Universität Trier – Fachbereich IV – Informatik

Untersuchung der Graphstruktur von Web-Communities am Beispiel der Informatik

Diplomarbeit

Betreuung:

Prof. Dr. Bernd Walter
Dipl.-Inform. Gerd Hoff

eingereicht von:

Christoph Schmitz
Fasanenweg 13
54329 Konz

Matr.-Nr. 441570

Trier, den 23. Oktober 2001

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle wörtlichen und sinngemäßen Übernahmen aus anderen Quellen sind als solche kenntlich gemacht worden. Die Arbeit wurde bisher keinem Prüfungsamt in gleicher oder vergleichbarer Form zur Erlangung eines akademischen Grades vorgelegt.

Trier, den 23. Oktober 2001

Christoph Schmitz

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API *Application Programming Interface*

Schnittstelle für eine Programmbibliothek

ASCII *American Standard Code for Information Interchange*

7-Bit-Standardzeichensatz für englischsprachige Texte

BFS *Breadth-First Search*

Breitensuche in einem Graphen [Meh84]

BIOS *Basic Input/Output System*

Basissoftware zur Ansteuerung von Hardwarekomponenten in einem PC

CGI *Common Gateway Interface*

Standard zur Anbindung von Programmen, die dynamisch Inhalte erzeugen, an Webserver

CLOB *Character Large Object*

Datentyp in einem RDBMS, der die Speicherung großer Textstücke als Feld in einer Tabelle erlaubt

CRUD *Create-Read-Update-Delete*

minimaler Satz von Operationen für Objekte, die sich in einem RDBMS persistent machen können [Yod98] (siehe auch Abschnitt 47)

DBMS *Database Management System*

allgemeiner Begriff für Software zur Verwaltung von Datenbanken

DFS *Depth-First Search*

Tiefensuche in einem Graphen [Meh84]

DKS *Dense-k-Subgraph Problem*

Maximierungsproblem: finde den dichtesten Subgraphen eines Graphen G mit k Knoten [Fei97]

DNS *Domain Name Service*

Namensdienst zur Auflösung von symbolischen Namen zu numerischen Adressen im Internet

ER *Präfix: Entity-Relationship-...*

Das ER-Datenmodell unterscheidet Datenobjekte (Entities) und Beziehungen zwischen Datenobjekten (Relationships).

FIFO *First-in-first-out*

Datenstruktur, die Elemente in der Reihenfolge der Einfügung wieder zurückgibt (auch *Queue* oder *Schlange* genannt)

GMT *Greenwich Mean Time*

Basis-Zeitzone, auf die sich viele globale Zeitangaben beziehen

GUI *Graphical User Interface*

graphische Benutzeroberfläche

HITS *Hyperlink-Induced Topic Search*

Algorithmus zur Bestimmung von wichtigen Seiten und Überblicksseiten zu einem Thema im WWW ([Kle99], Abschnitt 3.6)

HTML *Hypertext Markup Language*

Auszeichungssprache für Dokumente im WWW [Wor01]

HTTP *Hypertext Transfer Protocol*

Protokoll zur Übermittlung von Dokumenten im WWW [Fie97]

JDBC *Java Database Connectivity*

Standard-API in Java zur Benutzung von relationalen Datenbanken

JDK *Java Development Kit*

Entwicklungsumgebung für Java, bestehend aus JVM, Standardbibliotheken, Debugger, usw.

JVM *Java Virtual Machine*

abstrakte Maschine zur Ausführung von Java-Bytecode

LEDA *Library of Efficient Data Types and Algorithms*

Programmbibliothek für Standard-Datenstrukturen wie Mengen, Graphen, Schlangen usw. [Meh99]

MIME *Multipurpose Internet Mail Extensions*

Standard zum Einbinden von Multimedia-Inhalten in EMails [Bor93]

NP Menge der Probleme, die von einer nichtdeterministischen Turingmaschine in Polynomialzeit entschieden werden können

OID *Object Identifier*

systemweit eindeutiger Schlüssel für ein Datenobjekt

RDBMS *Relational Database Management Systems*

DBMS für Datenbanken im relationalen Datenmodell

SQL *Structured Query Language*

Abfragesprache für relationale Datenbanken

SQLJ *Embedded SQL for Java*

Standard zur Einbettung von SQL-Code in Java-Programme.

UML *Unified Modeling Language*

Modellierungssprache für objektorientierte Systeme

URL *Uniform Resource Locator*

Adresse eines Dokuments im WWW

W3C *World Wide Web Consortium*

Organisation für Standards im WWW-Umfeld

WWW *World Wide Web*

Gesamtheit der über HTTP im Internet abrufbaren Dokumente

XML *Extensible Markup Language*

Standard zur Definition von Auszeichnungssprachen für Dokumente

1 Einleitung

1.1 Einführung

Das WWW[1] als Hypertext-System setzt sich zusammen aus Dokumenten (Seiten), die durch Verweise aufeinander Bezug nehmen können. Durch diese Struktur liegt es nahe, das WWW als einen gerichteten Graphen zu betrachten: die Dokumente sind die Knoten dieses Graphen, und die Verweise bilden die Kanten zwischen den Knoten.

Eine solche Sichtweise auf Hypertext-Systeme gab es schon vor der Verbreitung des WWW. So schreibt z. B. Schnupp in seinem Buch aus dem Jahr 1992 [Sch92]:

> [Diese Terminologie] hat den Vorteil, daß sie die unmittelbare Abstraktion des Informationsnetzes als (mathematischen) *Graphen* und damit auch den Einsatz der *Graphentheorie* [. . .] zur Formalisierung und Diskussion auftretender Sachverhalte und Probleme nahelegt. [Sch92, S. 34]

Dementsprechend gibt es nach dem großen Erfolg des WWW eine Reihe von Arbeiten, die mit Hilfe der Graphentheorie Eigenschaften des Web untersuchen. Für die vorliegende Arbeit sind vor allem die Veröffentlichungen von Broder u. a. [Bro00], Kleinberg [Kle99] sowie Page und Brin [Pag98] von Bedeutung.

Broder u. a. haben für das WWW als Ganzes – oder zumindest einen großen Teil davon – eine sogenannte „Bowtie“-Struktur festgestellt [Bro00, Abschnitt 3]. Sie untersuchen dazu einen aus etwa 200 Millionen Seiten bestehenden Subgraphen des WWW und stellen eine Reihe charakteristischer Teilmengen vor, in die sich der Webgraph zerlegen lässt.

[1] World Wide Web

Die Arbeiten von Page/Brin und Kleinberg stellen Graphalgorithmen vor, um wichtige oder thematisch zusammengehörige Seiten durch Untersuchung des Graphen zu erkennen. Dabei ergeben sich sog. *Communities* – Mengen von Seiten, die sich mit einem gemeinsamen Thema befassen.

1.2 Ziele der Arbeit

Ziel dieser Arbeit ist es, Strukturen ähnlich den oben genannten auf einem Teil des WWW nachzuweisen, der sich mit Informatik befasst.

Eine entsprechende Teilmenge des WWW wird mit Mitteln der Graphentheorie betrachtet, um interessante Strukturen herauszuarbeiten. Dazu sollen die oben erwähnten Algorithmen nachvollzogen oder eigene entwickelt werden.

Um dies zu ermöglichen, muss ein Webcrawler entwickelt werden, der automatisch Informatik-relevante Seiten aus dem WWW abruft und einer lokalen Verarbeitung zugänglich macht.

1.3 Aufbau der Arbeit

Kapitel 2 beschreibt Grundlagen der Graphentheorie, Eigenschaften des WWW und seine Betrachtung als gerichteten Graphen. Außerdem wird beispielhaft der Aufbau eines Crawlers zum automatischen Traversieren des Web vorgestellt.

In Kapitel 3 werden zwei Strategien beschrieben und verglichen, um ausgehend von einer Menge von Startseiten im Web weitere thematisch verwandte Seiten zu lesen. Es wird eine Reihe von Graphalgorithmen vorgestellt und erprobt, die auf der so gefundenen Seitenmenge interessante Strukturen wie Cliquen und Verdichtungen, bipartite Cluster oder die Bowtie-Struktur finden können.

In den Kapiteln 4 und 5 wird ein datenbankgestützter Webcrawler entworfen und implementiert, der die Suchstragien aus Kapitel 3 umsetzt und so die untersuchten Daten zur Verfügung stellt.

Kapitel 6 beschreibt die praktische Benutzung der erstellten Software.

Kapitel 7 fasst die Ergebnisse zusammen.

1.4 Vorbemerkungen

Deutsch und Englisch

> Ich habe verstanden, daß man contemporary sein muß, das future-Denken haben muß. [...] Und für den Erfolg war mein coordinated concept entscheidend, die Idee, daß man viele Teile einer collection miteinander combinen kann. Aber die audience hat das alles von Anfang an auch supported.
>
> Jil Sander im FAZ-Magazin

Gerade in der Informatik kommt man oft ohne englische Fachbegriffe nicht aus. Es gibt einfach keine präzisen und gängigen deutschen Wörter für „Crawler“, „Client“ oder „Server“ – „Kriecher“, „Klient“ und „Bediener“ sagen jedenfalls nicht das Gewünschte aus.

Dennoch werden in dieser Arbeit soweit möglich deutschsprachige Begriffe verwendet, um nicht zu oft deutsch-englische Sätze wie die oben angedeuteten zu schreiben.

Klassennamen, Bezeichner und Kommentare im Quelltext sind in Englisch gehalten, da sie sich nach Meinung des Verfassers so besser in die Bezeichner, Schlüsselwörter und sonstigen Textbestandteile von Java, Javadoc und der verwendeten Bibliotheken einfügen.

Schreibweise von Zahlen

Um Mehrdeutigkeiten zu vermeiden, werden alle Zahlen mit Dezimalpunkt und ohne Dreiergruppierung geschrieben.

Dadurch wird die Schreibung von mathematischen Sachverhalten (z. B. Intervall [0.5, 0.6] statt [0,5, 0,6]), Programmaufrufen (programm -option 0.9) und Fließtext angeglichen, und Mehrdeutigkeiten ($3.000 = 3000$ oder $3.000 = 3.0 = 3$?) werden vermieden.

2 Grundlagen

Dieses Kapitel beschäftigt sich mit den Grundlagen, die für die Betrachtungen in dieser Arbeit notwendig sind.

In Abschnitt 2.1 werden zunächst einige Begriffe der Graphentheorie definiert, die für die Untersuchung des WWW zur Anwendung kommen. Abschnitt 2.2 zeigt, wie diese graphentheoretischen Konzepte auf das World Wide Web übertragbar sind.

Außerdem wird in Abschnitt 2.3 eine Klasse von Programmen, die sogenannten *Webcrawler*, vorgestellt. Diese dienen zum automatischen Traversieren des WWW, um größere Teile davon lesen und analysieren zu können. Der Aufbau eines solchen Crawlers wird an einem Beispiel aus der Praxis beschrieben.

2.1 Grundbegriffe der Graphentheorie

Dieser Abschnitt führt einige Definitionen und Begriffe aus der Graphentheorie ein, die in dieser Arbeit oft benutzt werden. Weitere Informationen finden sich z. B. in [Bon76].

Ein *gerichteter Graph* $G = (V, E)$ besteht aus einer Menge V von *Knoten* und einer Menge $E \subseteq V \times V$ von *Kanten*. Eine Kante $e = (u, v)$ ist eine *ausgehende* Kante von u bzw. eine *eingehende* Kante von v. u heißt dabei *Vorgänger* von v, v ist ein *Nachfolger* von u.

Um anzuzeigen, dass eine Kante (u, v) in der Kantenmenge E enthalten (nicht enthalten) ist, wird gelegentlich einfach $u \to v$ $(u \nrightarrow v)$ geschrieben, wenn die betrachtete Kantenmenge klar ist.

Der *Ingrad* (*indegree*) eines Knotens v ist die Anzahl der eingehenden Kanten:

$$\textit{indeg}(v) = |\{(u, v) \in E : u \in V\}|$$

Analog ist der *Ausgrad* (*outdegree*) eines Knotens v definiert als die Anzahl der ausgehenden Kanten:

$$\textit{outdeg}(v) = |\{(v, u) \in E : u \in V\}|$$

Der *Grad* eines Knotens ist die Summe aus In- und Ausgrad.

Ein *Pfad* von u nach v ist eine Folge $e_1 = (v_0, v_1), e_2 = (v_1, v_2), \ldots, e_k = (v_{k-1}, v_k)$ von Kanten aus E, so dass $v_0 = u$ und $v_k = v$. k ist die *Länge* dieses Pfades. Die minimale Länge eines Pfades zwischen u und v heißt *Entfernung* von u und v.

Ein *ungerichteter Pfad* von u nach v ist eine Folge $e_1 = (v_0, v_1), e_2 = (v_1, v_2), \ldots, e_k = (v_{k-1}, v_k)$ von Kanten, so dass $v_0 = u$ und $v_k = v$. Dabei können Kanten auch umgekehrt benutzt werden, d. h. $e_i \in E \cup \{(v, u) : (u, v) \in E\}$ für alle i. k ist die *Länge* dieses Pfades.

Eine *starke Zusammenhangskomponente* ist eine maximale Teilmenge $M \subseteq V$ von Knoten, so dass es für je zwei Knoten $u, v \in M$ einen Pfad von u nach v gibt.

Eine *schwache Zusammenhangskomponente* ist eine maximale Teilmenge $M \subseteq V$ von Knoten, so dass es für je zwei Knoten $u, v \in M$ einen ungerichteten Pfad von u nach v gibt.

Graphen werden üblicherweise in der Art von Abbildung 2.1 dargestellt. Der gezeigte Graph besteht aus den Knoten $V = \{1, 2, 3, 4, 5\}$ und den Kanten $E = \{(1, 2), (2, 1), (1, 3), (2, 3), (3, 4), (4, 5), (5, 3)\}$.

2.2 Das World Wide Web als Graph

Das *World Wide Web* (WWW) besteht aus einer Menge von Dokumenten, die unter ihrer jeweiligen Adresse, dem *Uniform Resource Locator* (URL) [1], über das Internet abgerufen werden können; die Übertragung der Dokumente regelt das *Hypertext Transfer Protocol* (HTTP) [Fie97]. Diese sog. Webseiten werden üblicherweise in der *Hypertext Markup*

[1] Obwohl *der* Uniform Resource Locator der grammatisch naheliegende Artikel ist, wird im Folgenden *die* URL verwendet. Beide Varianten werden allgemein benutzt, allerdings ist *die* URL gebräuchlicher.

Language (HTML) [Wor01] geschrieben, die es erlaubt, Auszeichnungen für Überschriften, Tabellen usw. mit Hilfe von Markierungen (*Tags*) festzulegen.

Die Besonderheit dabei ist, dass die Seiten auch Verweise (*Links* bzw. *Hyperlinks*) auf andere Dokumente tragen können, die dem Betrachter kenntlich gemacht werden und es ihm erlauben, von einem Dokument zu einem anderen zu wechseln. Diese Struktur von Seiten und Verweisen legt es nahe, das WWW als gerichteten Graphen zu betrachten.

Abbildung 2.2 stellt einen (fiktiven) Auschnitt des WWW als Graph dar. Die Seiten sind mit ihrem URLs als Knoten dargestellt; die Links bilden die Kanten. Knoten und Seiten sowie Kanten und Links werden dementsprechend in dieser Arbeit synonym betrachtet.

2.3 Automatisches Erfassen von Webseiten mit Crawlern

2.3.1 Allgemeines

Jedem Nutzer des WWW ist es relativ leicht möglich, selber Inhalte zu erzeugen und zu publizieren. Dabei gibt es keine zentralen Instanzen, die für die Verbreitung der Inhalte sorgen, wie es z. B. Verlage bei gedruckten Publikationen sind. Dadurch ist es schwierig, genauere

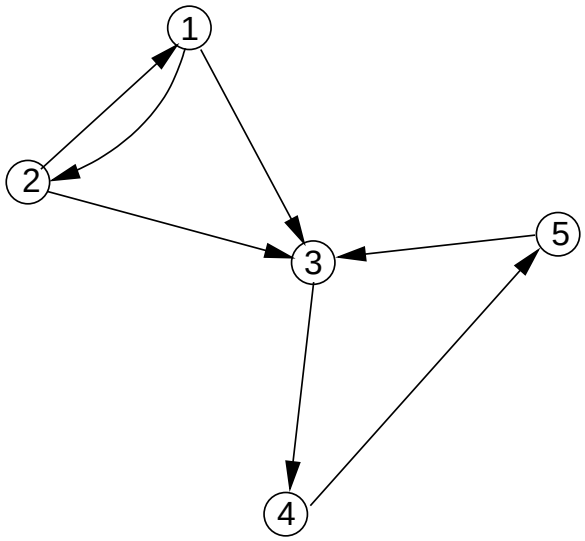


Abbildung 2.1: Beispiel: Darstellung eines Graphen

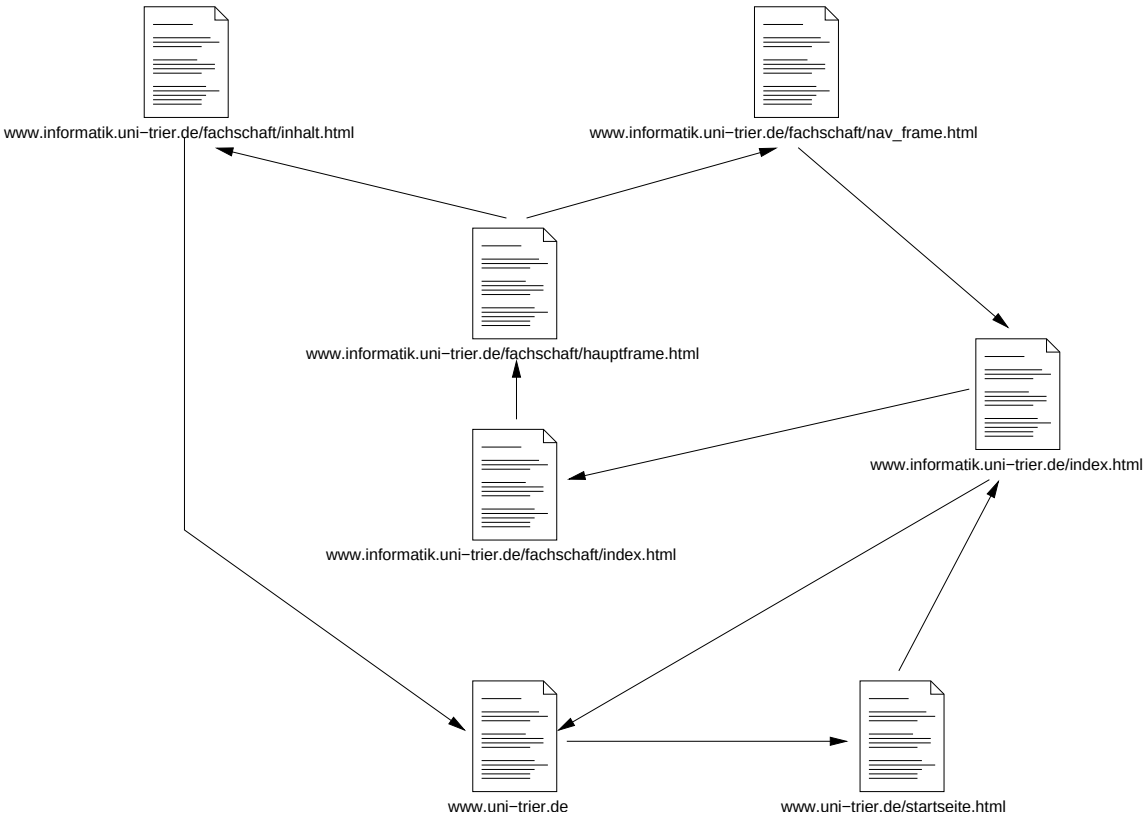


Abbildung 2.2: Das Web als Graph

Angaben über die Anzahl der abrufbaren Seiten zu machen. Die Firma BrightPlanet hat eine Studie [Bri01] veröffentlicht, die nahe legt, dass das Web je nach Betrachtungsweise zwischen einer und 550 Milliarden Seiten umfasst.[2]

Diese Größenangaben verdeutlichen den enormen Umfang an Daten, die aus dem Web abrufbar sind. Um diesen Datenbestand leichter nutzbar zu machen, werden Programme eingesetzt, die automatisch Seiten abrufen und verarbeiten. Auf diese Weise werden z. B. die Daten für die bekannten Suchmaschinen gesammelt.

Solche Programme, die automatisch das Web durchforsten, werden *Robot*, *Spider* oder *Crawler* genannt. Als Beispiel für einen Crawler, der auch für sehr große Datenmengen erfolgreich eingesetzt wurde, wird hier *Mercator* kurz vorgestellt.

2.3.2 Beispiel: Mercator

Im WWW sind etliche Webcrawler dokumentiert und oft auch im Sourcecode zugänglich. Allerdings sind diese oft nicht dazu geeignet, über längere Zeit zu laufen und dabei sehr viele Seiten zu bearbeiten.

Andererseits werden die Crawler, die im großen Maßstab eingesetzt werden, auf Grund kommerzieller Interessen in der Regel nicht dokumentiert.

Eine der wenigen Ausnahmen ist *Mercator*, der in einem Forschungsprojekt bei Compaq erstellt wurde. In [Hey99] ist dieses komplett in Java erstellte System ausführlich beschrieben.

Najork und Wiener haben ein Projekt durchgeführt, für das Mercator 328 Millionen Seiten erfasst hat [Naj01]. Das Programm hat sich also für sehr große Datenmengen bewährt.

Aufbau von Mercator Hier werden kurz die wesentlichen Komponenten von Mercator vorgestellt, um die Struktur eines erprobten Crawlers zu dokumentieren. Abbildung 2.3 auf der nächsten Seite zeigt eine schematische Darstellung. Die Zahlen geben die Reihenfolge des Ablaufs an.

[2] Der große Unterschied zwischen den beiden Zahlen erklärt sich dadurch, dass die zweite Betrachtungsweise auch dynamisch generierte Seiten zählt. Diese werden von außen oft nicht per Hyperlink referenziert und werden nur als Reaktion auf Formulareingaben o. ä. erzeugt.

① *Verwaltung der zu besuchenden Adressen:* Mercator verwaltet grundsätzlich die zu besuchenden Adressen in einer Schlange: am Kopf der Schlange wird die nächste URL entnommen, neue URLs werden am Ende eingefügt.

Da allerdings n Anfragen zur gleichen Zeit bearbeitet werden sollen, jedoch niemals zwei davon gleichzeitig auf einem Host, werden n Worker-Threads (s. auch 43) eingesetzt, von denen jeder eine eigene Schlange hat.

Zu bearbeitende URLs werden mittels Hashing auf die Worker verteilt, so dass URLs auf einem gegebenen Host immer dem gleichen Worker zugeteilt werden. Damit wird sichergestellt, dass pro Host jeweils nur eine Anfrage aktiv ist.

② *Abrufen von Dokumenten:* Zum Abrufen von Dokumenten wird ein eigenes HTTP-Modul implementiert, das effizienter ist als die Java-eigenen Klassen. Außerdem bietet es einen Timeout-Mechanismus für HTTP-Verbindungen, der in der Standardbibliothek fehlt.

Das Robot Exclusion Protocol [Kos94] wird dabei eingehalten. Ein Cache ist für die Informationen über die Robot Exclusion vorgesehen.

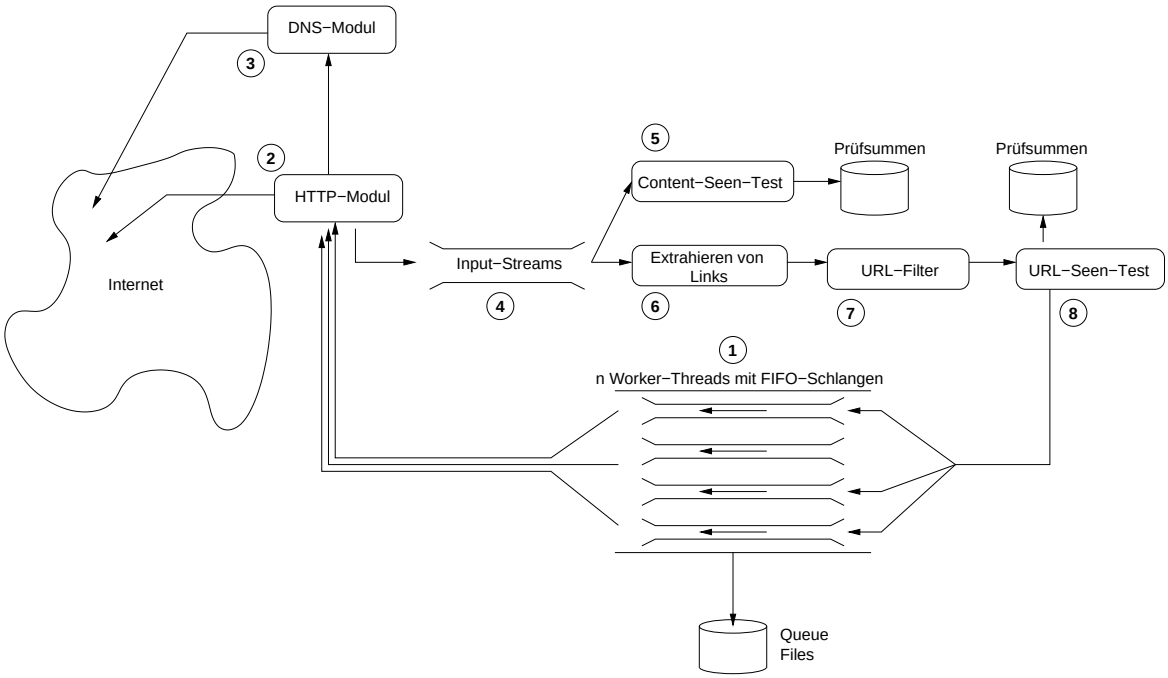


Abbildung 2.3: Aufbau von Mercator

③ *Auflösung von Hostnamen:* Die Autoren von Mercator haben festgestellt, dass die Namensauflösung von Hostnamen nach IP-Adressen beim DNS[3] eine sehr teure Operation ist. Dies führen sie darauf zurück, dass die benutzte Unix-Funktion gethostbyname synchronisiert ist. Dadurch erfolgt stets nur eine Anfrage beim DNS-Server gleichzeitig.

Um dies zu umgehen, implementieren sie ein eigenes DNS-Paket, das mehrere DNS-Anfragen gleichzeitig erlaubt.

④ *Input-Streams:* Da die empfangenen Daten möglicherweise mehreren Modulen zur Verfügung gestellt werden sollen, werden Eingabe-Ströme implementiert, die Daten puffern und mehrfach lesbar machen.

⑤ *Content-Seen-Test:* Viele Seiten im WWW werden unter mehreren URLs gespiegelt. Um solche mehrfach auftretenden Seiten zu erkennen, wird eine Menge von Prüfsummen der Seiteninhalte verwaltet.

⑥ *Extrahieren von Links:* Mercator ist nicht auf HTML-Dokumente beschränkt. Über den Typ des Dokuments (den sogenannten *Content-Type* [Fie97]) wird bestimmt, welches *Processing Module* ein Dokument verarbeitet.

Im Falle von HTML-Dokumenten ist dies ein Modul, das die Links aus dem HTML-Dokument extrahiert und neue abzurufende Seiten in die Schlangen einfügt.

⑦ *URL-Filter:* Um zu steuern, welche URLs bearbeitet werden, können Filter eingesetzt werden. So wäre z. B. ein Filter möglich, der URLs der Form `.../cgi-bin/...` von der Verarbeitung ausschließt.

⑧ *URL-Seen-Test:* Um zu testen, ob eine gegebene URL bereits bearbeitet wurde, dient eine Liste von Prüfsummen der bereits gesehenen URLs, wiederum mit einem Teil davon in einem Cache im Speicher.

Performanceaspekte Schon im ursprünglichen Artikel über den Entwurf von Mercator [Hey99] sind etliche Maßnahmen zur Leistungssteigerung beschrieben. Hauptsächlich sind dies eigens entwickelte Datenstrukturen, die eine effizientere Verwaltung der gesammelten

[3] Domain Name Service

Daten im Haupt- und Sekundärspeicher ermöglichen als es z. B. mit einem Datenbanksystem möglich wäre.

Daneben präsentieren Heydon und Najork in [Hey00] interessante Erfahrungen über die Entwicklung von Mercator, speziell im Hinblick auf Performanceprobleme mit der Java-Standardbibliothek. Diese ermittelten sie mit einer Reihe von Debugging- und Profiling-Werkzeugen.

Grundsätzlich bestätigen sie, dass Java als Programmiersprache und Laufzeitumgebung für ein derart leistungsfähiges System geeignet ist – es wurde ein Durchsatz von 10 Millionen Seiten pro Tag [Naj01, Abschnitt 3] erzielt. Dennoch kritisieren sie etliche Probleme in den Standardbibliotheken des JDK[4].

Dabei bezieht sich die Kritik einerseits auf die sehr defensive Weise, wie Synchronisierung in den Standardklassen eingesetzt wird. Dadurch verursachen einfache Operationen durch geschachtelte Aufrufe oft Dutzende von Sperr-Operationen zur Synchronisierung.

Andererseits besteht das Problem, dass bedingt durch den Entwurf der Standardbibliothek für viele eigentlich einfache Operationen eine große Anzahl von Objekten erzeugt und unmittelbar darauf wieder verworfen wird.

Durch diese beiden Aspekte sehen sich die Entwickler von Mercator gezwungen, etliche Bereiche der Standardbibliothek neu zu implementieren. Darunter befinden sich so zentrale Aufgaben wie Ein-/Ausgabeströme, Behandlung von Zeichenketten (StringBuffer) und Netzwerkklassen (HTTP, DNS-Namensauflösung).

Schlussbemerkung Mercator gibt guten Eindruck davon, welche Komponenten einen höchst leistungsfähigen und sauber strukturierten Crawler ausmachen. Um einen derart hohen Durchsatz wie Mercator zu erzielen, ist jedoch ein einfacher und sauberer Aufbau nicht ausreichend. Es muss dafür ein erheblicher Zusatzaufwand in Form von geeigneten Datenstrukturen und Optimierungsmaßnahmen geleistet werden.

Für die vorliegende Arbeit ist die Betrachtung von Mercator dennoch hilfreich als Beispiel für den Aufbau eines Crawlers.

Im Rahmen dieser Arbeit ist es zwar nicht möglich, ein vergleichbares System zu erstellen, aber die grundlegenden Abläufe sind bei dem hier entwickelten Crawler ähnlich.

[4] Java Development Kit

3 Untersuchung des WWW mit Graphalgorithmen

Dieses Kapitel beschreibt in Abschnitt 3.1 zunächst zwei Strategien, um ausgehend von einer Menge von vorgegebenen Startseiten das WWW zu traversieren und dabei weitere Seiten einzusammeln. Auf diese Weise werden zwei Teilgraphen des Web mit jeweils etwa 200000 Seiten ermittelt.

Auf diesen Graphen werden mit verschiedenen Graphalgorithmen Strukturen erkannt. Dabei werden sowohl globale Strukturen auf dem gesamten Graphen (Abschnitte 3.3, 3.6) als auch interessante kleinere Teilgraphen (3.4, 3.5) bestimmt.

3.1 Verwendete Durchmusterungsstrategien

Wie in Abschnitt 2.3.1 geschildert, dient ein Crawler dazu, automatisch Seiten aus dem WWW zu lesen. Es wird mit einer Startmenge von Seiten begonnen, die abgerufen werden. Eine solche Seite kann wiederum Verweise auf andere Seiten beinhalten. Diese neuen Seiten werden dann ihrerseits besucht, die enthaltenen Links ausgewertet, usw. Auf diese Weise kann der gesamte über Links erreichbare Teil des Web abgearbeitet werden.

In der Sprache der Graphentheorie wird ein solches Vorgehen als *Suche* auf Graphen, als *Durchmustern* oder *Traversieren* eines Graphen bezeichnet. Ziel ist dabei, jeden Knoten eines (gerichteten) Graphen $G = (V, E)$ genau ein Mal zu besuchen.

Die grundlegende Vorgehensweise dabei beschreibt Algorithmus 1 in Anlehnung[1] an Mehl-

[1] In [Meh84] wird markiert, welche *Kanten* schon benutzt wurden, während Algorithmus 1 *Knoten* markiert. Für die Abfolge der besuchten Knoten macht dies aber keinen Unterschied.

horn [Meh84, S. 17]. S ist dabei die Menge der zu bearbeitenden Knoten, I ist eine Menge von Startknoten.

Algorithmus 1: Allgemeine Graph-Suche

1: $S \leftarrow I$
2: **while** $S \neq \emptyset$ **do**
3: entferne ein $u \in S$ aus S
4: markiere u als bearbeitet
5: **for all** $w, (u, w) \in E$ **do**
6: **if** w noch nicht bearbeitet **then**
7: $S \leftarrow S \cup \{w\}$
8: **end if**
9: **end for**
10: **end while**

Durch die Art, wie die Menge S verwaltet und in Schritt 3 der Knoten u gewählt wird, lässt sich steuern, wie der Graph durchlaufen wird.

Die übliche Wahl bei Crawlern ist eine Verwaltung von S als *Schlange* (FIFO[2]); auch bei dem Crawler, der dieser Arbeit zu Grunde liegt, wird im Wesentlichen[3] eine Schlange benutzt. Diese Datenstruktur für S liefert in Schritt 3 immer denjenigen Knoten zurück, der zuerst in S eingefügt wurde. Dadurch werden die Knoten in der Reihenfolge besucht, wie sie in den Schritten 6 und 7 entdeckt wurden. Dies führt zu einer Breitensuche, wie sie in 3.1.1 beschrieben wird.

Der in dieser Arbeit eingesetzte Crawler (s. Kapitel 4) ist so ausgelegt, dass sich die verwendete Strategie zum Auffinden neuer Seiten leicht austauschen lässt. Speziell lässt sich Schritt 3 des Algorithmus 1 verfeinern, um die Suche genauer zu steuern.

Es werden hier zwei verschiedene Strategien eingesetzt:

- eine einfache Breitensuche mit fester maximaler Tiefe (Abschnitt 3.1.1)
- eine Breitensuche, die empfangene Seiten mit einem Textklassifizierer beurteilt und anhand dieser Information neue Seiten priorisiert (Abschnitt 3.1.2)

[2] First-in-first-out
[3] Auf Grund der zeitlichen Restriktionen an den Crawler (s. 4.1.2) kommt eine kompliziertere Datenstruktur (s. 42) zum Einsatz, die aber zumindest aus Sicht jedes Hosts einer Schlange entspricht.

3.1.1 Einfache Breitensuche

Eine Breitensuche (BFS[4]) durchläuft einen Graphen ausgehend von einer Menge von Startknoten in Schichten. Diese Schichten bestehen jeweils aus den Knoten, die in der gleichen *Tiefe* liegen. Tiefe bedeutet dabei die minimale Entfernung zu einem Startknoten (s. Abschnitt 2.1).

Es werden also zuerst die Startknoten besucht, dann die Knoten, die über eine Kante von den Startknoten aus erreichbar sind, dann die Knoten in Entfernung zwei von den Startknoten, usw. Abbildung 3.1 verdeutlicht dies.

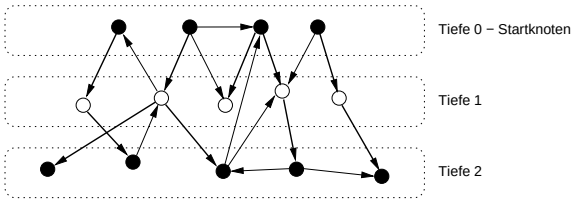


Abbildung 3.1: Beispiel: Einfache Breitensuche

Die Auswahl der Startseiten, die für diesen Lauf verwendet werden, beschreibt Abschnitt 3.1.3.

Von jeder eingesammelten Seite werden alle Hyperlinks extrahiert und die referenzierten Seiten wiederum in die Schlange eingefügt. Die Suche wird auf die Tiefe 2 beschränkt (s. auch Fußnote 8 auf Seite 21), wobei die Tiefe der Startseiten als 0 definiert ist; alle eingesammelten Seiten sind also von den Startseiten über maximal zwei Links zu erreichen. Den in 3.2.1 beschriebenen „Rand“ des Graphen bilden dementsprechend die Knoten der Tiefe 3.

