Diplomarbeit

Verbesserung von Suchindizes
im WorldWideWeb

Patrick Stein
Fachhochschule München
Betreuer: Prof. Ulla Kirch-Prinz

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das WorldWideWeb ist in den letzten drei Jahren zum größten Informationsspeicher und -verteiler der Welt gewachsen. Dies wird vor allem durch die zunehmende Akzeptanz in der breiten Öffentlichkeit gefördert. Diese Akzeptanz hat viele Firmen dazu bewegt, ihre Produkte und Informationen im WorldWideWeb zur Verfügung zu stellen.

Die Entwicklung „alles und jedes“ im WorldWideWeb anzubieten, hat neben vielen seriösen Informationsanbietern wie z.B. dem Universallexikon Encyclopaedia Britannica[1] oder dem Nachrichtendienst Reuters[2] allerdings auch Stilblüten wie z.B. dem virtuellem Weiterleben[3] oder einfach nur Quatschseiten[4] hervorgebracht.

Da das WorldWideWeb eigentlich nur für die institutsinterne Verteilung von Dokumenten vorgesehen war, wurden auch keinerlei Vorkehrungen getroffen, Dokumente zu finden oder in geordneten Gruppen zusammenfassen zu können. Mit einem Datenbestand von nun mehr als 150 Gigabyte (geschätzter Stand April 1997) wurde bei der Konzeption nicht gerechnet.

Um sich noch einigermaßen, im immer größer werdenden Informationsdschungel des WorldWideWeb zurechtzufinden wurden sehr bald sog. Suchmaschinen entwickelt. Suchmaschinen erlauben es dem Benutzer Dokumente via Schlagwortsuche zu finden.

Diese Suchmaschinen arbeiten jedoch momentan nur mit sehr rudimentären Strategien zur Informationsaufbereitung, weshalb ich dieser Diplomarbeit versuche, Konzepte für eine Verbesserung der Suchmachschinen aufzuzeigen und zu implementieren:

- Entwurf und Implementierung eines optimierten Crawlers.

[1] http://www.eb.com/
[2] http://www.reuters.com/
[3] http://www.ewigesleben.de/
[4] http://www.quackwatch.com/

Hier wird gezeigt, wie durch ein fehlertolerantes, verteiltes System die Suchgeschwindigkeit bestehender Crawler um ein vielfaches erhöht werden kann.

- Entwurf und Implementierung eines Sprachenerkenners für die Indexierung.

  Sprachenerkennung bezieht sich hierbei auf die Sprache in der die Dokumente abgefaßt sind, die von Suchmaschinen indiziert werden. Für Benutzer von Suchmaschinen sind nur Dokumente interessant, die sie auch verstehen. Ist die Sprache, in der Dokumente abgefaßt sind bekannt, kann man den Benutzern ihnen verständliche Dokumente zeigen.

- Entwurf einer kontextabhängigen Indexierung.

  Indexierung der richtigen Inhalte liefert wesentlich bessere Suchantworten als dies mit koventioneller Wortindizierung möglich ist.

Um eine möglichst große Portabilität der entwickelten Programme zu gewährleisten, sind die in dieser Arbeit vorgestellten Algorithmen plattformunabhängig in Perl5 und Objective-C implementiert.

Die Durchführung der Diplomarbeit fand am *Centrum für Informations- und Sprachverarbeitung (CIS)* der Ludwig-Maximilians Universität München statt.

Die Notwendigkeit der vorgestellten Verbesserungen wurde schon während der Erstellung dieser Arbeit sichtbar: Zur Zeit werden die hier beschriebenen Methoden zur Sprachenerkennung von *AltaVista* in den dortigen Suchdienst integriert.

Kapitel 2

Einführung

Dieses Kapitel will in die Thematik des WorldWideWeb und die der Suchmaschinen einführen. Das Themengebiet wird überblickhaft umrissen, die wichtigsten Begriffe werden erläutert und die vorhandenen Probleme werden aufgezeigt.

2.1 Konzept des WorldWideWeb

Das WorldWideWeb wurde eigentlich für die Verbreitung von Dokumenten am CERN[1] Ende 1990 von Tim Berners-Lee[2] entwickelt. Es war dafür vorgesehen, Dokumente der dort arbeitenden Wissenschaftler untereinander zugänglich zu machen.

Das eigentlich Neue am WorldWideWeb war, daß zum einen Dokumente mit einheitlichen Namen netzwerkweit zugänglich wurden und zum anderen, daß man Referenzen auf andere Dokumente mit aktiven Verweisen sog. *Links* erzeugen konnte. Es wurde somit sehr einfach, anstelle der Literaturhinweise, nun die referenzierten Dokumente einfach als aktiven Verweis in das eigene Dokument aufzunehmen.

Solche *Hypertext* genannten Konzepte zur Verknüpfung von Dokumenten existieren schon länger (z.B. HyperCard auf Macintosh Systemen). Sowohl die Einfachheit der Bedienung, als auch die freie Verfügbarkeit der Anzeigeprogramme auf verschiedenen Rechnerplattformen im Internet führten dann zur weiten Verbreitung des WorldWideWeb .

2.2 Konzept der Suchmaschinen

Sowohl eine Informationsaufbereitung als auch eine Überwachung der im Dokument befindlichen Verweise fand von Hause aus im WorldWideWeb nicht statt,

[1] http://www.cern.ch/

[2] http://www.w3.org/pub/WWW/People/Berners-Lee/

weshalb bald Suchmaschinen dieses Informationsloch zu stopfen suchten.

Suchmaschinen, beginnend an einem oder mehreren Startdokumenten, indexieren den in dem Dokument vorhandenen Text. Um neue Dokumente holen zu können, werten sie die in dem Dokument befindlichen Verweise aus. Nach bestimmten Regeln werden dann die Dokumente, auf welche verwiesen wurde, geholt, z.B. nur Seiten aus Deutschland.

Suchmaschinen indexieren so nach und nach ganze Myriaden von Dokumenten. *AltaVista*[3] indexiert derzeit rund 50 Mio. Dokumente weltweit.

Die so erstellten Indexe werden dann im WorldWideWeb zur Verfügung gestellt. So kann man Suchanfragen der Form „film and not movie“ starten, um Dokumente die das Wort „film“, nicht jedoch das Wort „movie“ enthalten, als Ergebnis zu bekommen.

2.3 Unzulänglichkeiten von Suchmaschinen

Das WorldWideWeb ist mittlerweile so groß, daß Verbesserungen am Dokumentstandard HTML[4] nur sehr träge Verbreitung finden. Die Veränderungen müssen zum einen in einem langwierigen Prozeß vom W3C verabschiedet werden, zum anderen muß daran anschließend die gesamte Software an die neuen Standards angepaßt werden.

Firmen wie Microsoft und Netscape, die den Web-Browser-Markt beherrschen, versuchen überdies noch eigene Standards mit Ihren Web-Browsern zu etablieren.

Aus diesem Grunde ist es sicher effizienter, Algorithmen für eine bessere Aquirierung und Aufbereitung der Informationen, welche Suchmaschinen anbieten, zu verbessern.

Um die Suchmaschinen von ihren Unzulänglichkeiten zu befreien, muß man sich derer erst einmal bewußt werden.

Die heute eingesetzten Suchmaschinen (z.B. Harvest[5], MomSpider[6] oder Scooter[7]) laufen jeweils nur auf einem einzigen Rechner und versuchen das Web von diesem Rechner aus zu indexieren. Wenn man sich vorstellt, 150 Gigabyte an Text mit einem einzigen, wenngleich großem Rechner aus dem Netz zu laden, kann man sich ausmalen, daß diese Suchmaschinen bald an ihre Grenzen stoßen.

Indizes kamen erstmals in Büchern vor und sind sicher eine große Hilfe um in Büchern die gewünschten Informationen schneller aufzufinden. Dies liegt zum einen an dem Hintergrundwissen über das Thema des Buches, das der Verlag einfließen läßt, zum anderen an dem Wissen von unrelevanten Informationen.

[3]http://www.altavista.digital.com/

[4]http://www.w3.org/pub/WWW/MarkUp/

[5]http://harvest.transarc.com/

[6]http://www.ics.uci.edu/pub/websoft/MOMspider/

[7]Scooter ist der von AltaVista, Lycos, Yahoo... eingesetzte, nicht frei verfübare Crawler.

So wird in einem Buch das Wort „das“ nur im Index zu finden sein, wenn es um den Artikel als solchen geht. In den meisten Büchern werden für das Themengebiet irrelevante Begriffe nicht indexiert.

Man kann nun natürlich dazu übergehen das WorldWideWeb mittels geschultem Personal indexieren zu lassen. In diese Richtung arbeiten auch schon einige Suchindizes, wie z.B. Lycos[8] oder WebDe[9], die versuchen, den in Deutschland befindlichen Teil des Internet manuell zu indexieren.

Dies macht sicher insofern Sinn, als daß Internetbenutzer in Deutschland zum Großteil die deutsche Sprache sprechen und wahrscheinlich nicht an Kochrezepten interessiert sind, welche in chinesisch verfaßt sind.

Mit der Größe des Internet und der Anzahl der dort zur Verfügung gestellten Dokumente, wird jedoch klar, daß man eine manuelle und akutelle Indexierung nur für ausgewählte Informationsgebiete machen kann. Ein vollständiger Index des WorldWideWeb wird, allein durch die Fluktuation der Daten, nur mit maschineller Hilfe möglich und verwaltbar.

2.4 Verbesserungsmöglichkeiten

Wie schon erwähnt, muß man versuchen die Strategien, die eine Suchmaschine ausmachen, zu verbessern. Sind die „brute force“ Algorithmen der momentan verfügbaren Suchmaschinen für kleine Datenmengen noch ausreichend, stoßen diese Algorithmen mit der aktuellen Größe des Internet an ihre Grenzen.

Verbesserungsmöglichkeiten fallen einem bei den genannten Unzulänglichkeiten sofort ins Auge:

- Verteilung der Suchmaschinen auf mehrere Rechner.
- Sprachenfilterung bei der Indexierung.
- Kontextabhängige Indexierung.

Zu der Kontextabhängigen Indexierung ist zu bemerken, daß hier das Internet sicher die größte Möglichkeit einer elektronischen Datenverarbeitung bisher versäumt:

Geht man heute in eine Bibliothek, so kann man z.B. nach „Mozart“ suchen und bekommt einige Bücher über das Leben und Werk von Mozart. Durch geschickte elektronische Datenverarbeitung kann man jedoch den Informationssuchenden im Internet zum einen auf die Dokumente in denen „Mozart“ behandelt wird verweisen, zum anderen kann man auch die mit „Mozart“ verbundenen Themen aufzeigen. Der Informationssuchende kann dann gleich einen Überblick über das Themengebiet bekommen, ohne vorher mehrere Bücher durchzulesen.

[8]http://www.lycos.de/
[9]http://www.web.de/

2.5 Eingrenzung der Arbeit

Im Rahmen dieser Diplomarbeit werden Lösungen für die genannten Verbesserungsmöglichkeiten aufgezeigt und implementiert.

Zu Beginn der Arbeit stand die Sprachenerkennung im Vordergrund. Um einen Sprachenerkenner implementieren zu können, benötigt man zunächst einen Web-Crawler[10]. Ein Web-Crawler ist ein Programm welches das WorldWideWeb durchsucht.

Aus diesem Grunde widme ich mich in den nun folgenden Kapiteln zunächst den Crawlern, um dann, darauf aufbauend, die Sprachenerkennung und folgend die Kontexterkennung näher zu untersuchen.

2.6 Grundbegriffe

Hier nicht erwähnte Begriffe befinden sich im Glossar.

2.6.1 Internet

Eigentlich versteht man unter dem Begriff „Internet“ nur das Netzwerk, welches mit dem *Internetprotokoll* arbeitet. Das *Internetprotokoll* wurde Mitte der 70'er Jahre vom amerikanischen Verteidigungsministerium in Auftrag gegeben; es sollte die Kommunikation von Militärcomputern nach einem atomaren Angriff sicherstellen.

Bis Ende der 80'er Jahre wurde das Internetprotokoll nur von den amerikanischen Militärbehörden und Universitäten weltweit benutzt. Erst mit dem WorldWideWeb und dessen Verbreitung in den letzten Jahren, wurde das Internet zu *dem* größten Computernetzwerk der Welt.

2.6.2 URL

Eine URL (Uniform Resource Locator) ist, wie der Name schon sagt, eine eindeutige Beschreibung einer Datenquelle. Über eine URL wird ein Dokument weltweit *eindeutig* benannt. Diese Eindeutigkeit der Namen erreicht man durch die drei verschiedenen Teile aus denen eine URL zusammengesetzt ist:

protocol: Hier wird das Übertragungsprotokoll genannt, über welches die Daten zur Verfügung gestellt werden.

host: Dies ist der Name des Rechners, auf welchem die Daten liegen.

[10] to crawl: *engl.* kriechen. Web-Crawler bewegen sich, den Links, die sie auf Dokumenten vorfinden, folgend (kriechend) durchs WorldWideWeb . Genauer wird auf Crawler im Kapitel 3 eingegangen.

path: Dies ist der Pfadname, unter dem das Dokument auf dem Rechner gespeichert ist.

Eine URL mit Namen „http://www.cis.uni-muenchen.de/index.html“ beschreibt also ein Dokument, welches auf dem Rechner mit dem Namen „www.cis.uni-muenchen.de“ unter dem Pfad „index.html“ gespeichert ist und man via „http“ abrufen kann.

2.6.3 HTTP

Das HTTP (HyperText Transfer Protokol) ist ein Protokoll, welches die Kommunikation von Web-Browsern mit Web-Servern beschreibt. Im wesentlichen besteht es aus einem Frage-Antwort Schema, welches ungefähr so aussieht:

Auf eine Anfrage des Clients: „Gib mir das Hypertextdokument mit der folgenden URL... mit Hilfe des genannten Protokolls.“

```
GET http://www.cis.uni-muenchen.de/index.html HTTP/1.1
```

Anwortet der Server: “Hallo Client, hier kommt das Dokument ...“

```
HTTP/1.0 200 Document follows
MIME-Version: 1.0
Server: CERN/3.0
Date: Tuesday, 20-May-97 09:22:13 GMT
Content-Type: text/html
Content-Length: 3768
Last-Modified: Monday, 10-Mar-97 15:53:29 GMT
```

...

2.6.4 HTML

Im WorldWideWeb werden Hypertextdokumente in HTML (HyperText Markup Language) publiziert. HTML ist eine Anwendung von SGML (Standard General Markup Language).

SGML entstand durch das Bestreben, formatierte Texte zwischen verschiedenen Rechnerplattformen austauschen zu können. Es handelt sich bei SGML um ein deskriptives Markup, d.h. es werden Textteile des Dokuments mit beschreibenden Marken versehen (z.B. Titel, Author, Überschrift,...).

2.6.5 HTML-Dokument

Ein HTML-Dokument, auch Webseite oder Hypertextdokument genannt, enthält neben dem Text noch weitere Informationen wie z.B. aktive Verweise, Textformatierung, Musik, Film, etc. .

Wie der Name schon sagt ist der Text in HTML verfaßt.

2.6.6 WorldWideWeb

WorldWideWeb war der Name des ersten Web-Browsers von Tim Berners-Lee. Heute ist der Begriff WorldWideWeb ein Synonym für alle Hypertextdokumente die im Internet zur Verfügung gestellt werden.

Das WorldWideWeb -Programm von Tim Berners-Lee lief auf dem wenig verbreiteten NeXTSTEP Betriebssystem. Durchschlagenden Erfolg hatte das WorldWideWeb erst, als der Web-Browser „Mosaic“ von NSCA auf Macintosh, Unix und Windows Systemen frei verfügbar wurde.

2.6.7 Web-Browser

Ein Web-Browser stellt die in HTML geschriebenen Dokumente formatiert auf dem Bildschirm des Benutzers dar. Wenn man einen aktiven Verweis in einem Hypertextdokument mit der Maus anklickt, wird daraufhin das Dokument von der angegebenen Datenquelle geholt und dargestellt.

2.6.8 Suchindex

Suchindizes bestehen aus drei Teilen:

Crawler: Crawler wird das Programm genannt, welches das Hypertextdokumente aus dem Internet holt. Mehr dazu im Kapitel 3.

Indexierer: Dieses Program indexiert alle Seiten, welche der Crawler aus dem WorldWideWeb holt. Man muß die Daten aus zwei wesentlichen Günden indexieren:

- **Datenkompression:** Ein Index benötigt meist weniger als $\frac{1}{3}$ der Größe des zugrundeliegenden Datenbestandes.
- **Geschwindigkeit:** Ein Index muß nicht erst die gesamten Daten durchsuchen, sondern verwaltet alle in den Hyptertextdokumenten vorkommenden Daten in speziellen Datenstrukturen. Diese Datenstrukturen ermöglichen es sofort, die Stelle zu finden, an der die gesuchten Informationen stehen.

  Die Zeit zum Auffinden der Informationen ist um Größenordnungen kleiner, als die zum linearen Durchsuchen der Daten.

SearchInterface: Damit Benutzer des WorldWideWeb die Informationen abrufen können, benötigt man noch eine Schnittstelle. Dies Schnittstelle nimmt zum einen die Suchanfragen des Benutzers an, zum anderen liefert es die Antworten des Indexierers in lesbarer Form zurück.

Kapitel 3

Crawler

In diesem Kapitel wird zuerst ein kleiner Überblick auf die verfügbaren Webcrawler gegeben, um daran anschließend die Konzeption und Implementation der verbesserten Crawling-Strategien zu erläutern.

