließend werden die beiden Seiten vertauscht und das Verfahren nochmal angewandt. Die Inversion des Mittelwerts all dieser so ermittelten normierten LD ergäbe ein aussagekräftiges Maß für die strukturelle und inhaltliche Ähnlichkeit von HTML-Seiten. Eine Größe, die hierbei jedoch noch ins Gewicht fallen sollte, ist die Größe der bewerteten Content-Einheiten, also die Anzahl der Zeichen, die sie enthalten. Dem steht die Überlegung zu Grunde, dass die Bedeutung eines Knotens innerhalb einer Seite zunimmt, je mehr Text er enthält. Dies wurde in der Implementierung bewerkstelligt, indem alle inversen normierten LDs der Knoten mit der Anzahl der Zeichen im jeweiligen Knoten multipliziert wurden und schließlich zur Berechnung des Mittelwertes deren Summe durch die gesamte Anzahl der Zeichen aller Knoten geteilt wurde. Auf diese Weise erhält man eine zuverlässige Größe, die hervorragend als Bewertungsmaß für unsere Filter dient. Problematisch bei dieser Ähnlichkeitsmessung ist lediglich, dass auch sehr unterschiedliche Content-Einheiten nicht unbedingt eine normierte LD von 1.0 (also die maximal mögliche Levenshtein-Distanz zueinander) haben müssen. Dadurch erhält man in einigen Fällen auch für Webseiten, die vom Menschen als völlig unterschiedlich betrachtet werden können, Ähnlichkeitsmaße > 0.0 (idR. jedoch < 0.3). Man muss also

unterstreichen, dass der mit dieser Methode erhaltene Ähnlichkeitsindex kein Maß für die vom Menschen subjektiv empfundene Ähnlichkeit von Webseiten ist, sondern vielmehr auf der Levenshtein-Distanz basiert.

Die HTML-Seiten der herangezogenen Webseiten können im Unterverzeichnis test/orig/ des Projektverzeichnisses gefunden werden, die von Hand gefilterten befinden sich in test/manual/, die automatisch gefilterten in test/lev/ bzw. test/ntpc/. Für die automatisierte Analyse wurden die Standardparameter verwendet (min. Relevanz 50%, Rekursionstiefe 1). In wenigen Fällen war zusätzlich der Parameter *-a* erforderlich, der es erlaubt, auch Links außerhalb des aktuellen Netzpfades (jedoch nicht außerhalb des Servers!) zu verfolgen, da manche Websites so stark in Verzeichnisse untergliedert waren, dass sich in den jeweiligen Pfaden keine weiteren verlinkten Dokumente für die Ähnlichkeitsanalyse befanden (z.B. microsoft.com).

Tabelle 5/2 enthält die Ergebnisse der Auswertung. Die erste Spalte gibt die Dokumentnummer der Test-Webseite an. Die HTML-Dokumente der Webseiten können im jeweiligen Verzeichnis (test/orig/, test/manual/, test/lev/ oder test/ntpc/) unter dem Dateinamen <n>.html – <n> sei dabei die Dokumentnummer laut Tabelle – eingesehen werden. Die zweite Spalte gibt die original URL an, von der die jeweilige Seite geladen wurde. Die dritte Spalte gibt den Ähnlichkeitsindex zwischen der manuell gefilterten Seite und der Originalseite an – also ein Hinweis darauf, wie viel vom gesamten Seiteninhalt nach dem manuellen Filtern erhalten geblieben ist. Die vierte Spalte gibt den Index zwischen der manuell- und der ntpc-gefilterten Seite und eine Genauigkeitsbewertung wieder (siehe Tabelle 5/1). In der fünften Spalte steht der Index zwischen manueller und Levenshtein-Filterung, ebenfalls mit Genauigkeitsbewertung und in der sechsten Spalte wird die Übereinstimmung zwischen ntpc und Levenshtein wiedergegeben.