Wie Najork und Wiener [Naj01] feststellen, führt eine Breitensuche trotz ihrer Einfachheit früh zu relevanten Seiten. Dies führen sie darauf zurück, dass wichtige Seiten von vielen anderen Seiten referenziert werden; damit ist es wahrscheinlich, früh Links auf relevante Seiten zu finden.

Damit ist eine einfache BFS-Strategie für die vorliegende Aufgabe geeignet.

[4] Breadth-First Search

3.1.2 Breitensuche mit Textklassifikation

Als Alternative zur einfachen Breitensuche wird eine Variante mit Textklassifikation verwendet. Durch die Klassifikation ist es möglich, Links auf relevanten Seiten bevorzugt zu verfolgen und so die Suche zu fokussieren. Diese Strategie wird im Folgenden entsprechend als *fokussierte Strategie* bezeichnet.

Die Speicherung der zu bearbeitenden URLs erfolgt dabei nicht in einer einfachen FIFO-Schlange, sondern in einer komplexeren Datenstruktur, die für jeden Host eine Prioritätswarteschlange verwaltet. Durch die Vergabe von Prioritäten in dieser sog. HostQueue kann die Durchmusterung des Webgraphen gesteuert werden.

Eine genauere Beschreibung der HostQueue erfolgt in Abschnitt 42.

Textklassifikation mit einem Naïve-Bayes-Klassifizierer

Textklassifikation bedeutet, Dokumente in eine vorgegebene Menge $\{C_1, \ldots, C_k\}$ von Klassen einzuordnen. Für jedes Dokument d wird also eine Klasse $C_i, i \in \{1, \ldots, k\}$ bestimmt, der dieses angehört.

Eine der am weitesten verbreiteten Techniken dazu ist ein *Naïve-Bayes-Klassifizierer* [Lew98]. Dieser soll hier kurz vorgestellt werden.

Das Ziel ist, die bedingten Wahrscheinlichkeiten $P(C_i|t_1, \ldots, t_n)$ zu bestimmen, dass ein Dokument der Klasse C_i angehört, wenn es die Terme $t_1, \ldots, t_n$ enthält. Mit dem Satz von Bayes erhält man:

$$P(C_i|t_1, \ldots, t_n) = \frac{P(t_1, \ldots, t_n|C_i) \cdot P(C_i)}{P(t_1, \ldots, t_n)} \tag{3.1}$$

Eine vereinfachende Annahme ist, dass die vorkommenden Terme stochastisch unabhängig sind. Daher kann man die Wahrscheinlichkeit für das Vorkommen aller Terme als Produkt der Einzelwahrscheinlichkeiten schreiben:

$$P(C_i|t_1, \ldots, t_n) = \frac{P(t_1, \ldots, t_n|C_i) \cdot P(C_i)}{P(t_1, \ldots, t_n)} = \frac{P(C_i) \cdot \prod_j P(t_j|C_i)}{\prod_j P(t_j)} \tag{3.2}$$

Der Klassifizierer wird „trainiert“, indem man Dokumente für die jeweiligen Klassen vorgibt; dann können die entsprechenden relativen Häufigkeiten aus den Trainingsdaten für die

verbliebenen Wahrscheinlichkeiten in Gleichung 3.2 eingesetzt werden.

Naïve-Bayes-Klassifizierer arbeiten am besten, wenn sie Dokumente in eine komplette Taxonomie aus mehreren Klassen einordnen sollen. Bei der Unterscheidung in relevante und nicht-relevante Dokumente gibt es allerdings nur zwei Klassen, und die Klasse „nicht relevant“ bildet keine inhaltlich zusammengehörige Dokumentenmenge.

Um die fokussierte Strategie zu erproben, werden manuell ausgesuchte Startseiten für die relevante Klasse vorgegeben (s. Abschnitt 3.1.3). Die nicht relevante Klasse wird mit Dokumenten aus dem Web initialisiert, die mit Yahoo Random Link (`http://random.yahoo.com/bin/ryl`) ausgewählt werden; der Internet-Katalog Yahoo bietet unter dieser Adresse die Möglichkeit, eine zufällige URL aus seinem Bestand zu erhalten.

Kombination der Breitensuche mit dem Klassifizierer

Als Klassifizierer wird für diese Arbeit *Bow* [McC96] eingesetzt, der unter anderem auch einen Naïve-Bayes-Modus bietet.

Bow liefert bei Eingabe eines Textes die Wahrscheinlichkeiten für jede Klasse, dass der Text dieser Klasse angehört. Die Wahrscheinlichkeit, der gleichen Klasse wie die vorgegebenen relevanten Seiten anzugehören, heiße *Relevanz*.

Die Suchstrategie RainbowStrategy[5] geht nun wie folgt vor: Jede Seite hat ein Attribut ranking, dass ihre Relevanz angibt. Trifft nun eine Seite wp ein, so wird sie klassifiziert und ihre Relevanz r bestimmt.

Falls r einen festen Grenzwert $t = 0.9$ überschreitet[6], werden ihre Nachfolgerseiten auch besucht. Dazu werden wie bei der Breitensuche die Links extrahiert und die von wp referenzierten Seiten $wp_1, \ldots, wp_n$ ermittelt. Diese werden nun in die HostQueue eingefügt, falls sie nicht bereits gelesen wurden. Dabei erhalten sie eine Priorität von $r' := f \cdot r$ mit $f := 1/2$. Diesem Wert von f liegt die Annahme zu Grunde, dass eine Seite mit der Relevanz r Links auf Seiten beinhaltet, die im Mittel eine Relevanz von $r/2$ haben.

Abbildung 3.2 auf der nächsten Seite zeigt ein Beispiel. Die Seite `www.cs.stanford.edu` wurde bereits gelesen und ihre Relevanz $r = 0.95$ vom Textklassifizierer bestimmt.

[5] benannt nach dem Frontend *rainbow* des Klassifizierers Bow; siehe auch Abschnitt 5.4
[6] Der genaue Wert von t hat keine große Bedeutung, wie in Abschnitt 12 erläutert wird.

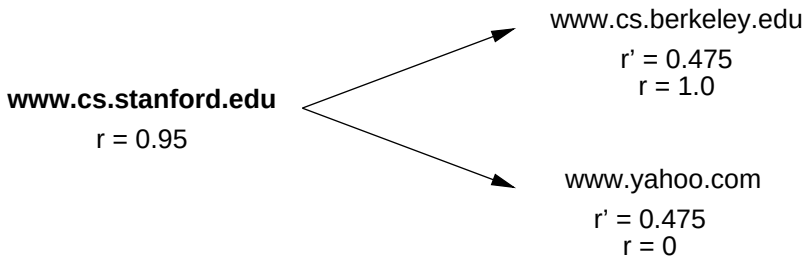


Abbildung 3.2: Beispiel: A-priori-Abschätzung der Relevanz

Die Nachfolgerseiten `www.cs.berkeley.edu` und `www.yahoo.com` werden demnach mit einer Priorität von $r' = r/2 = 0.475$ in die HostQueue eingefügt.

Erst wenn die beiden Nachfolgerseiten tatsächlich gelesen wurden, kann der Klassifizierer anhand des Seiteninhalts die Relevanz der beiden Seiten bestimmen; im Beispiel sind dies $r = 1$ für `www.cs.berkeley.edu` und $r = 0$ für `www.yahoo.com`;

Eine Seite wp_i wird also mit einer Priorität in die Schlange eingefügt, die einer a-priori-Abschätzung ihrer Relevanz entspricht. Diese Abschätzung wird anhand der ersten Seite wp errechnet, die einen Verweis auf wp_i beinhaltet. Die tatsächliche Relevanz von wp_i kann erst bestimmt werden, wenn diese Seite gelesen wurde.

Durch diese Höherpriorisierung vermutlich relevanter Seiten kommt es dazu, dass der Crawler schneller in die Tiefe vordringt, wie im Abschnitt 3.1.3 belegt wird.

Ein genauerer Wert für den oben eingeführten Faktor f ließe sich durch einen Feedback-Mechanismus ermitteln, der f für jede Seite oder global aus den bisher gelesenen Nachfolgerseiten errechnet. Ein solches Verfahren wird allerdings im Crawler nicht implementiert.

Beispiel für die Textklassifikation

Um ein Beispiel für die Arbeit des Textklassifizierers zu geben, kann man etwa die folgenden beiden Seiten betrachten:

Department of Computer Engineering, Middle East Technical University URL: `http://www.ceng.metu.edu.tr`

Diese Seite – die Startseite einer Informatik-Abteilung – wird mit 1 bewertet. Siehe auch Abbildung 3.3

University of Portsmouth URL: `http://www.port.ac.uk/index.htm`

Diese allgemeine Startseite einer englischen Universität wird mit 0 bewertet. Siehe auch Abbildung 3.4

Der Klassifizierer liefert fast immer Werte sehr nahe bei 0 oder 1; Bewertungen dazwischen sind selten. Siehe dazu auch Abschnitt 12 und Abbildung 3.6.

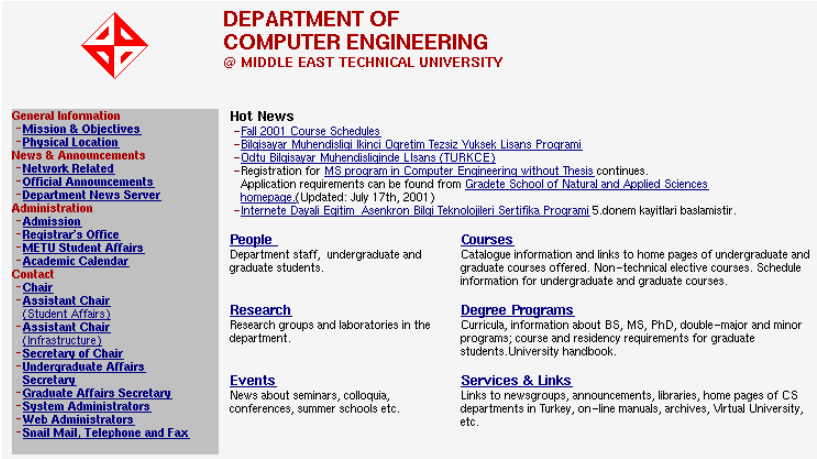


Abbildung 3.3: Beispiel: als relevant klassifizierte Seite

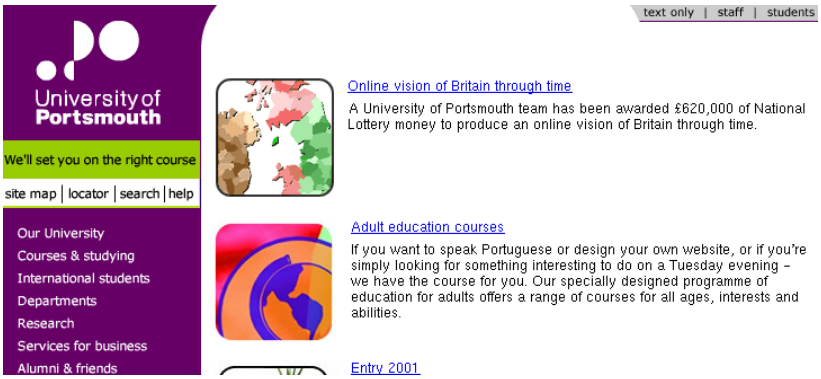


Abbildung 3.4: Beispiel: als nicht relevant klassifizierte Seite

3.1.3 Vergleich der Strategien

Ausgangslage

Es werden zwei Läufe des Crawlers betrachtet. Der erste Lauf benutzt die Breitensuche mit maximaler Tiefe 2, während der zweite Lauf die fokussierte Strategie mit unbeschränkter Tiefe nutzt.

Durch die Tiefenbeschränkung ist der BFS-Lauf nach etwa 180000 gelesenen Seiten beendet. Daher wird der zweite Lauf abgebrochen, wenn er vergleichbar viele Seiten abgearbeitet hat. Damit lassen sich die beiden Läufe quantitativ vergleichen.

Die Mengen Informatik-relevanter Startseiten werden manuell ermittelt aus Quellen wie

- dem Open Directory Project:
  `http://dmoz.org/Computers/Computer_Science/Academic_Departments/` und Unterseiten davon
  `http://dmoz.org/Science/Institutions/Research_Institutes/Computer_Science_Research/`
- Yahoo:
  `http://dir.yahoo.com/Science/Computer_Science/College_and_University_Departments/`
- einzelnen Auflistungen von Informatik-Abteilungen:
  `http://www.cs.haverford.edu/CS-Departments.html`
  `http://src.doc.ic.ac.uk/bySubject/Computing/UniCompSciDepts.html`
  `http://www-2.cs.cmu.edu/~ari/htmls/OtherCS.html`

Aus diesen Seiten werden nach Zusammenfassen und Entfernen von Duplikaten 1830 URLs gewonnen, die beim BFS-Lauf als Startseiten vorgegeben werden.

Für den Lauf mit der fokussierten Strategie werden zur Zeiterparnis die Seiten entfernt, die beim BFS-Lauf schon nicht gelesen werden konnten.[7] Dadurch startet dieser Lauf mit nur 1632 Seiten.

[7] Server existiert nicht, HTTP-Fehler 404 (Not Found), usw.

Vollständige Listen der Startseiten steht auf der CD im Verzeichnis auswertung/starturls.

Struktur der bearbeiteten Teilgraphen

Die Unterschiede in der Art, wie die BFS- und die fokussierte Strategie das Web traversieren, führt zu sehr verschieden strukturierten Graphen. Abbildung 3.5 und Tabelle 3.1 stellen die Anzahl der erfolgreich gelesenen Seiten in Abhängigkeit der Crawl-Tiefe dar.

Wie Tabelle 3.1 für die Tiefen 0–2 zeigt, wächst die Anzahl der Seiten beim BFS-Lauf anfangs exponentiell mit der Tiefe. Auch für Tiefe 3 trifft das noch zu; für diese Tiefe liegen 1556910 URLs vor.

Die ermittelten Daten lassen den Schluß zu, dass ein vollständiges Abarbeiten der dritten Ebene Dutzende Gigabytes von Speicherplatz und einige Monate Zeit in Anspruch nehmen würde.[8] Daher wird bei BFS die Crawl-Tiefe auf 2 beschränkt.

Im Gegensatz dazu hat die Bevorzugung relevanter Seiten bei der fokussierten Strategie zur Folge, dass schneller und mit etwa gleichbleibender Breite in die Tiefe vorgedrungen wird. So werden Seiten bis maximal zur Tiefe 15 gelesen, obwohl insgesamt nur unwesentlich mehr Seiten bearbeitet werden als beim BFS-Durchlauf.

Die dritte Kurve in Abbildung 3.5 zeigt die Anzahl der Seiten, die vom Klassifizierer mit einer Relevanz von mehr als 0.9 bewertet werden, *nachdem* sie tatsächlich gelesen wurden. Es sei darauf hingewiesen, dass hier ein zweifacher Auswahlprozess erfolgt ist: zum einen wird der Klassifizierer eingesetzt, um durch Beurteilung der verweisenden Seiten eine a-priori-Einschätzung über neue Seiten zu erhalten und so in der Warteschlange Seiten zu priorisieren, die *vermutlich* relevant sind. Andererseits werden unter den so gefundenen Seiten diejenigen ausgewählt, die nach ihrem Eintreffen *tatsächlich* als relevant beurteilt worden sind.

Die genaue Wahl der Schranke 0.9 spielt dabei keine große Rolle. Wie das Histogramm in Abbildung 3.6 bzw. die Werte in Tabelle 3.2 zeigen, liegen fast alle Relevanzwerte in den Intervallen $[0, 0.05)$ und $[0.95, 1]$ (logarithmische Skala!). Der Klassifizierer ist sich also fast immer „sicher“, ob eine Seite nicht relevant oder relevant ist.

[8] Die 178987 Seiten der Ebenen 0-2 belegen etwa 3 GB Speicherplatz, und es konnten im Mittel etwa 1000 Seiten pro Stunde abgearbeitet werden (s. Abschnitt 6.2.1). Wenn man die Zahlen entsprechend extrapoliert, bräuchte man 26 GB bzw. 65 Tage für die etwa 1.5 Millionen Seiten der Ebene 3.

Tiefe	**Anzahl erfolgreich gelesener Seiten**		
	BFS	fokussiert	fokussiert, Relevanz ≥ 0.9
0	1405	1374	908
1	18771	7565	5468
2	158811	10928	6960
3		15140	9080
4		21298	13199
5		22970	14800
6		27243	17946
7		29498	18926
8		27943	18062
9		22614	14543
10		14266	9637
11		1165	660
12		364	229
13		129	77
14		35	23
15		21	12
Summe	178987	202553	130530

Tabelle 3.1: Crawl-Tiefe und Anzahl der Seiten

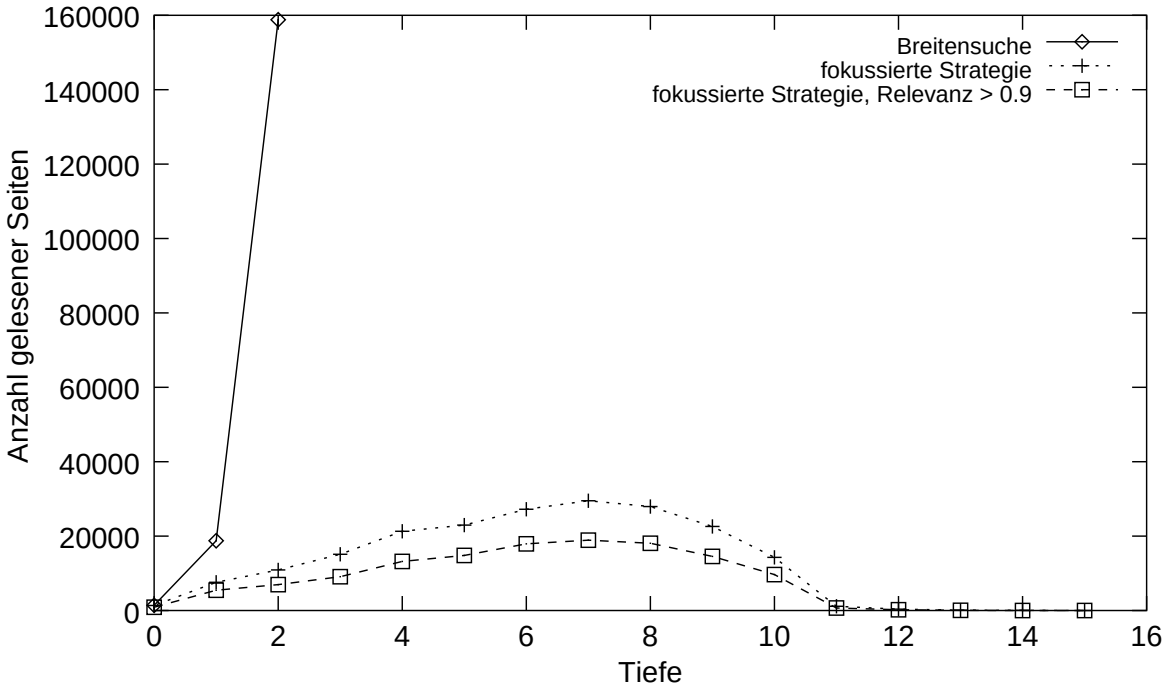


Abbildung 3.5: Crawl-Tiefe und Anzahl der Seiten

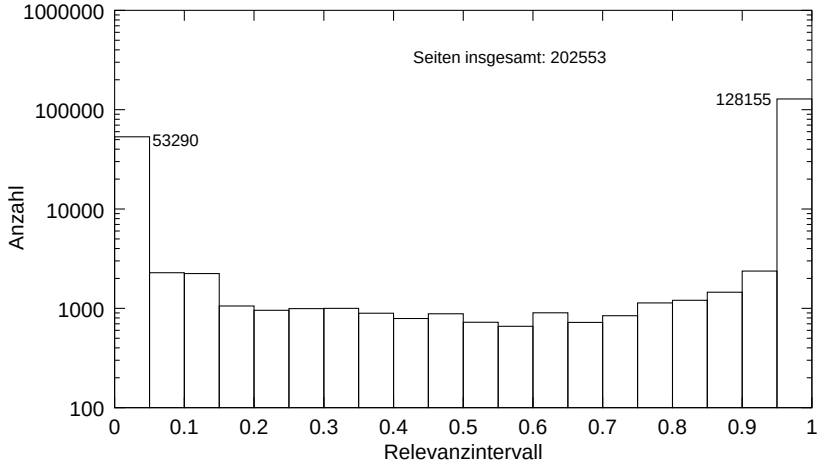


Abbildung 3.6: Histogramm der Relevanzwerte

Intervall	Anzahl Seiten
$[0, 0.05)$	53290
$[0.05, 0.1)$	2280
$[0.1, 0.15)$	2236
$[0.15, 0.2)$	1056
$[0.2, 0.25)$	956
$[0.25, 0.3)$	995
$[0.3, 0.35)$	1001
$[0.35, 0.4)$	893
$[0.4, 0.45)$	790
$[0.45, 0.5)$	883
$[0.5, 0.55)$	724
$[0.55, 0.6)$	659
$[0.6, 0.65)$	902
$[0.65, 0.7)$	722
$[0.7, 0.75)$	844
$[0.75, 0.8)$	1135
$[0.8, 0.85)$	1204
$[0.85, 0.9)$	1453
$[0.9, 0.95)$	2375
$[0.95, 1]$	128155

Tabelle 3.2: Verteilung der Relevanzwerte

Die BFS-Strategie arbeitet den von den Startseiten aus erreichbaren Graphen vollständig ab, kann aber mit den vorhandenen Ressourcen nur bis zur Tiefe 2 vordringen. Anfangs wächst der erreichbare Graph noch exponentiell, aber jenseits der Tiefe 3 kann über die Breite der einzelnen Ebenen keine sichere Aussage mehr gemacht werden.

Im Gegensatz dazu dringt die fokussierte Strategie bis Tiefe 15 vor, liest aber aus jeder Ebene nur einen Bruchteil der Seiten – schon bei Tiefe 2 liest BFS fünfzehn Mal so viele Seiten wie die fokussierte Strategie.

Es ist zu beachten, dass die Tiefe eines bestimmten Knotens in einem gegebenen Graphen bei den beiden Strategien verschieden sein kann. Abbildung 3.7 zeigt dies. Während der graue Knoten in der BFS-Strategie bei Tiefe 2 entdeckt würde, findet ihn die fokussierte Strategie erst bei Tiefe 4, da zunächst die von relevanten Knoten ausgehenden Kanten verfolgt werden.

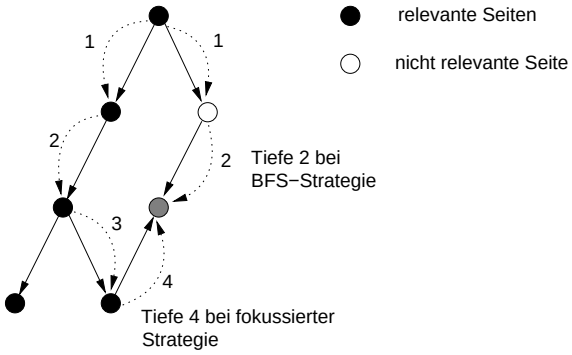


Abbildung 3.7: Tiefe eines Knotens

Abbildung 3.8 stellt Struktur der bearbeiteten Teilgraphen nochmals schematisch dar.

Beurteilung der ermittelten Seiten

Um die Qualität der gefundenen Seiten zu bewerten, werden zu den Strategien für jede Tiefe 100 zufällig gewählte Seiten[9] einer manuellen Beurteilung in Bezug darauf unterzogen, ob sie sich thematisch mit Informatik befassen. Beim zweiten Lauf wird zusätzlich die Menge der Seiten betrachtet, die der Klassifizierer mit einer Relevanz von mehr als 0.9 beurteilt.

Um die Beurteilung durchzuführen, wird ein Programm AssessPages erstellt. Dieses wählt aus einer vorgegebenen Menge von Seiten zufällig Seiten aus und zeigt sie in einem

[9] oder weniger, falls keine 100 Seiten in dieser Tiefe vorhanden waren

Fenster an. Der Benutzer kann dann entscheiden, ob eine gezeigte Seite relevant ist oder nicht; das Programm zählt entsprechend mit. Die Menge der betrachteten Seiten kann über eine SQL-Abfrage vorgegeben werden.

Abbildung 3.9 zeigt ein Bild des Programms. Da die HTML-Komponente der Java-Bibliothek Swing nicht alle Seiten fehlerfrei darstellen kann, wird zur Sicherheit auch noch der HTML-Quellcode angezeigt.

Tabelle 3.3 und Abbildung 3.10 zeigen die *Präzision* der Crawls für die Seiten der jeweiligen Tiefe. Die Präzision ist dabei definiert als der Anteil der relevanten Seiten an den insgesamt gefundenen Seiten. Dabei kann man folgende Beobachtungen machen:

- Auch bei BFS ist die Präzision mit mehr als 0.6 recht hoch, obwohl keinerlei Selektion beim Verfolgen der Links vorgenommen wird. Dies deckt sich mit den Beobachtungen aus [Naj01]. Die Frage, wie sich die Präzision bei größeren Tiefen verhält, kann mit den vorliegenden Daten allerdings nicht beantwortet werden.
- Die durch den Klassifizierer fokussierte Strategie liefert anfangs praktisch gleich gute Seiten wie BFS. Sie kann die Qualität der Seiten bis zu größeren Tiefen annähernd aufrecht erhalten.

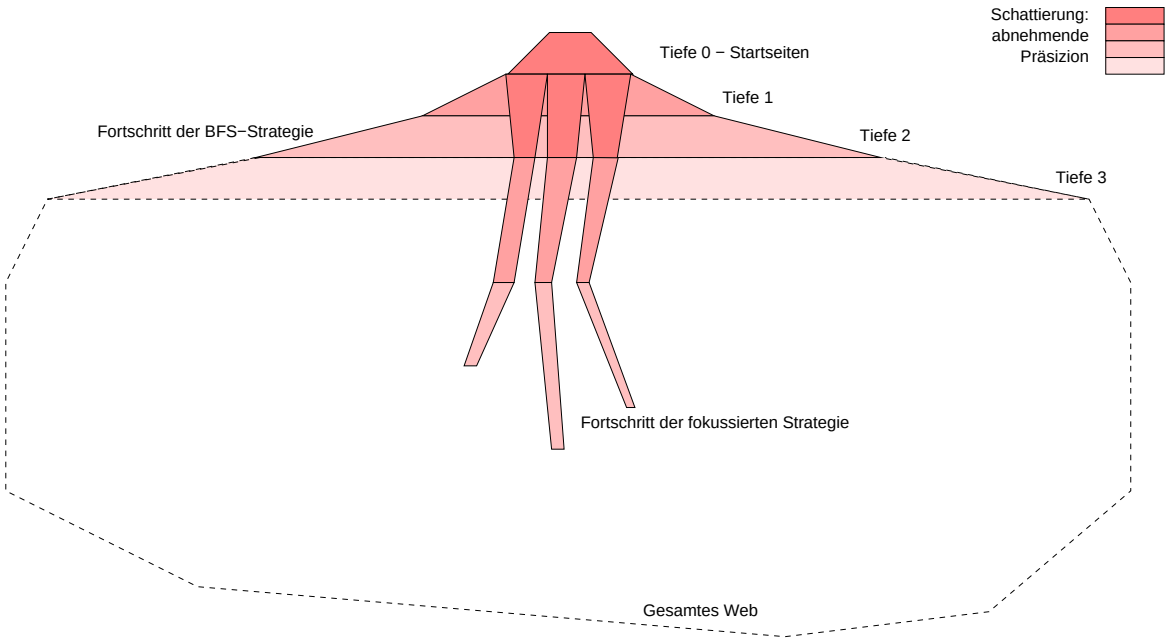


Abbildung 3.8: Vergleich der Strategien

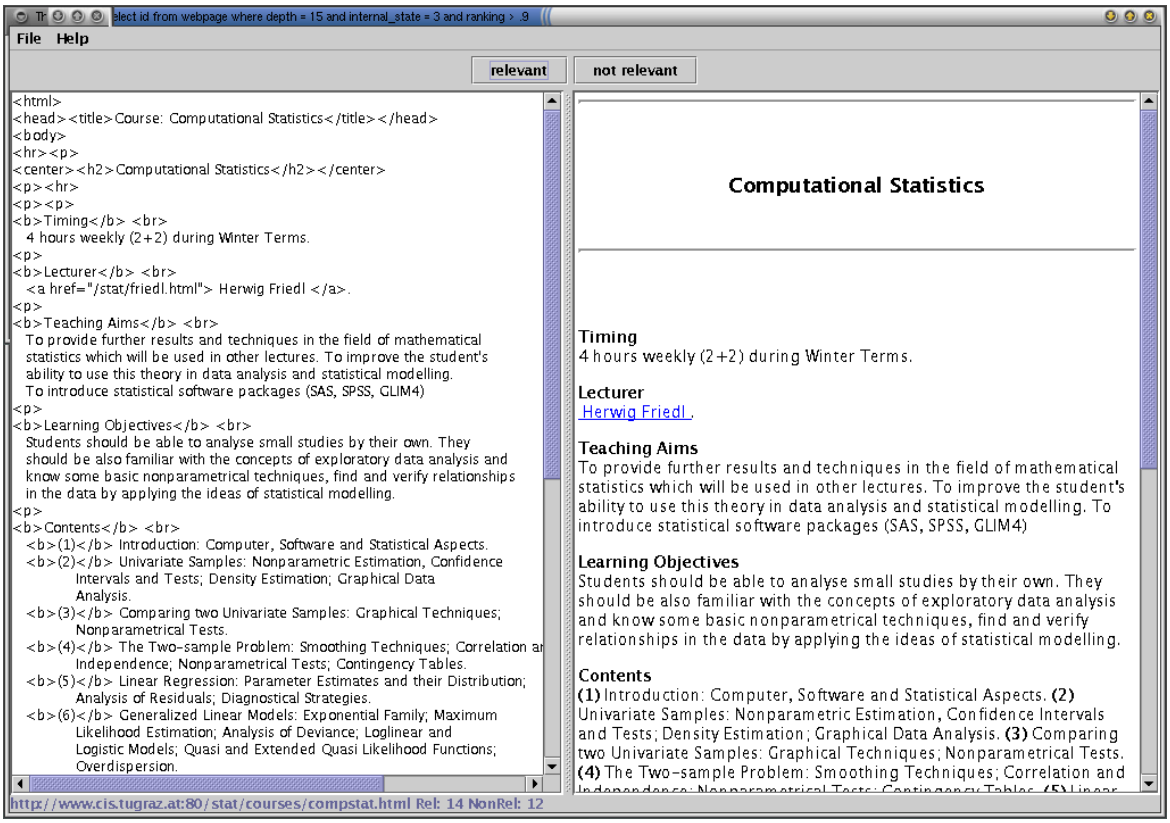


Abbildung 3.9: Beurteilung der Relevanz mit AssessPages

Tiefe	**Präzision**		
	BFS	fokussiert	fokussiert, Ranking ≥ 0.9
0	0.94	0.96	0.96
1	0.75	0.71	0.83
2	0.61	0.56	0.75
3		0.49	0.70
4		0.51	0.63
5		0.46	0.59
6		0.47	0.52
7		0.45	0.53
8		0.40	0.45
9		0.44	0.50
10		0.44	0.48
11		0.49	0.42
12		0.41	0.45
13		0.39	0.39
14		0.46	0.30
15		0.33	0.33

Tabelle 3.3: Crawl-Tiefe und Präzision

Angesichts des exponentiellen Wachstums bei der BFS-Strategie erscheint es fraglich, ob diese eine hohe Präzision bei dieser Tiefe noch erreichen würde. Dies gilt besonders unter dem Gesichtspunkt, dass der mittlere Abstand zweier beliebiger Webseiten als nur etwa 20 eingeschätzt wird [Alb99]. Damit hätte man bei Tiefe 15, die die fokussierte Strategie erreicht, schon einen beträchlichen Teil des gesamten Web traversiert.

- Wählt man innerhalb der Seiten der fokussierten Strategie nochmals diejenigen aus, die bei der automatischen Beurteilung eine hohe Relevanz erhalten haben, so erhöht sich die Präzision deutlich.

Der Klassifizierer liefert also Ergebnisse, die mit einer manuellen Beurteilung zum großen Teil übereinstimmen.

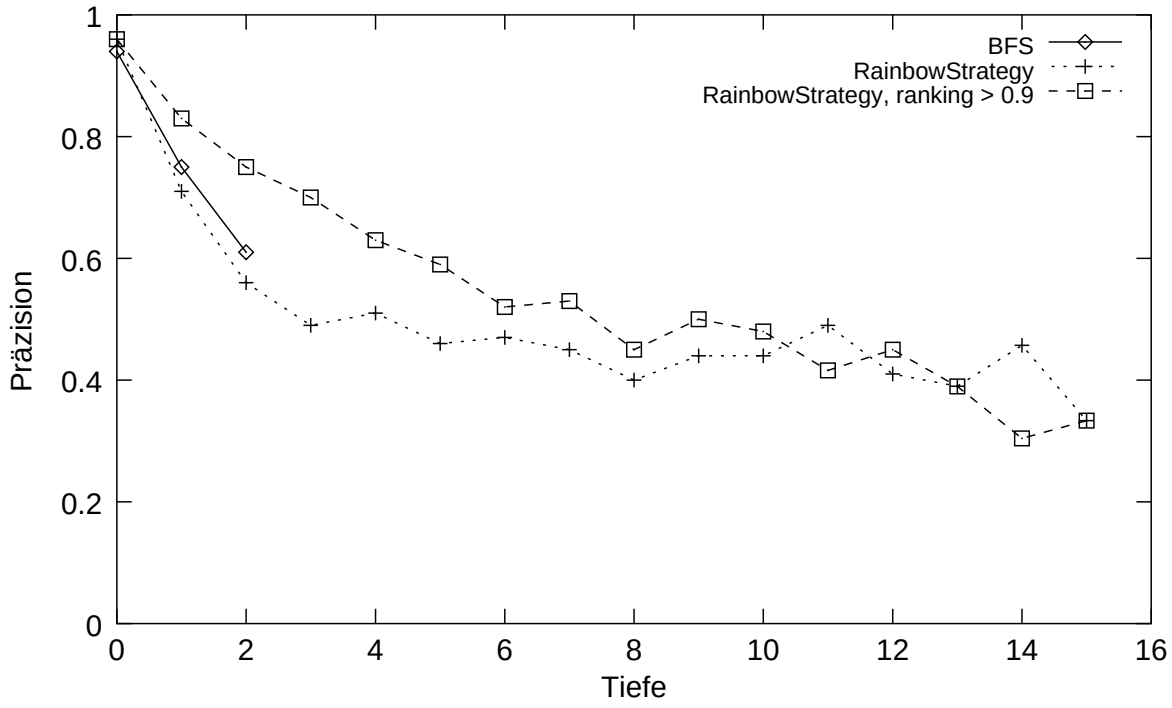


Abbildung 3.10: Crawl-Tiefe und Präzision

3.2 Beschaffenheit der ermittelten Daten

Der Crawler ermittelt – abgesehen von Status- und Fehlerinformationen – im Wesentlichen die Knoten (Seiten) und gerichteten Kanten (Hyperlinks) eines Teils des Webgraphen.

In den zu Grunde liegenden Protokollen für das WWW gibt es keine inhaltlich verschiedenen Arten von Knoten und Kanten.[10] Es wird z. B. keine Unterscheidung getroffen zwischen Kanten, die auf andere Dokumente verweisen, und solchen, die innerhalb eines mehrseitigen Dokuments zur Navigation dienen.

Für die weitere Betrachtung ist es sinnvoll, die ermittelten Knoten und Kanten zu differenzieren.

3.2.1 Unterscheidung der Knoten und Kanten nach ihrer Bedeutung im Graphen

Arten von Knoten

Die Größe und Dynamik des WWW lässt eine vollständige Betrachtung naturgemäß nicht zu. Selbst für die großen Suchdienste mit ihren enormen Ressourcen ist dies nicht möglich.

Abbildung 3.11 auf der nächsten Seite stellt die Sicht dar, die eine Betrachtung einer Teilmenge des WWW bietet.

Vom gesamten Graphen $G_{web} = (V_{web}, E_{web})$ des Web wird eine Teilmenge $V_{besucht}$ der Knoten besucht und ihr Inhalt gelesen. Diese Knoten sind weiß dargestellt.