3.1 Einführung in Crawler

Web-Crawler verhalten sich im wesentlichen so wie Web-Browser. Man gibt dem Web-Crawler eine *URL*, zu der er dann das zugehörige Hypertextdokument aus dem WorldWideWeb holt. Dann wird das geholte Hyptertextdokument nach *URL's* durchsucht. Die neu gefundenen *URL's* werden daran anschließend vom Web-Crawler geholt und wiederum nach *URL's* durchsucht.

Dies wird solange wiederholt, bis alle Hypertextdokumente dieses gerichteten Graphen geholt worden sind.

Meist werden die Web-Crawler nicht auf das gesamte WorldWideWeb „losgelassen“, sondern nur auf Teilbereiche. So kann man den Suchraum des Web-Crawlers durch Anwendung von regulären Ausdücken auf die Namen der *URL's* einschränken, z.B. „Hole nur *URL's* die ein „muenchen.de“ am Ende des Rechnernamenes haben“.

Bei den aktuell verfügbaren Crawlern[1] werden verschiedene Strategien verfolgt:

Harvest: Harvest ist ein komplettes WorldWideWeb -Indexierungssytem. Es wurde an der Universität von Colorado entwickelt. Harvest besteht aus den drei folgenden Komponenten:

- *Gatherer* ist der Web-Crawler des Pakets. Gatherer ist ein Programm das die Informationen der verschiedenen Dokumente sammelt (to gather, *engl.* sammeln, zusammensuchen).

[1] Für eine komplette Übersicht an Webcrawlern sei auf http://info.webcrawler.com/mak/projects/robots/active/html/type.html verwiesen.

- *Broker* ist ein Programm, das die gesammelten Informationen mit dem Glimpse-Index[2] indexiert und für Abfragen zur Verfügung stellt.
- *Squid* ist ein verteilter Web-Cache. Web-Caches sind Programme, die alle Dokumente die die Benutzer des WorldWideWeb ansehen, zwischenspeichern. Diese Zwischenspeicherung hat zwei Vorteile:
  - Veringerung der Netzlast.
    Benutzer des WorldWideWeb sehen sich teilweise Dokumente an, die schon andere Leute in der gleichen Firma oder in der gleichen Stadt gelesen haben. Wenn man diese Dokumente zwischenspeichert erspart man sich das aufwendige Holen über das Internet.
  - Erhöhung der Geschwindigkeit.
    Ist das Dokument schon von jemanden gelesen worden, der den gleichen Web-Cache benutzt, ist es somit im Cache gespeichert und kann direkt von dort geladen werden.
    Als Nebeneffekt wird die Netzlast ins Internet geringer und nicht im Cache vorgehaltene Seiten können über die nun weniger benutzten Internetleitungen schneller geladen werden.
    Damit *Squid* nicht nach kurzer Zeit Unmengen an Festplattenkapazität benutzt, werden Seiten welche lange nicht gelesen wurden wieder aus dem Cache entfernt.

  Verteilt ist *Squid* insofern, als daß ein Squidcache sog. Nachbarn haben kann. Wenn ein Dokument im lokalen Cache nicht vorliegt, so fragt *Squid* dann einen Nachbarcache, ob dort das gewünschte Dokument vorliegt.

MomSpider: MomSpider wurde eigentlich dafür geschrieben, Websites auf Korrektheit der Links zu überprüfen.

Verweise in HTML-Dokumenten müssen nicht unbedingt existieren. Wenn man zum Beispiel auf ein Dokument "NameDesAuthors/DiesDokumentBeschreibtHTML" verweist, der Author nun aber sein Dokument in "NameDesAuthors/DiesDokumentBeschreibtHTML3.2" umbenennt, dann kann ich auf dieses Dokument nicht länger zugreifen.

MomSpider wurde geschrieben, um alle Links, die in den Hypertextdokumenten eines Servers vorkommen, auf Korrektheit - sprich Existenz - zu überprüfen.

MomSpider beinhaltet zwei Teile, zum einen die Implementation eines "Minicrawlers", zum anderen Bibliotheksfunktionen um einfach auf das WorldWideWeb zugreifen zu können.

[2] Der Glimpse-Index ist ein frei zugängliches Indizierungspaket von der Universität von Arizona. Mehr dazu auf: http://glimpse.cs.arizona.edu/

Die von Perl aus zugänglichen Bibliotheksfunktionen machen es sehr einfach mit regulären Ausdrücken zu arbeiten, da diese schon in Perl selbst ausgewertet werden. Aus diesem Grunde wurden schon viele Abarten dieses „Minicrawlers" implementiert und an spezielle Aufgaben angepaßt.

Scooter: *Scooter* ist der von vielen großen Webindizes benutzte Web-Crawler. *Scooter* ist, nach Aussage von *AltaVista*, der momentan am schnellsten arbeitende Web-Crawler[3]. Die hohe Geschwindigkeit, mit der Scooter arbeitet, ist auf verschiedene Strategien zurückzuführen:

- Ausnutzung des Betriebssystems.

  Es wird davon ausgegangen, daß das Betriebssystem die Aufgaben der Speicherverwaltung, des Auslagerns von nicht benötigten Daten auf Festplatten und die des Schedulings der Prozesse und Threads optimal erledigt.

- Multithreading.

  Für jede zu holende Seite werden separate Threads gestartet, so daß man während des Ladens von Seiten über langsame Netzverbindungen, schon andere Webseiten holen kann.

- Virtueller Speicher.

  Der erstellte Index mit einer Größe von ca. 50 Gigabyte, wird komplett in den virtuellen Arbeitsspeicher geladen. Diese Strategie ist die wohl konzeptionell am verblüffendsten. Das Programm benutzt den Index, als würde sich dieser im Arbeitsspeicher des Rechners befinden, auch wenn in der Realität nur ein paar Gigabyte Arbeitsspeicher zur Verfügung stehen.

  Man stützt sich hiermit vollkommen auf die (hoffentlich) optimale Arbeitsweise des Betriebssytem.

  Den großen Aufwand, hohe Geschwindigkeit bei der Bearbeitung von riesigen Datenbeständen zu erzielen, spart man sich damit und kann mehr Intelligenz in die Algorithmen des Indexierers stecken.

Bei der Konzeption der genannten Crawler wurden verschiedene Zielsetzungen verfolgt, weshalb sich die zugrundeliegenden Strategien der Informationssuche so stark unterscheiden.

3.2 Perl Version

In diesem Punkt wird Konzeption und Implementierung des von mir in Perl geschriebenen Web-Crawlers betrachtet. Als Grundlage dieses Web-Crawlers wurde die Funktionsbibliothek von *MomSpider* benutzt.

[3] http://scooter.pa-x.dec.com/

Dieser Web-Crawler stellt als solcher zwar keinen Ansatz zur Verbesserung bestehender Webindizes dar; jedoch in der Praxis hat sich dieser Crawler als sehr nützlich erwiesen, so daß ich ihn hier kurz umreiße.

3.2.1 Vorüberlegung

Für die Sprachenerkennung wurde ein Crawler benötigt, der gezielt Webseiten folgt und mit regulären Ausdrücken die gefundenen Dokumente und Verweise bearbeiten kann. Als Grundlage dafür wurde die www-lib[4], auf die auch MomSpider aufsetzt verwendet. Diese ist konzeptionell dafür ausgelegt, das Schreiben von speziellen Crawlern zu erleichtern.

Da MomSpider und die davon abgeleiteten Webcrawler für die Dokumentverfolgung in lokalen Netzen ausgelegt sind, habe ich, um eine akzeptable Geschwindigkeit im WorldWideWeb zu erzielen, das verteilte Cachingsystem von *Harvest* vorgeschaltet.

3.2.2 Konzeption

Bei der Konzeption ging es darum die folgenden Anforderungen zu erfüllen:

- Einfache Handhabung um bestimmten Webseiten zu folgen
- Reguläre Ausdrücke, um Hosts ein- bzw. auszugrenzen, als auch die Hypertextdokumente zu parsen
- Persistenz der geholten Seiten gewährleisten
- Wiederaufsetzen des letzten Stands bei abgebrochenen Suchvorgängen
- Anbindung an Programme zu weiteren Bearbeitung der geholten Seiten

Da von keinem der genannten Crawler alle, der in den Anforderungen gestellten, Punkte erfüllt wurden, mußte ein neues System implementiert werden. Um dabei den Aufwand für eine Implementation so klein wie möglich zu halten, wurde versucht bestehende Komponenten aus frei verfügbaren Crawlern zu verwenden.

Die lib-www stellte dabei das Fundament für den zu schreibenden Crawler dar. Diese Bibliothek stellt Funktionen zum Zugriff auf Webseiten, als auch der Extraktion von Links aus Hypertextdokumenten, zur Verfügung. Da Perl keinerlei Methoden zur nebenläufigen Prozeßsteuerung besitzt, ist es zur Performanzsteigerung notwendig gewesen, die zu holenden Webseiten vor dem eigentlichen Zugriff zwischenzuspeichern.

Harvest - bzw. *Squid* als Webcache - bot sich für diese Aufgabe an, da dieser imstande ist nebenläufig, verschiedene Webseiten zu holen. Man kann *Squid* so

[4]http://www.ics.uci.edu/pub/websoft/libwww-perl/

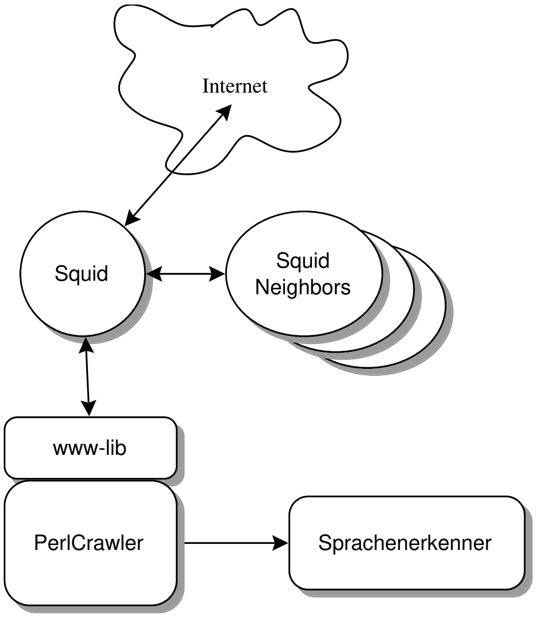

Abbildung 3.1: Aufbau des in Perl geschriebenen Crawlers: Der PerlCrawler holt mit den Funktionen der www-lib Hypertextdokumente und schickt diese an den Sprachenerkenner weiter. Falls die gewünschten Seiten nicht im Cache des lokalen *Squid* vorliegen fragt dieser seine Nachbarn nach den Dokumenten. Liegen bei den Nachbarn die Dokumente auch nicht vor werden die Dokumente aus dem Internet geladen.

konfigurieren, daß man eine Webseite, die sich nicht im Cache befindet, holt. Auf diese Weise ist es möglich an Squid Anfragen zu stellen und bei nicht vorliegen der entsprechenden Seite, diese von Squid holen lassen, während man sich unterdessen weiteren, zu holenden Seiten zuwendet.

Für den Rest des zu implementierenden Crawlers bot sich Perl als Programmiersprache an, da es in dieser zum einen sehr einfach ist reguläre Ausdrücke zu behandeln, als auch die Persistenz der geholten Daten sicherzustellen.

Damit war die Grundlage für die in Abbildung 3.1 dargestellte Struktur geliefert.

3.2.3 Implementierungs-Überlegungen

Für den inneren Aufbau des Programms mußten dann noch einige Designentscheidungen getroffen werden:

- Rechnerplattform bzw. Betriebssytemumgebung ?
- Welche Art der Konfiguration wird benutzt ?
- Wie werden die Daten persistent gehalten ?

- Welche Anbindung an die www-lib soll gewählt werden ?
- Wie soll die Anbindung an den Sprachenerkenner geschehen ?

Zum besseren Verständis der anstehenden Entscheidungen, seien die einzelnen Punkte näher erläutert:

Plattform: Die heterogene Arbeitsumgebung am CIS ließ mir bei dieser Entscheidung kaum Auswahlmöglichkeiten: ca. 40 Rechner laufen mit NeXT-STEP, zwei Rechner mit Digital Unix und fünf mit SunOS 4.*.

Da Perl relativ Plattformunabhängig ist, konnte ich - durch die Wahl von Perl - alle vorhandenen Hardwareplattformen als künftige Zielsysteme einplanen.

Konfiguration: Es gibt unter Unix zwei Arten Konfigurationen mit denen Programme meist arbeiten: Konfigurationsfiles oder Übergabeparameter. Ich entschied mich für die letztere Methode, da das Programm als nützliches Tool benutztbar sein soll und nicht erst mühsam mit Konfigurationsfiles eingerichtet werden muß.

So kann jeder künftige Benutzer die Parameter, die das Programm benötigt, einfach und schnell beim Programmaufruf übergeben. Dem Programm wird mittels regulärer Ausdrücke übergeben, welche Dokumente aus dem World-WideWeb zu holen sind und wie die Seiten für den Sprachenerkenner gefiltert werden sollen.

Datenhaltung: Für die persistene Datenhaltung bietet sich unter Unix, und besonders bei Benutzung der Programmiersprache Perl, die Verwendung von *dbm*-Datenbankfiles an. Speziell Perl bietet hier an, daß man Assioziative Arrays[5] automatisch als *dbm*-Files[6] verwaltet.

www-lib: Es stehen zwei Versionen der www-lib zur Auswahl: eine Funktionale und eine Objektorientierte.

Bei kurzen Testläufen mußte ich feststellen, daß die objektorientierte Version für das Durchsuchen von einigen dutzend Dokumenten schon Stunden benötigte, die funktionale Variante nur einige Sekunden. Dies machte die Wahl nicht schwer und so entschied ich mich die funktionale Version zu verweden.

[5] Assioziative Arrays sind „normale"-Arrays, die jedoch nicht mit natürlichen Zahlen, sondern mit Zeichenketten indiziert werden.

[6] *dbm*-Filefunktionen stehen in einer C-Bibliothek unter Unix zur Verfügung. Man speichert die Daten wie in einer Datenbank als Schlüssel/Wert - Paar ab. Die Bibliotheksfunktionen sind dann für das Erzeugen der Files, das Einfügen, Löschen und Wiederfinden der Daten verantwortlich.

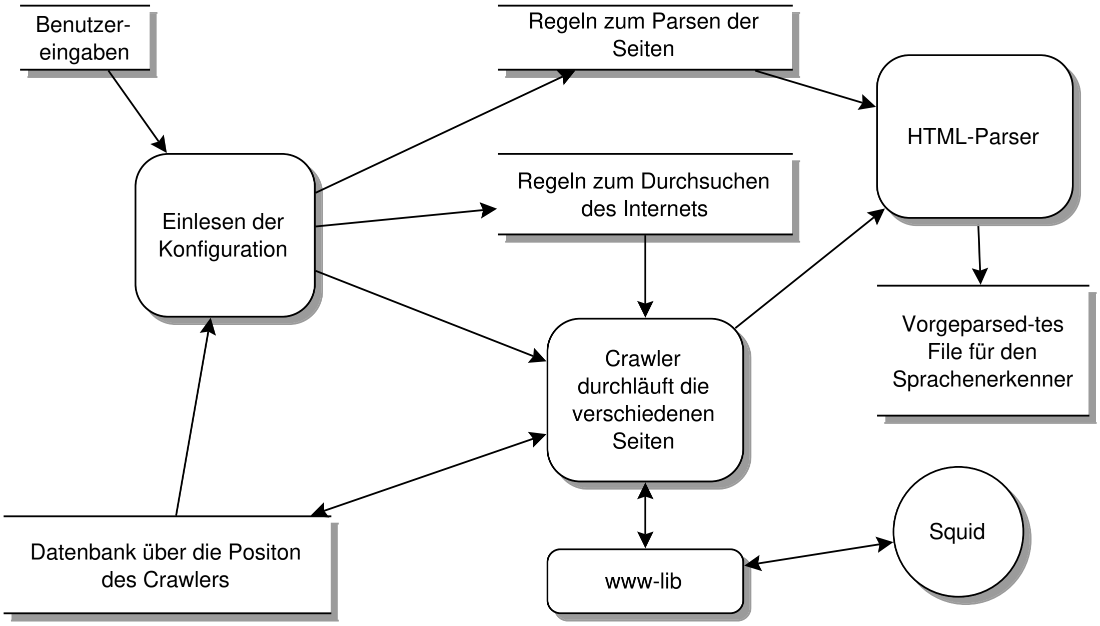

Abbildung 3.2: Datenflußdiagramm des in Perl geschriebenen Crawlers

Anbindung: Hier gibt es verschiedene Strategien im Unix- und Perl- Umfeld, um Programmteile miteinander zu verbinden. Via Pipes und Files bindet man meist konzeptionell verschiedene Programme an. Eng miteinander verwobene Programme kann man allerdings auch zu einem Programm zusammenlinken.

Ich entschied mich hierbei für eine Hybridlösung: So werden die Hypertextdokumente mit dem PerlCrawler geholt. Das Laden von Hypertextdokumenten beansprucht wenig Rechenzeit, dauert jedoch recht lange. Deshalb werden die gerade geholten Dokumente durch einen HTML-Parser geschickt, der die Seiten schon für die spätere Verwendung des Sprachenerkenners vorbereitet.

In dem HTML-Parser werden die Hypertextdokumente in Sätze und Paragraphen zerlegt, da dies für den Sprachenerkenner gutes Ausgangsformat ist.

Die Ausgabe der gefilterten Hypertextdokumente speichert dann der Crawler in ein File ab, aus dem dann später der Sprachenerkenner die notwendigen Daten lesen kann.

Nach den so gefällten Entscheidungen für das Design des Perl-Crawlers sah das Design wie in Abbildung 3.2 auf Seite 18 dargestellt aus.