Die oben erwähnte Genauigkeitsbewertung wurde nach folgendem Schlüssel gemacht:

Index	Genauigkeit
1.0 bis ausschließlich 0.9	sehr gut (++)
0.9 bis ausschließlich 0.8	gut (+)
0.8 bis ausschließlich 0.7	befriedigend (0)
0.7 bis ausschließlich 0.6	ausreichend (-)
<= 0.6	ungenügend (--)

Tabelle 5/1 Schlüssel für die Genauigkeitsbewertung

Ausschlaggebend für die Genauigkeitsbewertung waren einige Tests mit manuell bearbeiteten Webseiten. So wurden z.B. aus dem HTML-Quelltext der Seite 1 (siehe Tabelle 5/2) beabsichtigt die falschen Seitenabschnitte entfernt, wonach die so gefilterte Seite mit der ntpc- bzw. Levenshtein-gefilterten Seite verglichen wurde. Erwartungsgemäß fielen die Ähnlichkeitsindizes bedeutend geringer aus; so z.B. bei der Beispieldatei, die unter test/manual/1.example.false.html im Projektverzeichnis zu finden ist: hier wurden eben diejenigen Seitenabschnitte entfernt, die als relevant betrachtet wurden. Die Ähnlichkeitsindizes waren für ntpc 0.1033 und für lev 0.0794.

Nachfolgend finden sich die Ergebnisse der Auswertung:

Nr.	URL	man	ntpc	lev	ntpc/ lev
1	http://www.faz.net/s/Rub8A25A66CA9514B9892E0074EDE4E5AFA/Doc~EBBE84835D0394E3D83A1B06EC9EBFD83~ATpl~Ecommon~Scontent.html	.9525	.9424 ++	.9481 ++	.9937
2	http://www.de.tomshardware.com/praxis/20050729/index.html	.5530	.6929 -	.6929 -	1.0
3	http://de.news.yahoo.com/wahl/index.html	.6927	.6837 -	.6821 -	.9946
4	http://www.digitalkamera.de/Software/default.asp	.7641	.9641 ++	.9392 ++	.9453
5	http://www.stiftung-warentest.de/online/computer_telefon.html	.9016	.9107 ++	.9504 ++	.9639
6	http://de.today.reuters.com/news/newsArticle.aspx?type=topNews&storyID=2005-08-28T153250Z_01_DEO855699_RTRDEOC_0_DEUTSCHLAND-WAHL-CDU-MERKEL-20050828.xml	.7142	.9676 ++	.9737 ++	.9941
7	http://focus.msn.de/	.6059	.8955 +	.7939 0	.8823

Nr.	URL	man	ntpc	lev	ntpc/ lev
8	http://www.microsoft.com/security/incident/zotob.mspcx	.8839	.9388 ++	.9485 ++	.9789
9	http://www.heise.de/	.9018	.9215 ++	.9215 ++	1.0
10	http://de.wikipedia.org/wiki/Levenshtein	.8626	.9353 ++	.9283 ++	.9937
11	http://wireservice.wired.com/wired/headlines.asp?section=Breaking&firstStory=1	.8975	.8680 +	.8710 +	.9983
12	http://www.richtigfit.de/	.9010	.9752 ++	.9450 ++	.9693
13	http://onnachrichten.t-online.de/c/53/07/72/5307726.html	.6877	.8360 +	.8452 +	.9713
14	http://pkw.freenet.de/cardetail.php?car=1384	.6745	.7413 0	.7377 0	.9843
15	http://www.stadtentwicklung.berlin.de/wohnen/mietspiegel/	.9261	.1203 --	.4810 --	.3164
16	http://www.bpb.de/themen/YIDWCA,0,Der_11_September_2001_und_die_Folgen.html	.8363	.8514 +	.8631 +	.9274
17	http://www.amazon.de/exec/obidos/ASIN/3462036068/ref=ed_xsoc_b_1_1/028-3688527-7042109	.5955	.6857 -	.6577 -	.9688
18	http://www.welt.de/data/2005/08/31/768387.html	.7765	.9107 ++	.9205 ++	.9842
19	http://www.pcwelt.de/news/software/118911/index.html	.4857	.8309 +	.7634 0	.8762
20	http://www.sensor-test.de/main/dshe36qh/dshf78fi/page.html	.5999	.6808 -	.6808 -	1.0
21	http://www.zeitzuleben.de/inhalte/we/stress/antistresstipps_1.html	.7718	.0 --	.8511 +	.0
22	http://www.spiegel.de/politik/deutschland/0,1518,370344,00.html	.6396	.9493 ++	.9436 ++	.9955
23	http://www.radsport-aktiv.de/sport/sportnews_35682.htm	.7133	.9632 ++	.9639 ++	.9421
24	http://www.nicht-bei-mir.de/	.5792	.8339 +	.8440 +	.9930
25	http://www.rootprompt.org/	.9265	.9380 ++	.9380 ++	1.0
26	http://www.gnu.org/	.5817	.6765 -	.5761 --	.8560
27	http://www.nasa.gov/centers/glenn/research/warp/warp.html	.7708	.0 --	.9103 ++	0.0
28	http://www.findarticles.com/p/articles/mi_go1596/is_200410/ai_n9755487	.3718	.7966 0	.0 --	0.0
29	http://news.bbc.co.uk/1/hi/world/europe/default.stm	.9293	.7942 0	.9713 ++	.8177
30	http://www.br-online.de/bayern-heute/artikel/0508/30-pfahls/index.xml	.8870	.9311 ++	.9311 ++	.9933