Eine weitere Menge V_{rand} von Knoten wird von $V_{besucht}$ referenziert, also:

$$V_{rand} = \{v \in V_{web} : v \notin V_{besucht} \wedge \exists (u, v) \in E_{web} : u \in V_{besucht}\}$$

Die Knoten in V_{rand} sind in der Abbildung schwarz dargestellt. Diese Menge wird im Folgenden als *Rand* bezeichnet.

Sowohl in $V_{besucht}$ als auch in V_{rand} sind nur die Kanten bekannt, die von $V_{besucht}$ ausgehen, also

[10] Auf verschiedene technische Varianten von Hyperlinks wird in Abschnitt 4.2.1 eingegangen.

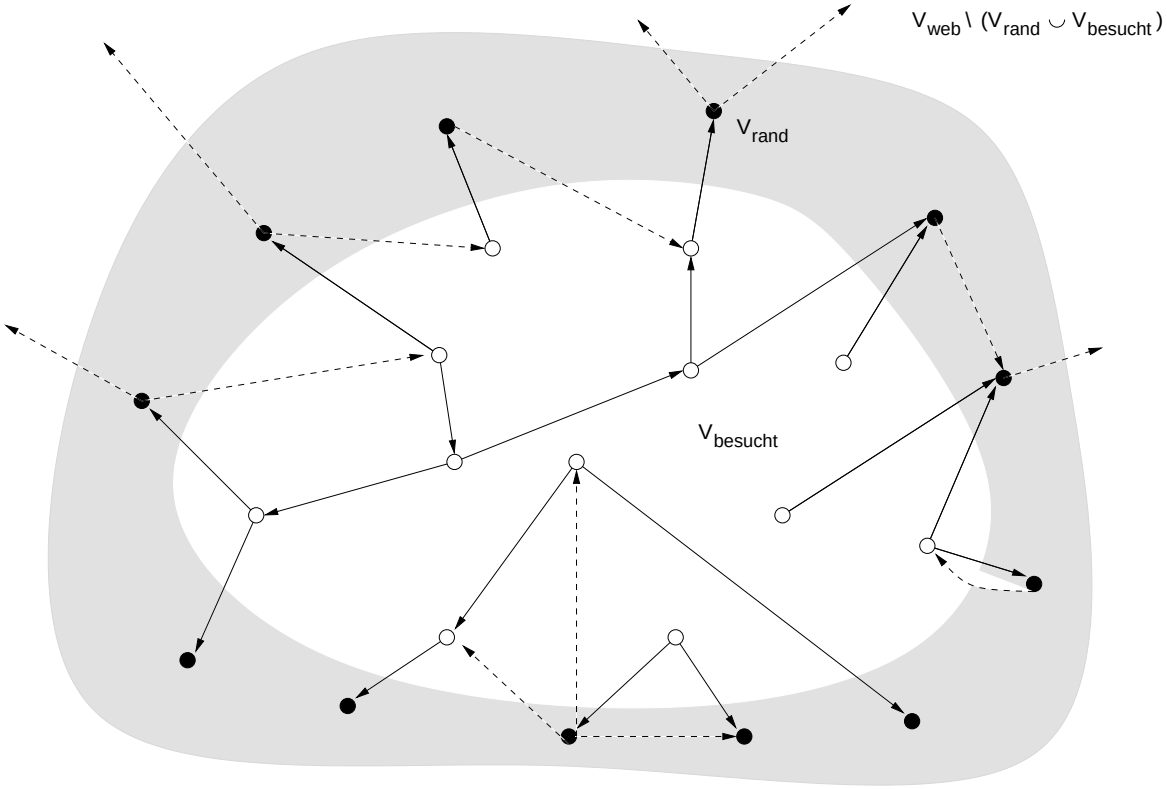


Abbildung 3.11: Rand des betrachteten Graphen

$$E_{besucht} = \{(u, v) \in E_{web} : u \in V_{besucht}\}$$

Von den Knoten im Rand kennt man nur die Adresse, weil ein Link auf sie zeigt. Es ist jedoch nicht bekannt, ob sie überhaupt noch existieren und welche Links von ihnen ausgehen – diese unbekannten Links sind hier gestrichelt dargestellt. Hyperlinks im WWW sind unidirektional, d. h. sie können nur in einer Richtung verfolgt werden. Damit können die Links, die von Knoten im Rand ausgehen, aus den bekannten Daten nicht ermittelt werden.

Schließlich gibt es noch Knoten, die außerhalb des Randes liegen, also in $V_{web} \backslash (V_{besucht} \cup V_{rand})$. Über diese ist überhaupt nichts bekannt.

Arten von Kanten

In manchen traditionellen Hyperlink-Systemen werden verschiedene Arten von Links semantisch unterschieden [Con87]. Es gibt dabei Links für verschiedene Zwecke: Querverweise, hierarchische Strukturierung, Rückwärtsverweise, Fußnoten, usw.

Im WWW wird diese Unterscheidung auf der Ebene der zu Grunde liegenden Protokolle nicht gemacht; hier sind zunächst alle Links gleichartig. (Die in Abschnitt 4.2.1 vorgestellte Unterscheidung nach der technischen Realisierung von Links hat damit nichts zu tun.)

Die Beziehung zwischen den Hostnamen von Ausgangs- und Zielseite eines Hyperlinks liefert allerdings ein Unterscheidungsmerkmal. Dazu werden für Hyperlinks zwei Eigenschaften unterschieden, die angeben, ob ein Link die Grenzen eines Hosts (spans_hosts) oder einer Website (spans_sites) überschreitet.

spans_hosts: Diese Eigenschaft gibt an, ob die Ausgangs- und Ziel-URL eines Links l von p_1 nach p_2 die gleichen Hostnamen und Portnummern haben:

$$\text{spans\_hosts}(l) = \begin{cases} 1 & \text{falls } p_1, p_2 \text{ unterschiedliche Hostnamen} \\ & \text{oder Portnummern haben} \\ 0 & \text{sonst} \end{cases}$$

spans_hosts ist also genau dann 0, wenn der Link innerhalb eines Hosts bleibt.

spans_sites: Diese Eigenschaft gibt an, ob ein Link über die Grenzen einer *Site* hinweggeht:

$$\text{spans\_sites}(l) = \begin{cases} 1 & \text{falls } p_1, p_2 \text{ in verschiedenen Sites (s. u.) liegen} \\ 0 & \text{sonst} \end{cases}$$

spans_sites ist genau dann 0, wenn der Link innerhalb einer Site bleibt.

Anmerkung: Mit diesen Definitionen gilt für alle Links l die Implikation

$$\textit{spans\_sites}(l) \Rightarrow \textit{spans\_hosts}(l)$$

da spans_sites $= 1$ verschiedene Hostnamen impliziert.

Eine Site ist dabei zu verstehen als eine Menge zusammengehöriger Seiten, die aber nicht notwendig auf einem Host vereinigt sind. So finden sich oft Unterteilungen wie

- `www.stanford.edu`
- `theory.stanford.edu`
- `weather.stanford.edu`
- `suif.stanford.edu`
- `vision.stanford.edu`
- `www.cs.stanford.edu`

Dies sind nur einige der Hostnamen, die in der Domain `stanford.edu` vorkommen.

Um solche Fälle zu erkennen, wird eine Heuristik benutzt, die anhand der Hostnamen von Ausgangs- und Zielseite des Links entscheidet, ob der Link Site-übergreifend ist:

1. Es werden die Hostnamen von Ausgangs- und Ziel-URL betrachtet.
2. Von beiden Hostnamen wird jeweils der erste Teil bis zum Punkt abgetrennt, falls er mit `www` beginnt, genauer: falls er dem regulären Ausdruck[11] `www[^.]*\.` entspricht.
3. Falls die resultierenden Namen aus drei Teilen oder mehr bestehen, wird der erste Teil abgeschnitten.

[11] in der Syntax von Unix-Tools wie grep oder sed

4. Es wird geprüft, ob eine der übriggebliebenen Zeichenketten Suffix der anderen ist. Falls ja, so werden die Namen als der selben Site angehörig angesenen.

Beispiel: Liegen die URLs `http://publications.wep.uni-trier.de/index.html` und `http://www1.informatik.uni-trier.de/` in der selben Website?

1. Hostnamen:

   `publications.wep.uni-trier.de` `www1.informatik.uni-trier.de`

2. Abtrennen eines Präfixes `www[^.]*\.`, falls vorhanden; für den zweiten Hostnamen ist dies „`www1.`“

   `publications.wep.uni-trier.de` `informatik.uni-trier.de`

3. Kürzen der Namen um einen Teil, falls sie drei oder mehr Bestandteile haben.

   `wep.uni-trier.de` `uni-trier.de`

4. Da `uni-trier.de` ein Suffix von `wep.uni-trier.de` ist, werden diese beiden Hostnamen und damit die beiden genannten URLs der selben Site zugerechnet.

3.2.2 Vorauswahl der untersuchten Daten

Wie Tabelle 3.4 zeigt, liegen bei der BFS- bzw. der fokussierten Strategie 77 (61) Prozent der Links innerhalb eines Hosts und 88 (82) Prozent innerhalb einer Site.

Durch diese sehr viel stärkere Verbindung innerhalb von Sites werden die im Folgenden vorgestellten Algorithmen gestört: es werden Personen oder Institutionen gefunden, die *innerhalb* ihrer Website interessante Strukturen aufweisen, anstatt Verbindungen *zwischen* solchen Sites zu finden.

Diese inneren Strukturen variieren stark, z. B. durch die Art der eingesetzten Navigationselemente, und sagen nichts aus über Verbindungen nach außen. Diese Beobachtung – zumindest die Unterscheidung zwischen Links innerhalb eines Hosts und denen, die Hostgrenzen überspannen – hat beispielsweise auch Kleinberg in seinem Papier zu HITS[12] [Kle99, Abschnitt 2] gemacht.

Ebenso führt die Situation am Rand des Graphen zu eine Verfälschung der Betrachtung. Würde man die Knoten im Rand auch betrachten, so würde der Graph

$$G_{rand'} = (V_{rand}, E_{besucht})$$

ausgewertet. In diesem Graphen fehlen aber die Kanten, die von Knoten im Rand ausgehen, da diese Knoten nicht gelesen wurden. Diese Knoten haben aus Sicht der Algorithmen nur eingehende, aber keine ausgehenden Kanten.

Dadurch „saugen“ z. B. solche Knoten die Gewichte bei den PageRank- und HITS-Algorithmen aus dem Graphen auf (s. 3.5 bzw. 3.6).

Um diese Probleme zu vermeiden, werden im Folgenden nur die tatsächlich gelesenen Seiten und die Site-übergreifenden Links berücksichtigt, falls nichts anderes angegeben ist. Der betrachtete Graph ist also

$$G_{besucht,spans\_sites} = (V_{besucht}, \{e \in E_{besucht} : \mathsf{spans\_sites}(e)\})$$

[12] Hyperlink-Induced Topic Search

Strategie	**Art der Links**	**Anzahl**	**Anteil**
BFS	gesamt	4966248	100 %
	Host-übergreifend	1125450	23 %
	Site-übergreifend	618371	12 %
fokussiert	gesamt	3857757	100 %
	Host-übergreifend	1527650	39 %
	Site-übergreifend	717724	18 %

Tabelle 3.4: Anzahl der Hyperlinks

3.3 Die Bowtie-Struktur

3.3.1 Die Bowtie-Struktur auf dem gesamten Web

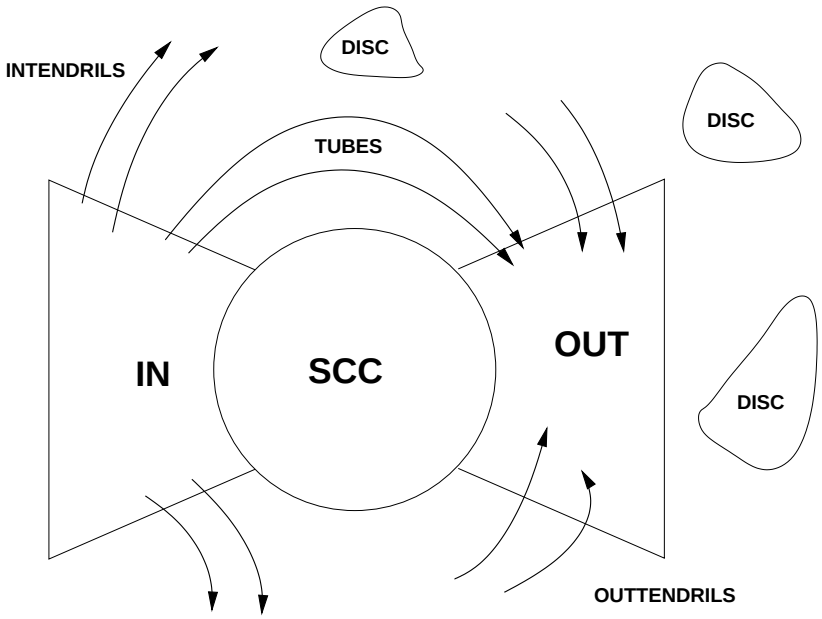


Abbildung 3.12: Bowtie-Struktur

Broder, Kumar u. a. beschreiben in [Bro00] eine Struktur im WWW-Graphen, die der Form einer Fliege (engl. *bowtie*) entspricht. Diese Struktur haben sie bei der Untersuchung von etwa 200 Millionen Seiten des Webgraphen entdeckt.

Das Zentrum der Fliege wird dabei von einer starken Zusammenhangskomponente (*strongly connected component, SCC*) gebildet, in der zwischen je zwei Knoten ein gerichteter Pfad existiert.

Die Menge *IN* besteht aus den Knoten, von denen ein Pfad nach *SCC* besteht, die aber nicht mit *SCC* stark verbunden sind. Analog seien *OUT* die Knoten, die von *SCC* aus erreichbar, aber damit nicht stark verbunden sind.

Weiterhin existieren sogenannte *TENDRILS* (Ranken). Diese unterteilen sich in *INTENDRILS*, *OUTTENDRILS* und *TUBES*. *INTENDRILS* sind solche Pfade, die von *IN* ausgehen, aber nicht nach *SCC* führen. Analog führen die *OUTTENDRILS* nach *OUT*, gehen

aber nicht von *SCC* aus. Ein Sonderfall davon sind sogenannte *TUBES*, die von *IN* nach *OUT* führen, ohne durch *SCC* zu gehen.

Die bisher genannten Mengen *SCC*, *IN*, *OUT* und *TENDRILS* bilden eine schwache Zusammenhangskomponente. Außerhalb davon gibt es noch Knoten, die mit *SCC* nicht verbunden sind. Diese werden als *DISC* (*disconnected*) bezeichnet.

Broder u. a. haben die Größenverhältnisse festgestellt, die in Tabelle 3.5 angegeben sind. Allerdings wird dabei nicht aufgeschlüsselt zwischen *INTENDRILS*, *OUTTENDRILS* und *TUBES*; diese Mengen werden auch in dem von ihnen angegebenen Rechenverfahren nicht unterschieden und sind hier unter *TENDRILS* zusammengefasst.

Tabelle 3.5 zeigt, dass *SCC* etwas mehr als ein Viertel des Graphen ausmacht, *IN*, *OUT* und *TENDRILS* jeweils etwa ein Fünftel und *DISC* den Rest.

Menge	**Anzahl**	%
SCC	56463993	27.74 %
IN	43343168	21.29 %
OUT	43166185	21.21 %
TENDRILS	43797944	21.52 %
DISC	16777756	8.24 %
Graph gesamt	203549046	100.00 %

Tabelle 3.5: Größenverhältnisse bei der Bowtie-Struktur im gesamten Web

3.3.2 Berechnung der Struktur auf den betrachteten Teilgraphen

Während die Berechnung der einzelnen Größen in [Bro00] teilweise indirekt über Abzählargumente erfolgt, geht das hier vorgestellte Programm konstruktiv vor. Für jeden Knoten des Graphen G wird explizit berechnet, welcher Teilmenge er angehört:

① Berechne mit dem LEDA[13]-eigenen Algorithmus alle starken Zusammenhangskom-

[13] Library of Efficient Data Types and Algorithms

ponenten. Unter diesen wird die größte Komponente *SCC* durch Auszählen[14] bestimmt.

② Bestimme *OUT*, indem von *SCC* aus eine Breitensuche durchgeführt wird. Dabei besuchte Seiten, die nicht zu *SCC* gehören, liegen in *OUT*.

③ Bestimme *IN*, indem von *SCC* aus eine Breitensuche durchgeführt wird; allerdings werden hier die Kanten *rückwärts* traversiert. Gefundene Seiten gehören zu *IN*, wenn sie nicht in *SCC* liegen.

④ Bestimme die *INTENDRILS* mittels Breitensuche von *IN* aus auf $G \backslash SCC$. Alle in dieser Suche gefundenen Knoten, die nicht in *IN* liegen, gehören zu *INTENDRILS*.

⑤ Bestimme die *OUTTENDRILS* mittels Breitensuche von *OUT* aus rückwärts (wie in ③) auf $G \backslash \mathrm{SCC}$. Analog zum vorigen Schritt gehören die gefundenen Knoten zu *OUTTENDRILS*, wenn sie nicht in *OUT* liegen.

⑥ Knoten in der Schnittmenge von *INTENDRILS* und *OUTTENDRILS* gehören auf jeden Fall zu *TUBES*.

⑦ Ausgehend von *TUBES*, finde mit Tiefensuche[15] (*Depth-First Search*, DFS) vorwärts und rückwärts Wege nach *OUT* bzw. *IN*. Stößt man dabei auf einen Knoten in *OUT* (*IN*), so gehören alle Knoten auf dem Weg auch zu *TUBES* und können aus den *OUTTENDRILS* (*INTENDRILS*) entfernt werden.

Abbildung 3.14 auf der nächsten Seite verdeutlicht diesen Schritt. Die Knoten auf dem Weg zwischen *TUBES* und *OUT* (*IN*), die dabei hinzugenommen werden, sind schwarz gekennzeichnet. Die gestrichelten Pfeile geben die Richtung der Tiefensuche an.

⑧ Alle Knoten, die nicht einer der anderen Mengen zugehören, sind nicht mit dem Rest verbunden (*disconnected*, *DISC*).

[14] LEDA liefert die Komponenten nicht direkt als Mengen, sondern für jeden Knoten eine Komponentennummer. Um die größte Komponente zu bestimmen, müssen diese Komponentennummern ausgezählt werden.

[15] Tiefensuche ist eine Variante der allgemeinen Graph-Suche aus Abschnitt 3.1, die einen Stack zur Verwaltung der Menge S benutzt. Dadurch wird der jeweils zuletzt entdeckte neue Knoten als erstes besucht, so dass immer möglichst weit in die Tiefe vorgedrungen wird.

Abbildung 3.13 verdeutlicht die Reihenfolge der Berechnung. Die gestrichelten Pfeile geben dabei die Richtung an, in der der Graph traversiert wird; diese ist in den Schritten ③, ⑤ und in einem Teil von Schritt 20 der Richtung der Kanten entgegengesetzt.

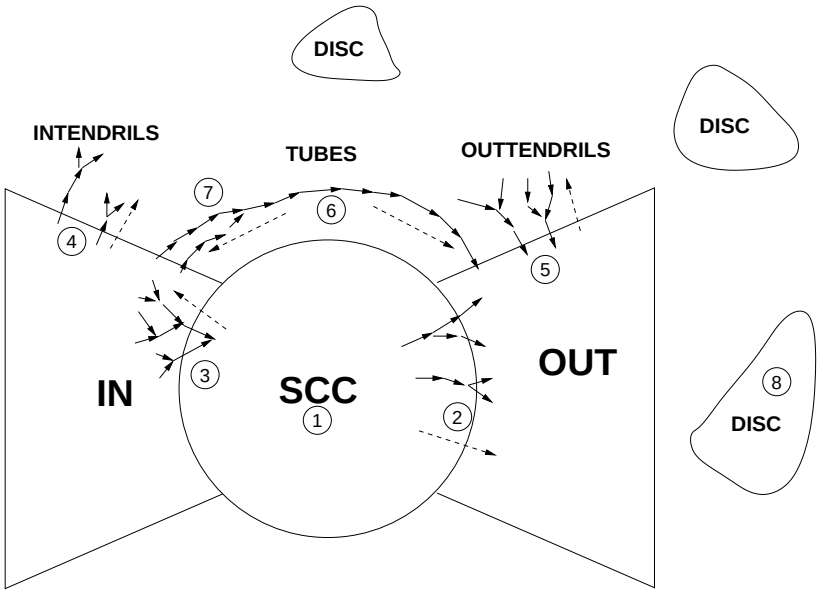


Abbildung 3.13: Ablauf des Bowtie-Algorithmus'

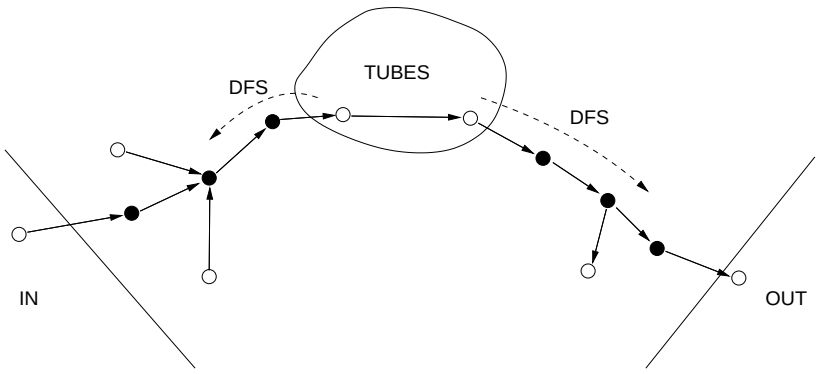


Abbildung 3.14: Ablauf des Bowtie-Algorithmus': Finden der *TUBES*

Dieser Algorithmus wird implementiert im Programm bowtie.cc. Zur Benutzung des Programms siehe auch Abschnitt 6.1.2.

3.3.3 Ergebnisse der Berechnung

Die oben geschilderte Berechnung wird auf den Graphen der beiden Strategien (s. Abschnitt 3.1) durchgeführt. Um die Vergleichbarkeit mit den Ergebnissen des Originalpapiers zu gewährleisten, wird neben der Variante mit Site-übergreifenden Links auch die Variante mit allen Links berechnet.

Zusätzlich wird der Algorithmus angewendet auf einen Graphen, der durch Zusammenfassen von Seiten entsteht, die jeweils auf einem gemeinsamen Server liegen.

Die Ergebnisse der Auswertung liegen im Verzeichnis auswertung/ergebnisse/bowtie auf der CD.

Variante 1: alle Links

Hier wird die Variante betrachtet, die alle Links berücksichtigt.

	Strategie					
Menge	**BFS**		**fokussiert**		**fokussiert, ranking > 0.9**	
SCC	103486	57.82 %	55480	27.69 %	53577	41.46 %
IN	11810	6.60 %	1670	0.83 %	2023	1.57 %
OUT	54709	30.57 %	134636	67.19 %	67425	52.20 %
TUBES	641	0.36 %	752	0.38 %	303	0.23 %
INTENDRILS	6469	3.61 %	4356	2.17 %	2059	1.59 %
OUTTENDRILS	576	0.32 %	1328	0.66 %	951	0.74 %
TENDRILS gesamt	7686	4.29 %	6436	3.21 %	3313	2.56 %
DISC	1295	0.72 %	2160	1.08 %	2851	2.21 %
Summe	178986	100.00 %	200382	100.00 %	129169	100.00 %

Tabelle 3.6: Bowtie-Struktur bei Berücksichtigung aller Links

In Tabelle 3.6 lässt sich beobachten:

1. BFS-Strategie:
   a) Über 88 Prozent der Knoten verteilen sich auf die Mengen *SCC* und *OUT*, etwa zwei Drittel davon in *SCC*.
   b) *IN* macht nur 6.6 Prozent der Knoten aus.
   c) Nur 4.29 Prozent der Knoten sind *TENDRILS*; der größte Teil davon befindet sich in *INTENDRILS*.
   d) *DISC* umfasst nur 0.72 Prozent der Knoten.
2. Fokussierte Strategie:
   a) Hier sind mehr als 94 Prozent der Knoten in $SCC \cup OUT$. Mehr als zwei Drittel davon befinden sich in *OUT*.
   b) *IN* besteht nur aus 0.83 Prozent der Knoten.
   c) *TENDRILS* ist mit 3.21 Prozent klein; der größte Teil befindet sich wiederum in *INTENDRILS*.
   d) *DISC* besteht aus nur 1.08 Prozent der Knoten.
3. Fokussierte Strategie, Relevanz > 0.9:
   a) Hier ist *OUT* wesentlich kleiner als in der vorherigen Messung. Ansonsten sind die Ergebnisse beinahe gleich.

Ein weiterer Vergleich zeigt die Größenverhältnisse, wenn man die fokussierte Strategie wie die BFS-Strategie auf Tiefe 2 beschränkt; siehe dazu Tabelle 3.7.

Hier ist *SCC* sehr klein; *OUT* und *DISC* sind verhältnismäßig groß.

Bemerkenswert ist auch, dass die Beschränkung auf Seiten mit einer Relevanz größer als 0.9 *SCC* und *IN* praktisch unverändert lässt.

Variante 2: nur Site-übergreifende Links

Tabelle 3.8 zeigt die Größenverhältnisse für die Berechnung, wenn nur Site-übergreifende Links einbezogen werden.

	Strategie			
Menge	**fokussiert, Tiefe 0–2**		**fokussiert, Tiefe 0–2** ranking > 0.9	
SCC	701	3.53 %	699	5.24 %
IN	254	1.28 %	254	1.90 %
OUT	9986	50.26 %	6428	48.20 %
TUBES	84	0.42 %	41	0.31 %
INTENDRILS	586	2.95 %	375	2.81 %
OUTTENDRILS	2193	11.04 %	1491	11.18 %
TENDRILS gesamt	2863	14.41 %	1907	14.29 %
DISC	6063	30.52 %	4048	30.35 %
Summe	19867	100.00 %	13336	100.00 %

Tabelle 3.7: Bowtie-Struktur bei Berücksichtigung aller Links

	Strategie					
Menge	**BFS**		**fokussiert**		**fokussiert, ranking > 0.9**	
SCC	375	0.21 %	1683	0.84 %	1659	1.28 %
IN	1926	1.08 %	3728	1.86 %	3709	2.87 %
OUT	5909	3.30 %	28305	14.13 %	14140	10.95 %
TUBES	1781	1.00 %	2272	1.13 %	1426	1.10 %
INTENDRILS	5500	3.07 %	14420	7.20 %	6529	5.05 %
OUTTENDRILS	24187	13.51 %	21101	10.53 %	13412	10.38 %
TENDRILS gesamt	31468	17.58 %	37793	18.86 %	21367	16.54 %
DISC	139308	77.83 %	128873	64.31 %	88294	68.26 %
Summe	178986	100.00 %	200382	100.00 %	129169	100.00 %

Tabelle 3.8: Bowtie-Struktur bei Berücksichtigung Site-übergreifender Links

Hier besteht *SCC* nur noch aus sehr wenigen Knoten. Durch den Wegfall der Site-internen Links wird also der starke Zusammenhang zerstört, der in der Variante mit allen Links festgestellt werden konnte. Abbildung 3.15 zeigt, wie dies zu Stande kommt.

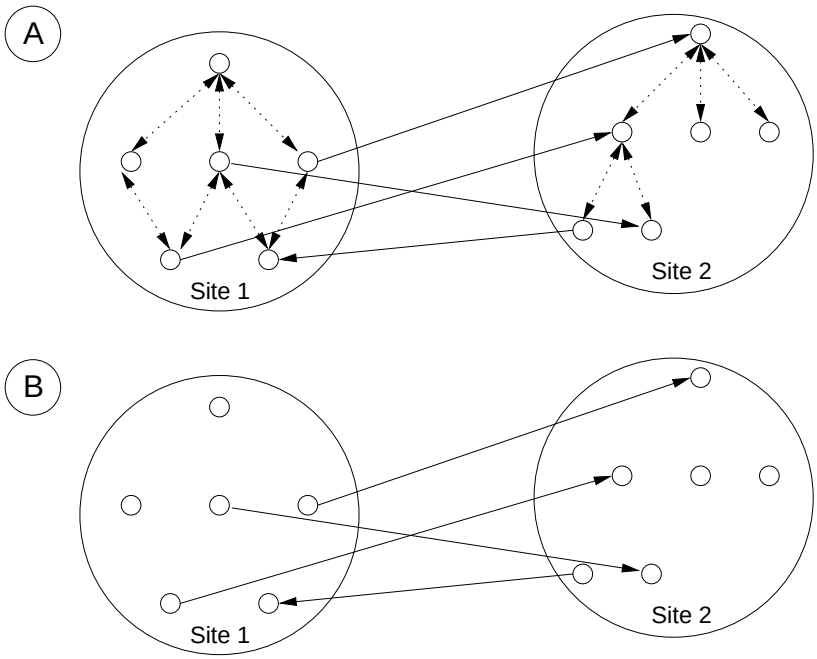


Abbildung 3.15: Zerfallen einer Komponente durch Wegfall Site-interner Links

Obwohl die Sites 1 und 2 eine starke Zusammenhangskomponente bilden (A), zerfällt diese nach Entfernen der Site-internen Links in starke Zusammenhangskomponenten bestehend aus einzelnen Knoten.

Eine weitere Betrachtung der anderen Mengen, die definiert sind über ihre Beziehung zu *SCC*, ist hier nicht sinnvoll.

Variante 3: Links auf Host-Ebene

Für diese Berechnung werden Seiten zu einem Knoten zusammengefasst, die auf einem gemeinsamen Host liegen.

Im so enstandenen neuen Graphen führt ein Link von Host h_1 nach h_2, wenn *irgendeine* Seite auf h_1 auf *irgendeine* Seite auf h_2 verweist.

Abbildung 3.16 verdeutlicht diese Zusammenfassung mehrerer Knoten zu einem Knoten pro Host.

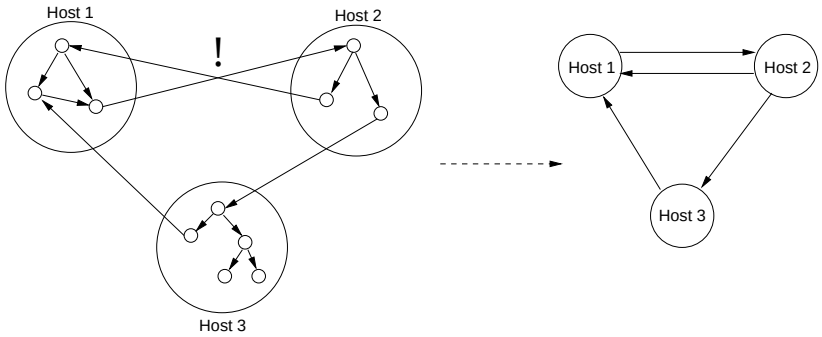


Abbildung 3.16: Zusammenfassung der Seiten eines Hosts

Zur Motivation für dieses Zusammenfassen kann die Beobachtung dienen, dass Links auf fremde Angebote bevorzugt auf deren Startseite oder Seiten nahe der Startseite gesetzt werden.[16] Andererseits verweisen Startseiten von Websites selten auf andere Websites. Dadurch kommt es u. U. zu einer „Über-Kreuz“-Situation, wie sie in der Abbildung durch das Ausrufezeichen gekennzeichnet ist – durch die Zusammenfassung wird daraus eine Zweierclique.

Zur Berechnung dieses zusammengefassten Graphen dient das SQL-Skript calc_hostlink.sql. Dabei werden nur die Hosts berücksichtigt, von denen mindestens eine Seite tatsächlich gelesen wurde, sowie die Hyperlinks, die innerhalb dieser Menge bleiben.

Tabelle 3.9 zeigt die Größenverhältnisse auf diesem Graphen.

Die Größenverhältnisse auf Hostebene sind also ähnlich wie auf der Ebene einzelner Seiten. Verglichen mit Tabelle 3.6 fallen nur zwei Unterschiede auf:

- Bei BFS-Strategie ist der Anteil von *OUT* auf Hostebene größer als auf Seitenebene.

  Das erscheint zunächst paradox, weil die Zusammenfassung von Knoten in der oben genannten Art eine „Verdichtung“ des Graphen darstellt – aus einer starken Zusammenhangskomponente wird durch Zusammenfassung wieder eine starke Zusammenhangskomponente. Trotzdem ist *SCC* relativ zu *OUT* kleiner als auf Seitenebene.

[16] Das Setzen von Links auf tiefer liegende Seiten, sogenanntes *deep linking*, wird oft nicht gerne gesehen und hat sogar schon zu Rechtsstreitigkeiten geführt [Kap00].

Diese Situation ist so zu erklären, dass die Seiten in *OUT* sich auf mehr verschiedene Hosts verteilen als die in *SCC*. Dadurch können weniger Hosts in *SCC* als in *OUT* sein, obwohl das Verhältnis auf Seitenebene andersherum ist.

- Auf Hostebene liegen 93.14% (BFS) bzw. 98.90% (fok. Str.) der Knoten in $SCC \cup OUT$; auf Seitenebene sind es nur 88.39% bzw. 94.88%.

  Das ist ein Indiz dafür, dass die „abweichenden“ Seiten, die nicht in *SCC* liegen oder von *SCC* aus erreicht werden können, sich auf relativ wenigen Hosts befinden.

Struktur der Informatik-Community im Web

Aus den oben dokumentierten Ergebnissen lassen sich einige Eigenschaften der betrachteten Graphen ableiten. (In Klammern stehen die entsprechenden Nummern der Beobachtungen bei Variante 1.)

- Beim Graphen der BFS-Strategie ist *SCC* im Verhältnis deutlich größer als im gesamten Web (1a). Selbst bei der fokussierten Strategie ist der Anteil von *SCC* am gesamten Graphen noch vergleichbar groß wie im gesamten Web (2a), obwohl diese

	Strategie			
Menge	**BFS**		**fokussiert**	
SCC	6863	45.26 %	14277	28.53 %
IN	461	3.04 %	145	0.29 %
OUT	7260	47.88 %	35171	70.27 %
TUBES	40	0.26 %	11	0.02 %
INTENDRILS	422	2.78 %	158	0.32 %
OUTTENDRILS	23	0.15 %	40	0.08 %
TENDRILS gesamt	485	3.20 %	209	0.42 %
DISC	95	0.63 %	248	0.63 %
Summe	15164	100.00 %	50050	100.00 %

Tabelle 3.9: Bowtie-Struktur bei Betrachtung auf Hostebene

Strategie nur einen Bruchteil der Seiten in jeder Tiefe liest. (Siehe auch Abschnitt 12 zur Struktur der Graphen, die die beiden Strategien ergeben.)

- Die Mengen *IN* sind jeweils extrem klein (1b, 2b). Die Startseiten befinden sich also nah an *SCC* oder liegen in *SCC*.

  Im Vergleich zu der Betrachtung des gesamten Web in [Bro00] ist die Menge *IN* hier viel kleiner. Das lässt sich dadurch begründen, dass Broder u. a. eine Gesamtsicht auf das Web untersucht haben; im Gegensatz dazu wird mit den hier verwendeten Strategien aus Abschnitt 3.1 von wenigen Informatik-relevanten Startseiten ausgegangen, die nahe an *SCC* oder in *SCC* liegen.

- Es gibt nur wenige *TENDRILS* (1c, 2c). Also führen die meisten Wege aus den Startseiten nach *SCC*.

- *DISC* ist jeweils sehr klein (1d, 2d). Die von den jeweiligen Startknoten aus traversierten Graphen sind also fast alle untereinander schwach zusammenhängend.

- Beschränkt man sich bei der fokussierten Strategie auf Seiten, die der Textklassifizierer als relevant eingestuft hat (3a), so fallen lediglich Seiten aus *OUT* weg. Das ist ein zusätzliches Indiz dafür, dass es sich bei *SCC* tatsächlich um eine thematisch zusammenhängende Menge von Seiten handelt, die sich mit Informatik befasst.

- Die Beschränkung der fokussierten Strategie auf Tiefen 0–2 führt ähnlich wie das Entfernen Site-interner Links zu einem Graphen mit weniger starkem Zusammenhang.

  Wie im Abschnitt 12 auf Seite 25 erläutert, liest die fokussierte Strategie schon bei Tiefe 2 nur noch einen Bruchteil der Seiten, die die BFS-Strategie liest. Damit ist der geringere Zusammenhang zu erklären.