3.2.4 Implementierungs-Probleme

Bei der Implementierung galt es dann einige Hindernisse zu überwinden:

Gleich zu Anfang fiel auf, daß die gewünschte Datenhaltung der noch zu holenden *URL's* mittels *dbm*-Files zu langsam war.

Die *URL's* wurden in Paaren: *UrlName/BinSchonGeholt* abgespeichtert. Nachdem einige tausend Dokumente geholt waren, wurde diese *dbm*-Datenbank einige hundert Megabyte groß, und der Zugriff auf die Daten entsprechend langsam.

Der Fehler liegt daran, daß die *UrlNamen* ihren Status verändern und die *dbm*-Bibliothek bei jeder Änderung einfach das alte *Schlüssel/Wert* Paar löscht und an das Fileende den neuen Wert schreibt.

Um diesen Fehler zum umgehen benutzte ich daraufhin die *gdbm*-Bibliothek[7]. Dies führte jedoch zu einer Verschlimmerung: das Betriebssystem fing schon nach kurzer Zeit an zu „thrashen", denn ein Fehler in der *gdbm*-Implementierung ließ den PerlCrawler nach ein paar Minuten einige hundert Megabyte Hauptspeicher anfordern.

Im Endefekt blieb nichts anderes übrig, als die Namen der schon bearbeiteten Hypertextdokumente in Textfiles abzulegen, um eine für das Projekt akzeptable Abarbeitungszeit zu erhalten.

Eine Dokumentation zur Benutzung des Perl-Crawlers ist im Anhang A beschrieben.

3.3 Verteilte Version

Im Folgenden widme ich mich nun voll einem der Hauptziele dieser Diplomarbeit: Das Crawlingverhalten der aktuellen Suchmaschinen zu verbessern. Dies wird durch ein, von mir entwickeltes, verteiltes, fehlertolerantes System Namens *Hoover* geschafft, welches nun erläutert wird.

Das Programm bekam den Namen *Hoover*, da es über den Daten des Internet schweben soll (*engl.* to hover) und das doppelte „oo" ist ein Bezug auf *Scooter*.

3.3.1 Vorüberlegung

Von den genannten Webcrawlern (Harvest, MomSpider und Scooter) sind eigentlich nur zwei im Hinblick auf ein schnelles Hypertextdokument-Retrieval geschrieben worden:

- Harvest versucht von einem Rechner aus, über mehrere Squid-Caches, auf das Internet zuzugreifen.
- Scooter holt von einer Maschine aus, in mehreren Threads, gleichzeitig verschiedene Seiten.

[7] *gdbm* (GNU dbm) ist eine frei verfügbare Version der *dbm*-Bibliothek.

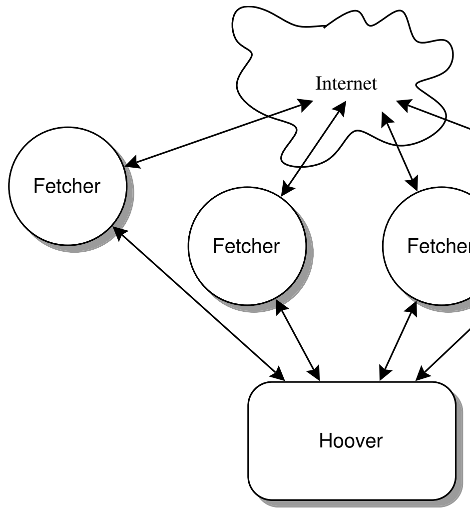

Abbildung 3.3: Grundgedanken bei der Konzeption des verteilten Crawlers.

Beiden Konzepten inherent ist das Vermeiden von Verzögerungen beim Holen von Hypertextdokumenten.

Von der Idee ist dieser Ansatz zwingend notwendig, will man den Programmablauf nicht, durch das zeitintensive Holen von Dokumenten über langsame Netzwerkverbindungen, unnötig verzögern.

Der Ablauf des Programms in mehreren Threads hat sicher den Vorteil, sich nicht dem Overhead der Programmierung von fehlertoleranten, verteilten Systemen auseinanderzusetzen. Sieht man sich die Zugriffsgeschwindigkeit von Scooter an, gibt dieses Konzept den Entwicklern sicher Recht.

Der Nachteil von Scooter ist, daß Scooter auf einem Rechner in mehreren hundert Threads arbeitet, und so einen großen Teil der CPU-Leistung, für Verwaltungs- und Schedulingaufgaben dieser Threads verbraucht.

Eine Symbiose dieser beiden, schon hochgradig optimierten, Suchmaschinen ist ein, auf mehreren Rechnern verteiltes, und auf den einzelnen Rechnern jeweils multithreaded ablaufendes Crawlen.

3.3.2 Konzeption

Bei der Konzeption dieser Symbiose mußte gleich zu Beginn große Sorgfalt bei der Auswahl der einzelnen Komponenten, in Hinblick auf die später daraus resultierende Geschwindigkeit, verwand werden. Überblickshaft ergab sich durch die in 3.3.1 angestellten Vorüberlegungen das in Abbildung 3.3 dargestellte Bild.

Die dargestellten, Fetcher genannten Prozesse, laufen jeweils auf einem eigenen Rechner. Von diesen Rechnern wird dann das Internet nach Seiten durchsucht, die der kontrollierende, Hoover genannte Prozeß dem jeweiligen Fetcher vorgibt. Die Fetcher laufen zudem multithreaded, damit damit nicht unnötig Rechenkapazität vergeudet wird.

Der Hoover Prozeß läuft auch auf einem separaten Rechner ab und übernimmt die Koordination der Fetcher. Zum einen werden Anfragen nach zu holenden Webseiten auf die Fetcher verteilt, als auch die von den Fetchern geholten Webseiten an andere Programme (z.B. Indexer) weitergereicht.

Durch die folgenden Anforderungen, die dieses verteilte System zu erfüllen hat konnte dann weiter die Spezifizierung der einzelnen Komponenten vorangetrieben werden:

Handhabung: Suchmachinen die komplette Netzwerke mit hoher Performanz durchsuchen sollten nicht „mal eben so“ gestartet werden können. Wenn jeder Benutzer Daten sehr schnell aus dem Internet mit verteilten Systemen holt, setzt das die gesamte Geschwindigkeit des Netzes herab.

Zudem sollte es eine spezielle Konfiguration erlauben Netzwerke regelmäßig, mit bestimmten Regeln, zu durchsuchen.

Reguläre Ausdrücke: Reguläre Ausdrücke sind sehr rechenzeitintensiv. Deshalb ist ein vollständiges parsen von Hypertextdokumenten nicht sehr sinnvoll.

Es sollte trotzdem möglich sein, daß man bei der Konfiguration die zu durchsuchenden Netzwerke mit einfachen Regeln beschreiben kann.

Persistenz: Hier ist die Persistenz viel wichtiger als bei einem nicht verteilt arbeitendem System. Da die geholten Datenmengen viel umfangreicher sind, ist es im industriellen Einsatz sehr kostenintensiv diese Datenmengen erneut zu holen.

Wiederaufsetzen: Das Wiederaufsetzen ist aus dem gleichen Grund wie die Persistenz zu implementieren.

Anbindung: Da hierbei die anzubindenden Programme zeitlich versetzt auf den Daten arbeiten, ist hier nur ein entsprechendes Übergabeformat der Daten zu konzipieren.

Die aufgelisteten Anforderungen sind von *Scooter* und *Harvest* schon erfüllt, so daß eine Verbesserung bestehender Suchmaschinen nur in Bezug auf die Datenmenge und Geschwindigkeit erzielt werden kann.

Um die Geschwindigkeit zu erhöhen wird, wie schon angesprochen die Arbeit des Holens von Hypertextdokumenten, als auch die Koordination welche Seiten geholt werden sollen, auf verschiedene Rechner verteilt. Man kann nun natürlich auch weitere, notwendige Arbeiten auf die Fetcher verteilen:

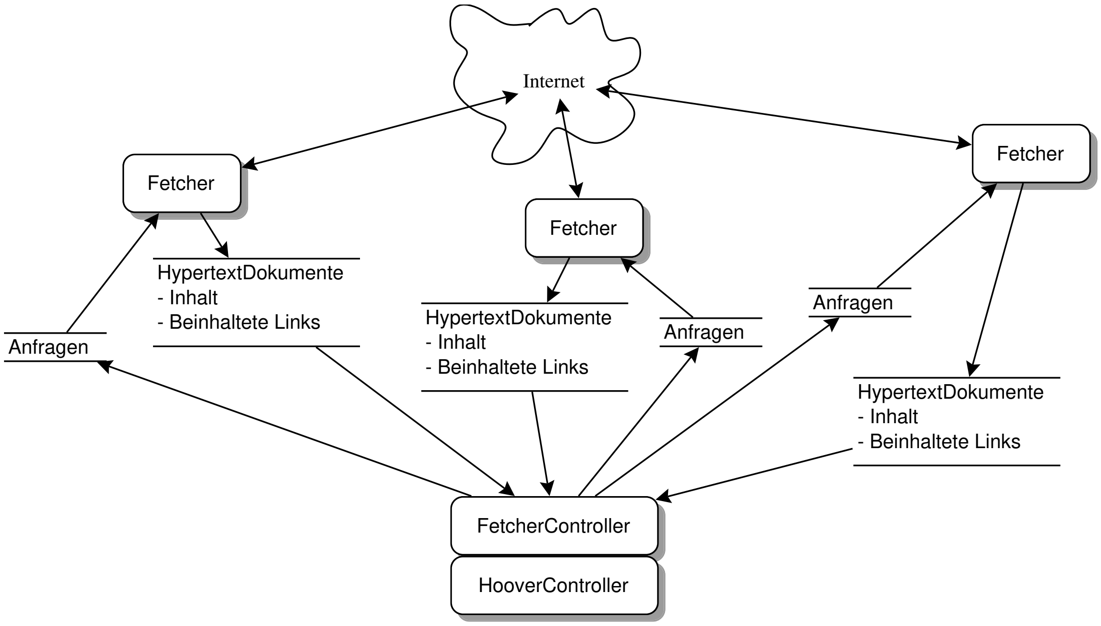

Abbildung 3.4: Erweiterte Konzeption des verteilten Crawlers.

Weitere Arbeiten von Suchmaschinen sind das Parsen der Dokumente nach Hyperlinks, das Parsen des „/robots.txt" Files[8] und die Normalisierung[9] von Hyperlinks. Von diesen Aufgaben kann man ohne Umstände das Parsen der Dokumente nach Hyperlinks und deren Normalisierung auf die Fetcher verteilen, so daß man den kontrollierenden Hooverprozeß noch einmal von Rechenlast erleichtert.

Die einzige Aufgabe des Hoverprozesses besteht dann im Verwalten der gefundenen und zu holenden Hypertextdokumenten und dem Verteilen der Aufträge an die existierenden Fetcher (Abbildung 3.4).

3.3.3 Implementierungs-Überlegungen

Um mit der Implementierung der Suchmaschine beginnen zu können mußten vorab noch einige Entscheidungen des zu wählenden Designs getroffen werden. Hierbei war die Auswahl der Komponenten wesentlich freier, als dies beim Perlcrawler der Fall war. Es konnte nicht auf schon voher implementierte Teilelemente zurückgegriffen werden weshalb die folgende Frage die Entwicklung maßgeblich bestimmte:

- Wie soll die Verteilung des Systems implementiert werden ?

Diese Frage setzt allerdings schon eine gewisse Vorkenntnis über verteilte Systeme voraus; es sei an dieser Stelle auf das Buch von Sape Mullender[Mullender]

[8] Wird im Abschnitt der Implementierung noch näher erklärt

[9] Normalisieren ist der Prozeß, in dem versucht wird den Links die man in Hypertextdokumenten folgt eindeutige Namen zu geben. (Wir noch erläutert).

verwiesen. Es sind hier zwei gegensätzliche Kriterien bei der Implementation des beschriebenen Systems in Einklang zu bringen.

Auf der einen Seite muß man davon ausgehen, daß die Fetcher und die Computer auf denen sie laufen zu jedem beliebigen Zeitpunkt nicht länger verfügbar sein können. So können die Programme abstürzen, die Computer vom Netzwerk getrennt oder einfach nur das Haus, in dem sie sich befinden, abbrennen. Solche Situationen dürfen jedoch nicht zu einem Stillstehen der Suchmaschine führen.

Auf der anderen Seite ist durch die Übersendung von Datenstrukturen eine hohe Abstraktionsebene gegeben, die die meisten Implementationen von verteilten Systemen vermissen lassen.

Ich entschied mich für die „portable distributed objects“ (*pdo*[10]) genannte Enwicklungsumgebung von NeXT, da diese es ermöglicht ohne großen Aufwand komplexe Datenstrukturen und sogar ganze Programme im Netzwerk zu verteilen. Die Fehlererkennung und Behandlung ist in dieser Umgebung recht einfach zu implementieren, so daß die gewünschten Anforderungen von dieser Entwicklungsumgebung erfüllt wurden.

Da *pdo* auf Objective-C als Programmiersprache aufsetzt, war mit der Wahl des Verteilungsmechanismus auch schon fast die Wahl der Programmiersprache gefallen. Man kann zwar C++ oder C-Programme an das *pdo*-System anbinden, da die Klassenbibliotheken von Objective-C jedoch schon sehr komfortable Funktionen zur Stringbearbeitung, als auch persistenten Abspeicherung von Datenobjekten bieten, entschied ich mich diese direkt in Objektive-C zu benutzen und das System in Objective-C zu implementieren.

3.3.4 Implementierung

Nachdem nun alle Designfragen geklärt waren, konnte mit der Implementierung begonnen werden.

Eine Einarbeitung in die zur Verfügung stehenden Klassenbibliotheken ließ mich daraufhin den Hooverprozeß wie in Abbildung 3.5 dargestellt unterteilen.

Die einzeln dargestellten Objekte haben folgenden Aufgaben:

HooverController: Dieses Objekt hat die Aufgabe, das Konfigurationsfile zu lesen und mit den darin enthaltenen Daten den *GeneralScanner*, das *Sites-Dictionary*, das *Sites sorted by next accessdate array* und den *FetcherController* zu initialisieren.

Die im Konfigurationsfile benannten Startseiten werden ins *Sites sorted by next accessdate array* gelesen.

[10] *pdo* wird von NeXT (http://www.next.com/) commerziell auf Digital-Unix, HP-UX, NeXT-OpenStep/Mach, Sun-Solaris und Microsoft-Windows NT/95 angeboten. Es gibt auch eine unter dem GNU-Copyleft veröffentlichte freie Version. Ich habe bei der Implementierung mit der OpenStep/Mach Version auf Motorola- als auch Intel-Hardware gearbeitet.

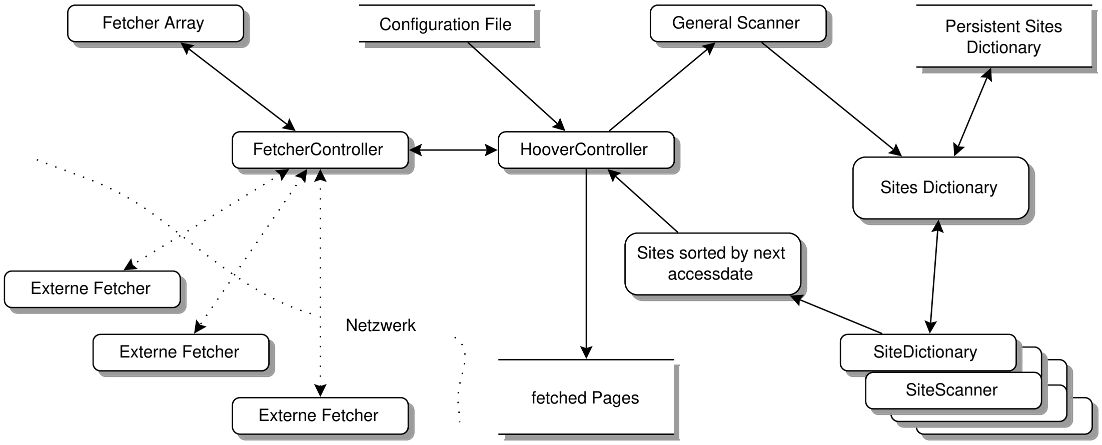

Abbildung 3.5: Erweiterte Konzeption des verteilten Crawlers.

Aus dem *Sites sorted by next accessdate array* wird dann das erste Objekt (*SiteDictionary*) entnommen und nach einer unbekannten Seite befragt. Diese Seite wird dann an den *FetcherController* zum Retrieven weitergereicht.

Kommen Hypertextdokumente vom FetcherController zurrück, so wird deren Inhalt in der *fetched Page*-Datei gespeichert und die in der Seite enthaltenen Hyperlinks an den *GeneralScanner* geschickt.

GeneralScanner: Vom *HooverController* wird dieses Objekt mit Regeln für zulässige Hypertextdokumente instanziert, z.B. Nur Dokumente die „.html“ am Ende des Pfadnamens haben und aus der „.gov“ oder „.mil“ Internetdomain kommen.

Während des eigentlichen Programmablaufs bekommt der *GeneralScanner* dann die von den Fetchern geholten Arrays von URL's zurrück, um diese Mittels der Regeln auszusortieren und die gültigen URL's an das *SitesDictionary* weiterzureichen.

SiteDictionary: Solch ein Objekt enthält alle persistenten Daten über eine *Site*. Hypertextdokumente werden über einen Sitenamen und einen Dokumentnamen angesprochen.