Nr.	URL	man	ntpc	lev	ntpc/ lev
31	http://www.nytimes.com/2005/09/01/opinion/01thu1.html?ex=1125806400&en=6256fd8a7aed7811&ei=5070	.6940	.8216 +	.7656 0	.9412
32	http://www.ezinearticles.com/?Do-What-Works&id=64011	.3877	.6569 -	.6635 -	.9829
33	http://www.historynet.com/we/blmasoncountywar/	.6979	.9781 ++	.9815 ++	.9964
34	http://straylight.law.cornell.edu/constitution/constitution.amendmentxiv.html	.9738	.9826 ++	.9826 ++	1.0
35	http://www.sitepoint.com/article/rich-media-email-best-practices	.8691	.8851 +	.9482 ++	.9268
36	http://www.linuxdevcenter.com/pub/a/linux/2005/09/01/keepalived.html	.7900	.9623 ++	.9422 ++	.9811
37	http://www.heise.de/ct/04/01/018/	.8336	.8585 +	.8568 +	.9976
38	http://www.autobild.de/aktuell/neuheiten/artikel.php?artikel_id=1478	.4205	.8863 +	.8876 +	.9983
39	http://www.netzeitung.de/autoundtechnik/interviews/325718.html	.8547	.9587 ++	.9620 ++	.9898
40	http://www.giga.de/index.php?storyid=125242	.9379	.6592 -	.8468 +	.8142
41	http://www.oasis-open.org/news/oasis_news_08_08_05.php	.9102	.9580 ++	.9580 ++	1.0
42	http://www.thecarconnection.com/Vehicle_Reviews/Luxury_Cars/Preview_2007_Mercedes-Benz_S-Class.S183.A9157.html	.7814	.9262 ++	.9191 ++	.9922
43	http://www.hydrogenus.com/DOE-role.asp	.9860	.0 --	.9572 ++	.0
44	http://www.scientific.de/produkte/maple/maple-10/index.html	.9339	.9729 ++	.9539 ++	.9854
45	http://www.free-av.com/antivirclassic/index.html	.8780	.7700 0	.9718 ++	.7972
46	http://solarsystem.nasa.gov/planets/profile.cfm?Object=Earth&Display=Facts	.7961	.8962 +	.8891 +	.8658
47	http://www.raceacrossamerica.org/Default.aspx?tabid=124	.7756	.7113 0	.6622 -	.7214
48	http://www.imdb.com/title/tt0388482/	.3628	.5199 --	.5011 --	.9676
49	http://www.bmwusa.com/vehicles/7/750iSedan	.2318	.5079 --	.9475 ++	.5440
50	http://www.eweek.com/article2/0,1895,1855188,00.asp	.4352	.9619 ++	.9676 ++	.9944

Tabelle 5/2 Ergebnisse der Auswertung

Bei einer durchschnittlichen Ähnlichkeit der manuell gefilterten Seiten mit den Originalseiten von 0,7386 beträgt die durchschnittliche Genauigkeit der Ergebnisse für

den ntpc-Algorithmus 0,7822 und für den Levenshtein-Algorithmus 0,8408. Die durchschnittliche Ähnlichkeit der Ergebnisse von Levenshtein- und Wortzählungsalgorithmus beträgt 0,8567.

Die folgenden Abbildungen zeigen ein Beispiel für eine recht gut gefilterte Seite:

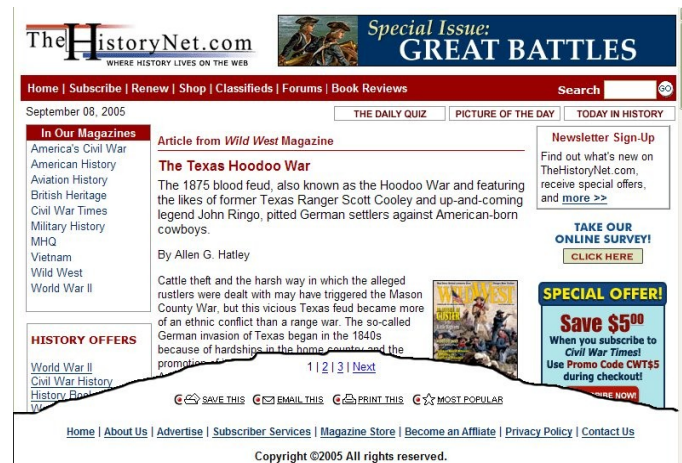


Abb. 5/1 Original-Seite Nr. 33

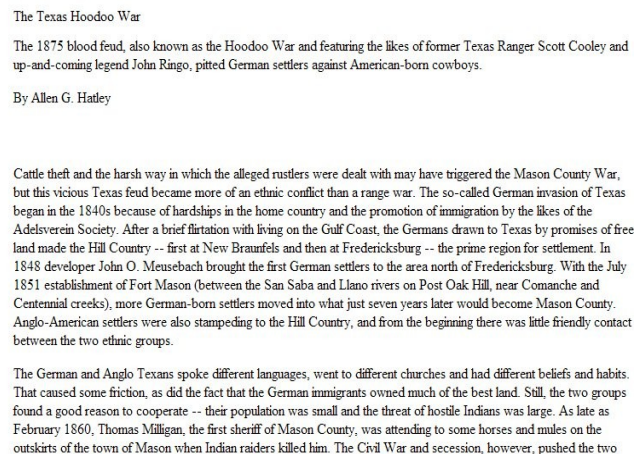


Abb. 5/2 Levenshtein-Gefilterte Seite Nr. 33

Als Conclusio lässt sich beim Betrachten dieser Ergebnisse feststellen, dass der

Levenshtein-Algorithmus im Schnitt signifikant besser abschneidet als der Wortzählungs-Algorithmus (um fast 6 Prozentpunkte!). Dieser Vorteil wird jedoch durch die zum Teil erheblich längere Laufzeit des Levenshtein-Algorithmus ausgeglichen. Es sei an dieser Stelle angemerkt, dass diese Ergebnisse, wie bereits festgestellt, mit den Standardparametern erzielt wurden – würde man für jede Webpräsenz die Parameter durch Probieren optimal einstellen, könnte man wesentlich bessere Ergebnisse erzielen. Außerdem gab es auch beim manuellen Filtern Situationen, in denen man sich nicht eindeutig entscheiden konnte, ob ein Bereich relevant ist oder nicht (z.B. bei Verlinkungen auf themennahe Artikel oder Werbung, die mit dem Inhalt der Seite zusammenhängt), was die manuelle Filterung zum Teil subjektiv und daher nicht zu 100% genau erscheinen lässt. Diese Tatsache wirkt sich auch negativ auf die gemessene durchschnittliche Genauigkeit der Ergebnisse aus.

Vor allem die „Ausreißer“ bei den Seiten Nr. 21, 27, 28 und 43, bei denen in den automatisch gefilterten Seiten zum Teil überhaupt kein Inhalt mehr übrig geblieben ist, ließen sich durch Veränderung der Relevanzgrenze mit dem Parameter -l vermeiden und damit das Gesamtergebnis wesentlich verbessern.

An dieser Stelle möchte ich noch Dinah Frielingsdorf, Carsten Jurenz und Zoran Golic danken, die sich an der manuellen Filterung der Testseiten beteiligt haben.

6 Zusammenfassung und Ausblick

Das Erarbeiten dieses Gewichtungs-Verfahrens für einzelne Webseiten-Abschnitte hat gezeigt, dass es durchaus einige bisher noch weniger untersuchte Ansätze gibt, die im Bereich des Web Content Mining brauchbare Ergebnisse liefern und somit von großem Nutzen sein können.

Die ähnlichkeitsbasierte Vorverarbeitung von Webseiten macht sich gemeinsame Strukturen innerhalb von Webpräsenzen zu Nutze, um mit unterschiedlichen Algorithmen – im Rahmen dieser Arbeit wurden zwei erörtert und implementiert – einzelne Seitenabschnitte zu gewichten, und damit ein Relevanzmaß für den darin enthaltenen Text zu liefern. Es wurde bei der Implementierung der Java-Klassenbibliothek besonders darauf geachtet, die Erweiterung derselben mit weiteren Ähnlichkeitsanalyse-Algorithmen zu ermöglichen, was eine Weiterentwicklung und Verbesserung erleichtern soll.