  Bemerkenswert ist dabei, dass die fokussierte Strategie bei unbeschränkter Tiefe dennoch einen Graphen mit relativ großem *SCC* ermittelt. Dies ist ein Anzeichen dafür, dass die gefundenen Seiten thematisch zusammenhängen, obwohl bei größeren Tiefen nur ein sehr kleiner Anteil der erreichbaren Seiten betrachtet wird.

Um diese Punkte zu verdeutlichen, werden in Tabelle 3.10 nochmals die Ergebnisse aus [Bro00] denen aus Variante 1 gegenüber gestellt. Es werden die Daten des BFS-Laufs aufgeführt, da diese am ehesten der Vorgehensweise des Crawls entsprechen, der die Daten aus [Bro00] gewonnen hat.

Menge	**Anteil**	
	gesamtes Web nach [Bro00]	Graph der BFS-Strategie
SCC	27.74 %	57.82 %
IN	21.29 %	6.60 %
OUT	21.21 %	30.57 %
TENDRILS	21.52 %	4.29 %
DISC	8.24 %	0.72 %

Tabelle 3.10: Vergleich: Bowtie-Struktur auf dem gesamten Web und auf dem Graphen der BFS-Strategie

Insgesamt kann also nachgewiesen werden, dass die ausgewählten Startseiten von Informatik-Institutionen (s. 3.1) fast ausnahmslos nahe oder in einer starken Zusammenhangskomponente aus Informatik-relevanten Seiten liegen. Diese ist auf dem Graphen der BFS-Strategie im Verhältnis mehr als doppelt so groß wie *SCC* im gesamten Web.

Die fokussierte Strategie, die in Tiefen größer als zwei nur einen Bruchteil der Knoten in jeder Tiefe gelesen hat, hat immer noch ein *SCC*, das vergleichsweise groß ist. Zusammen mit den Beobachtungen am Graphen der Breitensuche läßt sich ableiten, dass die Seiten der betrachteten Informatik-verwandten Institutionen deutlich dichter verbunden sind als das gesamte Web.

3.4 Dichte Subgraphen im WWW - Eine Familie von Algorithmen

Ein Hyperlink (u, v) im WWW bedeutet, dass der Verfasser von Seite u die Seite v zum weiteren Lesen vorschlägt. Daher ist es ein Indiz für eine inhaltliche Verwandtschaft, wenn in einem Subgraphen des WWW besonders viele Kanten vorkommen.

Dieser Abschnitt stellt eine Familie von Algorithmen vor, die zum Auffinden solcher Subgraphen mit vielen Kanten geeignet sind. Diese Graphen werden *dicht* (*dense*) genannt (s. z. B. [Fei97]).

Die einfachste Form eines dichten Subgraphen ist eine *Clique*. Eine Clique ist dabei ein Teilgraph $G' = (V', E')$ von G, in dem *alle* möglichen Kanten vorkommen, d. h. für alle u, v aus V' ist die Kante (u, v) in E' enthalten:

$$\begin{aligned} V' &\subseteq V \\ E' &= (V' \times V') \backslash \{(v, v) : v \in V\} \end{aligned}$$

Eine abgeschwächte Forderung für dichte Teilgraphen ist, dass *viele* der möglichen Kanten vorkommen sollen. Die *Dichte* eines Teilgraphen $G' = (V', E')$ wird dazu definiert als der Anteil der möglichen Kanten zwischen verschiedenen Knoten, die im Graphen vorkommen. Ein Graph mit nur einem Knoten hat per Definition die Dichte 1.

$$\begin{aligned} D(G') &= \begin{cases} \frac{|E'|}{|(V' \times V') \backslash \{(v,v) : v \in V'\}|} & \text{falls } |V'| > 1 \\ 1 & \text{falls } |V'| = 1 \end{cases} \\ &= \begin{cases} \frac{|E'|}{|V'| \cdot (|V'| - 1)} & \text{falls } |V'| > 1 \\ 1 & \text{falls } |V'| = 1 \end{cases} \end{aligned}$$

Eine Clique hat nach dieser Definition Dichte 1, da alle möglichen Kanten vorkommen..

Das Auffinden von Cliquen oder dichter Subgraphen in Graphen ist NP-schwer [Fei97]. Damit sind diese Probleme für einen Graphen mit vielen tausend Knoten praktisch nicht lösbar, da nach bisherigem Wissen nur Algorithmen mit exponentieller Laufzeit existieren.[17]

Für diese Arbeit wird eine Reihe von Algorithmen zum Auffinden von dichten Subgraphen entwickelt. Diese benutzen eine Greedy-Heuristik, die in einem Graphen dichte Subgraphen

[17] Falls $P \neq NP$ gilt, wie bisher allgemein angenommen, so kann es zumindest keine Algorithmen mit polynomieller Laufzeit geben.

ermitteln kann, aber nicht notwendig optimale Lösungen findet.

Algorithmen werden *greedy* (engl. gierig) genannt, wenn sie durch Verfolgen einer jeweils lokal besten Teillösung versuchen, ein globales Optimum zu erreichen.

In den folgenden Abschnitten werden drei Varianten dieser Algorithmen vorgestellt.

3.4.1 Greedy-Algorithmus zum Auffinden gerichteter Cliquen

Der Algorithmus für Cliquen startet für jeden Knoten $v \in V$ mit der Menge $M = \{v\}$. Diese Einer-Clique M wird so oft wie möglich zu einer größeren Clique erweitert. Dazu werden nacheinander alle mit M benachbarten Knoten w getestet; ist w Vorgänger und Nachfolger aller Knoten aus M, so kann w zur Clique hinzugenommen werden.

Um den Vorgang zu beschleunigen, werden durch die erste Schleife von Algorithmus 3 nur Knoten mit hinreichend großem Grad als Kandidaten für eine Erweiterung betrachtet; siehe dazu Abschnitt 3.4.5.

Algorithmen 2 und 3 zeigen den gesamten Ablauf.

Algorithmus 2: greedy-clique

for all $v \in V$ **do**
  $M = \{v\}$
  grow(M)
  gebe M aus
end for

Es ist leicht einzusehen, dass die ausgegebenen Mengen stets Cliquen sind – „M ist Clique“ ist eine Invariante von *grow*.

Abbildung 3.17 zeigt den Ablauf eines Wachstumsschrittes.

Algorithmus 3: grow (M)

{C: Kandidatenmenge}
$C \leftarrow \emptyset$
for all $w \in M$ **do**
  for all $(v, w) \in E \vee (w, v) \in E$ **do**
    {Kandidat ist jeder Nachbarknoten v mit hinreichend großem In- und Ausgrad}
    if $(v \notin M) \wedge (indeg(v) \geq |M|) \wedge (outdeg(v) \geq |M|)$ **then**
      $C \leftarrow C \cup \{v\}$
    end if
  end for
end for
for all $v \in C$ **do**
  {Kandidat v wird hinzugefügt, falls alle $w \in M$ Vorgänger und Nachfolger von v sind.}
  Pred $= \{w \,|\, w \rightarrow v\}$
  Succ $= \{w \,|\, v \rightarrow w\}$
  if $(M \subseteq$ *Pred*$) \wedge (M \subseteq$ *Succ*$)$ **then**
    $M \leftarrow M \cup \{v\}$
  end if
end for

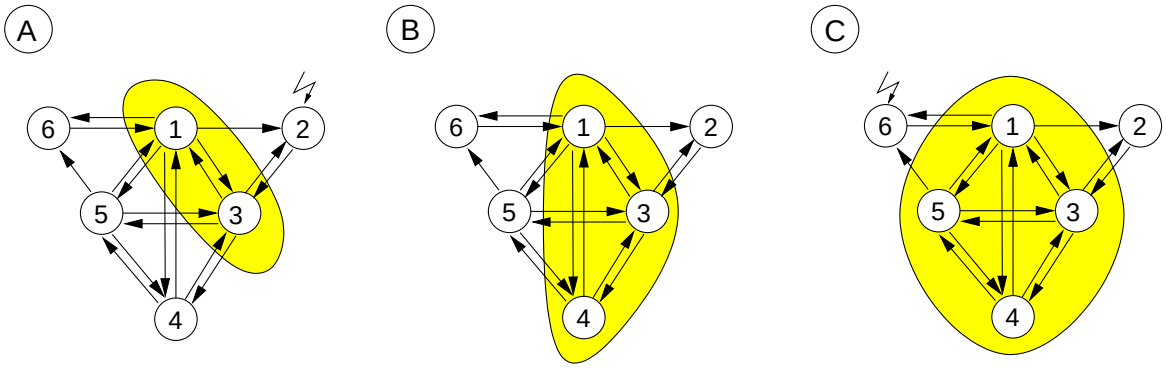


Abbildung 3.17: Wachstumsschritt im Greedy-Algorithmus für Cliquen

Abbildung	**Kandidat**	**Geprüfte Kanten**	**Menge** M
Startmenge			$\{1\}$
A	2	$1 \rightarrow 2, 2 \nrightarrow 1$ ↯	$\{1\}$
	3	$1 \leftrightarrow 3$ ✓	$\{1, 3\}$
B	4	$1 \leftrightarrow 4, 3 \leftrightarrow 4$ ✓	$\{1, 3, 4\}$
C	5	$1 \leftrightarrow 5, 3 \leftrightarrow 5, 4 \leftrightarrow 5$ ✓	$\{1, 3, 4, 5\}$
	6	$1 \leftrightarrow 6, 5 \rightarrow 6, 6 \nrightarrow 5$ ↯	$\{1, 3, 4, 5\}$

Start: Die Startmenge besteht aus dem Knoten 1. Es werden nacheinander alle Nachbarn von 1 abgearbeitet.

A. Knoten 2 wird nicht hinzugenommen, weil die Kante $2 \rightarrow 1$ fehlt. Knoten 3 wird hinzugenommen.

B. Knoten 4 wird hinzugenommen.

C. Knoten 5 wird hinzugenommen, Knoten 6 nicht, da u. a. die Kante $6 \rightarrow 5$ fehlt.

Allerdings garantiert der Algorithmus nicht, dass auch alle Cliquen gefunden werden. Abbildung 3.18 auf der nächsten Seite zeigt ein Gegenbeispiel.

A. Ausgehend von einem der äußeren Knoten werden zwei Nachbarn gefunden, die mit diesem zusammen eine Dreier-Clique bilden (Schritte 1, 2)

B. Diese Dreierclique kann nicht mit Nachbarknoten erweitert werden. Die innere Schleife von Algorithmus 2 terminiert.

C. Von einem der inneren Knoten werden zwei zulässige Nachbarn gefunden, die mit dem Startknoten eine Dreier-Clique bilden (Schritte 3, 4). Alle weiteren Nachbarn (5, 6, 7) werden nicht zur Clique hinzugenommen, obwohl der Startknoten mit zweien von ihnen Teil einer Vierer-Clique ist.

D. Damit ist eine weitere Clique gefunden worden, die nicht erweitert werden kann.

Analog gibt es von jedem Knoten aus eine Bearbeitungsreihenfolge, so dass die innere Vierer-Clique nicht gefunden wird.

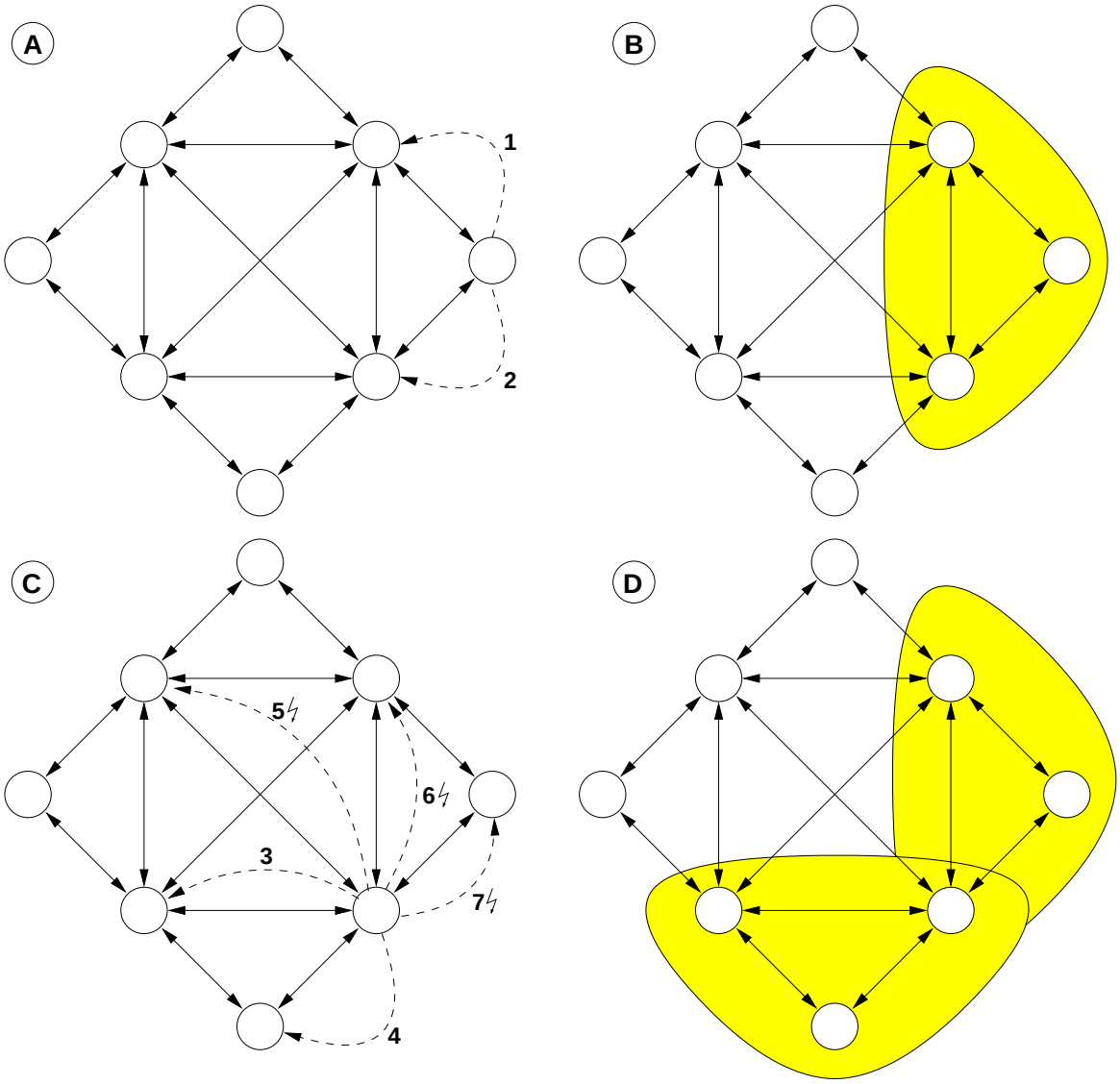


Abbildung 3.18: Beispiel: Versagen des Greedy-Algorithmus

Laufzeit des Algorithmus' in einem dünnen Graphen

Die Prozedur *grow(M)* arbeitet alle Nachbarn $N(M) = \{w | \exists v \in M : (w,v) \in E \vee (v,w) \in E\}$ der bereits gefundenen Menge M ab, benötigt also $|M| \cdot |N(M)| = O(|V|^2)$ Schritte.

greedy-clique ruft für jeden der $|V|$ Knoten möglicherweise $|V|$ mal *grow* auf. Insgesamt ist die worst-case-Laufzeit also $O(|V|^4)$.

In einem dünn besetzten Graphen mit $|E| = O(|V|)$ hat allerdings jeder Knoten im Mittel nur eine konstante Zahl c_N von Nachbarn. Ebenso können die gefundenen Cliquen im Mittel eine konstante Größe c_M nicht überschreiten.

Damit läuft *grow* in Zeit $O(c_M) = O(1)$, da $|M|$ im Mittel nur eine konstante Größe haben kann.

Damit hat man eine Laufzeit von

#Anfangsmengen·#Aufrufe von grow·#Nachbarn·Größe der Clique $= O(|V| \cdot c_M \cdot c_N \cdot c_M) = O(|V|)$

Im Falle eines dünn besetzten Graphen ist also eine lineare Laufzeit zu erwarten.

Die betrachteten Graphen sind dünn besetzt, wie die Tabelle 3.11 an einigen Beispielen zeigt. (Es werden nur die besuchten Seiten und Links innerhalb der Menge der besuchten Seiten betrachtet.)

Graph	**Knoten**	**Kanten**	**max. Kanten**	**Dichte**
BFS, alle Links	178986	1855775	$3.2 \cdot 10^{10}$	$5.8 \cdot 10^{-5}$
BFS, Site-übergreifende Links	178986	169815	$3.2 \cdot 10^{10}$	$5.3 \cdot 10^{-6}$
fok. Str., alle Links	200382	1048697	$4 \cdot 10^{10}$	$2.6 \cdot 10^{-5}$
fok. Str, Site-übergreifende Links	200382	255339	$4 \cdot 10^{10}$	$6.4 \cdot 10^{-6}$

Tabelle 3.11: Dichte der betrachteten Graphen

3.4.2 Greedy-Algorithmus zum Auffinden dichter Subgraphen

Mit einer Abschwächung kann der obige Algorithmus *grow* so angepasst werden, dass er nicht nur echte Cliquen findet, sondern auch Subgraphen mit einer Dichte $D(G)$, die über einem gewissen Schwellwert t liegt (im Folgenden *Cluster* genannt).

Dazu wird im Wachstumsschritt nicht mehr gefordert, dass der hinzuzufügende Knoten Vorgänger und Nachfolger *aller* bisher gefundenen Knoten von M ist; stattdessen soll jeweils eine Teilmenge M' von M ausreichen mit $|M'| \geq t \cdot |M|$, wobei $0 \leq t \leq 1$.

Algorithmus 4 zeigt den geänderten Ablauf beim Wachstumsschritt. Auch der aufrufende Algorithmus 5 hat sich geändert. Der Wachstumsschritt beim Algorithmus für die Clique kann nur ein Mal erfolgreich sein: alle Knoten, die in die Clique aufgenommen werden, sind schon Nachbarn des Startknotens. Im Gegensatz dazu kann beim Wachstumsschritt für einen dichten Subgraphen ein Knoten hinzukommen, dessen Nachbarn auch Kandidaten sein können. Daher kann $grow-with-threshold(M)$ mehrmals zu einem Wachstum von M führen.

Algorithmus 4: grow-with-threshold (M)

{C: Kandidatenmenge}
$C \leftarrow \emptyset$
$t = \mathit{threshold}(t_{min}, t_{max}, \mathit{decay}, |M|)$
for all $w \in M$ **do**
  for all $(v, w) \in E \vee (w, v) \in E$ **do**
    {Kandidat ist jeder Nachbarknoten v mit hinreichend großem In- und Ausgrad}
    if ($v \notin M$) $\wedge$ $(indeg(v) \geq t \cdot |M|) \wedge (outdeg(v) \geq t \cdot |M|)$ **then**
      $C \leftarrow C \cup \{v\}$
    end if
  end for
end for
for all $v \in C$ **do**
  {Kandidat v wird hinzugefügt, wenn hinreichend viele Vorgänger und Nachfolger von v bereits Mitglieder sind.}
  $\mathit{Pred} = \{w \mid w \rightarrow v\}$
  $\mathit{Succ} = \{w \mid v \rightarrow w\}$
  if $(|\mathit{Pred} \cap M|/|M| \geq t) \wedge (|\mathit{Succ} \cap M|/|M| \geq t)$ **then**
    $M \leftarrow M \cup \{v\}$
  end if
end for

Der Schwellwert t wird dabei der Größe des wachsenden Clusters angepasst. Wäre t fest, so wäre es am Anfang zu leicht, einen Knoten zu finden, der mit $t \cdot |M|$ vielen Knoten benachbart ist, aber mit wachsendem $|M|$ würde es immer schwieriger.

Algorithmus 5: greedy-cluster

for all $v \in V$ **do**
  $M = \{v\}$
  repeat
    grow(M)
  until M nicht gewachsen
  gebe M aus
end for

Um dies auszugleichen, wird t mittels einer Funktion *threshold* zwischen Grenzwerten $t_{min} \leq t \leq t_{max}$ angepasst:

$$\textit{threshold}(t_{min}, t_{max}, \textit{decay}, |M|) = t_{min} + (t_{max} - t_{min}) \cdot \textit{decay}^{(|M|^2)}$$

decay $\in [0, 1]$ gibt dabei an, wie schnell der Funktionswert abfällt; für Werte nahe 0 fällt er schneller, für *decay* nahe 1 langsamer. Abbildung 3.19 auf der nächsten Seite zeigt ein Beispiel für $t_{min} = 0.4, t_{max} = 0.8$ und verschiedene Werte für *decay*.

Beispiel: Abbildung 3.20 zeigt einen Berechnungsschritt. Es seien $M = \{1, 2, 3\}$ die bereits gefundenen Knoten, Knoten 4 ist der Kandidatenknoten. Die Vorgänger von 4 sind $Pred = \{1, 3\}$, die Nachfolger $Succ = \{1, 2\}$. Damit gilt also

$$\frac{|Pred \cap M|}{|M|} = \frac{2}{3} \quad \text{und} \quad \frac{|Succ \cap M|}{|M|} = \frac{2}{3}$$

Knoten 4 wird also zum Cluster hinzugenommen, falls $t \leq \frac{2}{3}$ ist.

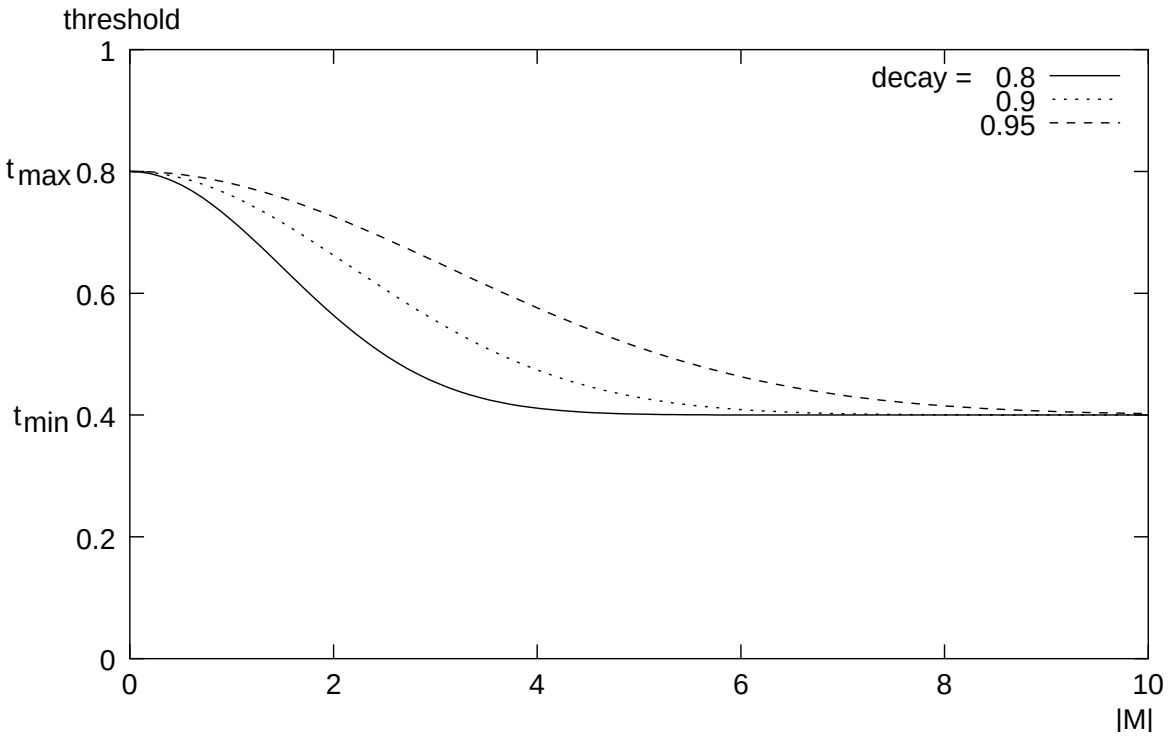


Abbildung 3.19: Graph der *threshold*-Funktion

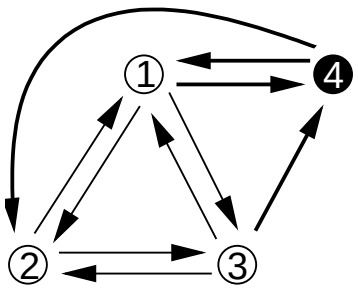


Abbildung 3.20: Beispiel: Greedy-Algorithmus für Cluster

3.4.3 Greedy-Algorithmus zur Annäherung bipartiter Cores

Eine weitere Variante der vorgestellten Algorithmen kann benutzt werden, um Strukturen zu finden, die den sogenannten *bipartiten Cores* [Kum99] ähneln. Ein bipartiter Core ist dabei ein vollständiger bipartiter gerichteter Subgraph[18]; alle Kanten gehen von einer Menge sogenannter *Fans* in eine Menge sogenannter *Centers*. In der Terminologie von Kleinberg [Kle99] (s. auch Abschnitt 3.6) sind dies die *Hubs* und *Authorities*. Abbildung 3.21 auf der nächsten Seite zeigt ein Beispiel.

Auch hier kann man die Forderung abschwächen, dass *alle* Kanten zwischen den Fans und Centers vorkommen müssen. Man verlangt also wieder, dass dieser bipartite Subgraph eine bestimmte Dichte hat, d. h. dass ein gewisser Anteil der möglichen Kanten zwischen den beiden Fans und Centers in diesem bipartiten Subgraphen vorhanden ist.

Die Dichte ist für diese bipartiten Graphen anders definiert als am Anfang des Abschnitts 3.4. In einem bipartiten Graphen bestehend aus Fans F und Centers C kann es maximal $|F| \cdot |C|$ Kanten geben, die zwischen Fans und Centers verlaufen. Die Dichte in einem bipartiten Subraphen $G' = (V', E')$ mit

$$\begin{aligned} V' &= F \cup C \\ E' &= \{(u, v) \in E : u \in F \vee v \in C\} \end{aligned}$$

sei wiederum definiert als der Anteil der möglichen Kanten, die im Graphen vorhanden sind:

$$D_{bip}(G') = \frac{|E'|}{|F| \cdot |C|}$$

[18] Im Folgenden werden nur Kanten *zwischen* den beiden Mengen der Bipartition betrachtet.
Die Kanten *innerhalb* einer der beiden Mengen werden hier nicht ausgewertet; ansonsten wäre ein „bipartiter Core“ nicht unbedingt bipartit im graphentheoretischen Sinne.

Algorithmus 6 geht ähnlich vor wie Algorithmus 4. Allerdings benutzt man nun zwei Mengen *Fans* und *Centers* von Knoten, die bereits gefunden wurden. Ein neuer Knoten ist ein Kandidat für die Fans (Centers), wenn hinreichend viele Centers (Fans) mit ihm verbunden sind. Ein Kandidat wird als Fan hinzugenommen, wenn *threshold* $\cdot$ |*centers*| viele Kanten aus *centers* zu ihm führen und umgekehrt.

Abbildung 3.21 illustriert dieses Vorgehen. Die weißen Knoten seien eine bereits gefundene Core-ähnliche Struktur. Der schwarze Knoten ist ein Kandidat für ein neues Center. Er hat drei der vier Fans als Vorgänger. Damit wird er zur Struktur hinzugefügt, falls der Schwellwert *threshold* kleiner oder gleich $3/4$ ist.

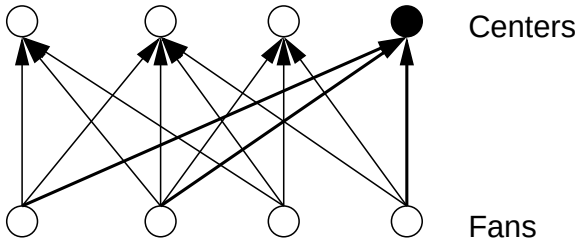


Abbildung 3.21: Beispiel: Greedy-Algorithmus für Bipartite Cluster

3.4.4 Korrektheit der Algorithmen

Es muss noch bewiesen werden, dass die Graphen, die die vorgestellten Algorithmen *grow-with-threshold* (Algorithmus 4) und *grow-bipartite* (Algorithmus 6) finden, wirklich eine Dichte größer als t_{min} aufweisen.

Es wird gezeigt: für alle Cluster M, die ein Wachstumsschritt erzeugt, gilt $D(M) \geq t$ bzw. $D_{bip}(M) \geq t$. t liegt dabei in $[t_{min}, t_{max}]$, so dass speziell eine Dichte größer oder gleich t_{min} nachgewiesen wird.

Beweis für die Korrektheit von Algorithmus 4 (Cliquen und dichte Subgraphen):

Der Beweis erfolgt per Induktion über die Knotenzahl $|M|$ des Clusters M mit Kantenmenge E'. Da der Fall $|M| = 1$ einen Sonderfall darstellt (s. Anfang von Abschnitt 3.4), beginnt die Induktion mit $|M| = 2$.

Algorithmus 6: grow-bipartite (M)

$Fan\_Cand \leftarrow \emptyset$
$Center\_Cand \leftarrow \emptyset$
$t = \textit{threshold}(t_{min}, t_{max}, \textit{decay}, (|Fans| + |Centers|)/2)$
{Kandidaten für Centers sammeln}
for all $v \in Fans$ **do**
  for all $\{w|(v,w) \in E\}$ **do**
    if $(w \notin Centers \wedge indeg(w) \geq t \cdot |Fans|)$ **then**
      $Center\_Cand = Center\_Cand \cup \{w\}$
    end if
  end for
end for
{Kandidaten für Fans sammeln}
for all $v \in Centers$ **do**
  for all $\{w|(w,v) \in E\}$ **do**
    if $(w \notin Fans \wedge outdeg(w) \geq t \cdot |Centers|)$ **then**
      $Fan\_Cand = Fan\_Cand \cup \{w\}$
    end if
  end for
end for
{Abwechselnd versuchen, Fan- und Center-Kandidaten hinzuzunehmen}
while $Fan\_Cand \neq \emptyset \vee Center\_Cand \neq \emptyset$ **do**
  if $Fan\_Cand \neq \emptyset$ **then**
    $t = \textit{threshold}(t_{min}, t_{max}, \textit{decay}, (|Fans| + |Centers|)/2)$
    wähle f aus Fan_Cand
    $Fan\_Cand \leftarrow Fan\_Cand \backslash \{f\}$
    $Succ = \{w \in Centers|(f,w) \in E\}$
    if $|Succ \cup Centers| \geq t \cdot |Centers|$ **then**
      $Fans \leftarrow Fans \cup \{f\}$
    end if
  end if
  if $Center\_Cand \neq \emptyset$ **then**
    $t = \textit{threshold}(t_{min}, t_{max}, \textit{decay}, (|Fans| + |Centers|)/2)$
    wähle c aus $Center\_Cand$
    $Center\_Cand \leftarrow Center\_Cand \backslash \{c\}$
    $Pred = \{w \in Fans|(w,c) \in E\}$
    if $|Pred \cup Fans| \geq t \cdot |Fans|$ **then**
      $Centers \leftarrow Centers \cup \{f\}$
    end if
  end if
end while

Induktionsanfang: $|M| = 2$

Der zweite Knoten, der zu einem Cluster M hinzugenommen wird, muss den ersten Knoten als Vorgänger und Nachfolger haben. Damit muss der Cluster zwei Kanten beinhalten.

Die Dichte ist also $D(M) = \frac{|E'|}{|M|\,(|M|-1)} = \frac{2}{2\cdot 1} = 1 \geq t$

Induktionsschritt: $|M| \rightarrow |M| + 1$

Der Cluster vor dem Wachstumsschritt habe die Dichte

$$D^{alt}(M) = \frac{|E'|}{|M|(|M|-1)} \geq t$$

Daraus folgt

$$|E'| \geq t\,|M|\,(|M|-1) \tag{3.3}$$

Nach dem Wachstumsschritt ist $|M|$ um eins gewachsen. Da der neue Knoten mindestens $t\,|M|$ Vorgänger und $t\,|M|$ Nachfolger in M hat, ist die Kantenzahl um mindestens $2\,t\,|M|$ gewachsen.

Für die neue Dichte gilt also:

$$\begin{aligned}
D^{neu}(M) &= \frac{|E'| + 2\,t\,|M|}{|M|(|M|+1)} \\
&\geq \frac{t(|M|-1)|M| + 2\,t\,|M|}{|M|(|M|+1)} \qquad \text{(nach Gleichung (3.3))} \\
&= t\,\frac{|M|^2 - |M| + 2\,|M|}{|M|(|M|+1)} \\
&= t\,\frac{|M|^2 + |M|}{|M|^2 + |M|} \\
&= t
\end{aligned}$$

also insgesamt

$$D^{neu}(M) \geq t$$

Damit ist gezeigt, dass die resultierenden Cluster stets eine Dichte größer als oder gleich t haben.

Beweis für die Korrektheit von Algorithmus 6 (bipartite Cluster):

Der Beweis erfolgt wieder per Induktion über die Knotenzahl $|M|$ des bipartiten Clusters M mit Kantenmenge E'. Die Induktion startet mit $|M| = 2$, da ein bipartiter Cluster mindestens einen Fan und ein Center beinhaltet, die mit einer Kante verbunden sind. Die Menge der Fans sei F, die der Centers C.

Induktionsanfang: $|M| = 2$

Ein bipartiter Cluster bestehend aus zwei Knoten und einer Kante hat die Dichte 1:

$$D_{bip}(M) = \frac{|E'|}{|F|\,|C|} = \frac{1}{1 \cdot 1} = 1$$

Induktionsschritt: $|M| \rightarrow |M| + 1$

M kann wachsen, indem ein Fan oder ein Center hinzukommt. Hier wird der Fall betrachtet, dass ein Fan hinzugenommen wird. Der andere Fall ist symmetrisch.

Für die Dichte vor dem Wachstumsschritt gilt:

$$\begin{aligned} D_{bip}^{alt}(M) &= \frac{|E'|}{|F|\,|C|} \\ &\geq t \end{aligned}$$

Daraus folgt

$$|E'| \geq t\,|F|\,|C| \tag{3.4}$$

Ein neuer Fan kommt hinzu ($|F| \rightarrow |F|+1$ und damit $|M| \rightarrow |M|+1$), der mit mindestens $t\,|C|$ Centers verbunden ist. Also

$$\begin{aligned} D_{bip}^{neu} &= \frac{|E'| + t\,|C|}{(|F|+1)\,|C|} \\ &\geq \frac{t\,|F|\,|C| + t\,|C|}{|F|\,|C| + |C|} \qquad \text{(nach Gleichung (3.4))} \\ &= t\,\frac{|F|\,|C| + |C|}{|F|\,|C| + |C|} \\ &= t \end{aligned}$$

Damit ist auch für den Fall der bipartiten Cluster gezeigt, dass die Dichte stets größer oder gleich t ist.

3.4.5 Vorverarbeitung des Graphen

Um den Ablauf der oben genannten Algorithmen zu beschleunigen, kann eine Vorverarbeitung des Graphen erfolgen. Dabei werden solche Knoten entfernt, die auf keinen Fall zu

einer dichten Struktur gehören werden. Kumar u. a. [Kum99] nennen ein solches Vorgehen „Zurückschneiden“ (engl. *pruning*).

Im Fall der Cliquen können beispielsweise alle Knoten aus dem Graphen entfernt werden, deren In- oder Ausgrad kleiner ist als eine geforderte Mindestgröße für die Cliquen.

Sind lediglich dichte (bipartite) Subgraphen gesucht, wird eine Heuristik benutzt:

Dichte Subgraphen Es wird eine Mindestgröße k vorgegeben, die ein dichter Subgraph erreichen muss, um ausgegeben zu werden.

Bei der Vorverarbeitung werden alle Knoten entfernt, deren In- oder Ausgrad kleiner als $k \cdot t_{min}$ ist.

Bipartite Cluster Es wird eine Mindestanzahl f von Fans und eine Mindestanzahl c von Centers vorgegeben, die ein bipartiter Cluster erreichen muss, um ausgegeben zu werden.

Bei der Vorverarbeitung werden alle Knoten entfernt, deren Ingrad kleiner als als $f \cdot t_{min}$ und deren Ausgrad kleiner als $c \cdot t_{min}$ ist.

Da beim Entfernen solcher Knoten auch die adjazenten Kanten wegfallen, sinkt auch der Grad anderer Knoten möglicherweise unter den Schwellwert. Man kann das Zurückschneiden so lange wiederholen, bis ein Fixpunkt erreicht ist.