Damit eine Übersicht über die schon geholten Dokumente besteht, werden die Dokumente nach dem Status (schon geholt, noch zu holen) in dem jeweiligen *SiteDictionary* gespeichert. Überdies halten *SiteDictionary* Objekte noch weitere Informationen zur Verfügung:

- Internet Domainname des Hosts
- Internet Adresse (und Portnummer)
- Datum des letzten Zugriffs

- Transferrate des letzten Zugriffs
- Datum des nächsten Zugriffs
- SiteScanner
- Inhalt der „/robots.txt"-Datei
- Schon bekannte (geholte) Hypertextdokumente
- Noch unbekannte (zu holende) Hypertextdokumente

Das Objekt arbeitet wie ein Wörterbuch[11]: man fragt z.B. nach „lastaccess" um das Datum des letzten Zugriffs zu erfahren. Diese Datenstruktur ermöglicht es sehr einfach neue Attribute einzuführen, um evtl. geänderten Standards zu genügen.

SitesDictionary: In diesem Objekt sind alle *SiteDictionary* Objekte über den Namen der *Site* gespeichert. Fragt man z.B. nach dem Namen „www.cis.uni-muenchen.de" bekommt man das dazugehörige *SiteDictionary*.

Neben dieser Wörterbuchfunktion hat dieses Objekt noch die Aufgabe die beinhaltenten *SiteDictionary* Objekte in regelmäßigen Abständen konsistent abzuspeichern.

Beim Start des Systems werden dann aus den gespeicherten Daten die *SiteDictionary*-Objekte wieder erzeugt.

SiteScanner: Dieses Objekt bekommt jeden Hypertextlink, der durch den *GeneralScanner* nicht herausgefiltert wurde und zum InternetSite des dazugehörigen *SitesDictionary* gehört. Hier wird nach Regeln, die im „/robots.txt" der jeweiligen Internetsite stehen, der neue Hypertextdokumentname gefiltert.

An dieser Stelle bietet es sich an über die „/robots.txt"-Datei zu sprechen.

Viele Internetdokumente werden entweder für den Benutzer erzeugt (z.B. Resultate auf Suchanfragen), täglich geändert oder sind sonst für Suchmaschinen uninteressant (z.B. Webseiten die nur über Passwörter zugänglich sind). Sie können vom Betreiber einer Internetsite als uninteressant für Roboter (Suchmaschinen) makiert werden.

Um nun dem sog. *robotsexclusion protocol*[12] zu entsprechen muß man das Hypertextdokument „/robots.txt" auf der jeweiligen Internetsite einlesen und die dort enthaltenen Regeln anwenden.

So kann man im „/robots.txt" sagen, welche Pfadnamen zugänglich sind, und welche nicht. Ein Beispiel macht dies anschaulicher:

[11] dictionary *engl.* Wörterbuch.

[12] http://info.webcrawler.com/mak/projects/robots/robots.html

Auf der Internetsite „thestar.com.my“ beinhaltet das „/robots.txt“-File folgendes:

```
User-agent: *
Disallow: /cgi-bin/
```

D.h, jedem („*“) UserAgent (Roboter) ist es verboten, Dokumente zu holen, die mit „/cgi-bin/“ anfangen.

Es sei noch angemerkt das dieses Protokol kein „Muß“ für Suchmaschinen ist; es gehört jedoch zum „guten Ton“.

Sites sorted by nextaccess: Dieses Objekt bietet im Endefekt die gleiche Datenstruktur, wie die des *Sitedictionary*-Objekts an, außer daß man hier einfach nach dem ersten Objekt in dem Array fragt, um den Namen des als nächstes zu holenden Hypertextdokumentes zu erfahren.

Der Aufwand, die *SiteDictionary* Objekte nach dem nächsten Datum zu sortieren, hat folgenden Hintergedanken: Die Suchmaschine beginnt z.B. mit dem Dokument

- „MeinInstitutsServer/MeineEigeneInternetSeite“

und holt dieses. Nach dem Parsen des Dokumentes sind die folgenden Seiten bekannt :

- MeinInstitutsServer/DieSeiteDesChefs
- MeinInstitutsServer/DieSeiteSeinesStellvertreters
- MeinInstitutsServer/MeineEigeneInternetSeiteMitWitzen
- MeinInstitutsServer/DieSeiteMeinesArbeitskollegen

Die Suchmaschine versucht folglich, im nächsten Durchlauf die vier genannten Seiten vom *InstitutsServer*-Rechner zu holen. Wenn die Suchmaschine hochoptimiert arbeitet, dann holt sie die genannten Seiten gleichzeitig vom *InstitutsServer*.

Falls die Suchmaschine keine Vorkehrungen trifft, dieses „hammering“[13] genannte Verhalten zu vermeiden, dann kann es in der Praxis passieren, daß einige hundert Hypertextdokumente zum gleichen Zeitpunkt, von einund demselben Rechner geholt werden.

Dieses Verhalten führt dann zu zwei Resultaten:

Zum einen wird der nun „thrashen“-de Rechner sehr zögerlich auf die Anfragen reagieren und die Performanz der Suchmaschine leidet erheblich; zum

[13]to hammer *engl.* hämmern. *hier* niederknüppeln, da der Rechner so lange mit Anfragen bombadiert wird, bis er zu *thrashen* anfängt und sich nicht mehr „regen“ kann.

anderen macht man sich keine Freunde in der Internetgemeinde, so daß die Serverbesitzer die Suchmaschine verfluchen und via *robotsexclusion protocol* den Zugriff auf ihre Site total verbieten.

FetcherController: Diese Objekt übernimmt den kompletten Anschluß des Systems an die auf anderen Rechnern laufenden Fetcherprozesse. Dabei wird ein *Hoover*-Server netzwerkweit zur Verfügung gestellt, an welchem sich dann Fetcher anmelden und sich „zur Arbeit bereit“ anmelden.

Fetcher die momentan wenig ausgelastet sind, werden mit neuen Anfragen nach Webseiten beauftragt. Fetcher die Webseiten geholt haben senden diese dem *FetcherController*, der diese Seiten dann an dem *HooverController* weiterreicht.

Um festzustellen ob Fetcher noch am Arbeiten sind, werden diese, wenn sie eine zeitlang keine bearbeiteten Aufträge abgeliefert haben „angepingt“[14].

Melden sich die Fetcher auf dieses „Ping“ ein paar Mal nicht, werden sie aus der Gruppe der arbeitenden Fetcher herausgenommen und alle ihnen anvertrauten Aufträge werden wieder in die „noch zu erledigen“-Warteschlange gestellt.

FetcherArray: Hier werden die momentan zur Verfügung stehenden Fetcher, in der Reihenfolge ihrer noch freien Arbeitskraft nach einsortiert. Auch werden die Aufträge, die die einzelnen Fetcher erledigen, hier festgehalten. So kann man bei Ausfall eines Fetchers, die Arbeit, die er zu erledigen hatte, nachgehalten werden.

Die einzelnen Fetcherprozesse sind wie in Abbildung 3.6 dargestellt aufgebaut.

Fetcher: Dieser nimmt Kontakt mit dem *FetcherController* des im Netwerk laufenden *Hoover* auf und meldet sich dort mit seiner zur Verfügung stehenden Arbeitskapazität an.

Wenn Anfragen vom *FetcherController* kommen wird der *ThreadController* nach einem freien *Worker* gefragt und dieser mit der Anfrage beauftragt.

Von den *Worker*-Objekten fertig geholte Webseiten werden an *Hoover* weitergeleitet.

ThreadController: Bei diesem melden sich freie *Worker* an und diese gibt er dann dem *Fetcher* falls dieser danach verlangt.

Dieses Objekt bildet, die eigentliche asyncrone Kommunikation mit Threads als syncrone ab. Dies vereinfacht multithreaded ablaufende Programme zu implementieren.

[14]Das *Ping-Pong* Protokoll funktioniert so: Man sendet dem zu überprüfenden Prozeß ein „Ping“ (einen Glockenschlag) worauf dieser mit „Pong“ antwortet. Antwortet der Prozeß nicht innerhalb einer festgelegten Zeit wird er als „tot“ erklärt.

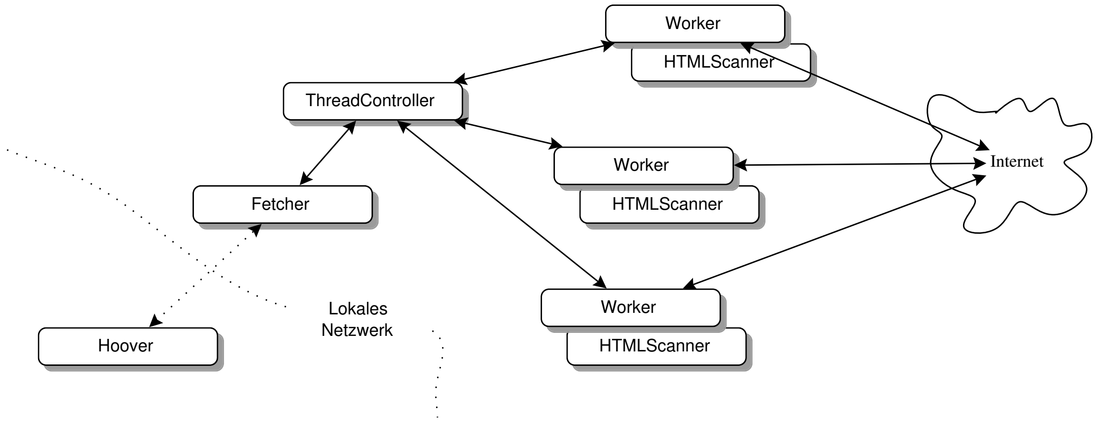

Abbildung 3.6: Aufbau der Fetcherprozesse.

Intern geschieht die Kommunikation über Semaphore. Normalerweise ist es recht schwierig eine Kommunikation mit Semaphoren zu implementieren. In diesem Objekt wird nur diese Kommunikation betrachtet, und nicht die speziellen Probleme des Hauptprogramms. So ist der Aufwand, Threads zu programmieren, aufgeteilt und die gesamte Implementation wird vereinfacht.

Worker: Dieses Objekt bildet die eigentliche Kommunikation mit dem Internet. Er holt Seiten von den verlangten Websites und parsed diese mit dem *HTMLScanner* nach gültigen Hyperlinks.

Wenn die Arbeit beendet ist, sendet er das geparste Hypertextdokument an den *Fetcher* zurück. Wenn ein Dokument nicht geholt werden konnte (z.B. der Rechner, von dem das Dokument geholt werden sollte, nicht erreichbar war), dann meldet der *Worker* auch dies dem *Fetcher* zurück.

Danach meldet sich der *Worker* beim *ThreadController* zurück.

HTMLScanner: Dieser parsed die HTML Seiten (sonstige Hypertextdokumente werden in der akuellen Version noch nicht unterstützt) nach Hyperlinks und gibt die gefundenen Links in einem Datenobjekt zurrück an den *Worker*.

3.3.5 Implementierung-Probleme

Bei der eigentlichen Implementierung gab es dann noch einige Probleme zu umschiffen.

So mußte gleich zu Anfang festgestellt werden, daß die Sortierroutine der mitgelieferten Klassenbibliotheken nur das komplette Array sortiert. Da dies bei einigen zehntausend Sites zu zeitaufwendig ist, mußte dann noch eine *SortedArray*-Klasse geschrieben werden.

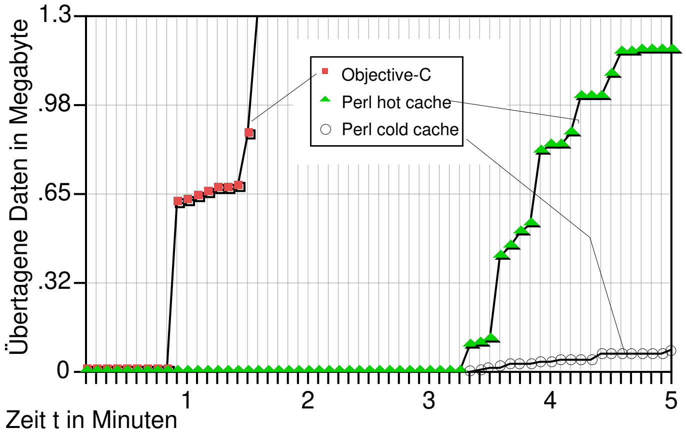

Abbildung 3.7: Vergleich zwischen dem der Perl und der Objective-C Version. Es durften nur HTML-Hypertextdokumente aus Norwegen geholt werden. Startpunkt waren 10 000 norwegische Links.

Bei dieser Klasse wird nun jedes Objekt beim Einfügen gleich an der richtigen Stelle eingefügt. Implementiert wurde das „richtige Eingfügen“ mittels binärer Suche.

Bei der Implementierung der *Fetcher* wurde zuerst die Arbeit des *ThreadController*-objekts im *Fetcher*-Objekt erledigt. Da das *pdo*-Objekt, welches für die Netzanbindung zuständig ist, nicht threadsafe ist, mußte der *ThreadController* dazwischengesetzt werden.

3.3.6 Leistungsanalyse

Nachdem nun das komplette System fertig implementiert war, blieb abzuwarten wie die vorher getroffenen Designentscheidungen sich auf die Geschwindigkeit der Suchmaschine ausgewirkt haben.

Die in Abbildung 3.7 zeigt die Performancesteigerung von der Perl/squid Version zur Objective-C Version. Die Perl Version lief auf einer Alpha5000 von Digital, die Objective-C Version auf einem PentiumPro System. Die Perl Variante sieht man dabei mit zwei unterschiedlichen Kurven:

Cache cold: Dies ist der erste Lauf der Perl Version, hier wird nur ein Teil der Daten gleich geholt. Seiten welche nach 3 Sekunden nicht geholt wurden werden von *squid* geholt.

Cache hot: Dies ist der zweite Lauf, hier sind alle Seiten schon im Cache und es werden nur noch nicht bekannte Seiten von *squid* im Hintergrund geholt.

Objective-C: Zum Vergleich die Objective-C Version, welche nach 5 Minuten schon 17 Megabyte direkt aus Norwegen geholt hat, ohne das die Daten schon lokal vorgelegen wären.

Nach einer Minute ist die Seite mit den 10000 Links geparsed und die „/robots.txt“-Seiten werden geholt.

Nach eineinhalb Minuten sind alle „/robots.txt“-Seiten geholt und die eigentlichen Seiten werden von den Fetchern dann aus Norwegen geladen.

Wie zu erwarten ist *Hoover* wesentlich schneller als die Perl Version. Von *Scooter* ist bekannt, daß das Programm auf einer Digital-TurboLaser rund 500 kByte/s aus dem Internet holt. *Hoover* ist mit rund 120 kByte/s auf einem Intel-PentiumPro System nicht schneller, der Hardwarekostenaufwand ist aber um einige Größenordnungen geringer.

3.3.7 Verbesserungsmöglichkeiten

Noch während der Implementierung wurde ersichtlich, daß nun die Performance der Suchmaschine zum Großteil von der Kommunikation der *Fetcher* mit dem *Hoover* abhängt.

In der ersten Version wurden die Hypertextdokumente als Proxyobjekte den *Fetcher*prozessen zugänglich gemacht.

Proxyobjekte sehen für den lokalen Prozeß aus, wie lokale Objekte. Werden nun Nachrichten an ein Proxyobjekt geschickt, verarbeitet es diese Nachrichten nicht selbst, sondern schickt die Nachrichten an den Server, wo die „wirklichen“ Objekte benachrichtigt werden.

Dies hatte zur Folge, daß jedesmal wenn ein Hypertextdokument geholt werden sollte eine entsprechende Nachricht an den *Fetcher* geschickt wurde. Dieser informierte sich dann ersteinmal über die zu holende Seite indem er das Objekt, welches in Wirklichkeit nur im *Hoover* existierte, befragte, welches Dokument denn zu holen sei.

Als erste Optimierung wurden die zu holenden Hypertextdokumente als Kopie den *Fetcher*prozessen geschickt, so daß die Fetcher die Objekte „direkt“ und nicht über den Netzwerkumweg befragen konnten. Den Geschwindigkeitszuwachs dieser Verbesserung ist in Abbildung 3.8 dargestellt.

Als weitere Optimierung könnte man die Aufträge die an *Fetcher* vergeben werden bündeln. D.h. nicht für jedes Hypertextdokument welches geholt werden soll eine eigene Anfrage senden, sondern zu Kolonnen von mehreren Aufträgen.

Dies würde die Größe der Nachrichten, die über das Netzwerk geschickt werden , vergrößern, jedoch die Anzahl erheblich verkleinern. Da bei jeder Nachricht immer ein kleiner, gleich großer Overhead entsteht, hätte diese Verbesserung Aussichten auf eine leichte Geschwindigkeitssteigerung der Suchmaschine.