Die Auswertung der Ergebnisse hat gezeigt, dass die hier implementierten Verfahren recht gute Ergebnisse liefern konnten. Nicht nur durch die eben erwähnte Erweiterung der ähnlichkeitsbasierten Analyse lässt sich die Ergebnisgenauigkeit noch steigern; es gibt auch einige grundverschiedene Ansätze zur Lösung desselben Problems – nämlich der Auffindung relevanter Seitenabschnitte in Webseiten – es sei hier z.B. das im Abschnitt 2 bereits beschriebene sichtbarkeitsbasierte Zerlegen von Webseiten (VISION-based Page Segmentation) [4] erwähnt, bei dem Seitenabschnitte auf Grund ihrer Position im Browser-Fenster gewichtet werden. Die Kombination der Ergebnisse solcher anderer Verfahren mit denen der ähnlichkeitsbasierten Analyse wäre ein mögliches Thema zur Weiterentwicklung und -forschung und lässt vermuten, dass sich auf diese Weise die Ergebnisgenauigkeit weiter verbessern ließe.

Primäres Ziel der Arbeit war, es durch ein automatisiertes Verfahren zu ermöglichen, gezielt Inhalt aus Internet-Seiten zu extrahieren. Verschiedene Einstellungs-

Möglichkeiten wie das Verschieben der Relevanzgrenze oder das Ändern der Rekursionstiefe für die Link-Weiterverfolgung sollten je nach Webpräsenz eine möglichst fehlerfreie und präzise Extraktion des gewünschten Inhalts gewährleisten. Selbstverständlich sind auch völlig andere Anwendungsgebiete denkbar. So kann durch eine solche Vorverarbeitung die dem Indizieren von Webseiten für Suchmaschinen vorgeschaltet ist, die Präzision von Such-Ergebnissen gesteigert werden. Oder aber man verwendet die ähnlichkeitsbasierte Analyse für den Vergleich von Internetseiten.

Die Arbeit an diesem Thema hat sich für mich als eine gleichermaßen interessante wie lehrreiche Aufgabe mit praktischem Schwerpunkt erwiesen. Der Nutzen, der aus ihr für andere Projekte und Arbeiten hervorgehen könnte, rundet dies noch ab und motiviert zur weiteren Auseinandersetzung mit diesem Thema.

Literaturverzeichnis

- [1] The w3c Document Object Model: <http://www.w3.org/DOM/>
- [2] The Nutch Project: <http://www.nutch.org/>
- [3] The HTML Parser: <http://htmlparser.sourceforge.net/>
- [4] Deng Cai et al.: Vision Based Page Segmentation
<http://www.ews.uiuc.edu/~dengcai2/VIPS/VIPS.html>
- [5] Cohen, W. William: Recognizing Structure in Web pages using Similarity Queries
<http://citeseer.ist.psu.edu/cohen99recognizing.html>
- [6] Bing Liu et al.: Mining Data Records in Web Pages
<http://www.cs.uic.edu/~liub/publications/KDD-03-techReport.pdf>
- [7] Salton, G.: Automatic Text Processing, Addison Wesley, 1989
- [8] Kristina Lerman et al.: Using the Structure of Websites for Automatic Segmentation of Tables: <http://www.isi.edu/info-agents/papers/lerman04-sigmod.pdf>
- [9] Crescenzi, Mecca, Merialdo: The RodeRunner Project
<http://www.dia.uniroma3.it/db/roadRunner/publications/daswis2001.pdf>
- [10] Lerman et al.: Using the Structure of Web Sites for Automatic Segmentation of Tables: <http://www.isi.edu/~lerman/papers/lermanSIGMOD04.pdf>
- [11] Lerman, Knoblock, Minton: Automatic Data Extraction from Lists and Tables in Web Sources: <http://www.isi.edu/~lerman/papers/lerman-atem2001.pdf>
- [12] Knoblock et al.: Accurately and Reliably Extracting Data from the Web: A Machine Learning Approach: <http://www.isi.edu/~lerman/papers/ieee-de.pdf>

Eidesstattliche Versicherung

Ich versichere hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbständig und nur unter Benutzung der angegebenen Literatur und Hilfsmittel angefertigt habe. Wörtlich übernommene Sätze und Satzteile sind als Zitate belegt, andere Anlehnungen hinsichtlich Aussage und Umfang unter Quellenangabe kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und ist auch noch nicht veröffentlicht.

Ort, Datum: _____ Unterschrift: _____