Dieses Vorgehen ist nur eine Heuristik: in einem Cluster der Dichte t mit m Knoten können durchaus auch Knoten mit einem In- oder Ausgrad kleiner als $t \cdot m$ enthalten sein; diese würden bei der Vorverarbeitung aber entfernt.

Andererseits bewirkt diese Vorverarbeitung, dass die gefundenen Cluster in einem gewissen Sinne „gleichmäßig dicht“ sind, da alle Knoten einen In- und Ausgrad größer als die untere Schranke haben.

3.4.6 Probleme dieser Algorithmen

Die drei vorgestellten Algorithmen haben auf Grund ihrer Vorgehensweise das Problem, dass sie mitunter gleiche oder ähnliche Cliquen oder Cluster mehrfach ausgeben. (Da dieses Problem und seine Lösungsansätze für alle drei Algorithmen gleich sind, wird im Folgenden einheitlich der Begriff „Cluster“ verwendet.)

Auffinden eines Clusters von verschiedenen Startknoten aus

Wie Abbildung 3.22 zeigt, kann ein Cluster M von jedem enthaltenen Knoten aus gefunden werden. Dadurch wird M im ungünstigsten Fall $|M|$ mal ausgegeben.

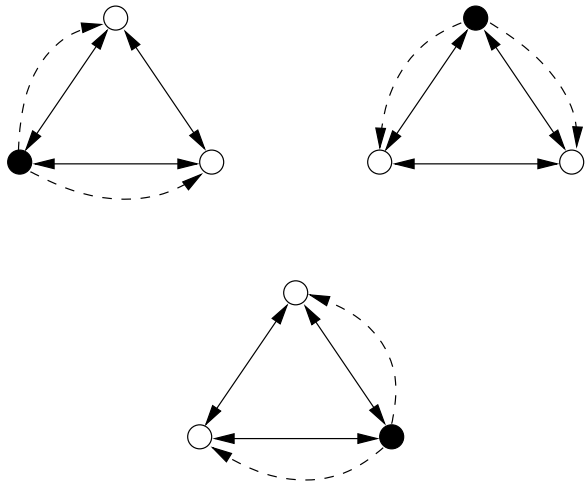


Abbildung 3.22: Verschiedene Startknoten für den selben Cluster

Auffinden ähnlicher Cluster

Eine Verallgemeinerung des vorgenannten Problems ist die Beobachtung, dass oft ähnliche Cluster gefunden werden.

Die *Ähnlichkeit* zweier nichtleerer Knotenmengen M_1, M_2 sei dabei definiert als

$$\text{Sim}(M_1, M_2) := \frac{|M_1 \cap M_2|}{\min(|M_1|, |M_2|)} \in [0, 1]$$

M_1 und M_2 sind also ähnlich, wenn sie viele gemeinsame Knoten haben.

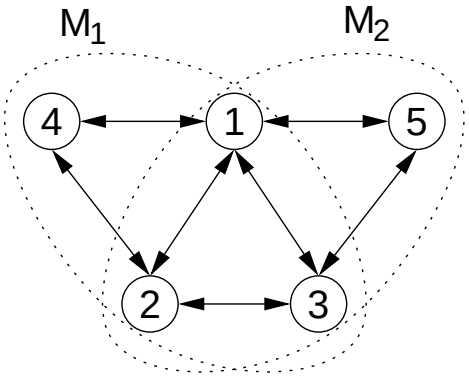


Abbildung 3.23: Auffinden ähnlicher Cluster

Abbildung 3.23 illustriert, wie es zu solchen ähnlichen Clustern kommen kann:

Sei die Schranke t aus Algorithmus *grow-with-threshold* (Abschnitt 3.4.2) fest, um die Betrachtung zu vereinfachen, also z. B. $t = t_{min} = t_{max} = 2/3$.

Dann können in Abbildung 3.23 zwei ähnliche Cluster M_1 und M_2 mit $\text{Sim}(M_1, M_2) = 3/4$ zu Stande kommen:

M_1: Bearbeitungsreihenfolge 1-2-3-4-5; Knoten 5 wird nicht hinzugenommen, da er nur mit der Hälfte der Knoten $\{1, 2, 3, 4\}$ verbunden ist.

Präziser: im Wachstumsschritt *grow-with-threshold (M)* von Algorithmus 4 mit $M = \{1, 2, 3, 4\}$ gilt für den Kandidaten $v = 5$:

$$\begin{aligned} M &= \{1, 2, 3, 4\} \\ \textit{Pred}(v) &= \{1, 3\} \\ \textit{Succ}(v) &= \{1, 3\} \end{aligned}$$

$$\frac{|\mathit{Pred}(v) \cap M|}{|M|} = 1/2 < t = 2/3$$
$$\frac{|\mathit{Succ}(v) \cap M|}{|M|} = 1/2 < t = 2/3$$

Damit wird $v = 5$ nicht hinzugenommen und der Algorithmus ist beendet.

M_2: Bearbeitungsreihenfolge 1-2-3-5-4: Knoten 4 wird nicht hinzugenommen (Begründung symmetrisch zu M_1)

In der Praxis kommt diese Situation häufig für große Cluster M_i und M_j vor, die sich nur in wenigen Knoten unterscheiden, so dass Sim(M_i, M_j) sehr nahe 1 ist.

Abhilfen

Um die oben genannten Probleme zu behandeln, können verschiedene Ansätze benutzt werden:

Erkennen bereits betrachteter Mengen durch Hashing Ein Ansatz, um die mehrfache Berechnung des selben Clusters zu vermeiden, ist *Hashing*. Dabei wird eine Hash-Funktion h auf alle Zwischenergebnisse M_j in der Schleife in Algorithmus 5 angewendet. Der erhaltene Wert $h(M_j)$ wird mit allen vorher berechneten $h(M_1), \ldots, h(M_{j-1})$ verglichen. Falls $h(M_j)$ darin schon vorkommt, wird der Algorithmus ohne Ausgabe abgebrochen. Die Hash-Werte sind jeweils nur 16 Byte lang und können problemlos im Hauptspeicher gehalten werden.

In der Praxis wird dazu in den Programmen clust.cc und bip_clust.cc eine MD5-Prüfsumme [Riv92] über alle Knotennummern (entsprechend der id der Webseiten) benutzt. Da es extrem unwahrscheinlich ist, zwei Mengen mit der gleichen MD5-Summe zu erhalten[19], kann davon ausgegangen werden, dass mit dieser Vorgehensweise genau die bereits bekannten Zwischenergebnisse ausgelassen werden.

[19] Um bei einer 128-Bit-Hashfunktion wie MD5 mit einer Wahrscheinlichkeit größer als 1/2 eine Kollision zu erhalten, müssen etwa $1.17 \cdot 2^{64}$ Funktionswerte berechnet werden. Siehe z. B. [Sti95, Abschnitt 7.3]

Mit dieser Vorgehensweise kann allerdings nur die mehrfache Berechnung *gleicher* Cluster vermieden werden.

Vergleich mit allen berechneten Clustern Eine Möglichkeit, um mehrfache Berechnung ähnlicher Cluster zu vermeiden, ist die tatsächliche Berechnung von $\text{Sim}(M_i, M_j)$ für den jeweils aktuell berechneten Cluster M_j und alle vorherigen $M_i, 1 \leq i < j$. Falls die Ähnlichkeit mit einem M_i eine festgelegte Schranke überschreitet, wird M_i durch $M_i \cup M_j$ ersetzt.

Problematisch hierbei ist, dass die Speicherung und der Vergleich mit allen vorherigen Clustern extrem zeit- und speicheraufwändig ist, da für jedes M_i nicht nur ein Hash-Wert, sondern die komplette Menge im Speicher gehalten werden muss.

Zudem kann die Vereinigung zweier sich überschneidender Cluster, die jeweils die vorgegebene Dichte t_{min} überschreiten, eine Dichte kleiner t_{min} haben. Kommt eine solche Vereinigung mehrfach vor, so kann der resultierende Cluster deutlich weniger dicht als die geforderte Mindestdichte t_{min} sein.

Aus diesen beiden Gründen wird dieser Ansatz nicht verwendet.

Eine endgültige Lösung für dieses Problem sich überschneidender ähnlicher Cluster scheint allerdings unmöglich: man kann immer einen Fall wie den auf Seite 65 konstruieren, in dem das Zusammenführen zweier Cluster inhaltlich sinnvoll erscheint, aber die geforderte Mindestdichte unterschreitet.

3.4.7 Experiment: Finden von Clustern und bipartiten Clustern mit den Greedy-Algorithmen

Dieser Abschnitt stellt die Ergebnisse vor, die die oben genannten Greedy-Algorithmen in verschiedenen Durchläufen liefern.

Die dazu benutzten Programme sind clust.cc für den Algorithmus *grow-with-threshold* (Algorithmus 4) und bip_clust.cc für den bipartiten Fall (*grow-bipartite*, Algorithmus 6). Zur Benutzung dieser Programme siehe auch Abschnitt 6.1.2.

Tabelle 3.12 zeigt die Parameter-Kombinationen der Läufe, die durchgeführt werden. Für alle Läufe ist $decay = 0.9$.

Algorithmus	**Mindestgröße**		$\mathbf{t_{min}}$	$\mathbf{t_{max}}$
grow-with-threshold	3		1	1
clust.cc	4		0.7	1
	6		0.2	0.5
	6		0.5	0.8
	8		0.3	0.6
	Fans	Centers		
grow-bipartite	3	3	1	1
bip_clust.cc	6	6	0.2	0.5
	6	6	0.5	0.8
	8	8	0.3	0.6
	8	8	0.7	1

Tabelle 3.12: Parameter für die Läufe der Greedy-Algorithmen

Es werden die gelesenen Seiten der BFS-Strategie, der fokussierten Strategie und der fokussierten Strategie beschränkt auf Seiten mit Relevanz größer als 0.9 betrachtet.

Die Auswertung beschränkt sich auf Site-übergreifende Links (s. 3.2.1) in diesen Seitenmengen, da sonst vor allem dichte Subgraphen *innerhalb* von Sites gefunden werden, wie z. B.

```
http://access.ncsa.uiuc.edu:80/
http://access.ncsa.uiuc.edu:80/Subscribe/
http://access.ncsa.uiuc.edu:80/Funding/
http://access.ncsa.uiuc.edu:80/Events/
http://access.ncsa.uiuc.edu:80/Archive/
```

oder

```
http://postgraduate.cs.uwa.edu.au:80/
```

```
http://research.cs.uwa.edu.au:80/
http://undergraduate.cs.uwa.edu.au:80/
http://www.cs.uwa.edu.au:80/people/
```

Die gesamten Ergebnisse sind sehr umfangreich und können hier nicht im Einzelnen aufgeführt werden. Daher werden hier verschiedene Arten von gefundenen Teilgraphen vorgestellt, die für den jeweiligen Algorithmus typisch sind. Die vollständigen Ausgaben liegen in den Verzeichnissen auswertung/ergebnisse/clust (Cluster) und auswertung/ergebnisse/bip_clust (bipartite Cluster) auf der CD.

Cluster: Algorithmus *grow-with-threshold*

Dieser Algorithmus (Abschnitt 3.4.2) liefert folgende Arten von Clustern:

(In Klammern stehen beispielhaft die Strategie (BFS oder fokussierte Strategie, fS), bei der solch ein Cluster gefunden wurde, die Größe des Clusters sowie seine Dichte.)

Gespiegelte Angebote Besonders große und dichte Cluster entstehen durch akademische Angebote oder Open-Source-Projekte, die vielfach gespiegelt werden, wie z. B.

EMIS (fS/30/1)	European Mathematical Information Service
OpenSSH (fS/35/0.98)	Open Secure Shell
ZMATH (BFS/7/0.71)	Zentralblatt MATH
DBLP (BFS/7/0.45)	Digital Bibliography & Library Project
IPSC (BFS/6/0.7)	Internet Problem Solving Contest
Evonet (fS/4/1)	European Network of Excellence in Evolutionary Computing
RCSB/PDB (fS/4/1)	Protein Data Bank

Organisationen mit mehreren Domainnamen

Sparc Die Organisation Sparc International tritt unter den Adressen `www.sparc.org` und `www.sparc.com` auf, die untereinander zahlreiche Links tragen (BFS/11/0.45).

Embry-Riddle Aeronautical University Diese Universität verteilt ihre Web-Präsenz auf www.erau.edu und www.embryriddle.edu (BFS/10/0.49).

Kommerzielle Anbieter mit mehreren Produkten Viele kommerzielle Anbieter im WWW vermarkten mehrere Produkte, die jeweils unter einer URL entsprechend ihrem Namen präsentiert werden.

Verlag mit verschiedenen Zeitschriften Eine Verlagsgruppe ist mit 8 verschiedenen Zeitschriften (www.tomorrow.de, www.net-business.de usw.) im WWW präsent (fS/8/1).

Entwickler-Portal Das Entwickler-Portal DevX.com bietet auf seiner Site verschiedene Themen wie Java-Entwicklung (www.java-zone.com) oder ein XML-Magazin (www.xmlmag.com) an, die untereinander verbunden sind (fS/10/0.8).

Zusammengewachsenen Webangebote Durch Zusammenwachsen von Webangeboten, die ursprünglich verschiedene Adressen benutzten, können dichte Graphen mit vielen Site-übergreifenden Links entstehen.

Ein Beispiel dafür ist www.allstudents.de, das aus den Angeboten www.studis-online.de, www.studentenpreise.de und www.zvs-opfer.de entstanden ist (fS/19/0.64).

Diesen verschiedenen Arten von Clustern ist gemeinsam, dass sie in gewisser Weise „künstlich“ entstanden sind. Sie sind nicht dadurch gebildet worden, dass verschiedene Autoren aus Interesse an den Dokumenten der jeweils anderen Links auf diese gesetzt haben.

Es werden also keine nennenswerten Mengen von Web-Präsenzen verschiedener Autoren gefunden, die durch Bezugnahme aufeinander eine Clique oder einen dichten Subgraphen bilden.

Bipartite Cluster: Algorithmus *grow-bipartite*

Die Ergebnisse des Algorithmus’ *grow-bipartite* (Abschnitt 3.4.3) lassen sich ebenfalls in verschiedene Klassen einteilen:

(In Klammern stehen wiederum beispielhaft die Strategie (BFS oder fokussierte Strategie, fS), bei der solch ein Cluster gefunden wurde, die Größe des Clusters (x Fans, y Centers) sowie seine Dichte.)

Ähnliche oder gespiegelte Verweislisten: wenige Fans, viele Centers Diese Cluster bestehen aus wenigen Fans, die Verweise auf die gleichen oder ähnliche Mengen von Centers beinhalten.

Dies sind zum einen gespiegelte Linklisten, wie z. B. beim Virtual Museum of Computing (fS/11 F/62 C/0.74), dessen Seiten in vielen ähnlichen Fassungen im Web gespiegelt werden.. Zum anderen gibt es auch Verweislisten, die durch ihre thematische Verwandtschaft die gleichen Ressourcen auflisten. So gibt es z. B. grosse bipartite Cluster mit Verweisen auf allgemeine Informatik-Ressourcen (BFS/10 F/244 C/0.30) oder auf Informatik-Abteilungen von Universitäten (BFS/10 F/241 C/0.30). Diese sind dadurch, dass sie von verschiedenen Autoren stammen, naturgemäß nicht so dicht wie gespiegelte Linklisten.

Viele Verweise auf eine verwandte Menge von Ressourcen: viele Fans, wenige Centers
Zu einem Thema gibt es oft einige wichtige Centers, die von sehr vielen Fans referenziert werden.

Beispiele dafür sind bipartite Cluster, deren Fans die bekannten Suchmaschinen sind (fS/111 F/17 C/0.36), oder Verweise auf die bekannte LAMP[20]-Software aus dem Open-Source-Bereich (BFS/145 F/7 C/0.33). Ein anderer Cluster beschäftigt sich z. B. mit Software Engineering (fS/117 F/11 C/0.40).

Interessant ist, dass Centers, die gemeinsam in einem bipartiten Cluster auftreten, nicht notwendig thematisch verwandt sein müssen.

Ein Beispiel dafür ist ein bipartiter Cluster (BFS/210 F/12 C/0.33), dessen Centers einerseits die wissenschaftlichen Organisationen ACM[21], IEEE[22] und CRA[23] umfassen, andererseits aber auch Seiten über Linux (Redhat/Debian) bzw. FreeBSD. Die Fans dieses Clusters liegen überwiegend auf Seiten universitärer Informatikabteilungen.

[20] LAMP steht für Linux – MySQL – Apache – PHP oder Perl. Diese Kombination wird von vielen Website-Betreibern benutzt. Siehe z. B. http://www.onlamp.com.
[21] Association for Computing Machinery
[22] Institute of Electrical and Electronics Engineers
[23] Computing Research Association

Dies zeigt, dass verschiedene Autoren mehrere gemeinsame Interessengebiete haben können.

Bipartite Teilgraphen von großen Clustern: viele Fans, viele Centers Die großen Cliquen oder Cluster aus Abschnitt 3.4.7 haben entsprechend große bipartite Teilgraphen. Diese besitzen viele Fans *und* viele Centers und sind dichter als die oben genannten bipartiten Cluster.

Ein Beispiel dafür ist ein bipartiter Cluster, der aus Spiegelungen des OpenSSH-Servers besteht (fS/26 F/37 C/0.9).

Insgesamt konnten mit der Suche nach bipartiten Clustern mehr natürlich gewachsene Strukturen im Web nachgewiesen werden als im nicht-bipartiten Fall.

Die Annahme von Kleinberg, die HITS zu Grunde liegt (s. Abschnitt 3.6), wird dadurch unterstützt: Communities im Web gliedern sich oft in Überblicksseiten einerseits und thematisch zusammengehörige Ressourcen andererseits.

3.5 Der PageRank-Algorithmus

3.5.1 Beschreibung des Algorithmus'

Der von Brin und Page vorgestellte Algorithmus *PageRank* [Pag98] basiert auf der Idee, dass wichtige Seiten Links auf andere wichtige Seiten beinhalten.

Dieser Betrachtung liegt das Modell eines Benutzers zu Grunde, der eine Seite besucht und sie über einen zufällig gewählten Link wieder verlässt. Seiten, die häufig besucht werden, lenken also entsprechend viele Besucher zu ihren Nachfolgerseiten.

PageRank wird beispielsweise von der Suchmaschine *Google* benutzt, um unter den gefundenen Seiten die wichtigsten zuerst aufzulisten.

Formal wird die Relevanz einer Seite durch einen Relevanzvektor R ausgedrückt. Dabei

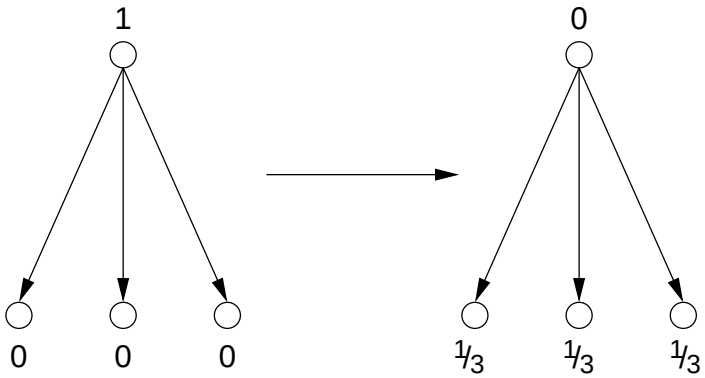


Abbildung 3.24: Verteilung der Gewichte bei Pagerank

sind:

$G(V, E)$	der betrachtete Graph
$R(v)$	Relevanz einer Seite $v \in V$

Um R zu berechnen, wird eine Fixpunktiteration durchgeführt. Dabei wird jeweils das Gewicht jedes Knotens anteilig an alle Nachfolgerknoten verteilt. Abbildung 3.24 verdeutlicht diese Verteilung für den einfachen Fall eines Knotens mit drei Nachfolgern.

Um in jedem Iterationsschritt das Gewicht jedes Knotens v zu berechnen, summiert man über alle eingehenden Kanten:

$$R_{neu}(v) := c \cdot \sum_{u, u \to v} \frac{R_{alt}(u)}{\mathit{outdeg}(u)}$$

In Matrixschreibweise:

$$R_{neu} := c \cdot A \cdot R_{alt} \quad \text{mit} \quad A_{uv} := \begin{cases} 1/\mathit{outdeg}(u) & (u, v) \in E \\ 0 & \text{sonst} \end{cases}$$

c ist dabei ein Normierungsfaktor, so dass $\|R_{neu}\|_1 = 1$ wird. Normiert wird in der L_1-Norm:

$$\|R_{neu}\|_1 := \sum_{v \in V} |R_{neu}(v)|$$

Diese Iteration für den neuen Wert von R wird so lange wiederholt, bis sich R nicht mehr wesentlich ändert, d. h. bis $\|R_{neu} - R_{alt}\|_1 < \varepsilon$ für einen Grenzwert ε.

Allerdings kann es durch Zyklen im Graphen dazu kommen, dass diese Berechnung nicht konvergiert. Außerdem kann ein Zyklus, in den Kanten hineinführen, aber nicht wieder heraus, zu einem „Abfluss“ (*rank sink*) werden, in dem die Gewichte sozusagen versickern.

Daher wird eine Quelle S für Gewichte in die Berechnung eingebunden:[24]

$$\begin{aligned} R_{neu} &:= c \cdot A \cdot R_{alt} + c \cdot S \\ &= c(AR_{alt} + S) \end{aligned}$$

c dient wieder zur Normierung, so dass $\|R_{neu}\|_1 = 1$.

Im Modell steht diese Quelle dafür, dass der Benutzer nicht einen Link auf der gerade besuchten Seite verfolgt, sondern zu einer anderen Seite springt – wie er es z. B. tun würde, wenn er sich in einem Zyklus befindet. Die Wahrscheinlichkeit, dass eine Seite v auf diese Weise angesprungen wird, wird dabei durch $S(v)$ ausgedrückt.

3.5.2 Experiment: Auffinden von verwandten Seiten mit PageRank

In [Pag98] wird die Quelle S als Präferenzvektor eingesetzt, um persönliche Vorlieben eines Benutzers zu simulieren. Es wird z. B. nachgebildet, dass der Benutzer eine Reihe von Web-

[24] Im Originalpapier heißt diese Quelle E; um eine Verwechslung mit der Kantenmenge E zu vermeiden, wird hier S benutzt.

seiten hat, die er anspringt, wenn das Verfolgen von Links keine relevanten Seiten zu einem Thema mehr liefert. Solche Seiten könnten etwa die Portale der großen Suchmaschinen sein.

Um diese vorzugeben, wird beispielsweise die Seiten $v_1, \ldots, v_k$ der bekannten Suchmaschinen mit $S(v_i) = 1/k$ für $i = 1 \ldots k$ initialisiert. Für alle anderen Seiten u wird $S(u) = 0$ gesetzt.

Dieser Präferenzvektor S lässt sich einsetzen, um thematisch verwandte Communities aufzufinden. Im genannten Benutzermodell entspricht dies dem Fall, dass die beliebten Seiten eines Benutzers einem thematisch zusammenhängenden Gebiet angehören.

Um dies zu demonstrieren, wird PageRank für die vorliegende Arbeit implementiert (Programm pagerank.cc; siehe auch Abschnitt 6.1.2).

S wird für ausgewählte Seiten v (im Folgenden einfach Vorgaben genannt) mit $S(v) = 1$ initialisiert, für alle anderen Seiten u ist $S(u) = 0$. Anschließend wird S normiert, so dass $\|S\|_1 = 1$.

Die vollständigen Ergebnisse liegen auf der CD im Verzeichnis auswertung/ergebnisse/pagerank.

Beispiel: Bibliographie-Server

Für diesen Versuch werden die Seiten des DBLP-Bibliographie-Servers der Uni-Trier, des Bibliographie-Servers ResearchIndex und der Networked Computer Science Technical Reference Library (NCSTRL) vorgegeben. Die von PageRank ermittelten Seiten werden in absteigender Relevanz ausgegeben.

Die zu Grunde liegenden Daten sind die Seiten des BFS-Laufes mit Site-übergreifendenden Links, beschränkt auf die tatsächlich gelesenen Seiten (s. Abschnitt 3.2.1). Der Grenzwert für die Konvergenz ist $\varepsilon = 0.02$.

Dabei werden unter anderem die folgenden Seiten entdeckt:

Pos.	URL/Erläuterung
1	http://www.researchindex.com:80/ Startseite ResearchIndex
2	http://www.ncstrl.org:80/ Startseite NCSTRL
3	http://www.informatik.uni-trier.de:80/~ley/db/ Startseite DBLP
6	http://www.acm.org:80/sigmod/ ACM SIGMOD
9	http://www.ieee.org:80/ Startseite IEEE
11	http://www.acm.org:80/ Startseite ACM mit Verweisen auf Digital Library, Konferenzen, Publikationen
12	http://www.vldb.org:80/ Startseite VLDB mit Verweisen auf Konferenzen und Publikationen
20	http://www.research.microsoft.com:80/ Microsoft Research; u. a. Konferenzen, Publikationen
21	http://liinwww.ira.uka.de:80/bibliography/index.html The Collection of Computer Science Bibliographies
22	http://computer.org:80/publications/dlib/index.htm IEEE Computer Digital Library
23	http://www.nzdl.org:80/ New Zealand Digital Library
24	http://computer.org:80/ IEEE Computer
25	http://theory.lcs.mit.edu:80/~dmjones/hbp/ The Hypertext Bibliography Project
28	http://www.acm.org:80/dl/ ACM Digital Library
36	http://www.eecs.umich.edu:80/digital-review/ ACM SIGMOD Digital Review

Pos.	URL/Erläuterung

Tabelle 3.13: PageRank-Ergebnis: Bibliographie-Server

Die ausgelassenen Seiten bestehen aus Unterseiten der angegebenen URLs, aus Seiten der Institutionen, die die drei vorgegebenen Dienste anbieten (Universität Waikato, NEC Research Institute bzw. Universität Trier), sowie aus Mirrors und alternativen Adressen für die Dienste.

Beispiel: Linux

Hier werden alle Seiten im Präferenzvektor S mit 1 initialisiert, deren URL den Text linux enthielt; S wird anschließend wieder normiert zu $\|S\|_1 = 1$.

Trotz dieser sehr groben Vorgehensweise findet PageRank recht viele wichtige Linux-Seiten, wie die Tabelle zeigt.

Die betrachteten Daten sind wieder die gelesenen Seiten des BFS-Laufes mit Site-übergreifendenden Links, $\varepsilon = 0.02$.

Pos.	URL/Erläuterung
1	`http://explorer.msn.com:80/`
2	`http://www.microsoft.com:80/windows/ie/` Downloadseiten Microsoft Internet Explorer
3	`http://www.netscape.com:80/computing/download/` Downloadseiten Netscape
4	`http://www.newsforge.com:80/` Nachrichten zu Open Source Software
5	`http://www.linuxbios.org:80/` Linux als BIOS-Ersatz

Pos.	URL/Erläuterung
6	http://www.linux.com:80/ Allgemeines Linux-Portal
7	http://slashdot.org:80/ Nachrichten zu Open Source Software
8	http://www.usenix.org:80/events/xfree86 XFree86 Conference
9	http://www.osdn.com:80/terms.shtml Open Source Development Software
10	http://www.debian.org:80/intro/cn Debian (Linux-Distribution)
11	http://www.linuxshowcase.org:80/ Linux-Konferenz
12	http://www.spi-inc.org:80/ Software in the Public Interest (Open Source Software)
15	http://www.stallman.org:80/ Richard Stallman
16	http://www.redhat.com:80/ Redhat (Linux-Distribution)
18	http://www.gnu.org:80/ Free Software Foundation
26	http://www.suse.com:80/ SuSE (Linux-Distribution)
28	http://freshmeat.net:80/ Freshmeat (Linux-Software)
29	http://www.turbolinux.com:80/ TurboLinux (Linux-Distribution)
30	http://www.linuxdoc.org:80/ Linux Documentation Project
32	http://linuxtoday.com:80/ Nachrichten zu Linux

Pos.	URL/Erläuterung

Tabelle 3.14: PageRank-Ergebnis: Linux

Bemerkenswert ist, dass an den ersten drei Positionen die Download-Seiten von Internet Explorer bzw. Netscape stehen. Dies rührt vermutlich daher, dass diese Seiten durch ein extremes Verhältnis von Eingangs- und Ausgangsgrad Senken bilden: die Seiten an Position 2 und 3 haben jeweils einen Eingangsgrad von etwa 200 und einen geringen Ausgangsgrad. Die Seite an Stelle 2 hat sogar nur eine einzige ausgehende Kante, nämlich zu der Seite an Position 1. Damit leitet sie ihr Gewicht vollständig an diese Seite weiter.

Zusammenfassung

PageRank ist auch alleine geeignet, um zu einigen vorgegebenen Seiten inhaltlich verwandte Dokumente zu finden. Die Suchmaschine Google, für die der Algorithmus entwickelt wurde, benutzt das PageRank-Gewicht lediglich zur Anordnung der Resultate. Im hier vorgestellten Experiment wurde gezeigt, dass sich das Verfahren auch zur Ähnlichkeitssuche bei vorgegebenen Seiten eignet.

Die Annahme, dass relevante Seiten wieder auf relevante Seiten zeigen, führt auf der hier betrachteten Datenbasis zum erwarteten Ergebnis. Zu vorgegebenen Seiten werden weitere verwandte Seiten gefunden.

Dabei ist allerdings ein thematisches „Abdriften“ möglich, wie das Beispiel der Browser-Seiten zeigt.

3.6 Der HITS-Algorithmus

3.6.1 Einführung

Ähnlich wie PageRank basiert auch der Algorithmus HITS (*Hyperlink-Induced Topic Search*) von Kleinberg u. a. auf der Idee, dass Seitengewichte über Links weitergegeben werden und dass durch eine Fixpunktiteration die Wichtigkeit der einzelnen Seiten ermittelt werden kann.

Anders als PageRank unterscheidet HITS allerdings zwei Arten von Seiten. Neben den *Authorities*, d. h. den für das Thema relevanten Dokumenten, werden *Hubs*[25] betrachtet, die auf Authorities verweisen. Ein Hub könnte z. B. eine der häufig zu findenden Sammlungen sein, die für ein bestimmtes Thema relevante Links enthält.

Jeder Knoten v trägt dementsprechend zwei Gewichte; eines davon, $h(v)$ ist das *hub weight*; je größer $h(v)$, desto eher ist v ein Hub. Analog ist $a(v)$ das *authority weight*.

Wiederum liegt der Berechnung die Idee zu Grunde, dass eine Seite wichtig wird, wenn wichtige Seiten auf sie verweisen. Diesmal wird allerdings zwischen Hubs und Authorities unterschieden: ein wichtiger Hub ist ein solcher, der auf wichtige Authorities zeigt. Eine Authority wird wichtig, wenn wichtige Hubs sie referenzieren.

In Gleichungen lässt sich das wie folgt ausdrücken:

$$h_{neu}(u) := \sum_{u:(u,v)\in E} a_{alt}(v)$$

Das Hub-Gewicht eines Knotens u ist also die Summe der Authority-Gewichte aller Knoten v, auf die u verweist.

[25] engl.: Radnabe

$$a_{neu}(u) := \sum_{v:(v,u)\in E} h_{alt}(v)$$

Dazu symmetrisch ist das Authority-Gewicht eines Knotens u die Summe der Hub-Gewichte aller Knoten v, die auf u verweisen.

Diese Gleichungen lassen sich auch übersichtlicher in Matrixschreibweise formulieren:

$$h_{neu} := A\, a_{alt} \quad \text{und} \quad a_{neu} := A^T\, h_{alt}$$

Dabei ist A die Adjazenzmatrix des Graphen, d. h.

$$A_{uv} = \begin{cases} 1 & \text{falls } (u,v) \in E \\ 0 & \text{sonst} \end{cases}$$

Um die Konvergenz zu beschleunigen, kann ein Dämpfungsfaktor $0 \leq d \leq 1$ eingeführt werden, so dass die Gewichte nur zum Teil weitergegeben werden:

$$h_{neu} := d\, h_{alt} + (1-d)\, A\, a_{alt} \quad \text{und} \quad a_{neu} := d\, a_{alt} + (1-d)\, A^T\, h_{alt}$$

Die Hub- und Authority-Gewichte werden in jedem Schritt normiert, so dass jeweils gilt $\|h\|_2 = 1$ und $\|a\|_2 = 1$.

Dabei wird die L_2-Norm benutzt:

$$\|h\|_2 := \sqrt{\sum_{v\in V} h(v)^2}$$

Die Iteration wird fortgesetzt bis $\|h_{neu} - h_{alt}\|_2 \leq \varepsilon$ und $\|a_{neu} - a_{alt}\|_2 \leq \varepsilon$ für ein vorgegebenes ε.

3.6.2 Experiment

Der HITS-Algorithmus wird implementiert im Programm kleinberg.cc; siehe dazu auch Abschnitt 6.1.2.

In der Originalveröffentlichung wird HITS eingesetzt, um ausgehend von den Ergebnissen einer textbasierten Suchmaschine eine entsprechende Menge von verwandten Seiten aus Hubs und Authorities zu finden. Man geht dabei von den Ergebnissen einer textbasierten Suche aus und erweitert diese Knotenmenge um Knoten, die durch ein bis zwei Kanten erreichbar sind. Darauf wird dann die HITS-Berechnung durchgeführt.

Im Gegensatz dazu wird hier untersucht, welche Strukturen HITS auf dem schon vorliegenden Datenbestand entdecken kann, den der Crawler erzeugt hat. Die Fixpunktiteration wird also auf dem *gesamten* Graphen durchgeführt.

Um zu untersuchen, inwiefern sich die Verteilung der Gewichte durch unterschiedliche Startmengen thematisch steuern lässt, werden verschiedene Mengen von Seiten als Hubs und/oder Authorities vorgegeben. Ihre Hub- oder Authority-Gewichte werden dazu mit dem Wert 1 initialisiert, alle anderen Seiten erhalten den Wert 0.[26] Der Algorithmus läuft bis zur Konvergenz, und die Seiten v, die Hub- oder Authority-Gewichte $h(v)$ bzw. $a(v)$ größer als einen Grenzwert t aufwiesen, werden ausgegeben.

Folgende Startmengen werden erprobt:

1.–3. Seiten, die eine URL der Form http://...cs...edu:.../... haben, werden als Authorities (1), Hubs (2) bzw. Hubs *und* Authorities (3) vorgegeben. (18086 Seiten bei BFS-Strategie, 3894 Seiten bei der fokussierten Strategie). Dies ist eine grobe Heuristik zum Erkennen von URLs an amerikanischen Universitäten, die sich mit Informatik

[26] Der Betrag des Werts für die Startseiten hat allerdings keine Bedeutung, da in jedem Schritt eine Normierung erfolgt.

(cs steht für *computer science*) befassen.

4.–6. Seiten, die eine URL der Form http://...informatik...de:.../... haben, werden als Authorities (Hubs/Hubs und Authorities) vorgegeben (4727 bzw. 2022 Seiten). Damit werden Seiten auf deutschen Informatik-Servern ausgewählt, wiederum mit einer groben Heuristik.

7.–9. Es werden die Seiten als Authorities (Hubs/Hubs und Authorities) vorgegeben, die den Text linux in der URL enthalten (352 bzw. 1431 Seiten).

10.-12. Die Startseiten dreier Dienste, die sich mit Online-Publikationen befassen (DBLP, ResearchIndex, NCSTRL) werden als Authorities (Hubs/Hubs und Authorities) vorgegeben (3 bzw. 1 Seite; die fokussierte Strategie hat von diesen dreien nur NCSTRL gelesen).

13. Es werden je 100 zufällige Hubs und Authorities vorgegeben.

14. Alle Seiten bekommen gleiche Hub- und Authority-Gewichte als Vorgabe.

Tabelle 3.15 zeigt die Ergebnisse für die Daten des BFS-Laufs, beschränkt auf gelesene Seiten und Site-übergreifende Links, mit einem Dämpfungsfaktor $d = 0.05$, einer Schranke $\varepsilon = 0.01$ für die Konvergenz und einem Grenzwert für die Ausgabe von $t = 0.0005$. Die kompletten Ausgabedateien liegen im Verzeichnis auswertung/ergebnisse/kleinberg auf der CD.