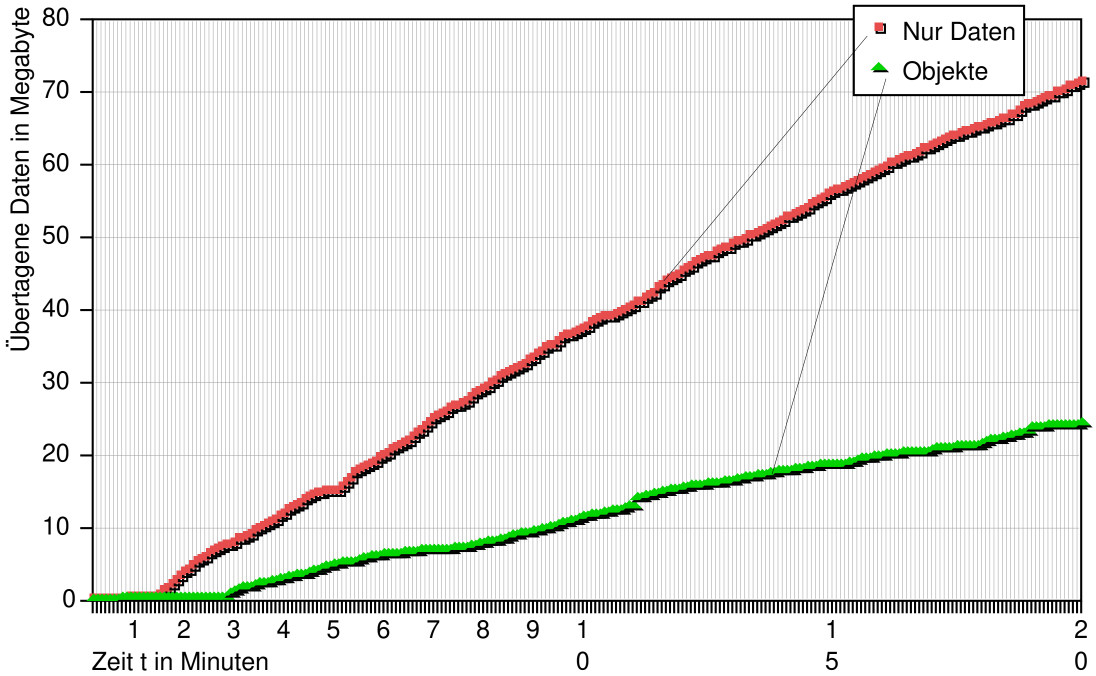

Abbildung 3.8: Vergleich zwischen dem Übertragen von Daten und dem Benutzen von Proxyobjekten. Es durften nur HTML-Hypertextdokumente aus Norwegen geholt werden. Startpunkt waren 10 000 norwegische Links.

Kapitel 4

Sprachenerkenner

In diesem Kapitel wird zurerst erläutert wozu eine Sprachenerkennung benötigt wird und wie man sie implementieren kann. Daran anschließend wird gezeigt wie man schrittweise die Erkennungsquoten verbessern kann.

4.1 Sinn und Ziel der Sprachenerkennung

Wenn man heutzutage im Internet Begriffe durch Eingabe eines Schlagwortes sucht, benutzt man meist einsprachige Wörter wie z.B. „Suchmaschinen“„Bundeskanzler“ oder „Gummistiefel“ und bekommt dann Seiten, welche das gesuchte Wort enthalten.

Schwieriger wird es da schon, wenn man nach Seiten, die etwas mit „Film“ zu tun haben, sucht. Hier erhält man bei den gänigen Suchmaschinen zum Großteil nur englischsprachige Seiten.

Man kann nun natürlich den Begriff versuchen etwas eingrenzen z.B. „Film and not Movie“. Man erhält damit alle Seiten in denen das Wort „Film“ vorkommt, nicht jedoch das Wort „Movie“ (*engl.* Film). Eine „richtige„ Sprachenerkennung ist dies nicht, speziell wenn man die Übersetzungen für die Schlagworte nicht kennt.

Bei einer „richtigen“ Sprachenerkennung sollte es möglich sein, die Sprache(n) in denen die Hypertextdokumente des Ergebnisraums auf eine Suchanfrage geschrieben sind, auszuwählen: „Suche nach *Film* in deutschen und französischen Dokumenten“.

Für Informationssuchende, welche der englischen Sprache nicht mächtig sind, würde damit das Internet viel einfacher benutzbar.

4.2 Vorüberlegung

Im Internet enthaltene Dokumente müssen keinen „Tag“[1] für eine Sprache haben, obwohl sog. Metatags hierfür existieren[2].

Man kann zum Teil die Sprache jedoch automatisch, durch die Kodierung der Buchstaben, erkennen. So sind z.B. Japanisch und Koreanisch durch die Zeichenwahl eindeutig bestimmt.

Wenn man sich vor Augen führt, wie ein Mensch eine Sprache erkennt, kann man evtl. Rückschlüsse für einen, auf einem Rechner implementierbaren Algorithmus bekommen. Als Mensch hat man einen Wortschatz der die wichtigen Wörter für die Muttersprache und die gesprochenen Fremdsprachen enthält. Zusätzlich weiß man, wie sich bestimmte Sprachen anhören, ohne die Sprachen jeweils zu können.

So kann man einigermaßen sicher Fränzösisch von Spanisch unterscheiden, auch wenn man beiden Sprachen nicht mächtig ist.

Man muß also irgendwie versuchen, dieses Wissen in einen Algorithmus zu integrieren. Erster Anhaltspunkt sind sicher Wörterbücher, die man auch als Mensch benutzt. Diese lassen sich einfach im Rechner handhaben - eine Spracherkennung auf akustischer Basis ist auch gar nicht nötig, da die im Internet vorliegenden Dokumente vorwiegend schriftlicher Natur sind.

4.3 Wörterbücher und Wortfrequenzen

Da man Anfangs noch keine Wörterbücher über die zu erkennenden Sprachen hat, muß man sich überlegen, wie man solche generieren kann, denn die gesuchten Wörter sind im Internet schon vorhanden.

Als wahrscheinlich kann man gelten lassen, daß die meisten Internetbenutzer ihre Dokumente in Englisch bzw. der jeweils üblichen Landessprache schreiben. Überdies sind die meisten Länder mit einer eigenen Internetdomain im Internet vertreten. Nur Amerika hat neben der *us* auch noch *edu,gov,mil,*... als Domainnamen.

Dies Wissen kann man einfach dazu verwenden, Wörterbücher automatisch zu generieren: Man holt sich Webseiten nach Internetdomänen getrennt und generiert aus den gefundenen Dokumenten Wortlisten.

Sehr schnell muß man nun aber feststellen, daß in Europa alle Wörter aus dem Englischen, Französizschen, Italienischen, Spanischen und Deutschen in allen Wortlisten vertreten sind. Sieht man sich jedoch die Worthäufigkeiten an, so überwiegen die Landesspezifischen Wörter und Englisch.

---

[1] tag *engl.* Markierung. - Die Dokumente sind nicht mit einer Markierung versehen die sie als „deutsch“, „französisch“ oder „deutsch und englisch“ ausweist.

[2] http://ds.internic.net/rfc/rfc1945.txt (RFC 1945 Abschnitt D.2.5)

der	.0223658143
und	.0202616321
die	.0195573927
the	.0192777032
in	.0166147651

Tabelle 4.1: Erstes Ergebnis einer Wortliste aus deutschen HTML-Dokumenten erzeugt. Zu sehen sind noch die englischen Wörter „the“ und „in“.

Für Deutsch erhält man mit dieser Methode Wörterbücher wie das in Tabelle 4.1 dargestellte.

Diese Tabelle bedeutet, daß jedes fünfzigste Wort (2%) das Wort „der“ auf deutschen Hypertextdokumenten vorkommt. Das englische Wort „the“ ist mit 1.9% auch sehr häufig.

Zum Erstellen dieser Wortliste habe ich allerdings schon vorrausgesetzt, daß man einen *lowercase*-Filter hat. Wie ein solcher zu implementieren ist, erkläre ich im nächsten Abschnitt.

4.4 Encodings und Kleinbuchstaben

Wie schon im vorangehenden Text kurz erläutert, kann man bestimmte Sprachen durch die Art der benutzten Zeichen erkennen. Dies funktioniert wie schon erwähnt für die meisten asiatischen Sprachen z.B. Chinesisch, Japanisch, Koreanisch ,. . . .

Für solche Hypertextdokumente bietet sich an, eine Art Vorfilter zu implementieren, welcher, über die Art der benutzten Zeichen, die Sprache herausfindet[3]. Übrig bleiben dann nur noch Dokumente, die in europäischen Zeichensätzen geschrieben sind, bei denen der Zeichensatz nicht so einfach erkannt werden kann.

Das Problem bei den europäischen Zeichensätzen ist, daß die Authoren von Hypertextdokumenten dieses Zeichensatzproblem gar nicht erkennen, da es auf Ihrem Computerbildschirm immer „richtig“ aussieht.

HTML-Dokumente sind im ISO-Latin1 Zeichensatz gesetzt - so der Standard. Authoren in Russland (kyrillisches Alphabet) oder den slavischen Ländern können aber mit ISO-Latin1 nicht alle Zeichen des jeweiligen Alphabets benutzen. Nun bietet sich zum einen eine Transliteration der nicht vorhandenen Zeichen in den ASCII-Code an (im Deutschen z.B. ä $\rightarrow$ ae). Zum anderen aber haben diese Authoren meist Schriftarten bei denen für Ihre Sprache die enthaltenen Zeichen auf Zeichen gelegt wurden, die sie nicht benötigen.

[3]Eine Implementierung eines solchen Filters hat Eric Kidd von AltaVista schon getätigt, weshalb ich darauf nicht näher eingehe.

An einem Beispiel läßt sich das einfacher erklären. Angenommen wir können drei Zeichen benutzen und der amerikanische Standard sieht eine Belegung wie folgt vor:

1.Zeichen → „f“, 2.Zeichen → „o“, 3.Zeichen → „r“.

Wir benötigen jedoch „f“,„ü“ und „r“ , nicht jedoch „o“. Belegen wir das dritte Zeichen mit „ü“ und wir können somit alle gewünschten Zeichen darstellen. Versucht jemand in Amerika jedoch dann unsere Dokumente zu lesen werden alle „für“ als „for“ erscheinen.

Da nicht immer eine Übersetzung mit dieser Methode erzielt wird, besteht das Problem darin, zu erkennen welcher Zeichensatz gerade vorliegt. Das ist bei den im Internet häufig benutzten Zeichensätzen nicht, bzw. nur unvollständig möglich.

Zwei Probleme werfen sich nun auf: Ambiguität und Normalisierung. Zum einen würde ein deutscher Benutzer alle Dokumente die „for“ enthalten auf eine Suchanfrage nach „für“ bekommen, zum anderen kann man nicht mehr einfach die Suchanfragen nach „FÜR“ auf die kleingeschriebene Variante „für“ konvertieren.

Die Ambiguität läßt sich durch den Sprachenerkenner auflösen. Eine Normalisierung kann nach eingehendem Studium der im Internet üblichen Standardencodings (Anhang C) ohne Kenntnis des verwandten Encodings erzielt werden.

Wenn man sich die ISO-8859 und KOI8-R Tabellen näher ansieht, so kann man eine Kleinschreibung dadurch erzielen, daß man bei der ISO-Latin1 Tabelle die dritte Zeile auf die fünfte und die vierte auf die letzte Zeile konvertiert.

Bei ISO-Latin2 ist es die erste auf die zweite, die dritte auf die fünfte und die vierte Zeile auf die letzte Zeile. Bei den anderen ISO-Tabellen verhält es sich ähnlich, selbst die russische KOI8-R Tabelle ist genauso zu konvertieren, ausgenommen daß dabei die Klein- in Großschrift konvertiert wird.

Einzuwenden bleibt, daß die so erhaltene Groß-Klein-Konvertierung nicht immer richtig ist. So wird z.B. das deutsche „ß“ in ISO-Latin1 auf den vermeintlichen Kleinbuchstaben „ÿ“ konvertiert wird. Dies spielt allerdings keine Rolle, da durch die Konvertierung keine Ambiguitäten in den jeweiligen Sprachen auftreten. Dies wäre der Fall würde man „ä“ auf „a“ konvertieren, so daß „älter“ zu „alter“ konvertiert.

Die so erhaltene „Kleinschreibung“ ist zwar nicht immer richtig, bietet jedoch einen gangbaren Ausweg aus der Problematik an.

Als beste Alternative bietet sich Unicode[4], als universelle Zeichenkodierung, an. Um Dokumente nach Unicode konvertieren zu können, muß man das Encoding vorher kennen. Dies ist im WorldWideWeb meist nicht der Fall, weshalb man erst langwierig das Encoding von Dokumenten zu erkennen versuchen müßte.

[4]Unicode ist ein 16bit Zeichensatz, d.h. man kann alle europäischen und asiatischen Schriftzeichen mit ein- und demselben Zeichensatz darstellen. http://www.unicode.org/

4.5 HTML-Parsing

Nach diesem kleinen Abstecher in die Encoding- und *lowercase*-Problematik, kann ich noch gleich eine weitere grundlegende Schwierigkeit bei der Benutzung des WorldWideWeb , das Parsing von HTML, näher erläutern.

Man kann Wörterbücher durch das Scannen von Hyptertextdokumenten erzeugen. *So* einfach ist dies jedoch nicht, da Hypertextdokumente nicht nur aus hintereinandergereihten Wörtern bestehen.

Grundsätzlich sind die Hyptertextdokumente im WorldWideWeb in HTML[5] abgefaßt. HTML besteht im wesentlichen aus sog. *Tags*, die der Beschreibung der Seite dienen.

Will man z.B. „Fonts können *ohne* <u>Probleme</u> gesetzt werden.“ schreiben, so sieht das entsprechende HTML-Dokument wie folgt aus:

```
<b>F</b>onts k&ouml;nnen <em>ohne</em>
<a href=‘‘http://www.problem.org/‘‘>
<font size=‘‘+3‘‘>P</font>robleme</a> gesetzt werden.
```

Will man aus diesem Dokument die eigentliche Information “Fonts können ohne Probleme gesetzt werden.“ herausfiltern, bedarf es eines Filters, welcher die reinen Textinformationen aus HTML-Dokumenten herausfiltern kann. In diesen Filter muß man einiges Wissen um den HTML-Standard einfließen lassen.

Beginnend mit einem Parser der alle *Tags* herausfiltert mußte ich feststellen, daß die resultierenden Wortlisten sehr „unsauber“ sind. Ich befaßte mich mit der Problematik daraufhin eingehender.

Sehr viele Dokumente im WorldWideWeb enthalten Tabellen, Adressen, Listen und Graphiken. Mit solchen Dokumenten kann man schlecht Wortlisten generieren, da die gefundenen Wortfrequenzen nicht mit den in geschriebenem Text vorkommenden Frequenzen übereinstimmen. Einfachstes Beispiel ist das Wort „Telefon“, welches mit einem Mal ein sehr wichtiges deutsches Wort wurde, da es auf fast jeder *personal Homepage* vorkommt.

Am einfachsten schien es mir die HTML-Dokumente in Wörter, Sätze und Paragraphen zu zerlegen und die Wortfrequenzen nur aus aus den gefundenen Paragraphen zu erzeugen. Bei der Generierung eines solchen Parsers hat man mit zwei grundsätzlichen Problemen zu kämpfen:

Austesten: Der Parser funktioniert für die ersten zehn Megabyte Text aus Europa tadellos und stößt dann auf Dokumente die schlecht geparsed werden. Das Testen auf eine volle Funktionsfähigkeit wird dadurch erheblich behindert.

[5] Der HTMT-Standard ist momentan in der Version 3.2 (offiziell) und auf http://www.w3.org/pub/WWW/MarkUp/ abzurufen

Unbekanntes: Ein Parser geht von einer bestimmten Struktur der Dokumente aus. Wenn die Dokumente anders strukturiert sind, muß der Parser angepaßt werden. Der oben genannte Satz „Fonts können *ohne* Probleme gesetzt werden.“ ist ein gutes Beispiel dafür:

Der Parser arbeitete zuerst nach dem Prinzip, daß z.B. <em> und <b> -Tags wegfallen, unbekannte und trennende Tags als Satztrennung dienen. So wurde aus dem genannten Satz das Folgende:

```
Fonts können ohne <HTMLTAG>
P <HTMLTAG>
robleme gesetzt werden <SENTENCE>
```

Der <FONT> Tag war im alten Standard noch nicht enthalten, weshalb das „P“ mit einem Mal ein eigener Abschnitt wurde.

An diesem Beispiel wird ersichtlich, daß man den Parser ständig an die neuesten Standards und die tatsächlich vorhandenen Gegebenheiten im WorldWideWeb anpassen muß.

4.6 Einfache Relativitätenerkennung

Mit den in 4.3 generierten Wörterbüchern kann man schon eine erste Spracheneerkennung implementieren. Man benutzt dazu die relativen Häufigkeiten der Wörter in den verschiedenen Ländern, so daß man eine Prozentuale Abschätzung machen kann, in welcher Sprache ein Satz geschrieben ist.

Mathematisch ausgedrückt:

$$\begin{aligned} WortWahrscheinlichkeit &= \frac{Worthäufigkeit\ im\ Land}{\sum Worthäufigkeit\ in\ Ländern} \\ SatzWahrscheinlichkeit &= \frac{WortWahrscheinlichkeit}{\sum WortWahrscheinlichkeiten} \end{aligned}$$

An einem Beispiel wird die Simplizität dieser Berechnung ersichtlich:

Als Grundlage nehmen wir den deutschen Satz: „Dies ist ein deutscher Satz.“. Für die Sprachenerkennung nehmen wir das, aus 4.3 erzeugte, deutsche und französische Wörterbuch. So ergibt sich für das Wort „ist“ die folgende Berechnung:

$$\begin{aligned} \text{Worthäufigkeit im Land(de)} &= 5.05510^{-3} \\ \text{Worthäufigkeit im Land(fr)} &= 5.58410^{-7} \end{aligned}$$

$$\Rightarrow \qquad \text{WortWahrscheinlichkeit(de)} =$$

$$
\begin{aligned}
&= \frac{\text{Worthäufigkeit im Land(de)}}{\sum \text{Worthäufigkeit im Land(de)+Worthäufigkeit im Land(fr)}} \\
&= \frac{5.05510^{-3}}{5.05510^{-3}+5.58410^{-7}} \\
&= 99.98\%
\end{aligned}
$$

Das bedeutet, daß das Wort „ist" zu 99.98% als Deutsch und zu 0.02% als Französisch erkannt wurde. Iteriert man diese Methode über alle Wörter des Satzes und über alle Wörterbücher, so erhält man die in Tabelle 4.2 dargestellte Erkennungstabelle.

de	87.28%
no	2.57%
is	1.64%
sk	1.04%
nl	1.04%
dk	0.95%
hu	0.74%
es	0.64%
fi	0.62%
pl	0.60%

Tabelle 4.2: Methode: Relative Häufigkeiten; Wörterbücher: 20 Länder, ungesäubert

Man sieht also schon an dieser einfachen relativen Häufigkeitsberechnung, daß die Erkennung von Sprachen mit dieser Methode funktioniert. Die Erkennungsqualität zu verbessern ist das Thema des nächsten Abschnitts.