Ein Haken (✓) bedeutet, dass die betreffende Seite bei der entsprechenden Startmenge als Hub („H“ in Spalte H/A) bzw. Authority („A“) ermittelt wird.

1: …cs…edu → AUTH	2: …cs…edu → HUB	3: …cs…edu → A+H	4: …informatik…de → AUTH	5: …informatik…de → HUB	6: …informatik…de → A+H	7: …linux… → AUTH	8: …linux… → HUB	9: …linux… → A+H	10: DBLP, NCSTRL, ResearchIndex → AUTH	11: DBLP, NCSTRL, ResearchIndex → HUB	12: DBLP, NCSTRL, ResearchIndex → A+H	13: zufällige Hubs, zufällige Authorities	14: uniform (alle H+A → 1)	**H/A**	**URL**
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	A	http://epubs.siam.org:80/
												✓	✓	A	http://www-ppg.dcs.st-and.ac.uk:80/People/Ron/
			✓											A	http://www.acm.org:80/
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	A	http://www.acm.org:80/dl/
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	A	http://www.acm.org:80/ubiquity
			✓											A	http://www.cisco.com:80/
												✓	✓	A	http://www.dcs.st-and.ac.uk:80/\%7Eal/
												✓	✓	A	http://www.dcs.st-and.ac.uk:80/\%7Ecolin/
													✓	A	http://www.dcs.st-and.ac.uk:80/Admin/People/Details/mkw.html
	✓		✓						✓		✓			A	http://www.hp.com:80/
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	A	http://www.idealibrary.com:80/
	✓		✓											A	http://www.microsoft.com:80/
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	A	http://www.sciencedirect.com:80/
	✓		✓						✓		✓			A	http://www.sun.com:80/
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://milton.mse.jhu.edu:80/research/ejournals.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.apnet.com:80/www/journal/hm.htm
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.apnet.com:80/www/journal/nt.htm
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.brics.dk:80/~omoeller/blip.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.cs.utexas.edu:80/users/nn/pages/resources/resources.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.cs.virginia.edu:80/brochure/library.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.cs.virginia.edu:80/brochure/profs/pearson.html
✓														H	http://www.csc.calpoly.edu:80/~alum/cgi-bin/alphaList.cgi
✓														H	http://www.csc.calpoly.edu:80/~alum/cgi-bin/chronList.cgi
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.csci.csusb.edu:80/dick/se.www.sites.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.daimi.au.dk:80/~omoeller/blip.html

1	2	3	4	5	6	7	8	9	10	11	12	13	14	H/A	URL
✓													✓	H	http://www.dcs.st-andrews.ac.uk:80/cgi-bin/cgiwrap/www/publications.cgi
													✓	H	http://www.dcs.st-andrews.ac.uk:80/cgi-bin/cgiwrap/www/publications.cgi?type=grant
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.informatik.uni-trier.de:80/~ley/db/journals/index.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.int-evry.fr:80/biblio/BER/periodique_liste.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.lib.csusb.edu:80/database/
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.lib.ic.ac.uk:80/ejournals/ejnls_sub.htm
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.lib.rochester.edu:80/database/ejournal.htm
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.jcu.edu.au:80/Resources/datasets.shtml
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/anthro.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/biology.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/business.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/chemistry.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/compsci.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/economics.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/environment.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/geology.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/mathematics.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/physics.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/psychology.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.library.mwc.edu:80/sociology.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.math.utah.edu:80/~beebe/digital-libraries.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.mcs.vuw.ac.nz:80/research/publications/1999.shtml
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.mpi-sb.mpg.de:80/services/library/news/news.html
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	H	http://www.people.virginia.edu:80/~wrp/pearson.html

Tabelle 3.15: Hubs und Authorities mit verschiedenen Startmengen

Wie die Tabelle verdeutlicht, hat die Wahl der Startmenge keinen großen Einfluss auf das Ergebnis. Der überwiegende Teil der Hubs und Authorities wird von allen Startmengen aus gefunden.

Es fällt auf, dass die gefundenen Hubs und Authorities beinahe ausnahmslos dem Bereich des wissenschaftlichen Publizierens angehören. So finden sich darunter Verweise auf Online-Bibliotheken und wissenschaftliche Journale, Literaturdatenbanken und dergleichen. Diese Dokumente bilden also offenbar eine Art Schnittmenge dessen, wofür sich die Autoren der Seiten in der Datenbasis interessieren.

Authorities sind hier solche Seiten, die digitale Bibliotheken zur Verfügung stellen, wie etwa die digitalen Bibliotheken der ACM (*Association for Computing Machinery*) und der SIAM (*Society for Industrial and Applied Mathematics*).

Hubs sind dagegen hauptsächlich Verweislisten, die als Überblick über vorhandene Resourcen aus dem Bereich der elektronischen Publikationen erstellt worden sind.

Die Auswertung für den Lauf mit der fokussierten Strategie sieht sehr ähnlich aus, allerdings sind die aufgeführten elektronischen Publikationen thematisch nicht auf Informatik beschränkt.

Es gibt bei diesem Lauf allerdings zwei Auffälligkeiten:

- Die Startmenge, die nur aus der Seite von NCSTRL als Hub bestand, lieferte keine Hubs oder Authorities. Dies liegt daran, dass der Knoten in dem betrachteten Graphen Ausgangsgrad 0 hat.

- Die Startmenge mit den deutschen Informatik-Seiten als Authorities (Spalte 4) liefert eine Menge von Hubs und Authorities, die beinahe vollständig disjunkt ist mit den Ergebnissen der anderen Startmengen. Die ermittelten Hubs und Authorities lassen auch keine thematische Zusammengehörigkeit erkennen.

3.6.3 Schlussfolgerungen

Aus den genannten Auswertungen lassen sich folgende Folgerungen ableiten:

- Auf einer großen Seitenmenge global angewendet, konvergiert HITS sehr stabil im Hinblick auf verschiedene Startmengen. Es wird eine feste Menge $M(G)$ von Hubs und Authorities gefunden, die nur vom Graphen, nicht aber von der Startmenge abhängt.
- Die gefundenen Hubs und Authorities entsprechen der Interpretation von relevanten Seiten einerseits und Überblicksseiten andererseits.
- Falls der Algorithmus nicht gegen $M(G)$ konvergiert, so zeigen die gefundenen Seiten keinen deutlichen thematischen Zusammenhang und keine inhaltliche Übereinstimmung mit dem Konzept von Hubs und Authorities.
- $M(G)$ für die beiden betrachteten Graphen besteht jeweils aus Hubs und Authorities, die dem Bereich der wissenschaftlichen Publikationen angehören.

3.7 Störende Strukturen im Graphen

Bei der Untersuchung der betrachteten Teilgraphen des WWW kommen einige Strukturen zum Vorschein, die weder wirkliche inhaltliche oder soziale Beziehungen ausdrücken, noch auf „natürliche“ Weise zu Stande gekommen sind.

Riesige Cliquen wie die unten genannten, verfälschen stark die Auswertung; so „saugt“ eine solche große Clique beispielsweise beim HITS-Verfahren (Abschnitt 3.6) regelrecht die Hub- und Authority-Gewichte auf, so dass wirklich interessante Strukturen unentdeckt bleiben.

Aus diesen Gründen werden diese Teilgraphen aus der Datenbank entfernt. Die einzelnen Fälle sind hier aufgeführt:

Fehlerhaft konfigurierter Webserver Beim deutschen Internet-Verein *handshake e. V.*, der für seine Mitglieder Web-Hosting anbietet, war ein fehlerhaft konfigurierter Webserver im Einsatz. Dieser Server beheimatet über 1100 Web-Präsenzen, die größtenteils unter verschiedenen .de-Domains eingetragen sind.

Durch den Fehler in der Konfiguration kommt es bei bestimmten HTTP-1.1-konformen Anfragen[27] zu einem Fehler: Es wird nicht die gewünschte Seite, sondern die Nutzerliste des *handshake e. V.* zurückgegeben.

Auf dieser Liste stehen Verweise auf alle Mitglieder, so dass 1116 mal die gleiche Seite heruntergeladen wird, die Verweise auf alle ihre Kopien trägt.

Spamming zur Beeinflussung von Suchmaschinen Der Domain-Anbieter *Service Partner* bietet seine Dienste unter 125 verschiedenen Domains wie www.webspace-profi.com, www.service-partner.at, www.domain-fuchs.com usw. im Netz an, die alle eine Kopie der gleichen Seite tragen.

Auf jeder dieser Kopien sind unsichtbare Hyperlinks[28] auf jeweils alle anderen Kopien enthalten. Diese für den normalen Benutzer nicht sichtbaren Links dienen offenbar nur dazu, möglichst viele Seiten dieses Anbieters in den Indizes der großen Suchmaschinen unterzubringen.

In Analogie zum Versand von Massen-EMails wird ein solches Vorgehen als *Spamming* bezeichnet [Sul01].

3.8 Bemerkungen zu den Programmen zur Auswertung

Zwar gelten die später in 5.1 angeführten Argumente für die Verwendung von Java als Programmiersprache teilweise auch hier. Es hat sich aber herausgestellt, dass für den Umgang mit sehr großen Graphen die verfügbaren Bibliotheken für Java, wie z. B. die *Graph Foundation Classes* von IBM (`http://www.alphaworks.ibm.com/tech/gfc`), nicht

[27] Host:-Feld mit Angabe der Portnummer im HTTP-Header
[28] Unsichtbar heißt hier: Es handelt sich also um Hyperlinks ohne Text.

geeignet sind. Besonders der Speicherplatzbedarf bei Verwendung dieser Bibliotheken lässt eine Behandlung grosser Graphen nicht zu.

Die in diesem Kapitel vorgestellten Graphalgorithmen werden daher mit Hilfe der C++-Bibliothek LEDA [Meh99] umgesetzt; diese ist im Hinblick auf den Hauptspeicherbedarf wesentlich sparsamer.

Neben Standarddatentypen wie Mengen, Listen usw. bietet LEDA einen Datentyp für Graphen, der für diese Arbeit besonders nützlich ist. Zudem sind die Datentypen von LEDA mit „syntaktischem Zucker“ in Form von Präprozessor-Makros versehen, so dass sich praktisch Pseudocode direkt als Programm schreiben lässt.

Das Codebeispiel in Abbildung 3.25 zeigt dies am Beispiel des Pagerank-Algorithmus (Abschnitt 3.5) – dieses Programmstück verteilt das Gewicht r_old(u) für alle Knoten u im Graphen g anteilig auf alle Nachfolger v von u, wobei die neuen Gewichte im Vektor r_new gespeichert werden.

Bemerkenswert sind an dieser Stelle auch die sogenannten *Node Arrays* r_old und r_new, die wie normale Felder in C++ verwendet werden. Allerdings sind die Node Arrays nicht mit Zahlen $0 \ldots n-1$ indiziert, sondern mit den Knoten eines Graphen.

```
// Node Arrays für Gewicht mit 0 initialisieren
node_array<double> r_old(g, 0);
node_array<double> r_new(g, 0);

...

node u;
forall_nodes (u, g) {
    edge e;
    forall_out_edges (e, u) {
        node v = target (e);
        r_new[v] += (1.0/g.outdeg(u)) * r_old[u];
    }
}
```

Abbildung 3.25: Codebeispiel LEDA

Durch die gezeigten komfortablen Möglichkeiten zur Programmierung von Graphalgo-

rithmen in LEDA lassen sich die Algorithmen aus diesem Kapitel unmittelbar in C++-Programme umsetzen. Die einzelnen Programme bowtie, clust, bip_clust, pagerank und kleinberg werden daher hier nicht weiter erläutert. Die Benutzung der Programme wird in Abschnitt 6.1.2 erklärt.

Das einzige größere Problem in diesem Zusammenhang ist das Einlesen des Graphen aus der Datenbank in die C++-Programme. Darauf wird in Abschnitt 6.1.2 eingegangen.

4 Entwicklung eines Crawlers: Analyse und Entwurf

Diese Arbeit untersucht eine Teilmenge des WWW mit Algorithmen der Graphentheorie. Es wäre zwar möglich, diese Algorithmen online auszuführen, d. h. jede Seite dann aus dem Web abzurufen, wenn der Algorithmus sie benötigt. Dies verbietet sich aber aus folgenden Gründen:

- Der Abruf einer Seite aus dem Web dauert sehr lange verglichen mit einem Datenbank- oder Hauptspeicherzugriff.
- Jede Seite sollte nur einmal abgerufen werden. Ein Graphalgorithmus besucht aber möglicherweise jeden Knoten viele Male. Die dadurch unnötig erzeugte Last für die benutzten Server wäre nicht akzeptabel.
- Die Kanten im Webgraphen können nur von den Seiten aus entdeckt werden, von denen sie ausgehen. Die eingehenden Kanten eines Knotens sind ohne weiteres[1] nicht zu bestimmen.

Daher ist es erforderlich, mit Hilfe eines Crawlers eine größere Teilmenge des WWW automatisch zu durchlaufen und die gelesenen Seiten lokal abzuspeichern.

[1] Zwar bieten die gängigen Suchmaschinen die Möglichkeit, eingehende Links zu einer Webseite zu finden, aber diese Funktion viele tausend Male im Ablauf eines Algorithmus zu verwenden ist nicht praktikabel.

Es ist mit modernen Programmiersprachen und ihren Bibliotheken vergleichsweise einfach, einen einfachen Webcrawler zu entwickeln, der automatisch Webseiten abruft und Links verfolgt. Die benötigte Funktionalität, z. B. für Netzwerkzugriffe oder die Verarbeitung von HTML-Daten, gehört in vielen Sprachen mittlerweile zur Standardbibliothek.

Für die Betrachtungen der Daten in Kapitel 3 ist es allerdings notwendig, eine große Zahl von Seiten zu sammeln. Außerdem müssen diese Seiten für eine weitere Bearbeitung leicht zugänglich sein.

Dieses Kapitel beschreibt einen Crawler, der diese Bedingungen erfüllt. Abschnitt 4.1 beschreibt zunächst die genauen Anforderungen, die an den Crawler gestellt werden. Der Entwurf des Crawlers und der zu Grunde liegenden Datenstrukturen wird in Abschnitt 4.2 vorgestellt.

4.1 Anforderungen

Die Anforderungen an den Crawler resultieren einerseits aus der Aufgabenstellung, andererseits gibt es allgemeine Verhaltensregeln für ein automatisches Ansprechen von Webservern.

4.1.1 Anforderungen durch die Aufgabenstellung

Um eine größere Teilmenge des WWW zu besuchen und die Ergebnisse für weitere Betrachtungen zugänglich zu machen, muss der Crawler vor allem diese Anforderungen erfüllen:

Durchsatz Das Programm soll einen möglichst hohen Durchsatz im Hinblick auf Seitenabrufe pro Zeiteinheit haben.

Datenunabhängigkeit und Persistenz Die gesammelten Daten, die größerer Menge anfallen, werden mit anderen Programmen weiter verarbeitet. Dazu müssen sie in einer

Form persistent gespeichert werden, die zum einen die Manipulation auch umfangreicher Datenmengen erlaubt, und zum anderen allgemein verwendbar ist.

Erweiterbarkeit und Flexibilität Es soll einfach möglich sein, den Crawler um neue Aspekte zu erweitern. Das Programm geht mit „fremden“ Daten um, deren genaue Eigenschaften nicht im Voraus bekannt sind. Deshalb muss damit gerechnet werden, dass im Laufe der Zeit noch Funktionalität nachzurüsten ist.

Die Strategie für das Auswählen neuer Ziele für den Crawler soll veränderbar sein. Sie kann leicht durch eine andere Strategie ersetzt werden.

Robustheit und Fehlertoleranz Die gesammelten Daten stammen aus vielerlei Quellen und sind daher von wechselnder Qualität. So ist beispielsweise ein Großteil aller HTML-Dokumente nicht konform zu den Standards des W3C[2]. Es treten ungültige Links auf, und es muss mit fehlerhaften Webservern gerechnet werden.

Mit diesen Situationen muß das Programm umgehen und sie an den betreffenden Daten vermerken.

Um hinreichend viele Seiten zu sammeln, muss das Programm außerdem über eine lange Zeit hinweg laufen. Deshalb ist es wichtig, dass es robust ist. Das heißt beispielsweise, dass nach einem Programmabbruch oder -absturz mit einem konsistenten Zustand und möglichst geringem Datenverlust wieder aufgesetzt werden kann. Weiterhin sind Mechanismen notwendig, die dafür sorgen, dass ein solches Wiederaufsetzen automatisch und unbeaufsichtigt erfolgt.

Beschränkungen Neben den im nächsten Abschnitt genannten Randbedingungen, die aus Gründen der sozialen Verträglichkeit[3] für Crawler im Allgemeinen gelten, werden weitere Beschränkungen eingehalten. Speziell können die Tiefe des Crawls, d. h. die größte Entfernung zu den Startseiten, sowie die maximale Anzahl von betrachteten Seiten auf einem Server beschränkt werden.

[2] World Wide Web Consortium

[3] In einigen Texten über Crawler, z. B. [Hey99, Abschnitt 3.2], wird gefordert, dass Crawler *socially acceptable*, also sozial verträglich, sein sollen.

4.1.2 Allgemeine Randbedingungen

Ein unbedacht entwickelter Webcrawler kann leicht die Administratoren der angefragten Server verärgern, Ressourcen verschwenden und zu Überlastung führen.

Martijn Koster [Kos93] hat daher eine Liste von Kriterien zusammengestellt, die ein Crawler erfüllen soll, um niemandem über Gebühr zur Last zu fallen. Die wichtigsten davon sind:

„Identify your Web Wanderer [. . .] *Identify yourself“* Der Crawler sowie die Person, die den Crawler entwickelt hat, sollten aus der HTTP-Anfrage identifizierbar sein. Dadurch wird dem betroffenen Administrator ermöglicht, sich über die Aufgabe des Crawlers zu informieren oder bei Fehlverhalten den Entwickler zu benachrichtigen.

Es empfiehlt sich dabei, im HTTP-Request die EMail-Adresse des Entwicklers anzugeben sowie die Adresse eines Dokuments, das den Crawler und seinen Verwendungszweck beschreibt. Dazu können die Felder Referer: und From: benutzt werden.

„Don’t hog resources [. . .] *Walk, don’t run“* Ein Hauptproblem bei Einsatz von Crawlern ist, dass sie leicht Server überlasten können, indem sie zu viele Objekte in zu kurzer Zeit anfordern. Daher sollten auf jeden Fall dafür gesorgt werden, dass der Crawler jeden einzelnen Server nur mit einer bestimmten Höchstfrequenz anspricht, z. B. mit maximal einer Anfrage pro Minute.

Diese Beschränkung steht in teilweisem Widerspruch zu der vorher geäußerten Forderung nach hohem Durchsatz. Beiden Forderungen gleichzeitig nachzukommen ist eines der Hauptprobleme für die Entwicklung des Crawlers.

Robots Exclusion Das informell standardisierte Robots Exclusion Protocol [Kos94] ist eine allgemeine Vereinbarung zwischen Betreibern von Crawlern und den Administratoren von Webservern.

Es beschreibt eine einheitliche Möglichkeit, wie Administratoren Dokumente auf Webservern benennen können, die nicht von Crawlern gelesen werden sollen. Außerdem dient die Robots Exclusion auch dazu, Seiten zu markieren, die ohnehin für Crawler nicht interessant oder verwirrend wären; so können beispielsweise CGI[4]-Skripte oder sonstige automatisch generierte Seiten zu regelrechten „Fallen“ für

[4] Common Gateway Interface

Crawler werden, da sie potentiell unendlich viele Seiten erzeugen können. Es gibt auch absichtlich eingerichtete „Spider Traps“ [Kol96], die dafür sorgen, dass Crawler sich in für sie verbotenen Bereichen von Webservern stecken bleiben.

Es ist daher notwendig, dieses Protokoll einzuhalten.

4.2 Entwurf

Dieser Abschnitt beschreibt den Entwurf eines Crawlers, der die in 4.1 gestellten Anforderungen erfüllt.

Die anfallenden Daten werden in einer relationalen Datenbank gespeichert. Dafür spricht zum einen, dass relationale Datenbank-Management-Systeme (RDBMS) ausgereift und auch für große Datenbanken vielfach erprobt sind. Andererseits bieten RDBMS eine Reihe von Möglichkeiten, auf die gespeicherten Daten zuzugreifen, sei es von Java oder C/C++ aus, oder aber per Export in ASCII[5]-Dateien.

Abschnitt 4.2.1 beschreibt das Entity-Relationship-Modell für die ermittelten Daten.

Die eigentliche Programmlogik wird objektorientiert modelliert; die objektorientierte Sichtweise ist schon seit Jahren Standard bei der Strukturierung komplexer Software und hat sich dafür auch bewährt. Das OO-Modell wird in 4.2.2 vorgestellt.

4.2.1 Entity-Relationship-Modell

Das Entity-Relationship-Modell (Abbildung 4.1 auf Seite 99) der betrachteten Daten gestaltet sich recht einfach. Die Entity Sets sind webpage und hyperlink.

[5] American Standard Code for Information Interchange

Entity Set webpage

Ein Datenobjekt diesen Typs repräsentiert eine Webseite. Im Kontext dieser Arbeit sind nur solche Seiten interessant, die HTML-Code enthalten (d. h. einen Content-Type text/html haben).

Tabelle 4.1 beschreibt die Attribute dieses Enitity Sets genauer.

Attribut	**Erläuterung**
id	Primärschlüssel (laufende Nummer)
content	Seiteninhalt (HTML)
content_type	Content-Type (z. B. text/html)
last_fetched	Zeitstempel des letzten Abrufs dieser Seite in ms seit dem 1.1.1970, 0 Uhr GMT (in Java: System.currentTimeMillis())
http_state	HTTP-Rückgabecode des letzten Seitenabrufs
internal_state	interner Zustand (siehe 4.2.3)
protocol	URL-Bestandteil Protokoll
userinfo	URL-Bestandteil User Info
host	URL-Bestandteil Hostname
port	URL-Bestandteil Port-Nummer
file	URL-Bestandteil Dateiname
ranking	Priorität in der Schlange
depth	Tiefe (Entfernung von Startseite)

Tabelle 4.1: Attribute des Entity Sets webpage

Eine genauere Beschreibung des Feldes internal_state findet sich in Abschnitt 4.2.3.

Entity Set hyperlink

Ein Hyperlink ist eine Beziehung zwischen zwei Webseiten; es liegt also nahe, einen Hyperlink als Relationship im Sinne der ER-Modellierung zu betrachten.

Allerdings sind Links in der vorliegenden Arbeit ein zentraler Aspekt der Betrachtung; außerdem ist jeder Link mit einer Reihe von Attributen versehen, so dass im Folgenden Links als eigene Entities geführt werden. [6]

Jeder Hyperlink steht mit zwei Webseiten in Beziehung: mit einer, von der er ausgeht, und mit einer, auf die er zeigt. Tabelle 4.2 beschreibt die Attribute dieses Entity Sets. Eine Beschreibung der Attribute spans_hosts und spans_sites erfolgt in Abschnitt 3.2.1.

Attribut	**Erläuterung**
id	Primärschlüssel (laufende Nummer)
text	Text dieses Links
type	Art des Links (siehe Text)
spans_hosts	wahr, falls Zielseite auf einem anderen Host liegt
spans_sites	wahr, falls Zielseite in einer anderen Website liegt

Tabelle 4.2: Attribute des Entity Sets hyperlink

Dabei werden nicht nur Hyperlinks im eigentlichen Sinne, also Konstrukte mit dem HTML-Element <A HREF="...">...</A>, sondern noch weitere Arten von Seitenreferenzen betrachtet. Im Einzelnen sind dies die folgenden Linktypen, die im Attribut type angezeigt werden:

- TYPE_HREF: Hyperlink mit Hilfe eines A-Tags in HTML.
- TYPE_META: Seitenumleitung über ein META-Tag mit REFRESH-Attribut.

[6] Für die Abbildung in SQL-Tabellen macht diese Frage keinen Unterschied, da eine m:n-Beziehung genau so in eine Tabelle umgesetzt wird wie ein Entity Set mit zwei 1:n-Beziehungen.

- TYPE_FRAME: Beziehung zwischen FRAMESET und FRAME bei geschachtelten Dokumenten mit sog. *Frames*, wie sie mit HTML 4 eingeführt wurden.
- TYPE_AREA: Hyperlink über eine Image-Map; mit Image-Maps ist es möglich, Teile eines Bildes als Quelle eines Links anzugeben.

Für eine genauere Beschreibung dieser HTML-Konstrukte siehe [Con99].

Im Unterschied zu den oben angegebenen Methoden, die Links über HTML-Elemente ausdrücken, kann ein Server eine Umleitung von einer Seite auf eine andere im HTTP-Header angeben. Auch dies wird als Hyperlink betrachtet:

- TYPE_HTTP_30x: Umleitung über HTTP-30x-Code im HTTP-Header.

(Siehe dazu auch die HTTP-1.1-Spezifikation in RFC 2068 [Fie97].)

Für die Betrachtung des Webgraphen machen die verschiedenen Arten von Verweisen zwischen Seiten keinen Unterschied. Allerdings ist es hilfreich, sie unterscheiden zu können. So kann z. B. im Falle eines unerwarteten Ergebnisses bei der Auswertung genau nachvollzogen werden, wie dieses zu Stande kam.

Redundanz vs. Performance

Dieses Modell scheint zunächst nicht optimal im Hinblick auf Redundanzvermeidung: es werden z. B. URL-Bestandteile, wie etwa der Hostname, wiederholt aufgenommen.

Abbildung 4.2 zeigt eine Alternative, die die URL und den Hostnamen in eigene Entity Sets auslagert.

Dieses Modell hat jedoch einige Nachteile:

- Bei jeder Einfügung in die Tabelle webpage muss sichergestellt werden, dass die entsprechenden Einträge in url und host vorhanden sind. Das bedeutet zwei Mal ein

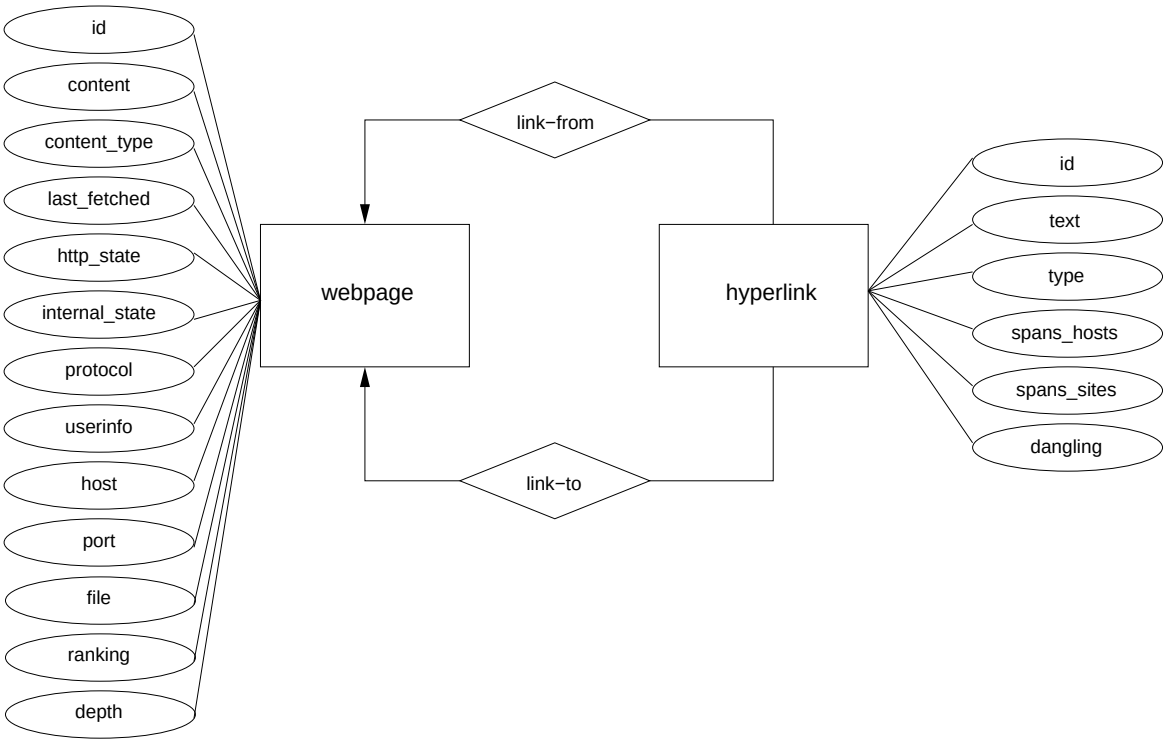


Abbildung 4.1: ER-Modell

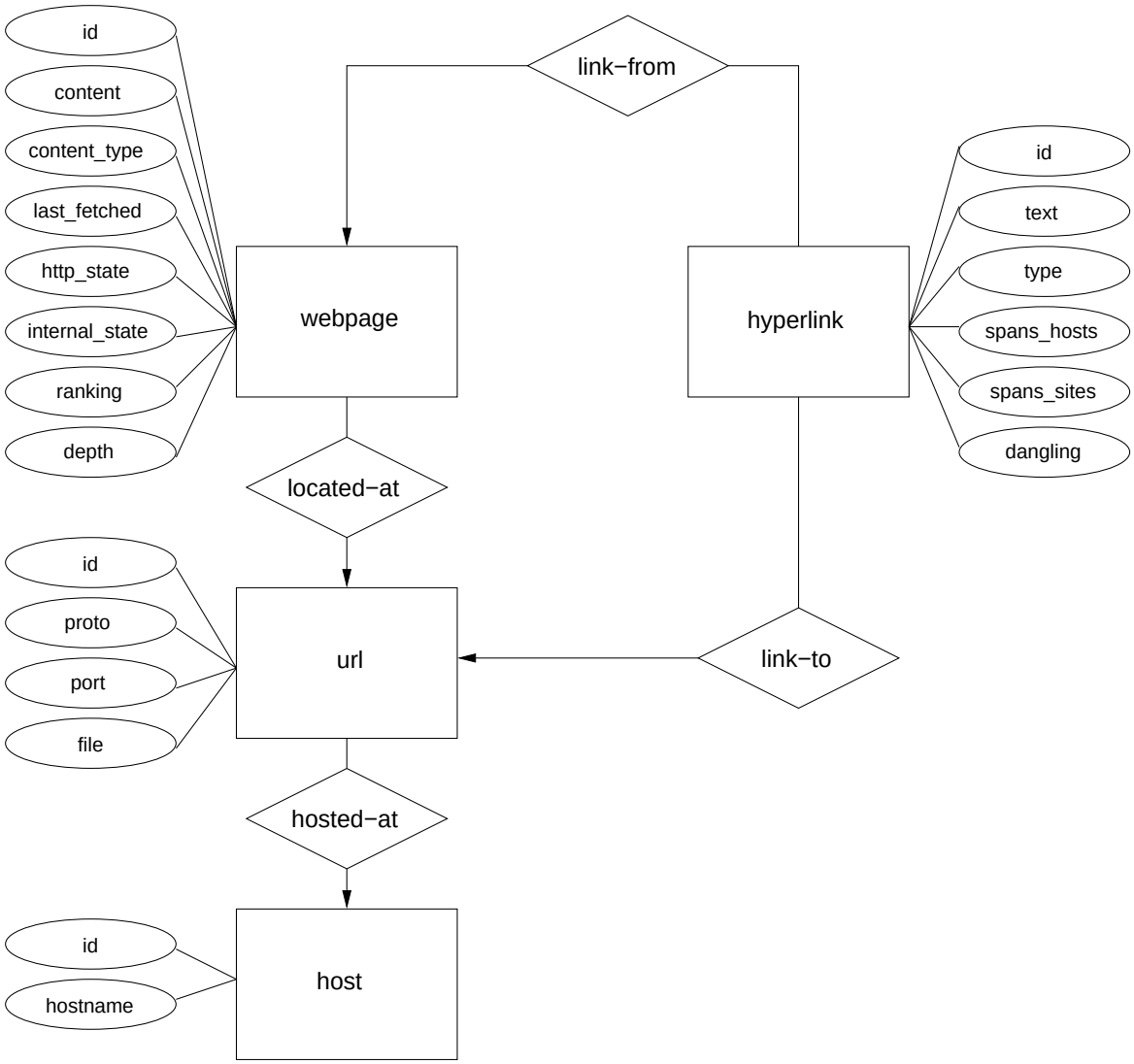


Abbildung 4.2: Alternatives ER-Modell

bis zwei Operationen: es muss nachgeprüft werden, ob die entsprechende Zeile schon vorhanden ist, und eventuell wird eine neue Zeile eingefügt.

- Die Tabelle host beinhaltet außer dem künstlichen Primärschlüssel nur den Namen des Hosts. Der einzige Grund, dafür eine Tabelle anzulegen, könnte die Platzersparnis in anderen Tabellen sein. Dort muss jeweils nur ein Fremdschlüssel gespeichert werden, nicht der ganze Name.

Aus diesen Gründen wird das Modell aus Abbildung 4.1 benutzt.

SQL-Tabellen

Das ER-Modell wird entsprechende Tabellendefinitionen in SQL[7] überführt. Die beiden 1:n-Beziehungen zwischen Webpage und Hyperlink werden über Fremdschlüsselbeziehungen realisiert. Für Einzelheiten über die Abbildung von ER-Modellen in Tabellen siehe z. B. [Sil97, Abschnitt 2.9].

Es ergeben sich folgende SQL-Definitionen[8]:

```
create table webpage (
        id bigint not null,

        -- URL-Komponenten:
        proto varchar(20) not null,
        userinfo varchar(200),
        hostname varchar(200) not null,
        port integer,
        file varchar(400) not null,

        -- Inhalt
        content_type varchar(50),
```

[7] Structured Query Language
[8] Die Darstellung ist hier etwas verkürzt; für den vollständigen Code siehe sql/create_tab.sql auf der CD.

```
        content clob(2000000) not logged compact, -- DB2-
spezifisch

        -- Zustand
        http_state integer,
        internal_state integer,

        -- sonstige
        last_fetched bigint,    -- ms seit 1.1.1970, 0 Uhr GMT
        ranking double,
        depth integer,

        constraint pk_webpage primary key (id)
);

-- Indizes über Hostnamen und Prioritäten für die HostQueue
create index ix_hostname on webpage(hostname);
create index ix_ranking on webpage(ranking);

-- View, um URL als Ganzes abfragen zu können
-- (mit symbolischen Zustandsnamen)
--
-- DB2-spezifisch
create view wp_view (id, url, content_type, content,
http_state, internal_state, last_fetched, ranking, depth)
as (
        select w.id, (proto CONCAT '://'
                      CONCAT hostname CONCAT ':'
                      CONCAT RTRIM(CHAR(port)) CONCAT file),
               content_type, content, http_state, name,
               last_fetched, ranking, depth
        from webpage w, state_names s
        where w.internal_state = s.id
);
```

```
create table hyperlink (
        id bigint not null,
        from_id bigint not null,
        to_id bigint not null,
        text varchar(2000),
        type integer not null,
        spans_hosts smallint with default 0,
        spans_sites smallint with default 0,

        constraint pk_hlink primary key (id),
        -- Relationship link-from
        constraint fk_hl_from foreign key (from_id)
                references webpage (id)
                on delete cascade,
        -- Relationship link-to
        constraint fk_hl_to foreign key (to_id)
                references webpage (id)
                on delete cascade
);

-- Indizes über beide Endpunkte des Links
create index hl_from on hyperlink(from_id);
create index hl_to on hyperlink(to_id);
```

4.2.2 OO-Modell

Die Programmstruktur des Crawlers wird objektorientiert entworfen. Abbildung 4.3 auf Seite 106 und Tabelle 4.3 auf Seite 105 zeigen die grundlegenden Klassen im Crawler. Nach einer kurzen Übersicht werden drei Blöcke daraus genauer betrachtet: die HostQueue, der

WorkerPool und die Strategy. Diese Blöcke sind im UML[9]-Diagramm (Abbildung 4.3) durch Rahmen kenntlich gemacht.

Die HostQueue und einige Hilfsklassen (mittlerer Block im Diagramm) verwalten die Informationen darüber, welche Seiten abzurufen sind und stellen dabei das korrekte Zeitverhalten der Crawlers sicher, um eine Überlastung von Servern zu vermeiden.

Die Crawl-Strategie (Strategy, rechter Block) ist dafür verantwortlich, nach dem Eintreffen einer Seite zu bestimmen, ob und welche anderen Seiten, die diese referenziert, ebenfalls einzusammeln sind.