4.7 Verbesserte Wörterbücher

Da die Qualität der Erkennung direkt mit der Qualität der zugrundeliegenden Wörterbücher korreliert, muß man versuchen die in 4.3 generierten Wörterbücher zu säubern.

Da die erzeugten Wörterbücher sehr viele englische Wörter enthalten muß man als erstes versuchen diese herauszufiltern. Im wesentlichen bieten sich hier nach kurzer Überlegung drei Strategien an:

1. Wörter, die in englischsprachigen Ländern vorkommen, herausfiltern.

   Die Wörter, die in englischsprachigen Ländern vorkommen herauszufiltern, sieht augenscheinlich als beste Methode aus. Überlegen wir uns zuersteinmal das Vorgehen: Wir wollen z.B. das deutsche Wörterbuch von allen, in England vorkommenden Wörtern säubern.

Da sicherlich Deutsche in England in deutsch geschriebene Hypertextdokumente schreiben, sollten wir nur die häufigsten, in England gefundenen Wörter herausfiltern. Allerdings gehen uns so auch seltene, englische Wörter verloren, die eventuell auch auf deutschen Webseiten zu finden sind.

Wenn wir dann die übriggebliebenen hochfrequenten Wörter aus der deutschen Wortliste herausnehmen gehen uns mulitlinguale Wörter verloren, z.B. „England", „Hypertext", ... Man kann den Verlust dieser Wörter verschmerzen, da diese Wörter nichts zur Spracherkennung beitragen.

Wörter wie z.B. „die" und „these" die in beiden Sprachen hochfrequent sind, und sehr wohl etwas zur Spracherkennung beitragen, darf man aber auf keinen Fall herausfiltern. Nun kann man „händisch" die Liste durchgehen und solche Wörter als „ok" markieren.

2. Wörter, die in mehr als $\frac{n}{m}; m = Anzahl\ der\ Wörterbücher$ vorkommen, herausfiltern.

   Hier behält man zum einen nicht sehr häufige Wörter in den Wortlisten, zum anderen filtert man dabei wieder Wörter wie „die" und "these" heraus, weshalb man diese Strategie gleich verwerfen kann.

3. Neue Wörterbücher generieren. Allerdings nur Sätze für die Generierung benutzen, die durch den gerade vorgestellten Sprachenerkenner als eindeutig zu einer Sprache zugehörig erkannt wurden.

   Diese Methode liefert wesentlich weniger Worte für die manuelle Nachkorrektur wie die erste Methode, weshalb ich mich hierfür entschied. Man nimmt also Sätze und läßt diese vom rudimentären Erkenner in eine Sprache einteilen:

```
hk 24.79% pl 12.07%
if you have any suggestions or you need more information on
any subject just send me an e-mail

fr 76.60% de  5.46%
liste des mots cl&eacute;s appartenant au
vocabulaire de cette discipline

nl 82.08% cz  5.27%
lees je liever een korte inleiding in het nederlands
```

   Wählt man nun die Parameter des Filters z.B. 70% minimale Zugehörigkeit, zweiterkannte Sprache $\leq 10\%$ entsprechend, so bekommt man Wörterbücher wie in Tabelle 4.3 zu sehen.

der	.0410957468
und	.0378103926
die	.0371554176
in	.0166678019
von	.0127589115

Tabelle 4.3: Deutsches Wörterbuch, erste Säuberung

Man sieht hierbei, daß die englischen Wörter aus Tabelle 4.1 herausgefallen sind und die Worthäufigkeiten sich quantitativ verbessert haben. Das liegt daran, daß nun die Gesamtzahl der vorkommenden Wörter nur noch aus deutschen Sätzen stammt. Der Nenner des Bruchs von Worthäufigkeit zu Gesamtwortzahl ist nun kleiner geworden, der Wert des Bruchs ist somit höher.

Mit den so erhaltenen Wortlisten wird der schon bekannte Satz: „Dies ist ein deutscher Satz." wie in Tabelle 4.4 dargestellt bewertet.

de	96.81%
no	1.15%
is	0.75%
nl	0.39%
pl	0.25%
se	0.19%
sk	0.13%
cz	0.11%
fr	0.09%
es	0.06%

Tabelle 4.4: Methode: Relative Häufigkeiten, Wörterbücher 20 - erste Säuberung

Man sieht am verbesserten Ergebnis, daß die Qualität der zugrundeliegenden Wörterbücher entscheidenden Einfluß auf die Güte der Sprachenerkennung hat.

Aus diesem Grunde wurden dann die so gesäuberten Wörterbücher von Muttersprachlern auf Fehler durchgesehen, um dadurch die Qualität noch weiter zu steigern. Das Ergebnis dieser „perfekten" Wörterbücher, angewandt auf den bekannten Satz, kann man in Tabelle 4.5 sehen.

de	98.29%
no	0.21%

Tabelle 4.5: Methode: Relative Häufigkeiten, Wörterbücher 20 - perfekt sauber

Man sieht, daß hier nur noch Deutsch und Norwegisch auftreten. Alle anderen Wortlisten enthalten keines der Wörter in dem Satz. Die norwegische Liste enthält das Wort „ein“, da dies auch norwegisch sein kann.

Abschließend zu Thema Wörterbücher bleibt zu bemerken, daß die von Muttersprachlern geprüften Wörterbücher kaum verbessert werden können.

Während der künftigen Spracherkennung könnte man allerdings automatisch alle unbekannten Worte herausschreiben. Wenn man z.B. jedes neue Wort, welches nur in einem Land auftaucht, herausfiltert, könnte man auf diese Weise die Wörterbücher ständig erweitern.

4.8 Zipf

Mit den qualitativ hochwertigen Wörterbüchern lassen sich unterschiedliche Sprachen sehr gut erkennen. Allerdings bricht die Erkennungsquote sehr schnell ein, wenn man „verwandte“ Sprachen, wie z.B. Dänisch und Norwegisch zu erkennen versucht.

Bei dänisch und norwegisch liegen die Fehlerraten bei 10% mit der oben benannten Methode und perfekten Wörterbüchern.

Überdies haben Suchmaschinen nicht immer genügend Rechnerkapazitäten zur Verfügung eine HTML-Seite „gut“ zu parsen, weshalb es notwendig wurde eine Art Sprachenverstärker einzubauen, der wie sich später herausstellte beide Problemzonen verkleinert.

Wenn man sich die Frequenzlisten der Wörterbücher ansieht und miteinander vergleicht ergibt sich das in Abbildung 4.1 dargestellte Bild.

Diese Kurven werden nach dem Entdecker Zipfkurven[Zipf] benannt und sind in jeder Sprache vorzufinden. Ich möchte hier nur auf den in der Abbildung „Zipfpunkt“ makierten Punkt eingehen.

Da die englische Sprache von Leuten überall auf dem Globus im WorldWideWeb benutzt wird, hat diese als einzigste Sprache ein unregelmäßige Kurve.

Für meisten Authoren im WorldWideWeb ist Englisch nicht die Muttersprache. Deshalb benutzen diese Authoren die bekannten englischen Wörter überproportional häufig (in der Graphik die ersten 1200 Wörter). Die nicht so häufig benutzten Wörter werden von diesen Authoren unterproportional wenig benutzt, weshalb die englische Kurve nach den ersten 1200 Wörtern unter die der anderen Sprachen fällt.

Abbildung 4.1: Vergleich der Frequenzlisten zwischen Englisch und drei europäischen Sprachen.

Man kann nun über diese Zipf-Wörter, welche mehr als 0.06% (abgelesen aus der Graphik) in einer Sprache verwendet werden, bestimmte Aussagen machen.

- In „normalen" Texten kommen zwischen 20% und 30% Zipfwörter vor.
- Zipfwörter sind zum Großteil kontextunabhängig aber sprachabhängig.

Wenn nun die Hypertextdokumente nicht gut geparsed sind (z.B. Tabellen und Listen), so enthalten diese keine Zipfverteilung und es kommen weniger als 20% Zipf-wörter darin vor. Dies stellt eine sehr einfache Methode dar festzustellen ob das zu erkennende Dokument überhaupt in einem sinnvollen Format vorliegt.

Zipfwörter sind in Sprachen wie Norwegisch und Dänisch sehr unterschiedlich, weshalb eine überproportionale Bewertung dieser Wörter zu besseren Ergebnissen führt. So konnte die die Erkennung von Dänisch und Norwegisch mit Hilfe von Zipf von 90% auf 95% verbessert werden.

4.9 N-gramme

Bei näherer Betrachtung der mit den geannten Methoden erzielten Ergebnisse kann man erkennen, daß Wörter, welche nicht in den Wörterbüchern enthalten sind, nichts zur Spracherkennung betragen. Als Mensch kann man aber sehr wohl „hören“ ob jemand spanisch spricht, auch wenn man kein spanisches Wort kennt.

Dieses “hören“ von Sprachen läßt sich natürlich so nicht im Rechner nachbilden weshalb man andere Methoden finden muß dies zu implementieren.

Hier bieten sich sog. N-gramme als Lösung an. N-gramme sind die ersten N Buchstaben eines Wortes sowie die N letzten. Zum Beispiel sind beim Wort die Sequenzen „bi“ und „mm“ die Bi-gramme[6] des Wortes „Bigramm“.

Bei vielen Wörtern einer Sprache werden die Anfangs- und Endsilben an die jeweilig zugrundeliegende Grammatik angepaßt. So gibt es im Deutschen viele Wörter die auf „nen“ und „ung“ enden bzw. Wörter die mit „Be“ und „Ver“ anfangen.

Wenn man nun die Häufigkeiten der N-gramme bestimmt, die man in den Wörterbüchern findet, so kann man wieder über einfache Relativitäten die Währscheinlichkeiten ausrechnen mit denen ein Wort zu einer Sprache gehört.

Über diese N-gramme kann man dann Wörter einer Sprache zuordnen, auch wenn man das Wort selbst nicht im Wörterbuch auffindet. Mit dieser Methode erhält man die in Tabelle 4.6 abgebildeten Erkennungsquoten mit dem bekannten Satz - ohne direkt Wörter aus Wörterbüchern zu übernehmen.

de	20.14%	de	45.02%	de	26.93%
lv	10.75%	en	7.89%	lv	7.80%
nl	6.78%	fr	1.99%	nl	7.48%
en	6.68%	lv	1.46%	ee	6.84%
ee	6.01%	pt	1.04%	fr	5.39%
fr	5.63%	nl	0.46%	en	5.11%
fi	5.26%	lt	0.21%	fi	5.01%
lt	4.81%	se	0.16%	dk	4.60%

Tabelle 4.6: Von links nach rechts: Bigramme, Trigramme und Quadgramme

Wie man sieht, bieten die Trigramme das beste Ergebnis im Vergleich zu den Bi- und Quadgrammen. Wenn man die beschriebene Methode der relativen Häufigkeiten mit Zipf und Trigrammen verbindet, kann man die Ergebnisse der Zipfversion noch verbessern.

Als Beispiel hierfür kann ich die Erkennung von Französisch in Abbildung 4.2 heranziehen. Hier sieht man deutlich, daß zum einen die sehr hochfrequente Erkennung von französischen Sätzen nicht mehr stattfindet, da Trigramme von französischen Wörtern auch in anderen Sprachen vorkommen.

Diese augenscheinliche Verschlechterung der Erkennungsrate wird aber durch die Unzahl von Sätzen ausgeglichen, die dadurch besser erkannt werden. Die

[6] Man nennt die häufig benutzten N-gramme nach den griechischen Buchstaben: Bigramm (2), Trigramm (3) und Quadgramm (4).

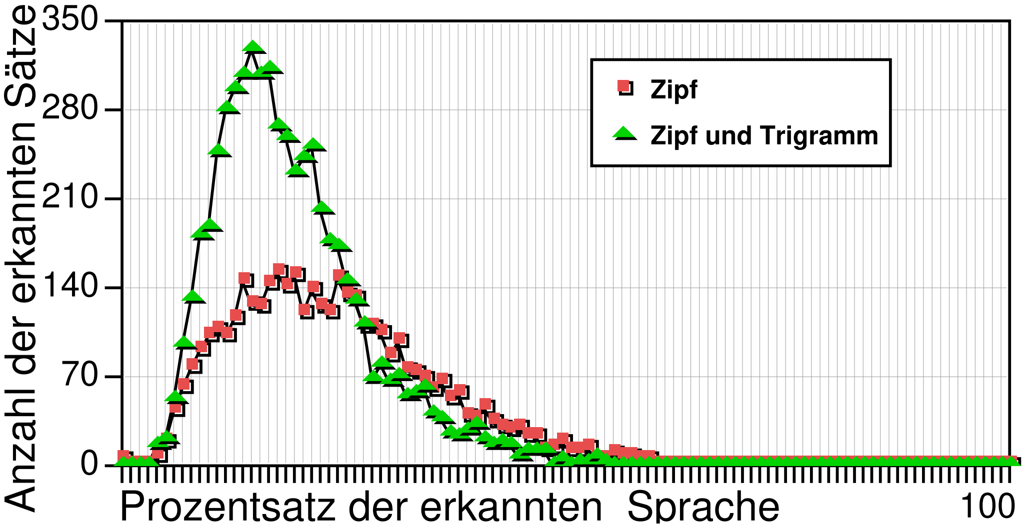

Abbildung 4.2: Vergleich zwischen der Erkennungsquote Zipf und Zipf mit Trigrammen. Grundlage 4000 französische Sätze.

Anzahl der als französisch erkannten Sätze ist die Fläche unter der Kurve, welche mit der Zipf-Trigramm-Erkennung größer ist.

4.10 Ausblick

Mit den bisherigen Methoden erhält man eine Erkennungsrate für deutsche Sätze von 99.88%. Die fehlerhaften Sätze sind allerdings auch nicht *sehr* Deutsch, wie man in Tabelle 4.7 sieht.

en	2.71	de	0.08	ihr antikolonialistisches rezept lautet somit don't eat curry don't worry and don't say sorry
en	0.65	de	0.10	suhrkamp verlag frankfurt am main taschenbuchausgabe kazimierz brandys warschauer tagebuch

Tabelle 4.7: Fehler bei 4000 deutschen Sätzen. Methode: Relative Häufigkeiten,Zipf und Trigramm. Die Zahlen geben Absolutwerte an.

Da bei einzelnen Sätzen eine solch hohe Erkennungsquote erzielt werden konnte, ist es offensichtlich, daß bei Dokumenten mit mehreren Sätzen die Erkennungsquote wesentlich höher ist.

Die hohe Erkennungsquote war auch ein Grund dafür weshalb während der Implementierungsphase der Suchdienst *AltaVista* sich sehr an den Algorithmen zur Sprachenerkennung interessiert zeigte.

Bei *Alta Vista* wird zur Zeit die Spracherkennung auf Basis dieser Arbeit in den Suchdienst integriert und Ende August 1997 der breiten Öffentlichkeit zugänglich gemacht. Eine Vorabversion kann der interessierte Leser über das WorldWideWeb auf *http://ie4.altavista.digital.com:7000/* ausprobieren.

Kapitel 5

Kontexterkenner

In diesem Kapitel beschäftige ich mit der Frage der Erkennung von Zusammenhängen im WorldWideWeb und wie man diese erkennen kann.

Die hier vorgestellten Algorithmen wurden von mir nur in Einzelversuchen auf ihre Gültigkeit überprüft. Da diese Algorithmen sehr speicherintensiv (sowohl Hauptspeicher als auch Plattenspeicher) sind, war es mir nicht möglich eine allgemein benutzbare Implementierung zu schaffen.

5.1 Überlegungen

Wie schon in der Einleitung erwähnt, bietet die große Informationsmenge die das WorldWideWeb darstellt neue Möglichkeiten der Informationsverabeitung. Kontexterkennung bezieht sich hierbei auf das weltweit verteilte Umfeld von Informationen.

Vor einigen Jahren publizierten Wissenschaftler neueste Erkenntnisse ihrer Forschung in Fachzeitschriften. Seit Einführung des WorldWideWeb verlagert sich die Publikation neuester Forschungsberichte zunehmend auf das Internet.

Da all diese Daten in aller Welt verstreut liegen und Suchserver momentan nur nach Stichworten suchen lassen, ist die Suche nach Hypertextdokumenten, die zu einem Themengebiet gehören, recht mühsam.

Abhilfe könnten hier jedoch die nun folgenden Strategien bieten.

5.1.1 Domainerkennung und „hot topics“

Mit dem im vorigen Kapitel vorgestellten Sprachenerkenner kann man eine Domainerkennung implementieren.

Domain meint hier einen abgegrenzten Wortschatz. So kann man medizinische Texte von mathematischen Texten durch den verwendeten Wortschatz unterscheiden. Hier kann man das Themengebiete mit Wörterbüchern beschreiben und Texte werden dann nicht mehr einer Sprache zugehörig angesehen, sondern einem Themengebiet.

Allerdings erleichtert dies einem die Suche nicht, sondern ermöglicht es neue Seiten, die zu einem Themengebiet hinzugefügt werden, zu erkennen. In einem solchen Szenario könnte man sich vorstellen, daß man so „brandaktuelle“ Themen innerhalb eines Themenkomplexes erkennt.

Für ein noch nicht erwähntes „Goodie“ der Spracherkennung kann man die Frequenzlisten aus 4.3 benutzen. Sehr wichtige Wörter sind hochfrequent, da häufig benutzt.