Der WorkerPool (linker Block) schließlich stellt eine Abstraktion für eine Menge von *Threads* dar. Diese Threads werden eingesetzt, um Seitenabrufe parallel auszuführen und so den Durchsatz zu steigern.

Neben den hier vorgestellten Klassen werden noch einige Hilfsklassen benötigt, die im Abschnitt 5 vorgestellt werden.

Ein weiteres Problem ist das Zusammenspiel von OO-Modell und relationaler Datenbank. Darauf wird in Abschnitt 47 eingegangen.

Klasse/*Interface*	**Erläuterung**
HostQueue	Eine Schlange von Host-Objekten, aufsteigend sortiert nach der Zeit des nächstmöglichen Zugriffs.
Host	Beinhaltet Informationen über einen Server sowie eine Schlange von WebPageWork-Objekten, die ausstehende Anfragen repräsentieren.
RobotsTxt	Informationen über die Robots Exclusion auf einem Host.
WebPage	Repräsentiert eine Webseite mit Inhalt, Content-Type, Datum des letzten Zugriffs usw.
WebPageWork	Objekte dieses Typs dienen dazu, das Aktualisieren eines WebPage-Objekts von einem WorkerPool erledigen zu lassen.

[9] Unified Modeling Language

Klasse/*Interface*	**Erläuterung**
WorkerPool	Menge von Threads (Worker), die Work-Objekte ausführen können.
Worker	Thread zur Verwendung im WorkerPool.
Work	Kapselt Code, der vom WorkerPool ausgeführt werden soll.
Strategy	Einheitlicher Zugriffspunkt für das jeweilige StrategyImpl-Singleton
StrategyImpl	StrategyImpl wird von Klassen implementiert, die einen Algorithmus zur Auswahl neuer abzurufender URLs nach Eintreffen einer Seite bereitstellen.
BFSStrategy	Eine Strategie, die ausgehend von einer gefundenen Seite alle URLs in die Warteschlange einfügt, die von dieser Seite referenziert werden. Neue URLs können unterdrückt werden, wenn eine gewisse Tiefe überschritten wird oder wenn die neue URL auf einem anderen Host liegt als die verweisende Seite.
RainbowStrategy	Eine Strategie, die mit Hilfe des Textklassifizierers *Rainbow* die Relevanz von eintreffenden Seiten bewertet und bevorzugt Links von relevanten Seiten verfolgt.

Tabelle 4.3: Klassenübersicht

Die HostQueue

In Abschnitt 4.1 wird ein hoher Durchsatz vom Crawler gefordert. Andererseits ist aber auch ein zeitlicher Mindestabstand zwischen zwei Zugriffen auf einen Server einzuhalten. Dieser scheinbare Widerspruch wird durch die HostQueue gelöst.

Um die zeitlichen Vorgaben einzuhalten, müssen alle auszuführenden Seitenanforderungen (Requests) zentral verwaltet werden. Ebenso muss für jeden Server bekannt sein, *wann* er als nächstes wieder benutzt werden kann. Schließlich muss verwaltet werden, *welcher* Server der nächste ist, auf den wieder zugegriffen werden kann.

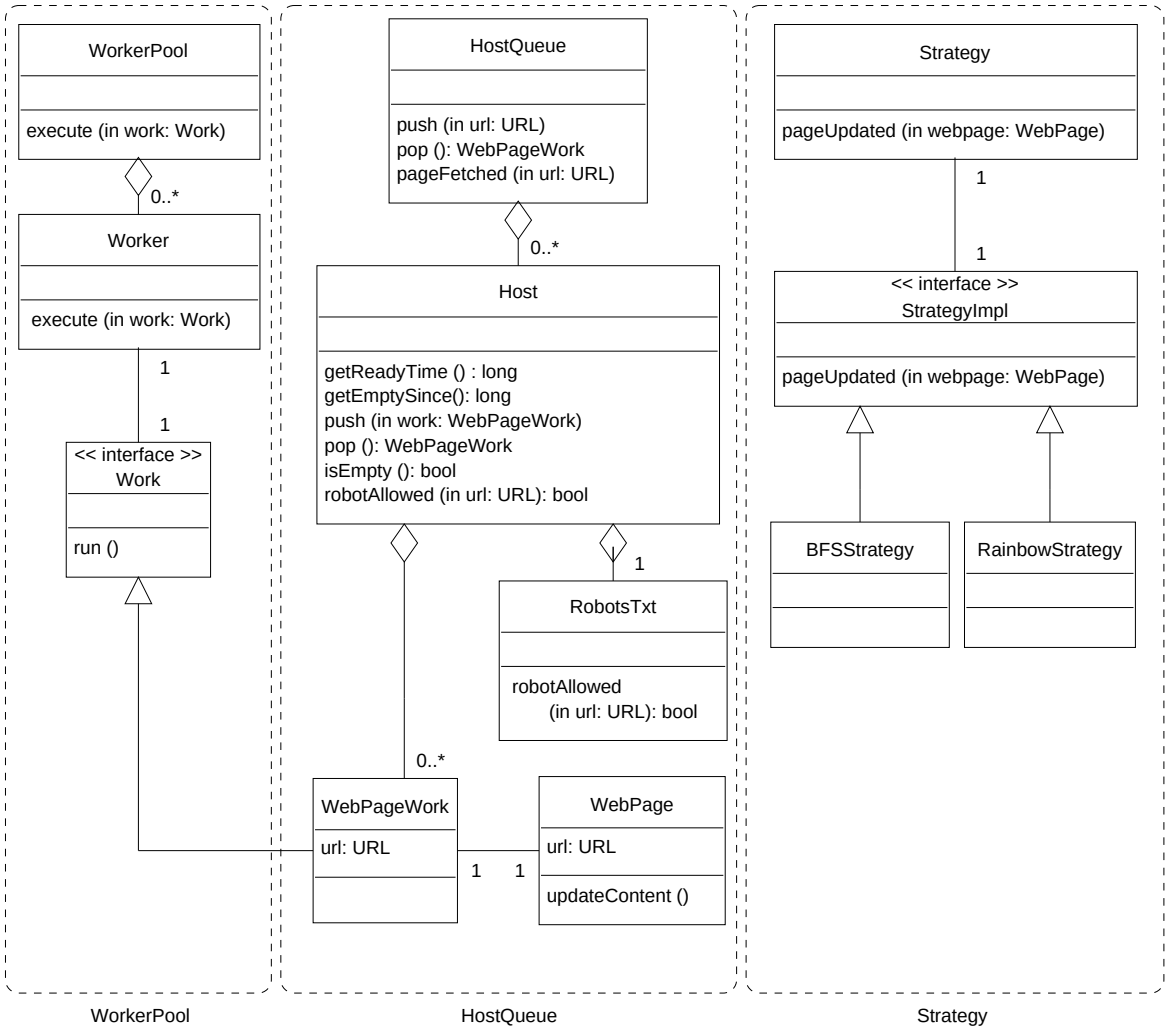


Abbildung 4.3: UML-Klassendiagramm

Abbildung 4.4 zeigt eine schematische Darstellung einer Warteschlange von Warteschlangen, der HostQueue.

In dieser Schlange steht für jeden Host ein Objekt der Klasse Host. Diese Objekte sind nach der frühesten Zeit geordnet, zu der wieder auf diesen Host zugegriffen werden kann (Bereitzeit). In jedem Host-Objekt wiederum ist eine Warteschlange von Requests zu diesem Host gespeichert; diese Schlange kann nach einem beliebigen Kriterium mit Prioritäten versehen sein, beispielsweise nach der erwarteten Relevanz der Seite (siehe z. B. Abschnitt 3.1.2).

Leere Hosts werden an das Ende sortiert, da ein leeres Host-Objekt überhaupt keine Anfragen hat, die derzeit bearbeitet werden können.

Mit dieser Struktur hat nun die HostQueue an ihrer Spitze eine globale früheste Bereitzeit zur Verfügung. Dadurch kann sie jeweils entsprechend lange warten, um die zeitlichen Vorgaben einzuhalten. Beim Einfügen oder Abarbeiten von Requests werden die HostQueue und die ensprechende Host-Warteschlange jeweils auf den korrekten Stand gebracht.

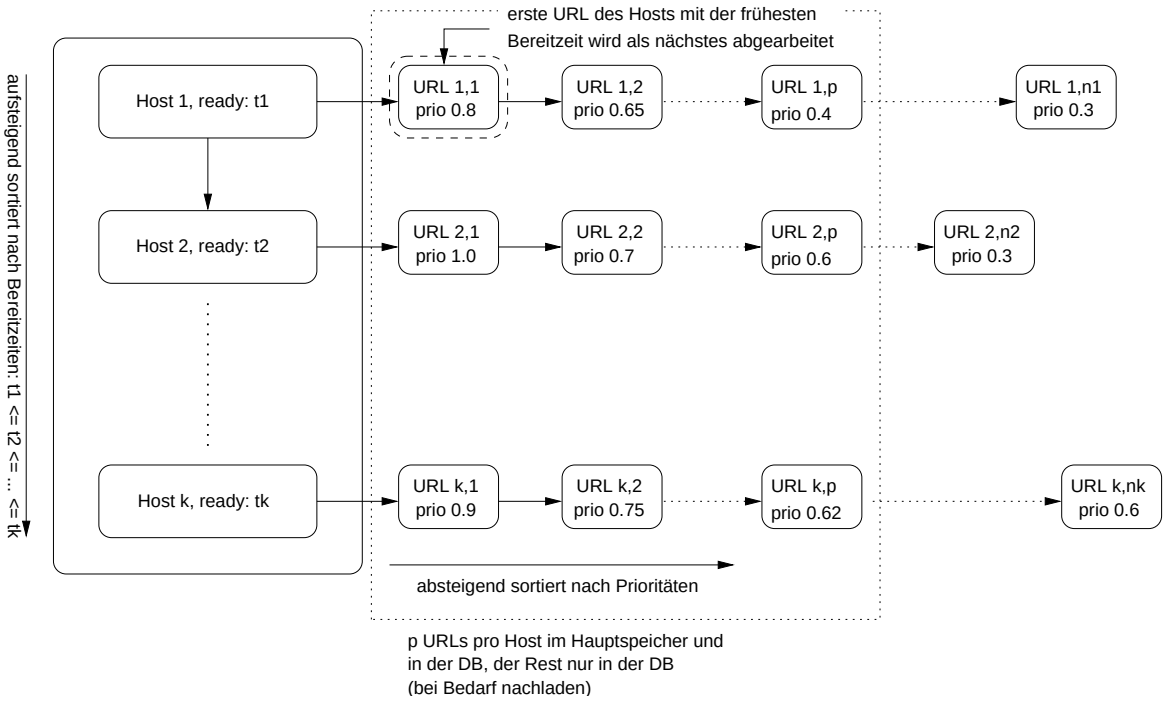


Abbildung 4.4: Die HostQueue

Insgesamt werden also innerhalb eines Hosts die Anfragen nach absteigenden Prioritäten geordnet, und die Hosts mit nicht-leeren Warteschlangen werden reihum abgearbeitet.

Um nach einem Fehler wieder aufsetzen zu können und um Hauptspeicher zu sparen, werden alle Requests in der Datenbank gespeichert; es werden aber jeweils nur p Requests pro Host im Hauptspeicher gehalten. Wenn diese abgearbeitet sind, werden wieder p Requests aus der Datenbank nachgeladen.

Hosts, für die derzeit keine Anfrage aussteht, werden nach einiger Zeit aus der HostQueue entfernt. Dies übernimmt ein eigener Thread (Klasse HostQueue.CleanUpThread), der periodisch aufwacht und jeweils alle leeren Hosts entfernt.

Es ist zu beachten, dass ein leerer Host nicht sofort aus der Schlange entfernt werden kann, da dabei die Information über die Zeit des letzten Zugriffs (getEmptySince()) verlorengeht. Falls unmittelbar nach dem Entfernen ein neuer Request für diesen Host erzeugt würde, könnte der zeitliche Mindestabstand unterschritten werden.

Vergleich mit Mercator Im Abschnitt 2.3.2 wurde beschrieben, wie bei Mercator die Verwaltung ausstehender Anfragen erfolgt. Dort wird mit einer festen Zahl von Schlangen gearbeitet, die jeweils von einem Thread abgearbeitet werden. Die Anfragen werden durch Hashing auf die Schlangen verteilt, so dass Anfragen auf einen Host stets in der selben Schlange bearbeitet werden.

Diese Vorgehensweise kam hier aus zwei Gründen nicht zum Einsatz:

- Es wird nur garantiert, dass nicht zwei Anfragen gleichzeitig auf einen Host ausgeführt werden. Ein zeitlicher Mindestabstand kann nicht gewährleistet werden.
- Wenn Seiten auf nur wenigen Hosts in wenigen Schlangen bearbeitet werden, sind Kollisionen beim Hashing wahrscheinlich. Dadurch bleiben dann einige Schlangen leer, während andere vergleichsweise viele Anfragen enthalten.

  Mercator bearbeitet Seiten von sehr vielen verschiedenen Hosts und benutzt extrem viele Threads (z. B. 500 in [Naj01, Abschnitt 3]), wodurch dieses Problem sich nicht

stark auswirkt. Bei kleineren Crawls und einer geringeren Zahl von Threads wird es wahrscheinlicher, dass diese Situation auftritt.

Dies haben auch die Autoren von Mercator erkannt [Hey99, Abschnitt 4.2] und durch eine dynamische Verteilungsstrategie von Aufträgen zu Schlangen behoben. Allerdings gehen sie nicht näher auf diese Strategie ein.

Diese beiden Probleme erfordern einen anderen Entwurf für die Verwaltung ausstehender Requests, so wie er in der HostQueue realisiert ist.

Die Strategie

Ein Webcrawler wird gestartet mit einer Menge von abzurufenden Seiten. Diese Seiten werden eingesammelt und die enthaltenen Hyperlinks extrahiert. Damit ergibt sich eine neue Menge von Seiten, die man wiederum aufsuchen kann. Die Reihenfolge, in der diese neuen Seiten bearbeitet werden, bestimmt eine *Crawl-Strategie* (s. Abschnitt 3.1).

Um diese Strategie leicht änderbar zu halten, wird der Algorithmus zur Bestimmung neuer aufzusuchender Seiten in ein eigenes Objekt gekapselt. Eine solche Vorgehensweise ist als *Strategy*-Entwurfsmuster (s. 4.2.4) bekannt.

Für mögliche Strategie-Klassen wird ein Interface vorgegeben, das eine oder mehrere Methoden deklariert, die den jeweiligen Algorithmus verwirklichen; im vorliegenden Fall ist das lediglich eine Methode pageUpdated (WebPage), die aufgerufen wird, wenn eine Seite aus dem WWW erfolgreich empfangen wurde.

Im System ist genau *eine* Strategie vorhanden, die in einer Klasse realisiert ist, die das Interface StrategyImpl implementiert. Eine solche Klasse, von der genau eine Instanz existiert, nennt man *Singleton*; siehe auch hierzu wieder 4.2.4.

Problematisch an dieser Stelle ist, dass genau ein Strategie-Objekt vorhanden ist, die Klasse dieses Objekts aber zur Übersetzungszeit nicht feststeht.[10] Der Zugriff auf eine solche

[10] Die Strategie-Klasse kann über die Konfigurationsdatei festgelegt werden.

Singleton-Instanz erfordert aber, dass man auf eine Klassenmethode oder Klassenvariable zugreift; dazu muss man die Klasse kennen.

Um dieses Problem zu umgehen, werden *zwei* Klassen eingesetzt: die Klasse Strategy bietet lediglich einen globalen Zugriffspunkt auf das Strategie-Objekt, angelehnt an das Singleton-Muster. Die Klasse, die den eigentlichen Algorithmus kapselt, implementiert StrategyImpl. Strategy muss vor der ersten Benutzung demnach mit dem einen zu benutzenden StrategyImpl-Objekt initialisiert werden.

Der WorkerPool

Moderne Betriebssysteme bieten als eine weitere Form von Kontrollflüssen neben Prozessen auch sogenannte *Threads* an. Threads laufen dabei in gemeinsamen Adressräumen und erlauben so den Zugriff auf gemeinsame Daten und schnelle Kontextwechsel.

Im Crawler werden Threads eingesetzt, um mehrere Anfragen an Webserver gleichzeitig auszuführen.[11] Da die Geschwindigkeit des Netzwerks und der angesprochenen Server hier der limitierende Faktor ist, kann so der Durchsatz erhöht werden, da nicht das Gesamtsystem auf eine einzige Netzwerkverbindung warten muss.

Da diese Threads zum einen teuer zu erzeugen sind, zum anderen wieder verwendet werden können, werden sie in einem Pool (s. 4.2.4) zusammengefasst.

Es existiert genau eine Instanz der Klasse WorkerPool, die eine Anzahl von Threads, sogenannte Worker, verwaltet. Eine zu erledigende Aufgabe wird als Work-Objekt übergeben. Der WorkerPool wartet, bis einer der Threads frei ist, und übergibt diesem dann das Work-Objekt zur Ausführung.

Diese Technik, eine auszuführende Funktion in ein Objekt zu verpacken, ist als *Command*-Pattern bekannt; siehe dazu auch Abschnitt 4.2.4.

[11] Eine andere Möglichkeit wäre asynchrone Ein-/Ausgabe, aber diese ist, falls sie von Betriebssystem und Programmierumgebung angeboten wird, kompliziert zu benutzen. Java bietet diese Möglichkeit erst ab JDK 1.4, das bei der Erstellung dieser Arbeit noch nicht vorliegt.

4.2.3 Programmablauf

Kontrollfluss

Der Kontrollfluss im Crawler ist eine Endlosschleife. Angenommen, die HostQueue hat eine zu besuchende URL bestimmt, deren Bereitzeit erreicht ist. Dann geschieht Folgendes:

1. Die HostQueue wartet, bis der WorkerPool einen freien Thread hat. Dies ist notwendig, falls die HostQueue mehr bereite Aufträge hat, als freie Threads vorhanden sind. Würde hier nicht gewartet, hätte man neben dem Scheduling durch die HostQueue eine zweite implizite Warteschlange am WorkerPool, die u. U. das Zeitverhalten stören könnte.
2. Die zu erledigende Seitenanforderung wird in Form eines WebPageWork-Objektes an den WorkerPool weitergegeben.
3. Der WorkerPool bestimmt einen freien Worker-Thread und übergibt diesem die Anforderung
4. Der Thread bearbeitet das WebPageWork-Objekt:
   a) Es wird ein WebPage-Objekt mit der entsprechenden URL erzeugt.
   b) Das WebPage-Objekt ruft den Inhalt der entsprechenden Seite vom Webserver ab.
   c) Danach wird die Crawl-Strategie vom Eintreffen der Seite in Kenntnis gesetzt.
   d) Die Strategie bestimmt neue URLs, die abzuarbeiten sind, und schiebt diese in die HostQueue.

Abbildung 4.5 auf der nächsten Seite zeigt den Ablauf als Sequenzdiagramm.

Lebenszyklus einer Seite

Ein webpage-Datenobjekt durchläuft in der Regel mehrere Zustände:

1. Anfangs ist der Zustand einer Seite STATE_INIT. Von der Seite ist nur die Adresse bekannt; sie steht in den Feldern für die URL-Bestandteile.

2. Wird die Seite in die HostQueue eingefügt, um ihren Inhalt aus dem Web zu aktualisieren, wird der Zustand auf STATE_QUEUED gesetzt.

3. Während die Daten aus dem WWW angefordert werden, steht der Zustand auf STATE_FETCHING.

4. Der Zustand STATE_FETCHED zeigt an, dass der Seiteninhalt erfolgreich aktualisiert worden ist.

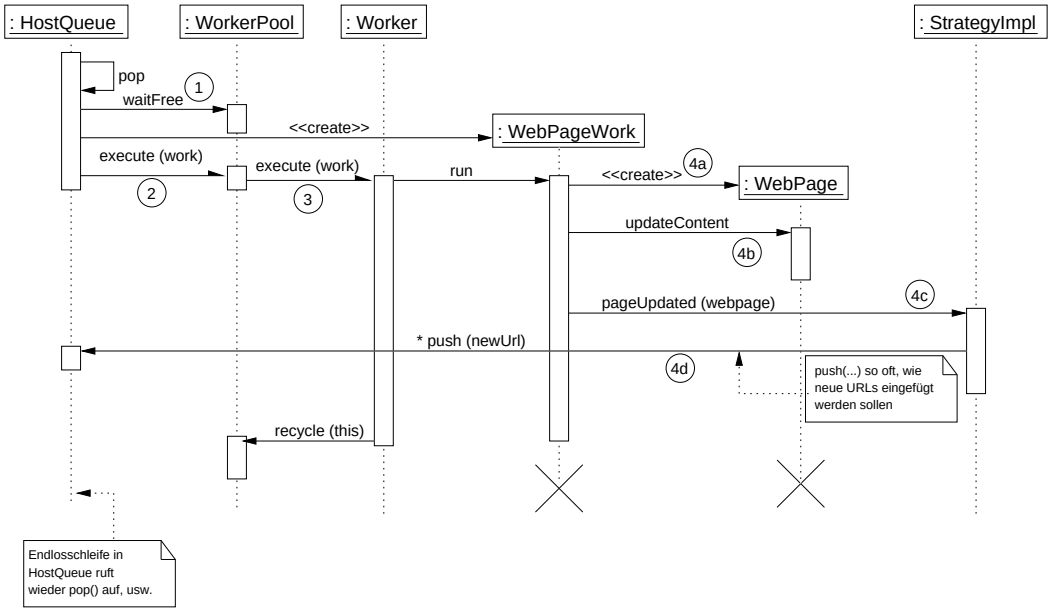


Abbildung 4.5: UML-Sequenzdiagramm

Daneben kann eine Seite noch Zustände annehmen, die einen aufgetretenen Fehler anzeigen:

- STATE_FAILED, wenn ein Fehler auf Serverseite (Dokument nicht gefunden, etc.) auftritt.
- STATE_UNSUPPORTED, wenn der Content-Type der Seite vom Crawler nicht verarbeitet werden kann, etwa bei Bilddateien.
- STATE_INTERNAL_ERROR, wenn ein interner Fehler im Crawler auftrat.
- STATE_ROBOT_EXCLUDED, wenn das Aufrufen der Seite durch das Robots Exclusion Protocol verboten wurde.
- STATE_TOO_MANY, wenn bereits zu viele Seiten von dem Host dieser Seite abgerufen wurden.

4.2.4 Entwurfsmuster

Wiederverwendung von bereits erstellten Lösungen spielt in der Softwareentwicklung seit jeher eine große Rolle; üblicherweise geschieht dies in der Form von Programmbibliotheken, die unter einer dokumentierten Schnittstelle Code für oft benötigte Funktionen bereitstellen.

Seit einigen Jahren gibt es die Idee, Wiederverwendung auch auf einer abstrakteren Ebene zu betreiben. Das Ziel dabei ist es, Erfahrungen und erprobte Lösungen im Entwurf von Software als sogenannte „Entwurfsmuster“ (engl. *design patterns*) zu dokumentieren. Im ihrem Standardwerk zu diesem Thema beschreiben Gamma u. a. dies wie folgt:

> Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is to capture design experience in a form that people can use effectively.
>
> [Gam95, S. 2]

Einige dieser Muster werden an verschiedenen Stellen der vorliegenden Arbeit verwendet. Sie sollen hier kurz aufgeführt werden; die Art der Präsentation ist dabei an [Gam95] angelehnt, woraus auch die Muster *Command*, *Singleton* und *Strategy* entnommen sind. Das Muster *Object Pool* ist beschrieben in [Gra98].

Command

Ziel: Kapseln eines Auftrags als Objekt.

Motivation: Manchmal ist es notwendig, einen Auftrag an ein Objekt zu übergeben, ohne vorher genaueres über den Auftrag oder das Empfängerobjekt zu wissen.

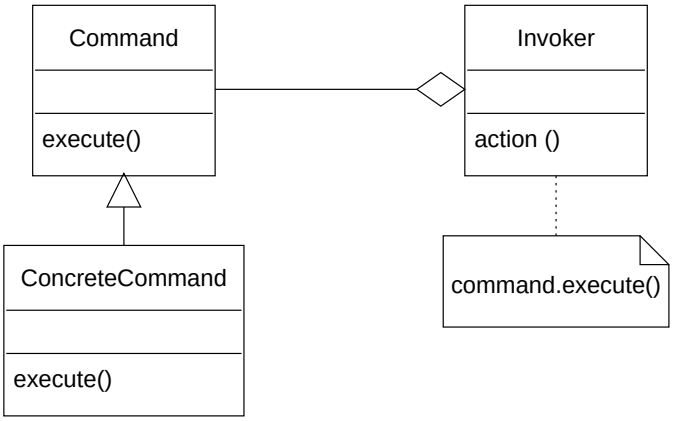
Anwendungsbeispiele:

HostQueue zeitliches Entkoppeln von Erteilung und Ausführung eines Auftrags durch eine Warteschlange

GUI[12]*-Toolkits* Kapseln der Aktionen, die an Menüeinträge oder Buttons gebunden werden

Undo, Transaktionen Bereitstellen einer Undo-Funktionalität oder von Transaktionseigenschaften

Struktur:



Beispielcode:

```
interface Command {
     public void execute();
}

// Ein ExitCommand könnte beispielsweise an eine
// Schaltfläche in einem GUI gebunden werden.
class ExitCommand implements Command {
     public void execute() {
          System.exit(1);
     }
}
```

Singleton

Ziel: Garantieren, dass von einer gegebenen Klasse genau eine Instanz existiert, und Bereitstellen eines globalen Zugriffspunktes auf diese Instanz.

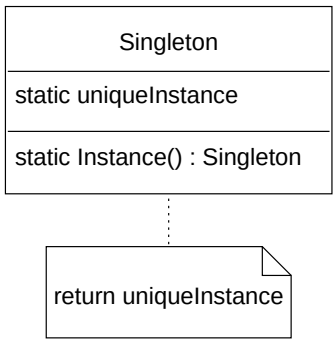
Motivation: Oftmals ist es notwendig, von einer Klasse genau eine Instanz zu haben. Dies ist beispielsweise der Fall, wenn die Klasse eine Systemresource repräsentiert, die genau einmal vorhanden ist, etwa einen Drucker. Dabei soll die Klasse weiterhin offen sein für Vererbung, so dass es nicht möglich ist, die Funktionalität über Klassenmethoden zu realisieren.

Anwendungsbeispiele:

Konfigurationsdaten Im Crawler stellt die Klasse Config einen globalen Zugriffspunkt auf die Konfigurationsparameter für den Crawler bereit, die aus einer Properties-Datei gelesen werden.

Systemweite Objektpools Von den Objekt-Pools WorkerPool und ConnectionPool ist jeweils genau eine Instanz vorhanden.

Struktur:



Beispielcode:

```
class MySingleton {
        static uniqueInstance;
```

```
        // static-Block wird bei erster
        // Benutzung der Klasse ausgeführt
        static {
                uniqueInstance = new MySingleton ();
        }

        public MySingleton Instance () {
                return uniqueInstance;
        }

        // "private"-Konstruktor kann nicht von aussen
        // aufgerufen werden!
        private MySingleton () { ... }
}
```

Strategy

Ziel: Kapseln einer Familie von Algorithmen, so dass diese untereinander austauschbar sind oder sich ändern können, ohne dass der benutzende Code geändert werden muss.

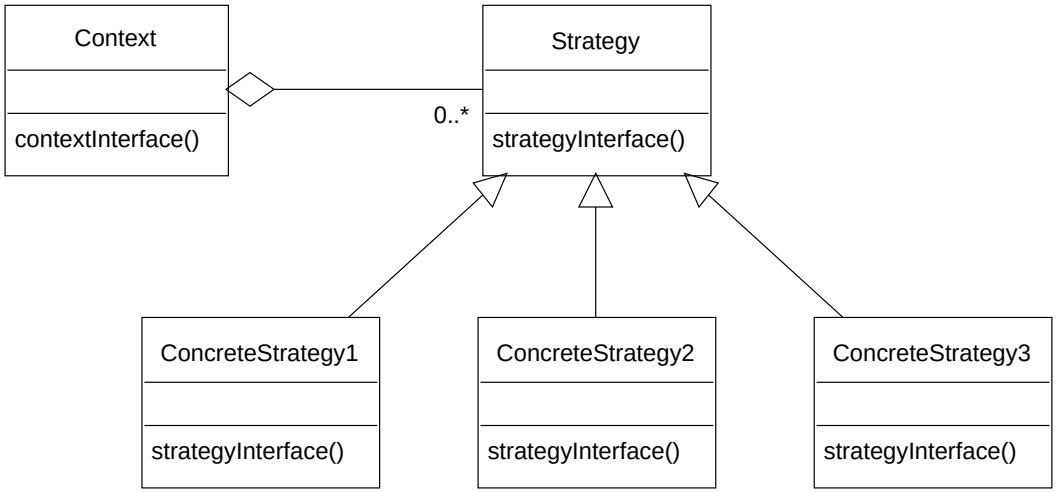
Motivation: Oftmals existieren für bestimmte Aufgaben mehrere Algorithmen. Dabei soll es einfach möglich sein, einen Algorithmus gegen einen anderen auszutauschen. Code außerhalb des eigentlichen Algorithmus' soll davon unberührt bleiben.

Anwendungsbeispiele:

Suchstrategien austauschbare Strategien BFSStrategy und RainbowStrategy im Crawler

Trade-off verschiedene Varianten von Algorithmen, die das Abwägen zwischen Zeit- und Platzbedarf ermöglichen

Struktur:



Beispielcode: (angelehnt an den Crawler)

```
interface StrategyImpl {
    void pageUpdated (WebPage newPage);
}

class BFSStrategy
  implements StrategyImpl
{
   ...
}

class Crawler {
    ... {
        StrategyImpl theStrategy = new BFSStrategy(...);
        ...
```

```
        // neue Seite eingetroffen
        theStrategy.pageUpdated (newPage);
    }
}
```

Object Pool

Ziel: Verwalten einer Menge von Objekten, deren Gesamtzahl beschränkt werden soll und/oder deren Erzeugung teuer ist.

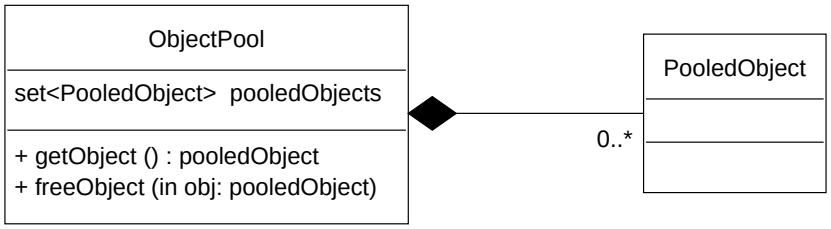
Motivation: Oft werden mehrere gleichartige Objekte benötigt, die aufwändig zu erzeugen sind oder von denen nicht mehr als eine bestimme Anzahl existieren soll. Wenn diese Objekte nach Benutzung wieder verwendet werden können, können sie in einem Pool verwaltet werden.

Anwendungsbeispiele:

WorkerPool Die Erzeugung von Threads ist eine teure Operation. Außerdem soll die Gesamtzahl von Threads beschränkt werden, um die Systemauslastung steuern zu können.

ConnectionPool Das Aufbauen von Datenbankverbindungen ist teuer; zudem sollten nicht zu viele Verbindungen auf einmal geöffnet sein, damit z. B. nicht zu viele Transaktionen wegen Verklemmungen abgebrochen werden.

Struktur:



Beispielcode:

```
interface PooledObject {
   // wird implementiert von Objekten im Pool
   ...
}

class ObjectPool {
        List pooledObjects;

        ObjectPool {
                // pooledObjects mit n Objekten der
                // Klasse PooledObject füllen
                ...
        }

        synchronized Object getObject () {
                // warten auf ein freies Objekt
                while (pooledObjects.isEmpty()) {
                        wait();
                }
                // erstes Objekt aus der Liste zurückgeben
                return (PooledObject) pooledObjects.remove(0);
```

```
        }

        synchronized void freeObject (PooledObject o)
                pooledObjects.add (o);
                // wartende Threads aufwecken, da jetzt
                // freies Objekt zur Verfügung steht
                notifyAll();
        }
}

// Benutzung:
...
PooledObject o = pool.getObject();
o.doSomething();
...
pool.freeObject (o);
```

5 Implementierung des Crawlers

Dieses Kapitel befasst sich mit Implementierungsaspekten des Crawlers, der im vorigen Kapitel entworfen wird.

Zunächst wird in Abschnitt 5.1 motiviert, warum Java als Programmiersprache zum Einsatz kommt.

Das Hauptproblem bei der Implementierung ist die Zusammenarbeit von objektorientiertem Programm und relationaler Datenbank. Abschnitt 5.2 zeigt verschiedene Möglichkeiten auf, wie dieses Problem gelöst werden kann. Daneben beschreibt Abschnitt 5.3 einige technische Gesichtspunkte der Datenbankanbindung.

Abschnitt 5.4 schließlich beschreibt die Anbindung des Textklassifizierers Bow (s. auch Abschnitt 3.1.2) an den Crawler.

5.1 Wahl der Programmiersprache

Als Implementierungssprache für den Crawler wurde Java gewählt. Dafür sprechen unter anderem folgende Gesichtspunkte:

Objektorientierung Durch den objektorientierten Ansatz beim Entwurf wird nahegelegt, zur Implementierung eine OO-Sprache wie Java zu verwenden.

Einfache Programmierung und Robustheit Die Sprache Java ist recht einfach gehalten und vergleichsweise frei von Überraschungen für den Programmierer. In Verbindung mit der Laufzeitumgebung und den Werkzeugen aus dem Java Development Kit (JDK) wird es leicht gemacht, robuste Programme zu schreiben und Fehler aufzufinden.

So bietet Java z. B. einen Mechanismus zur Ausnahmebehandlung (Exception Handling), der eine geordnete Behandlung von Fehlersituationen im Programm erzwingt. Im Fehlerfall wird eine *Ausnahme* erzeugt, die an übergeordneten Teile im Kontrollfluss (umgebender Block, aufrufende Prozedur) weitergereicht wird und schließlich behandelt werden *muss*. Dadurch wird ein defensiver Programmierstil erzwungen.

Das JDK enthält weiterhin einen *Debugger*, um Programme während des Ablaufs beobachten zu können, sowie einen *Profiler*, der Auskunft über das Zeitverhalten sowie die Speichernutzung eines laufenden Programms geben kann.

Ausrichtung auf Internet-Programmierung Java war von Anfang an zur Programmierung im Internet-Umfeld gedacht. Entsprechend einfach gestaltet sich die Benutzung der Bibliotheken für den Netzwerkzugriff.

Vielzahl von Bibliotheken und APIs Von Sun – den Entwicklern von Java – und von vielen anderen Anbietern gibt es eine sehr große Vielfalt an Bibliotheken und entsprechenden Application Programming Interfaces (APIs) für fast jeden erdenklichen Zweck, viele davon kostenlos und im Sourcecode. Als Beispiel seien hier JDBC[1] und SQLJ für den Datenbankzugriff, OpenXML zum Verarbeiten der gesammelten Webseiten oder Log4J zum komfortablen Logging genannt.

Multithreading Schon in der Sprachdefinition von Java ist die Möglichkeit zum Multithreading verankert; dazu ist u. a. ein Monitor-Konzept [Hoa74] zur Synchronisation vorhanden. Dieses erlaubt es, Methoden oder Codeabschnitte innerhalb von Methoden als „synchronisiert“ zu kennzeichnen; es ist dann für jedes Objekt ein wechselseitiger Ausschluss dieser sychronisierten Bereiche sichergestellt.

Gegenüber nachträglich „aufgesetzten“ Bibliotheken, wie sie z. B. für C/C++ angeboten werden, ist dieses in der Sprache integrierte Multithreading erheblich einfacher und übersichtlicher zu nutzen.

[1] Java Database Connectivity

Als Nachteile der Entwicklung in Java haben sich herausgestellt:

Performanceprobleme In Bezug auf Speicherverbrauch und Ausführungsgeschwindigkeit kann Java als interpretierte Sprache sicherlich nicht immer mit Sprachen wie C mithalten. Der Geschwindigkeitsaspekt tritt jedoch in der vorliegenden Situation in den Hintergrund: das Programm verbringt ohnehin die meiste Zeit in Netzwerk- und Datenbankzugriffen; deren Geschwindigkeit hat aber nichts mit der Implementierungssprache des Crawlers zu tun. Zudem verfügen die eingesetzten JDKs mittlerweile über sogenannte Just-In-Time-Compiler, die zur Laufzeit Teile des Java-Bytecodes in Maschinensprache übersetzen können.