Wenn nun Worte in der Frequenzliste „wandern“, d.h. in ihrer Wichtigkeit varieren, kann man erkennen, welche Worte akutelle Schlagworte sind. Man kann aber auch zukünftig Interessannte Themen schon vorher erkennen, da diese Worte in Fachkreisen schon vorher häufiger benutzt werden.

Benutzt man nun einen *Domainerkenner* und verfolgt dort das „Wandern“ von Worten, so kann man neue Entwicklungen in den verschiedenen Themengebieten von Anfang an mitverfolgen.

5.1.2 Clustering Algorithmen

Diese Strategie wird momentan vom *AltaVista-LiveTopics* Suchdienst benutzt. Man sucht nach einem Wort und es wird in einigen tausend Dokumenten gefunden. Diese Dokumente werden dann durch Clustering Algorithmen in zusammenhängende Gruppen unterteilt.

Dies funktioniert einfach ausgedrückt so:

Aus allen Dokumenten die das Suchwort enthalten werden jeweils alle Worte herausgefiltert. Dann werden die Worte, welche auf mehren Seiten zusammen auftauchen, in Gruppen zusammengefaßt. Die Übergänge zwischen den Gruppen werden dann als verschiedene Bereiche des Suchbegriffes aufgefaßt.

Da man alle Hypertextdokumente mit allen anderen Hypertextdokumenten des Ergebnisraums vergleichen muß (z.B. 172.000 Dokumente beim Wort „composer“), benötigt man bei Durchführung dieser Methode sehr viel Rechen- und Speicherkapazität.

5.2 Normalisierung

Normalisieren (lemmatisieren[1]) von Hypertextdokumenten bedeutet, daß man die Worte in den Dokumenten in die jeweiligen Grundformen zerlegt. Erst diese Grundformen benutzt man dann zum indexieren.

An einem Beispiel wird dies ersichtlicher: Wenn man den das Wort “Komponisten“ indexiert, dann kann man dieses Dokument nur über Eingabe des Wortes „Komponisten“, nicht jedoch über Eingabe von „Komponist“ erreichen.

[1]Lemmatisieren wird das Zerlegen von Worten in die einzelnen Komponenten genannt.

Normalisiert man das Wort „Komponisten“ zu „Komponist“ und normalisiert darüber hinaus noch die Eingaben der Suchanfragen, so bekommt man bei Suchanfragen auch Seiten, die das Suchwort nicht direkt enthalten.

Allerdings setzt dies auch voraus, daß man zum einen einen funktionierenden Sprachenerkenner hat und zum anderen einen Normalisierer für die entsprechende Sprache. Am *CIS* wurde ein solcher Normalisierer für die deutsche Sprache schon implementiert, so daß ich darauf schon zurrückgreifen konnte.

Diese Normalisierung bietet noch einen Vorteil bei der Bearbeitung von Hypertextdokumenten an: Die Indexe werden wesentlich kleiner, da weniger verschiedene Worte zu indexieren sind.

5.2.1 Linkauswertung

Weniger Rechenleistung benötigt man, wenn man sich die Intelligenz der Leute im Netz zunutze machen kann. So kennen die Authoren von Hypertextdokumenten schon einen kleinen Teil des WorldWideWeb und referenzieren diesen. Die Linkauswertung verlangt insofern wesentlich geringere Rechenkapazitäten, als daß die Links in der Suchmaschine sowieso vorgehalten werden müssen und nur die Bezeichner von Verweisen zusätzlich gespeichert werden müssen.

Man findet auf Seiten zu „Mozart“ Referenzen auf andere „Mozart“-Seiten, Seiten zu anderen klassischen und modernen Komponisten und viele in das Themengebiet zu Mozart gehörende Referenzen.

Wenn man sich also dieses Wissen der Authoren zunutze macht, kann man ein Themengebiet wesentlich umfangreicher beschreiben als dies mit einfachen Textsuchmaschinen möglich ist.

Es bieten sich hier verschiedene Möglichkeiten der Linkauswertung an:

direkt: Viele Links werden benannt nach dem Thema, auf welches sie verweisen, z.B. „Hier finden sie etwas über Mozart“.

Wenn ich nun nach „Mozart“ suche, benötige ich nur alle Referenzen in denen das Wort „Mozart“ auftaucht. Zum besseren Verständnis sei auf Abbildung 5.1 verwiesen.

Wenn viele Dokumente auf ein Dokument verweisen, kann man einfach feststellen, ob die Seite, auf die verwiesen wird (in der Abbildung das Dokument D), eine wichtige Seite ist.

indirekt: Alle Hypertextdokumente, die auf ein Hypertextdokument verweisen, haben zum Großteil auch etwas mit dessen Inhalt zu tun. So wird man auf einer Seite über Mozart selten einen Verweis auf *Gummistiefel* finden. Ein Bild zu diesem Konzept liefert Abbildung 5.2.

Schwieriger wird es schon, die Informationen auszuwerten, die einem die Seiten liefern, auf die bei diesem Verfahren gezeigt wird. Eine Auswertung

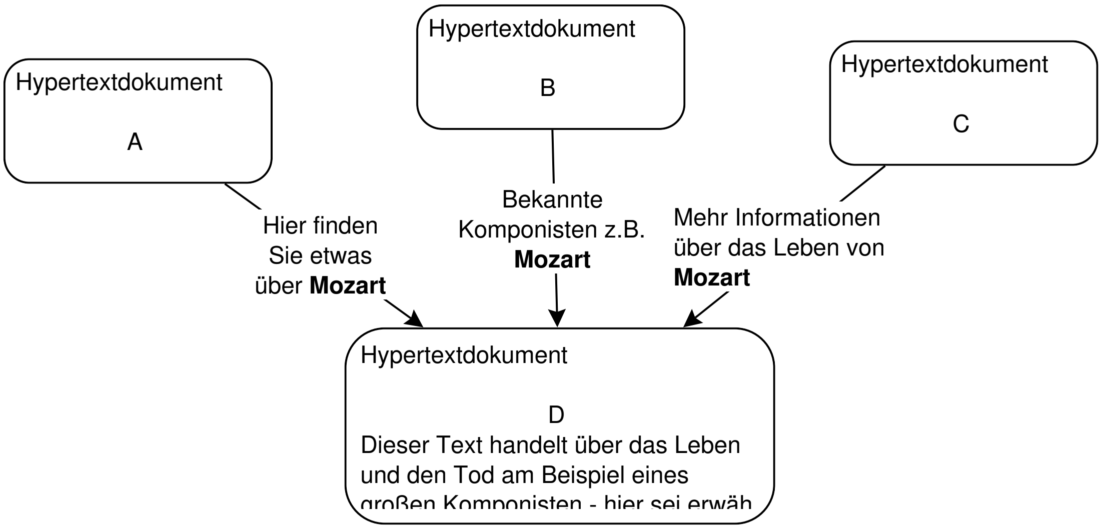

Abbildung 5.1: Direkte Auswertung der Namen von Verweisen.

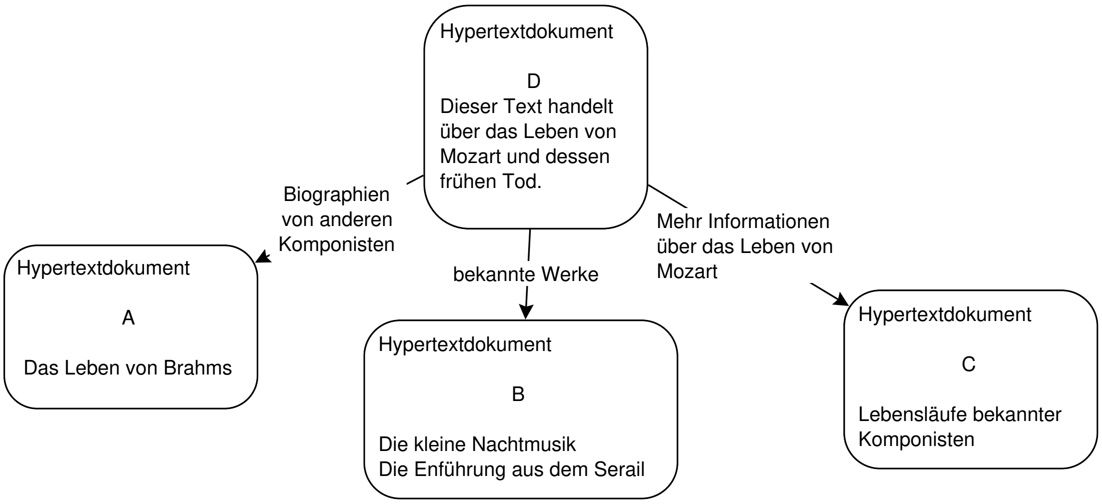

Abbildung 5.2: Indirekte Auswertung von Verweisen.

von Wörtern die auf allen so verwiesenen Seiten zu finden waren lieferte aber schon ansprechende Ergebnisse.

5.3 Link-Cluster

Die Benutzung der Links als Informationsträger und das Normalisieren der Dokumente bringt schon eine deutlich verbesserte Qualität der Suchantworten.

Die verbesserte Qualität ist darauf zuürckzuführen, daß Dokumente, die nicht mit dem Suchbegriff schon von anderen Dokumenten referenziert wurden, nicht auftauchen.

Um eine weitere Qualität den Suchantworten hinzuzufügen, entschied ich mich die normalisierten Links der Zieldokumente bei Suchantworten mit auszugeben. An einem Beispiel wird dies einfacher ersichtlich:

DokumentA enthält:	„Mozart“ auf DokumentB
	„Mozart“ auf DokumentC
DokumentB enthält:	„Klassische Komponisten“auf DokumentA
	„Beethoven“ auf DokumentC
DokumentC enthält:	„Mozart“auf DokumentB
	„Klassik“ auf DokumentA

Bei einer Suchanfrage nach „Mozart“ erhielt man daraufhin:

Mozart:	DokumentB
	DokumentC
Komponisten:	DokumentA
Beethoven:	DokumentC
Klassik	DokumentA

Die erreichte Qualität ist damit schon eine ganz andere, man sieht nun auch „nahe Verwandte“ des Themenkomplexes.

5.4 Frequenzlisten

Bei einem weiteren Ausbaus des „Link-Cluster“-Ansatzes benutzte ich die in den gefundenen Dokumenten vorkommenden Wörter als Gesamtwörterbuch. Aus diesem Wörterbuch generierte ich dann eine Frequenzliste.

Hintergedanke dabei war, daß Themengebiete Schlagwörter haben, die häufiger vorkommen als alle anderen Wörter. Die Generierung einer Frequenzliste ist mit wesentlich weniger Aufwand verbunden, wie die Clusterung der Daten.

Bei einer Anfrage nach „Bundeskanzler“ tauchten die folgenden Wörter dann hochfrequent auf: Saarland, Lafontaine, Bundestag, Kohl,. . . .

Interessant ist diese Ergebnis insofern, als daß der Name Lafontaine häufiger auftauchte als der von Kohl. Dies kann aber auch durch die sehr geringe

Datenmenge (200 Megabyte Text von deutschen Web-Sites) , die dieser Analyse zugrunde lag, hervorgerufen worden sein.

5.5 Ausblick

Erste Versuche mit den Verweisauswertungen und dessen Verbesserungen zeigten aufregende Ergebnisse. So wurde bei der Anfrage nach „Bundesliga“, ganz wider Erwarten, neben den „normalen“ Ergebnisseiten auch auf die Wirtschaftsuniversität von Frankfurt am Main verwiesen, da diese den größten Fußballergebnisserver betreibt.

Anhand solcher Ergebnisse wird klar, daß eine Linkauswertung mehr der Denkstruktur von Menschen ähnelt. Viele Menschen werden, zum Thema Bundesliga befragt,die Vereine und deren Spieler nennen, nur wenige werden von Datenbanken berichten.

Es läßt sich klar eine neue Qualität der Suchantworten erkennen, weshalb in einem weiteren Projekt am *CIS* die Verarbeitung der Linkinformationen umfangreicher untersucht wird.

Anhang A

Dokumentation zum PerlCrawler

A.1 Aufruf

Das Program wird durch Eingabe von

wcacheFiller.perl [option]... url=http://...

aufgerufen. Die Position der Argumente ist frei wählbar. Die folgenden Aufrufe sind beispielsweise gültig:

cacheFiller.perl url=http://www.next.com/
cacheFiller.perl database=remove url=http://www.next.com/
cacheFiller.perl url=http://www.next.com/ database=remove follow=text:!pictures

Wenn man eine falsche oder ungültige Argumentenliste übergibt, wird eine Erklärung der gesamten Optionen aufgelistet:

```
NAME
    cacheFiller.perl - get all html pages beginning at a specific url.

SYNOPSIS
    cacheFiller.perl [ option ]... url=http...

OPTIONS :
    defaults are marked with a *

    database=[remove|reuse*]
                The links to follow are stored in a database. This database
                consists of three files:
                    'links.visited'     - Contains links that the program
```

```
                                  already has followed.
              'links.nextstage'   - Links that are new in this round are
                                  stored here. Those links will be
                                  fetched in the next round.
              'links.unknown'     - Links that get fetched in this round.

htmlpages=[none*|save]
            If set, all pages fetched are saved to disk to the directory
            htmlpages.

languages=[none|examine*]
            If set, the program will exmaine every html document depending
            on one of the following algorithms.

    sentences=[none|examine*]
            If set, sentences of fetched url's are appended to the files
            country/name. Name is the topmost internet domain of the fetched
            url. It then appends : URL:urlname
                                   SEN:word1:word2:...
                                   SEN:...                     to the file.

    wordspersentence=<int>
            Only sentences with greater and equal number of words will be
            appended to the sentencesfiles. Default is 5.

    minword=<int>
            Minimal length a word has to have. Default is 2.
    maxword=<int>
            Maximal length a word has to have. Default is  (''== endless).
    languagemax=<int>
            Maximal size in megabytes of the language files. Default is 40.

debug=[database]:[sentences]:[contents]:[linkarrays]

follow/links=[option]:[option]:...
    with option: text       follow only .html and .htm documents
                 !pictures  never get picutres
                 [!]us      [never] get things from .edu .mil .com .net ...
                 [!]de      [never] get things from .de .leo.org ...
                 regex      enter regular expression like .uk/ to get
                            just links from the uk

url={urlname}
            Begin the searchtree with the url named. Default is
            http://www.w3.org/pub/DataSources/WWW/Servers.html.
```

```
    maxdepth=<int>
                Searchtree ends at depth. Default is 30.
    timeout=<int>
                Maximum time in seconds to wait for an url to resolve.
                Default is 5.
    retrytime=<int>
                Minimum time in seconds to wait before an url might be retried.
                Default is 1000.


EXAMPLES

    perl5 wwwCacheFiller.pl url=http://www.next.com/ database=remove
    perl5 wwwCacheFiller.perl links=\!us:\!de:text

FILES
    In case the environment variable "http_proxy" is set, the named proxy will
    be used.

BUGS
    perl5.002 or higher is mandatory.
```

A.2 Beispiel

Will man zum Beispiel sehr viele deutsche Seiten mit diesem Programm holen, dann kann man als Startseite eine Auflistung vieler Web-Sites auswählen und das Programm holt dann iterativ die Seiten, die dort benannt sind und mit den regulären Ausdrücken übereinstimmen.

Der Programmaufruf würde dann wie folgt aussehen:

```
cacheFiller.perl url=http://www.leo.org/demap/cities/Muenchen.html follow=de
```

Das Programm gibt dann die Daten zurück, mit denen es nun das *Crawlen* beginnt (z.B. die regulären Ausdrucke mit denen es arbeitet) und holt dann auch gleich die Seiten vom WorldWideWeb :

```
Never Follow  :
@links=grep(!/\/cgi-bin\//i,@links);
@links=grep(!/\.map$/i,@links);
@links=grep(!/^mailto:/i,@links);
@links=grep(!/^gopher:/i,@links);
@links=grep(!/^ftp:/,@links);

Only Follow   :
```

```
@linksOK=();
push(@linksOK,grep(/\.de[:\/]/,@links));
push(@linksOK,grep(/\.leo\.org[:\/]/,@links));
push(@linksOK,grep(/\.ch[:\/]/,@links));
push(@linksOK,grep(/\.at[:\/]/,@links));
@links=@linksOK;
undef @linksOK;
```

```
Database      :remove
HTML Pages    :none
Languages     :examine
Sentences     :examine
Words/sentence:5
min. Word     :2
max. Word     :
max. Langfile :40 MByte
Begin         :http://www.leo.org/demap/cities/Muenchen.html
Maxdepth      :30
Timeout       :5
Retrytime     :1000
Debugging     :
Removing link database:done.
Entering Stage
 0                  http://www.leo.org/demap/cities/Muenchen.html OK
Database swap:done.
Entering Stage
 1                                        http://www.leo.org/demap/ OK
 1 http://www.leo.org/demap/cities/http//www.muenchner-aidshilfe.de/ Not Found
 1                                              http://www.leo.org/ OK
 1                                       http://www.leo.org/muenchen/ OK
 1                   http://www.leo.org/images/icons/LEO/leoc.gif^C
Got signal - saving
```

⋮

Zuerst werden also die ganzen Statusmeldungen ausgegeben, dann wird die erste Stufe der Breitensuche im WorldWideWeb durchgefürht. Es ist als Startpunkt *http://www.leo.org/demap/cities/Muenchen.html* angegeben, so daß zuerst diese Seite geholt wird.

Nachdem nun alle Seiten dieser Stufe der Suche geholt worden sind beginnt die zweite Stufe. Alle in der vorhergehenden Stufe gefundenen Links wird nun nachgegangen. Die geholten Files werden, nach Internetdomänen geordnet. abgespeichert.

Das Programm speichert seinen momentanen Stand ständig ab, so daß bei Absturz des Rechners nicht alle Dokumente nocheinmal geholt werden müssen, sondern dort weitergemacht werden kann, wo aufgehört wurde.

Anhang B

Dokumentation zu Hoover

B.1 Starten der Fetcher

Wie schon ausführlich erörtert, laufen die Fetcher, die die eigentliche Arbeit des *Hoover*-Systems erledigen, auf vielen Rechnern. Auf jedem, dem System zuarbeitenden Rechner, muß also ein Fetcher laufen, der sich dann beim *Hoover* anmeldet. Der eigentliche Programmaufruf sieht wie folgt aus:

Fetcher [-threads <number>] [-host <hostname>]

Die Parameter sind optional. In der Voreinstellung wird mit fünfzig Threads gefetched und der lokale Rechner als *Hoover* angenommen.

B.2 Starten von Hoover

Hoover ist, wie schon in Kapitel 3 besprochen, mit einem Konfigurationsfile zu starten. Der eigentliche Programmaufruf sieht dann wie folgt aus:

Hoover [ConfigurationFileName]

Der Parameter ist optional. In der Voreinstellung wird mit „HooverConfiguration“ als Konfigurationsfile gearbeitet. Hoover gibt die erhaltenen HTML-Dokumente auf die Standardausgabe aus, so daß der Programmaufruf in einer Unix-Shell meist wie folgt aussieht:

Hoover >fetchedWebDokuments 2>hooverProgressInformation

Im Klartext: Alle geholten HTML-Dokumente werden in die Datei „fetchedWebDokuments“ geschrieben, die Informationen, was *Hoover* gerade macht, in die Datei „hooverProgressInformation“.

B.2.1 Das Hoover Konfigurationsfile

Das Konfigurationsfile von Hoover entspricht dem "PropertyList"-Format von OpenStep. Dieses Format erlaubt es Daten aus Textfiles einzulesen und in Objekte abzubilden.

Ein *NSDictionary*-Objekt erzeugt man durch Eingabe von „SchlüsselName = WertName", ein *NSString*-Objekt wird von Gänsefüschen umramt, ein *NSArray* erzeugt man, indem man die Werte mit Komma trennt und in runde Klammern setzt. Kommentare werden zwischen „/*" und „*/" geschrieben.

Soviel zur Theorie, in der Praxis sieht ein HooverKonfigurationsfile z.B. so aus:

```
{   /*
        Dies ist eine TestKonfiguration, um zu zeigen wie einfach es
        ist ein poropertyList Format von Hand einzugeben
    */
    urls =  (   "http://www.leo.org/demap/cities/Muenchen.html",
                "http://watzmann.cis.uni-muenchen.de/Deutschland.html"
            );
    general = {
            useragentname = "Hoover/0.1";
            useragentmail = "Patrick Stein <jolly@cis.uni-muenchen.de>";
            databasename =  "webdatabase" ;
            includehosts = ( ".de",".at","ch" );
            excludehosts = ();
            includepaths = ();
            excludepaths = ( "?", "cgi-bin", "cgi_bin" );
            httpproxy = { host="www.cis.uni-muenchen.de"; port = 8000 };
        };
}
```

Die Datei enthält zum gegenwärtigen Zeitpunkt ein Wörterbuch mit den folgenden zwei Einträgen:

urls: Hier kann man die Startseiten, mit denen *Hoover* beginnen soll, als *URL*-Array eingeben.

general: Hier werden Schlüssel-Wert Paare aufgelistet, die die verschiedenen Optionen im *Hoover* beeinflussen:

- **useragentname:** Mit diesem Namen meldet sich *Hoover* bei jeder Web-Site. Diese können dann den Zugriff über das *robots exclusion protocol* mit diesem Namen verweigern oder eingrenzen.
- **useragentmail:** Mite dieser E-Mail Adresse meldet sich *Hoover* bei jeder Web-Site. So können die Administratoren bei Problemen mit *Hoover* dies gleich der verantwortlichen Person mitteilen.

databasename: Unter diesem Dateinamen wird die persistente Datenspeicherung der *URLs* durchgeführt.

includehosts: Es werden Hosts nur von *Hoover* kontaktiert, wenn mindestens eine Zeichenkette in diesem Array auf den Hostnamen paßt (matched).

excludehosts: Hosts, deren Hostname auf mindestens eine Zeichenkette in diesem Array passt, werden niemals kontaktiert.

includepaths: Siehe includehosts, nur das hier auf den Pfadnamen gematched wird.

excludepaths: Siehe excludehosts, nur das hier auf den Pfadnamen gematched wird.

httpproxy: Existiert dieser Eintrag, dann wird das Internet von den Fetchern nur über den benannten Host (HTTP-Proxy) kontaktiert. Bei Nichtexistenz wird das Internet direkt von den Fetchern benutzt.

Es hat sich während der Implementierung der Vorteil dieser *variablen* Konfiguration gezeigt. Der Eintrag „httpproxy“ war nicht von Anfang an in der Konfiguration enthalten.

Der Eintrag wurde notwendig, als das *Hoover*-System über einen Firewall-Rechner auf das Internet zugreifen sollte. Ich ergänzte daraufhin die Spezifikation und Implementation und konnte nun das System über einen Firewall auf das Internet zugreifen lassen.

Bahnbrechend ist dies sicher nicht, jedoch ist es ohne Probleme möglich die neuen, erweiterten Konfigurationsfiles auf den *alten Hoovern* laufen zu lassen, da diese Systeme gar nicht nach einem „httpproxy“-Eintrag im “general“-Wörterbuch suchen.

Anhang C

Encoding Tabellen

	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Abbildung C.1: ISO Latin 1 oder auch ISO 8859-1

	Ą	˘	Ł	¤	Ľ	Ś	§	¨	Š	Ş	Ť	Ź	-	Ž	Ż
°	ą	˛	ł	´	ľ	ś	ˇ	¸	š	ş	ť	ź	˝	ž	ż
Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
Đ	Ń	Ň	Ó	Ô	Ő	Ö	×	Ř	Ů	Ú	Ű	Ü	Ý	Ţ	ß
ŕ	á	â	ă	ä	ĺ	ć	ç	č	é	ę	ë	ě	í	î	ď
đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	ţ	˙

Abbildung C.2: ISO Latin 2 oder auch ISO 8859-2

	Ħ	˘	£	¤		Ĥ	§	¨	İ	Ş	Ğ	Ĵ	-		Ż
°	ħ	²	³	´	µ	ĥ	·	¸	ı	ş	ğ	ĵ	½		ż
À	Á	Â		Ä	Ċ	Ĉ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
	Ñ	Ò	Ó	Ô	Ġ	Ö	×	Ĝ	Ù	Ú	Û	Ü	Ŭ	Ŝ	ß
à	á	â		ä	ċ	ĉ	ç	è	é	ê	ë	ì	í	î	ï
	ñ	ò	ó	ô	ġ	ö	÷	ĝ	ù	ú	û	ü	ŭ	ŝ	˙

Abbildung C.3: ISO Latin 3 oder auch ISO 8859-3

	Ą	ĸ	Ŗ	¤	Ĩ	Ļ	§	¨	Š	Ē	Ģ	Ŧ	-	Ž	¯
°	ą	˛	ŗ	´	ĩ	ļ	ˇ	¸	š	ē	ģ	ŧ	Ŋ	ž	ŋ
Ā	Á	Â	Ã	Ä	Å	Æ	Į	Č	É	Ę	Ë	Ė	Í	Î	Ī
Đ	Ņ	Ō	Ķ	Ô	Õ	Ö	×	Ø	Ų	Ú	Û	Ü	Ũ	Ū	ß
ā	á	â	ã	ä	å	æ	į	č	é	ę	ë	ė	í	î	ī
đ	ņ	ō	ķ	ô	õ	ö	÷	ø	ų	ú	û	ü	ũ	ū	˙

Abbildung C.4: ISO Latin 4 oder auch ISO 8859-4

	Ё	Ђ	Ѓ	Є	Ѕ	І	Ї	Ј	Љ	Њ	Ћ	Ќ	-	Ў	Џ
А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
№	ё	ђ	ѓ	є	ѕ	і	ї	ј	љ	њ	ћ	ќ	§	ў	џ

Abbildung C.5: ISO Latin 5 oder auch ISO 8859-5

─	│	┌	┐	└	┘	├	┤	┬	┴	┼	▀	▄	█	▌	▐
░	▒	▓	⌠	■	∙	√	≈	≤	≥		⌡	°	²	·	÷
═	║	╒	ё	╓	╔	╕	╖	╗	╘	╙	╚	╛	╜	╝	╞
╟	╠	╡	Ё	╢	╣	╤	╥	╦	╧	╨	╩	╪	╫	╬	©
ю	а	б	ц	д	е	ф	г	х	и	й	к	л	м	н	о
п	я	р	с	т	у	ж	в	ь	ы	з	ш	э	щ	ч	ъ
Ю	А	Б	Ц	Д	Е	Ф	Г	Х	И	Й	К	Л	М	Н	О
П	Я	Р	С	Т	У	Ж	В	Ь	Ы	З	Ш	Э	Щ	Ч	Ъ

Abbildung C.6: KOI8-R

Literaturverzeichnis

[AhoUllman] A.V. Aho,J.D. Ullman : *Foundations of Computer Science*, Computer Science Press, New York (1992).

[Comer] D.E. Comer: *Internetworking with TCP/IP Vol.I, Principles, Protocols, and Architecture*, Prentice-Hall Inc., Englewood Cliffs, New Jersey (1991).

[Denning] P.J. Denning: *„Thrashing: It's causes and prevention"* Proceedings of the AFIPS National Computer Conference, Seiten 915-922 (1968).

[Mullender] S.J. Mullender: *Distributed Systems* ACM Press New York, New York (1993)

[Zipf] G.K. Zipf: *Human Behavior and the Principle of Least-Effort* Addison-Wesley,Cambridge, MA (1949)

Glossar

Fehlertolerante Programme: Programme, die nach Systemabstürzen oder sonstigen Problemen dort weitermachen können, wo sie aufgehört haben.

In Verbindung mit einem verteilten System meint dies meist die Toleranz des Gesamtsystems gegenüber einzelnen Rechnerabstürzen.

Parsen: *Parsen* ist das Herausfiltern von (syntaktischen) Informationen aus einem Datenstrom.

Beim Parsen von HTML-Dokumenten werden alle Hypertext-Informationen herausgefiltert. Nachdem ein HTML-Dokument geparsed wurde, stehen alle Sätze des Dokuments in einer Datenstruktur zur Verfügung.

Ein Beispiel erhellt den Vorgang: Die Einstiegsseite der Fachhochschule, wie sie in Abbildung C.7 auf Seite 64 zu sehen ist, wird zu den folgenden Sätzen geparsed:

```
URL: http://www.informatik.fh-muenchen.de/welcome.html
SEN: Dies ist der Hypermedia-Informationsserver des Fachbereiches
07 an der Fachhochschule München
SEN: Das Informationssystem ist ein Teil des internationalen World
Wide Web
```

Die Überschrift des HTML-Dokuments wurde nicht als Satz erkannt, da diese das gestelltes Kriterium: ein Satz muß mindestens acht Wörter enthalten, nicht erfüllt.

Proxy-Server: Proxy-Server oder auch Web-Caches speichern alle von Benutzern des Internet geholten Hypertextdokumente. Damit werden Seiten, die von mehrmals angefragt werden, nur aus dem Speicher des „nahen“ Proxies geholt und müssen nicht erst langwierig vom Internet geladen werden.

Reguläre Ausdrücke: Mit regulären Ausdrücken kann man in algebraischer Form, Suchmuster definieren.[Foundations]

Einfach ausgedrückt kann man mit regülaren Ausdrücken Suchmuster sehr leicht definieren. So benutzen viele Unix-Werkzeuge reguläre Ausdrücke um

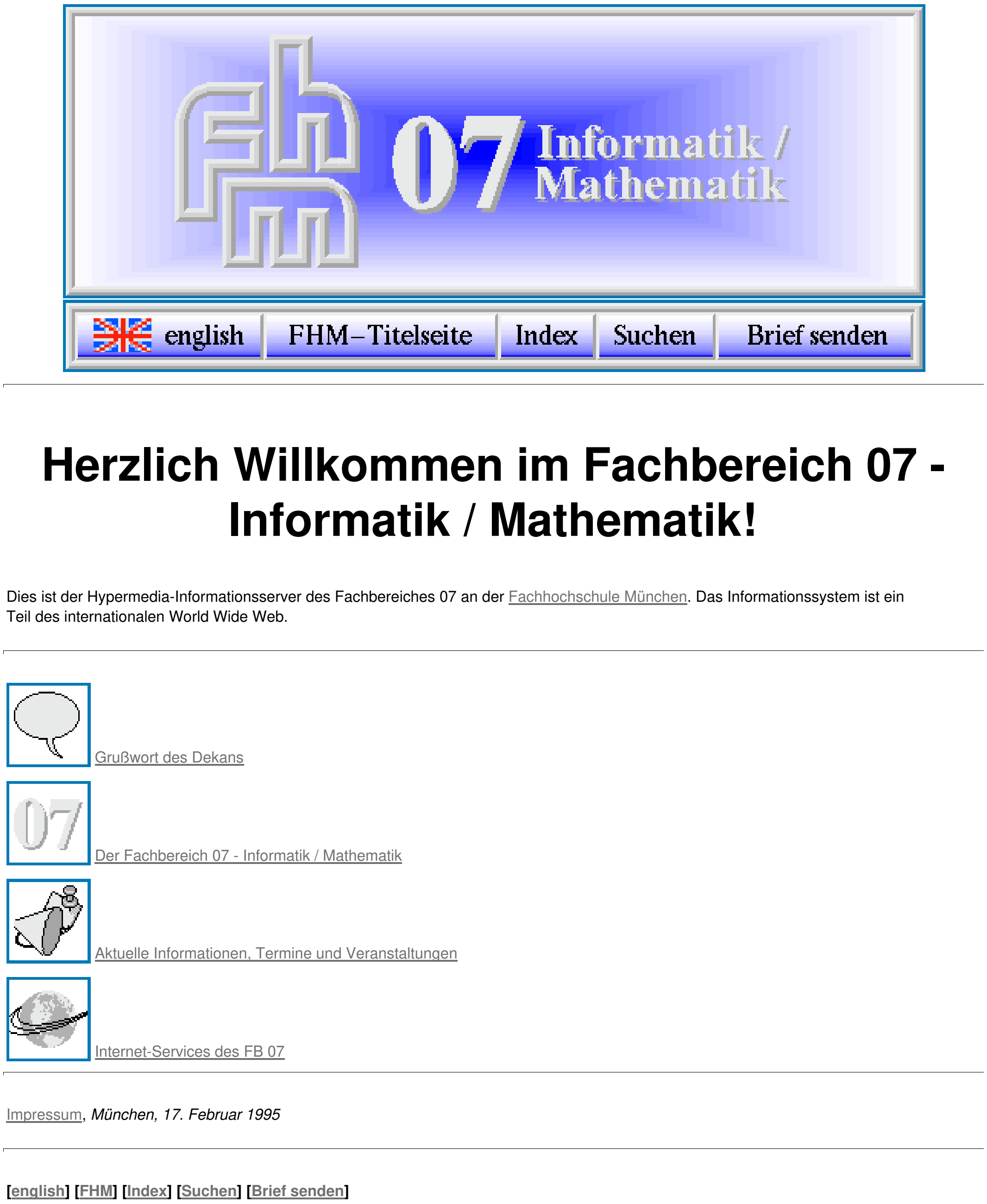

Abbildung C.7: Die Webseite der Fachhochschule München in einem Webbrowser dargestellt.

Dateinamen allgemein zu bennenen. In der Programmiersprache Perl bekommt man reguläre Ausdrücke gleich im Sprachumfang mitgeliefert, weshalb sich diese Sprache auch gut für Textparsing eignet.

Will man z.B. alle auseinandergezogenen Bindestrichwörter zu Mehrworten ohne Bindestrich konvertieren, dann kann man dies durch den Ausdruck:
`s/\s-/ /g;`
erledigen. Im Klartext: Ersetze (s) jedes Zeichenpaar, welches aus einem Leerzeichen oder Tabulator besteht (\s) und auf den ein Bindestrich folgt (-) durch ein Leerzeichen ( ). Führe diese Ersetztung für jedes Zeichen in der Zeichenkette aus (g).

Thrashen: *engl.* dreschen, *hier:* um sich schlagen.

So wird das Systemverhalten von Betriebssystemen genannt, welches auftritt, wenn Prozesse so groß werden, daß andere Prozesse nicht nur ausgelagert (paging), sondern überdies noch selber aufgrund von Seitenverdrängnungsalgorithmen auf die Platte ausgelagert werden. Das System arbeitet dann sozusagen die Programmbefehle nicht mehr aus dem Hauptspeicher heraus ab, sondern muß erst jeden Befehl von der Festplatte laden. Diese, aufgrund der Mechanik der Festplatte bis zu eine Million mal langsamere Abarbeitung der Befehle macht einen Rechner dann unbenutzbar.[Denning]

Verteilte Programme: Programme die auf mehreren Rechnern gleichzeitig Rechenkapazität nutzen und dies auch koordinieren.

Web-Site: Web-Sites sind Rechner im Internet, die Hypertextdokumente zum weltweiten Abruf zur Verfügung stellen.

Erklärung

Student:	Patrick Stein
Geburtsdatum:	20.09.1969
Studiengruppe:	IF8T
Matrikelnummer:	81 42 12

Hiermit wird erklärt, daß ich die Diplomarbeit selbständig verfaßt, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine andern als die angegebenen Hilfsmittel und Quellen benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

München, 2. Juli 1997 ______________________________

Unterschrift des Verfassers