Wie Abschnitt 3.8 erläutert, war der Speicherverbrauch einer der Gründe, für die Auswertung der Graphen *nicht* Java zu verwenden.

Probleme der Ausführungsumgebungen Es kommen zwei Versionen des JDK 1.3 von Sun und IBM zum Einsatz. Wie fast alle größeren Programme sind auch diese beiden nicht frei von Fehlern; bei beiden kommt es gelegentlich bei hoher Last – viele Threads, Netzwerk- und Datenbank-Verbindungen – zu Abstürzen (core dumps). Dennoch hält sich der dadurch verursachte Schaden in Grenzen. Abschnitt 49 beschreibt Maßnahmen, um solche Abstürze zu behandeln.

5.2 Abbildung vom objektorientierten in das relationale Modell

Ein immer wieder erkanntes Problem bei der Zusammenarbeit von objektorientierten Sprachen und relationalen Datenbanken ist der entstehende „impedance mismatch“[2] [Amb00, Zim00, Yod98].

Daher bedarf es einer Abbildung vom objektorientierten Modell in das relationale Datenmodell, so dass die Änderungen von Objekten in der Implementierungssprache entsprechend in der Datenbank widergespiegelt werden. Solche Abbildungen werden im Allgemeinen *O/R-Mapping* genannt.

[2] *impedance mismatch* ist in diesem Zusammenhand mit „Reibungsverlust“ frei zu übersetzen.

Für die Realisierung einer solchen Anbindung zwischen objektorientiertem und relationalem Modell gibt es verschiedene Möglichkeiten, die für die vorliegende Arbeit in Frage kommen; auch in der Literatur werden sehr verschiedene Meinungen zu diesem Thema vertreten. Hier sollen die Ansätze vorgestellt werden, um die schließlich getroffene Wahl zu motivieren.

- *Direkter Zugriff auf die Datenbank* mittels dynamischem oder statischem SQL (in Java: JDBC oder SQLJ[3]). Die Daten werden in der Datenbank modelliert. Die Klassen des Java-Programms bieten bestenfalls eine Hilfsfunktion; die Datensätze in der Datenbank werden nicht durch Java-Objekte repräsentiert.

  Vorteile:
  - hohe Performance
  - keine zusätzliche Software nötig

  Nachteile:
  - unübersichtlich
  - Wartung schwierig

- *CRUD-Operationen:* Yoder u. a. benennen in [Yod98] einen minimalen Satz von Operationen, die Objekte unterstützen müssen, damit sie sich selbst persistent machen können. Diese sind:

***C**reate*	Objekt in der Datenbank erzeugen
***R**ead*	Objekt aus der Datenbank lesen
***U**pdate*	geänderten Zustand des Objekts in die Datenbank schreiben
***D**elete*	Objekt aus der Datenbank löschen

  Eine bekannte Anwendung dieser Technik sind zum Beispiel Enterprise JavaBeans mit Bean Managed Persistence [Sun00].

  Vorteile:
  - einfach zu realisieren
  - keine zusätzliche Software nötig

  Nachteile:
  - jede Klasse muss Datenbank-Operationen anbieten
  - Beziehungen zwischen Objekten schwierig zu handhaben

[3] Embedded SQL for Java

- *Generische Zugriffsschicht:* eine sog. Zugriffsschicht (*access layer*) kann die O/R-Abbildung völlig transparent machen. Die herzustellenden Beziehungen zwischen Klassen- und Relationenmodell werden der Zugriffsschicht beschrieben (z. B. durch ein XML-Dokument). Danach kann der Programmierer persistente Objekte beinahe genau so verwenden wie gewöhnliche Objekte der Implementierungssprache. Die Zugriffsschicht sorgt mit Hilfe dieser Beschreibung dafür, dass sich alle Änderungen der Objekte – unter Beibehaltung der Transaktionseigenschaften – in der Datenbank widerspiegeln.

  Vorteile:

  – saubere Trennung von OO-Modell und Datenbankprogrammierung
  – Wartung erheblich erleichtert

  Nachteile:

  – Performanceprobleme
  – Zugriffsschicht ist selber ein komplexes Programmpaket

  Ein weiteres Problem mit generischen Zugriffsschichten ist, dass kommerzielle Produkte wie z. B. *TopLink* von WebGain oder *JavaBlend* von Sun sehr teuer sind und damit für die vorliegende Arbeit ausfallen; als einziger ernsthafter kostenloser Vertreter kommt *Castor* von der Open-Source-Initiative ExoLab in Frage. Versuche damit waren allerdings nicht zufrieden stellend im Hinblick auf die Ausführungsgeschwindigkeit; schwerwiegender jedoch ist das Problem, dass Castor noch unfertig erscheint. Dies äußert sich z. B. darin, dass auf der einschlägigen Mailing-Liste täglich sehr viele Fragen, Probleme und Patches veröffentlicht werden.

O/R-Mapping in der Literatur

Die Meinungen darüber, ob und wie ein O/R-Mapping zu gestalten ist, gehen auseinander. Ambler beispielsweise argumentiert recht dogmatisch für den Einsatz einer generischen Zugriffsschicht:

> You need to encapsulate your database. [...] You need to let your class diagram drive your database design. You shouldn't wrap a legacy database. If you ignore

> this advice you risk running into serious trouble.
>
> [Amb00, S. 28]

Im Gegensatz dazu raten Lawson u. a. unbedingt davon ab, *irgendeine* Form von Zugriffsschicht zu benutzen:

> The guideline is simple: never use generic I/O modules. In fact, never do generic anything with DB2, since generic means less than the best performance. We are trying to shorten code length, not lengthen it.
>
> [Law00, S. 47]

Wahl der Strategie für das O/R-Mapping

Wie vorher gezeigt, gehen die Meinungen zur Art und Weise, wie Datenbanken an die Programmlogik angebunden werden sollen, recht weit auseinander.

Folgende Beobachtungen beeinflussen die Wahl der O/R-Mapping-Strategie in dieser Arbeit:

- Die benutzten Daten haben eine sehr einfache Struktur. Die betrachteten Daten bestehen nur aus den Entity Sets webpage und hyperlink; nur diese werden persistent gespeichert.
- Es bestehen nur einfache Beziehungen zwischen den Datenobjekten.
- Es werden Millionen von Webseiten und Links bearbeitet. Daher ist Performance ein wichtiger Faktor.
- Daten werden nur an wenigen isolierten Stellen mit einfachen Operationen manipuliert (etwa: „Setze den Zustand dieser Seite auf STATE_QUEUED“).
- Die Datenobjekte haben nur wenig eigene Logik.

Angesichts dieser Faktoren wurde folgende Strategie gewählt:

- Für einfache Manipulationen, wie etwa das Umsetzen eines Zustandsflags, wird direkt mit eingebettetem SQL (SQLJ, siehe Abschnitt 6.3.3) auf die Datenbank zugegriffen.

- Es gibt einen relativ eng begrenzten Bereich im Programmablauf, wo Webseiten als Objekt agieren und eigene Logik einbringen, nämlich das Abrufen von Seiten von einem Server und darauf folgend das Extrahieren von Hyperlinks aus ihrem HTML-Code (Klasse WebPage). Dort wird der CRUD-Ansatz umgesetzt. Hyperlink-Objekte entstehen nur im Kontext der Webseite, in der sie enthalten sind, und sie benötigen keine eigene Logik. Deshalb werden sie als innere Klasse von WebPage realisiert.

  Damit die Klasse WebPage nicht mit Datenbankoperationen überfrachtet wird, werden diese von einer WebPageFactory bereit gestellt. Diese ist zuständig für die Erzeugung von WebPage-Objekten und deren Abgleich mit der Datenbank, stellt also die CRUD-Operationen zur Verfügung.

5.3 Technische Aspekte der Datenbankanbindung

Neben den Entscheidungen bezüglich des O/R-Mappings gibt es auch noch eher technische Gegebenheiten, die beim Datenbankeinsatz zu berücksichtigen sind.

5.3.1 Objekt-IDs

Alle Datenobjekte in der Datenbank besitzen als Primärschlüssel eine systemweit eindeutige Identifikationsnummer, die sog. *Objekt-ID*. In Anlehnung an [Yod98] wurde dazu ein OIDManager implementiert, der für die Vergabe von eindeutigen IDs verantwortlich ist. Der OIDManager stützt sich dabei auf eine Tabelle in der Datenbank (oidtable), um den Stand des ID-Zählers persistent zu halten.

Um nicht für jede neue ID einen Datenbankzugriff durchführen zu müssen, werden jeweils ganze Blöcke von Nummern zwischen zwei Zugriffen auf oidtable vergeben.

Die Vorteile dieser Vorgehensweise sind:

- Systemweit eindeutige IDs erleichtern mitunter die Identifikation eines Objekts, da nicht zwischen Webseite Nr. x und Hyperlink Nr. x unterschieden werden muss, sondern nur ein eindeutiges Objekt x existiert.
- Zwar bieten alle gängigen Datenbank-Management-Systeme Möglichkeiten zur Erzeugung laufender Nummern, aber diese Funktionalität ist nicht im SQL-Standard verankert und variiert von System zu System.
- Das verwendete System DB2 generiert laufende Nummern nur pro Tabelle; es existieren keine Zähler, die über mehrere Tabellen genutzt werden könnten – anders als etwa in Oracle.

5.3.2 Connection Pooling

Im Vergleich zu den eigentlichen Datenbankoperationen ist das Herstellen einer Verbindung zur Datenbank eine teure Operation – das Auf- und Abbauen einer Datenbankverbindung dauert in der verwendeten Umgebung 700-800 ms.

Um dieses Problem zu mildern, wird wie beim WorkerPool in Abschnitt 43 ein Objekt-Pool eingesetzt.

Die Klasse ConnectionPool bietet als Singleton Zugriff auf ein Objekt der Klasse ConnectionPoolImpl, das das eigentliche Pooling realisiert. (Zu dieser Trennung in ConnectionPool und ConnectionPoolImpl siehe auch Abschnitt 42. Dort wird das gleiche Problem mit Strategy und StrategyImpl besprochen.)

Es wurden Pools mit unbeschränkter (UnboundedConnectionPoolImpl) und beschränkter (BoundedConnectionPoolImpl) Größe implementiert, letzterer, um die Belastung der Datenbank steuern zu können. Als Besonderheit ist zu bemerken, dass einige Bestandteile

des Systems – speziell die HostQueue und der OIDManager – unter allen Umständen eine Verbindung bekommen *müssen*, um Verklemmungen zu vermeiden. Deshalb bieten alle Implementierungen eine Möglichkeit, dies zu erzwingen; normalerweise kann die Anforderung einer Verbindung blockieren, bis wieder eine Verbindung frei ist.

5.4 Anbindung des Textklassifizierers *Bow*

Der Textklassifizierer *Bow* steht zunächst in Form einer Programmnbibliothek zur Verfügung.

Um diese interaktiv nutzbar zu machen, wird ein Frontend namens *rainbow* mitgeliefert. Dieses nimmt Text von der Konsole entgegen und gibt Wahrscheinlichkeiten für die Zugehörigkeit zu den trainierten Klassen auf die Konsole aus. (S. Abschnitt 3.1.2)

Als Besonderheit bietet *rainbow* eine Server-Funktion, in der es Daten auf einem Netzwerksocket entgegennimmt und das Ergebnis auf diesen wieder ausgibt. Zur Anbindung an das Java-Programm wird eine Klasse RainbowClassifier entwickelt, die diesen Netzwerkzugriff und die Klassifizierung kapselt.

Das Training des Klassifizierers erfolgt über die Kommandozeile, die Trainingsdaten stehen dabei in Dateien.

6 Praxis

6.1 Praktischer Betrieb der Software

Im Folgenden soll kurz beschrieben werden, wie der Crawler praktisch benutzt wird.

6.1.1 Einsatz des Crawlers

Konfiguration

Der Crawler ist über eine Reihe von Parametern zu konfigurieren. Diese werden als Java-Properties, also Paare der Form „name=wert“, in der Datei cscrawler.properties eingetragen. Tabelle 6.1 zeigt eine Übersicht der möglichen Properties. Die Zahlen in Klammern geben Abschnitte an, in denen genauere Erläuterungen zu finden sind.

Die Properties aus der Konfigurationsdatei können beim Aufruf des Programms überschrieben werden, indem neue Werte mit der Option -D der JVM[1] übergeben werden:

```
java -DdbPassword=neuesPasswort de.cs75.crawler.Main
```

Mit diesem Aufruf würde z. B. ein Datenbank-Passwort übergeben.

[1] Java Virtual Machine

Property	Typ	Erläuterung
storeContent	boolean	gibt an, ob der Seiteninhalt in der Datenbank gespeichert werden soll
numberOfWorkers	int	Anzahl der Worker-Threads (43)
pooledConnections	int	Anzahl der Datenbankverbindungen, die offen gehalten werden (5.3.2)
maxConnections	int	maximale Anzahl von Datenbankverbindungen, die geöffnet werden, wenn das Öffnen nicht erzwungen wird (5.3.2)
maxConnAge	long	maximales Alter einer Datenbankverbindung in ms; ältere Verbindungen werden nicht wieder in den Pool aufgenommen (5.3.2)
timeBetweenHits	long	minimales Zeitintervall zwischen zwei Anfragen auf einem Host in ms (42)
cleanupInterval	long	Zeitintervall zwischen Aufräumvorgängen in der HostQueue (42)
oidRange	long	Größe des OID-Bereiches, der jeweils vergeben wird (5.3.1)
bfsDepth	int	maximale Tiefe der Suche; Startseiten haben Tiefe 0; -1 bedeutet unbeschränkte Tiefe (3.1)
bfsStayOnHost	boolean	gibt an, ob der Crawler auf einem Host bleiben soll (true), oder ob Hostgrenzen überschritten werden dürfen
dbDriver	String	Klassenname des Datenbanktreibers (5.3.2)
dbUrl	String	JDBC-Datenbank-URL (5.3.2)
dbUser	String	Datenbank-Benutzername (5.3.2)
dbPassword	String	Datenbank-Passwort (5.3.2)
threadMonitorSleepTime	long	Wartezeit zwischen zwei Aufrufen des ThreadMonitors (51)

Property	Typ	Erläuterung
watchDogInterval	long	Wartezeit zwischen zwei Aufrufen des WatchDogs (51)
robotsMaxAge	long	maximales Alter der Informationen über Robots Exclusion, bevor diese erneut angefordert werden (4.1.2)
pagesPerHost	int	maximale Anzahl von Seiten, die von einem Host heruntergeladen werden (4.1.1)
strategy	String	Java-Klassenname der zu verwendenden Strategie (implementiert StrategyImpl) (42)
rainbowHost	String	Hostname des Rainbow-Query-Servers (5.4)
rainbowPort	int	Port des Rainbow-Query-Servers (5.4)

Tabelle 6.1: Konfigurationsparameter

Konfiguration des Logging

In Abschnitt 6.3.1 wird motiviert, warum Logging für dieses Programm eine wichtige Diagnose- und Überwachungsmöglichkeit ist.

Hier soll nur kurz auf die Arbeitsweise von des verwendeten Logging-Pakets Log4j eingegangen werden. Eine genaue Beschreibung der Konfiguration findet sich in der Log4j-Dokumentation [Apa01]. Die Konfigurationsdatei für Log4j heißt hier log.lcf.

Die wichtigsten Konzepte von Log4j sind:

Category Jede Log-Ausgabe gehört zu einer Kategorie. Kategorien sind hierarchisch über ihren Namen verschachtelt; de.cs75 ist beispielsweise eine Subkategorie von de. Einstellungen von übergeordneten Kategorien werden vererbt, wenn für eine Subkategorie nichts vorgegeben ist. Es bietet sich an, vollständige Java-Klassennamen als

Kategorien zu übernehmen, so dass jede Klasse ihre eigene Kategorie hat.

Anmerkung: Ein Idiom bei der Verwendung von Log4j ist

```
package de.cs75.crawler;
...
public class Main {
    static Category logCat =
        Category.getInstance (Main.class.getName());
    ...
}
```

Damit erhält man eine Kategorie logCat, die dem vollständigen Klassennamen von de.cs75.crawler.Main entspricht.

Appender Ein Appender ist eine „Daten-Senke“, auf die Meldungen ausgegeben werden können. Es gibt Appender für Dateien, Netzwerkverbindungen, für die Konsole, und noch einige mehr.

Layout Ein Layout legt fest, welche Bestandteile einer Log-Meldung in welcher Form ausgegeben werden; die Beschreibung ähnelt der eines printf-Formatstrings in C.

Configurator Ein Configurator dienst als Zugriffspunkt für die Konfiguration des gesamten Logging-Systems. Er legt fest, wie Kategorien, Appender und Layouts kombiniert werden.

Neben dem BasicConfigurator, der durch Methodenaufrufe eingerichtet werden kann, gibt es Configurator-Klassen, die durch Properties-Dateien oder XML[2]-Dokumente gesteuert werden.

Für den Crawler wird die Konfiguration über eine Properties-Datei log.lcf verwendet.

Vorbereiten der Datenbank

Die erforderliche Tabellenstruktur in der Datenbank kann mit dem SQL-Skript sql/createTab.sql angelegt werden:

[2] Extensible Markup Language

```
db2 -vtf sql/createTab.sql
```

Trainieren von Bow

Falls die fokussierte Strategie (s. 3.1.2) benutzt werden soll, muss der Textklassifizierer Bow trainiert werden. Dazu werden Dokumente in zwei Verzeichnisse cspages (*computer science pages*, relevante Dokumente) und random (zufällige, nicht relevante Dokumente) als HTML-Dateien vorgegeben. Bow wird mit

```
rainbow -i -H cspages random
```

trainiert. Der Name cspages für die relevanten Seiten ist wichtig, da die fokussierte Strategie diesen Namen zur Klassifizierung benutzt.

Einfügen von Startseiten in die Datenbank

Der Crawler geht beim Traversieren des WWW von einer Menge von Startseiten aus. Diese müssen in der Datenbank mit der Tiefe 0 und im Zustand STATE_QUEUED eingefügt werden.

Diese Aufgabe erledigt das Programm InsertURL, das aufgerufen wird mit

java de.cs75.crawler.InsertURL <*url 1*> ... <*url n*>

Programmablauf und Überwachung

Starten des Servers für die Textklassifikation Falls die fokussierte Strategie zum Einsatz kommen soll, muss der Klassifizierungs-Server rainbow gestartet werden mit

```
rainbow --query-server=<port> -H
```

Der Rechner, auf dem der Klassifizierer läuft, und die Portnummer müssen in der Konfigurationsdatei (Abschitt 6.1.1) entsprechend angegeben werden.

Programmstart Nachdem Startseiten in die Datenbank eingefügt sind, kann der Crawler mit

```
java -Xmx250M de.cs75.crawler.Main
```

gestartet werden.[3]

Überwachung Je nach Konfiguration des Logging wird der Crawler Meldungen entweder auf die Konsole oder in Logdateien ausgeben, die man zur Überwachung seiner Tätigkeit mitlesen kann.

Mechanismen zum Abfangen von Abstürzen In der Praxis hat sich herausgestellt, dass gelegentlich der Crawler, die ganze JVM oder auch der komplette Rechner – z. B. durch Stromausfall – abstürzt. Da zumindest die letzten beiden Ereignisse außerhalb des Einflusses dieser Arbeit liegen, sind einige Überwachungsmechanismen vorgesehen, damit das System möglichst lange läuft:

Verlustfreies Wiederaufsetzen Im Falle eines Programmabbruchs oder eines Absturzes werden möglicherweise Seiten in den Zuständen STATE_FETCHING oder STATE_QUEUED verbleiben. Diese Zustände stehen persistent in der Datenbank, so dass bei einem Neuaufsetzen solche Seiten wieder in die HostQueue eingefügt werden können. (S. auch Abschnitt 4.2.3.)

Watchdog-Thread Unter hoher Last sterben gelegentlich Worker-Threads, ohne eine Exception auszulösen oder auf eine sonstige nachvollziehbare Art ihren Kontrollfluß zu verlassen.

[3] Die Angabe -Xmx... ist notwendig, da die JVM sonst ein relativ kleines Speicherlimit festlegt, was zu OutOfMemoryErrors führt. Die Syntax dieses Parameters ist allerdings von der jeweiligen JVM abhängig.

Daher lässt der WorkerPool einen Thread laufen, der periodisch aufwacht und prüft, ob die Worker-Threads noch lebendig sind (isAlive() in Java-Terminologie). Gegebenenfalls werden neue Worker gestartet.

Ein solches Vorgehen, mit einem eigenen Kontrollfluss andere zu überwachen, ist als „Watchdog“ bekannt. Die entsprechende Klasse heißt WorkerPool.WatchDog.

Watchdog-Skript Die gleiche Vorgehensweise kann auch auf Betriebssystemebene angewendet werden, um Abstürze der JVM oder beispielsweise Stromausfälle zu behandeln.

Dazu existiert ein Shell-Skript check.sh, das den Crawler neu startet, wenn eines der folgenden Indizien für einen Programmabsturz vorliegt:

- es hat ein Core-Dump stattgefunden (Datei core ist im Verzeichnis des Crawlers vorhanden)
- weniger als 5 Java-Prozesse laufen
- die Logdatei wurde seit 10 Minuten nicht mehr geändert

Dieses Skript kann z. B. durch cron periodisch aufgerufen werden. Es sorgt dann entsprechend für einen Neustart, falls dies notwendig ist.

Damit keine Informationen über den Grund eines eventuellen Absturzes verloren gehen, werden Logdateien usw. beim Neustart in ein Archiv-Verzeichnis gesichert.

ThreadMonitor Zur allgemeinen Überwachung registrieren sich die interessanten Threads im System bei der Singleton-Instanz von ThreadMonitor, die periodisch den Zustand aller Threads in die Logdatei schreibt.

SingletonThreadGroup In Java gehört jeder Thread zu einer ThreadGroup, die unter anderem Exceptions auffängt, die in einem ihrer Threads nicht gefangen wurden. Dies kann bei sog. *unchecked exceptions* auftreten; zu diesen gehören Ausnahmen wie NullPointerException, ArithmeticException usw., die praktisch überall auftreten können und deren Behandlung nicht erzwungen wird.

Eine solche ThreadGroup wird als Singleton-Instanz SingletonThreadGroup zur Besitzerin aller wichtigen Threads im Crawler gemacht, so dass alle eventuell ungefangenen Exceptions dort landen und mit einer entsprechenden Meldung im Log vermerkt werden können.

6.1.2 Benutzung der Programme zur Auswertung

Einlesen der Daten aus der Datenbank

Es gibt zwei Möglichkeiten, die Daten aus der Datenbank für die C++-Programme zur Auswertung zugänglich zu machen:

1. direkter Zugriff auf die Datenbank mittels eingebettetem oder dynamischem SQL
2. Export der Daten in ASCII-Dateien, die dann eingelesen werden

Für beide Varianten werden Ableitungen des LEDA-Datentyps für gerichtete Graphen implementiert, die gegeneinander austauschbar sind. DBGraph liest die Daten über dynamisches SQL,[4] während FileGraph ASCII-Dateien liest.

Die zweite Möglichkeit ist in einem gewissen Sinne unelegant, da sie einige der Vorteile zunichte macht, die ein Datenbanksystem bietet.

In der Praxis hat sie sich dennoch als die praktischere erwiesen: das Lesen der Daten aus einer „flachen“ Datei ist weitaus schneller als der Zugriff auf eine Datenbank. Da die Programme zur Auswertung jeweils große Teile des gesamten Datenbestandes lesen, ist der Unterschied in den Ausführungszeiten erheblich.

Zum Vergleich: während das Einlesen der Daten für die Kombinaton BFS-Strategie/nur gelesene Seiten/Site-übergreifende Links aus Dateien nur 19 Sekunden dauert, braucht die Variante mit Datenbankzugriff 434 Sekunden; das ist um den Faktor 22 langsamer.[5]

Aus diesem Grund sind alle Programme zur Auswertung mit FileGraph implementiert.

[4] Die Verwendung von eingebettetem SQL hätte einen größeren Installations- und Konfigurationsaufwand erfordert, so dass hier dynamisches SQL vorgezogen wird.

[5] Da der Rechner mit der Datenbank (Rechner 1 in Anhang A) nicht über genügend Speicher verfügt, um gleichzeitig auch noch die Programme für die Graphalgorithmen auszuführen, wird die Auswertung auf Rechner 2 durchgeführt. Daher erfolgen hier sowohl Datenbank- als auch Dateizugriffe über das 100-MBit-Netzwerk des Lehrstuhls. Der Dateizugriff erfolgt über NFS; für den Datenbankzugriff wird die Datenbank mittels CATALOG DATABASE bei einer lokalen DB2-Instanz angemeldet, die den Zugriff über das Netzwerk regelt.

Die Daten stehen dabei in zwei Dateien:

Knoten Die Knoten stehen in der Form *id,name* in einer Datei:

```
...
9349,"http://www.informatik.uni-trier.de:80/"
...
39117,"http://www.informatik.uni-trier.de:80/Reports/"

...
```

Kanten In einer zweiten Datei stehen die Kanten in der Form *from_id, to_id*. Der Hyperlink von der ersten der oben genannten Seiten zur zweiten sieht z. B. so aus:

```
...
9349,39117
...
```

Das Format für die Knoten- und Kantendateien ist so gewählt, dass es direkt mit export-Befehlen aus DB2 zu erzeugen ist. So könnten die beiden Dateien aus obigem Beispiel erstellt werden mit den DB2-Befehlen

```
export to webpages.del of del select id, url from wp_view

export to hyperlinks.del of del select from_id, to_id from hyperlink
```

(Zu wp_view siehe Abschnitt 39.).

Optionen der Programme

Tabelle 6.2 zeigt eine Übersicht über die fünf Programme zur Auswertung und die möglichen Optionen. In Klammern stehen Verweise auf die Abschnitte, in denen die entsprechenden Algorithmen erläutert sind.

Programm	**Aufgabe (Abschnitt im Text)**	
	Option	**Erläuterung**
Alle Programme	-n <*Dateiname*>	Name der Knotendatei
	-e <*Dateiname*>	Name der Kantendatei
	-h	Hilfe
bowtie	Berechnung der Bowtie-Struktur (3.3)	
	Keine weiteren Optionen.	
clust	Finden dichter Subgraphen (3.4.2)	
	-s <*Zahl*>	Mindestgröße für Cluster; nur Cluster mit mindestens dieser Größe werden ausgegeben
	-i <*Zahl*>	t_{min}
	-a <*Zahl*>	t_{max}
	-d <*Zahl*>	$decay$
bip_clust	Finden dichter bipartiter Subgraphen (3.4.3)	
	-f <*Zahl*>	Mindestanzahl der Fans eines Clusters für die Ausgabe
	-c <*Zahl*>	Mindestanzahl der Centers eines Clusters für die Ausgabe
	-i <*Zahl*>	t_{min}
	-a <*Zahl*>	t_{max}
	-d <*Zahl*>	$decay$
pagerank	Berechnen der Pagerank-Gewichte (3.5)	
	-i <*Name*>	Name einer Initialisierungsfunktion
	-p *Zahl*	Schranke ε für die Konvergenz
	-t *Zahl*	Schranke t für die Ausgabe: nur Knoten mit Gewicht $\geq t$ werden ausgegeben
kleinberg	Durchführen des HITS-Algorithmus (3.6)	
	-i <*Name*>	Name einer Initialisierungsfunktion

Programm	**Aufgabe (Abschnitt im Text)**	
	Option	**Erläuterung**
	-p *Zahl*	Schranke ε für die Konvergenz
	-d *Zahl*	Dämpfungsfaktor d
	-t *Zahl*	Schranke t für die Ausgabe: nur Knoten mit Gewicht $\geq t$ werden ausgegeben

Tabelle 6.2: Optionen der Auswertungsprogramme

Die Initialisierungsfunktionen bei kleinberg und pagerank sind dafür verantwortlich, die Startmengen für die HITS-Berechnung (Abschnitt 3.6) bzw. den Präferenzvektor bei Pagerank (Abschnitt 3.5) zu wählen. Eine Liste der möglichen Initialisierungsfunktionen liefert die Option -h.

Ein Aufruf des Programms für den HITS-Algorithmus könnte also z. B. so aussehen:

```
./kleinberg -n webpages.del -e hyperlinks.del -i init_uniform -p .01 -d .05 -t .0005
```

Dies startet die HITS-Berechnung mit

- Knotendatei webpages.del
- Kantendatei hyperlinks.del
- Initialisierungsfunktion init_uniform, also gleiche Hub- und Authority-Gewichte für alle Knoten als Vorgabe
- Schranke $\varepsilon = 0.01$ für die Konvergenz
- Dämpfung $d = 0.05$
- Schranke $t = 0.0005$ für die Ausgabe

6.2 Performance

6.2.1 Gesamtperformance des Crawlers

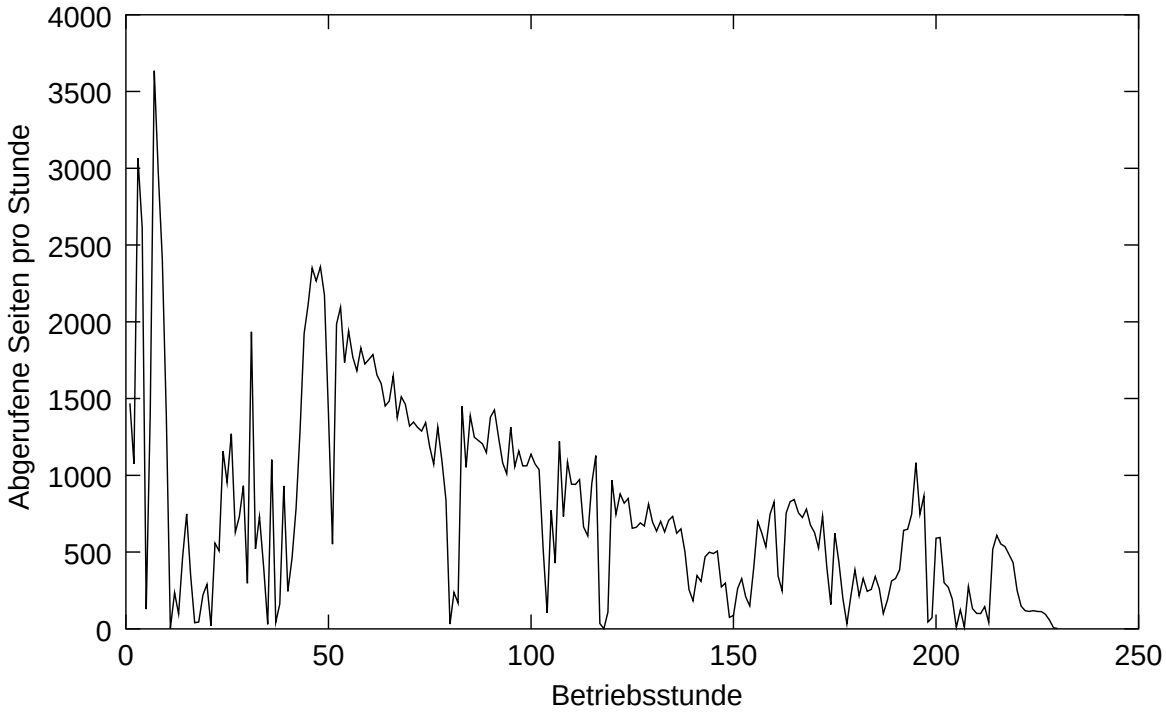


Abbildung 6.1: Durchsatz des Crawlers

Die erzielte Performance des Crawlers, speziell also der Durchsatz in Seiten pro Zeiteinheit, ist zufrieden stellend. Wie Abbildung 6.1 zeigt, konnten anfangs teilweise mehr als 3000 Seiten pro Stunde heruntergeladen werden. Dies hätte ohne Multithreading und gleichzeitige Benutzung mehrerer Netzwerkverbindungen nicht erreicht werden können.[6].

Dabei lief das Programm auf einem handelsüblichen PC (siehe Anhang A) mit einem Prozessor und 100-MBit-Anbindung an das Netzwerk der Universität.

Der Einbruch in den ersten 50 Stunden resultiert daraus, dass während dieser Zeit noch aktiv am Crawler gearbeitet wurde, was häufige Unterbrechungen und Neustarts erforderte. Die kurzzeitigen Schwankungen im Durchsatz rühren vom wechselnden Durchsatz des Uni-

[6] Zur Skalierbarkeit durch Multithreading siehe auch Abschnitt 6.2.3.

Netzes und anderer Teile des Internets her, die üblicherweise zu beobachten sind. Auffälliger allerdings ist die abnehmende Performance über einen längeren Zeitraum betrachtet.

Hier ist der Grund in der wachsenden Größe der Datenbanken zu suchen. Die Datenbank für den Lauf der BFS-Strategie umfasst 1.8 Millionen Webseiten, davon etwa 179000 mit Inhalt, und knapp 5 Millionen Hyperlinks. Der Lauf der fokussierten Strategie ermittelte 1.76 Millionen Seiten, davon etwa 202000 mit Inhalt, und knapp 3.9 Millionen Hyperlinks. Die Datenbanken benötigten jeweils etwa 3.5 GB Speicherplatz.

Vor allem die Verwendung von CLOBs ist bei wachsender Datenbankgröße für die Verlangsamung verantwortlich, wie Abschnitt 6.2.4 zeigt.

6.2.2 Durchsatzmessung im lokalen Netz

Um die grundsätzliche Leistungsfähigkeit des Crawlers zu testen, wird ein vollständiger binärer Baum der Tiefe 10 von Webseiten erzeugt; dieser umfasst 1023 Seiten. Diese Seiten werden auf 5 Rechner im lokalen Netz gespiegelt und mit dem Crawler abgearbeitet.

Mit 10 Worker-Threads, maximal 10 Datenbankverbindungen gleichzeitig, leerer Datenbank und ohne Zeitverzögerung zwischen zwei Zugriffen braucht der Crawler 895 Sekunden, um 5114 Seiten[7] abrurufen und zu speichern; das entspricht etwa 5.7 Seiten pro Sekunde oder 20570 Seiten pro Stunde. Angesichts der Tatsache, dass jeder Seitenabruf neben dem Netzwerkzugriff mehrere Datenbankzugriffe, einen Lauf des HTML-Parsers und eine Aktualisierung der internen Datenstrukturen erfordert, ist dieser Wert recht gut.

6.2.3 Skalierbarkeit durch Multithreading

Der Crawler benutzt Multithreading, so dass mehrere Seiten gleichzeitig aus dem Netz angefordert werden können; dadurch kann Zeit gespart werden.

[7] Die Zeit wird gemessen als die Differenz der minimalen und maximalen last_fetched-Attribute in der Datenbank. Insgesamt sind $5 \cdot 1023 = 5115$ Seiten abzurufen, die Zeit *zwischen* der ersten und letzten Seite gilt also für 5114 Seiten.

Um diese Zeiterparnis nachzuweisen, wird hier untersucht, wie sich die Gesamtzeit für das Abrufen von mehreren Webseiten durch Einsatz mehrerer Worker-Threads verhält.

Dazu werden 100 von einem CGI-Skript erzeugte Seiten vom Crawler abgearbeitet. Dieses Skript liefert die Seiten mit einer einstellbaren Verzögerung, um die Verhältnisse im WWW zu simulieren. Die Seiten bestehen aus nur einer Zeile Text ohne Hyperlinks, es müssen also keine Links gespeichert werden. Es wird der volle Mechanismus des Crawlers (BFS-Strategie, HTML parsen, etc.) benutzt.

Die Seiten werden auf einem zweiten Rechner erzeugt, der über ein 100-MBit-Netzwerk angebunden ist. Das Erstellen und Übertragen der Seiten an sich dauert ohne Verzögerung etwa 16 ms pro Seite, die hier vernachlässigbar sind. Es wird die Zeit zwischen den last_fetched-Attributen der ersten und letzen Seite bestimmt.[8]

Tabelle 6.3 zeigt die erreichten Zeiten. Aus der zweiten Spalte ist ersichlich, dass die minimale Zeitspanne für das Abarbeiten einer Seite etwa 13.5 s/99 Seiten = 136 ms/Seite beträgt. Hier zeigt sich auch, dass der Verzicht auf eine generische Zugriffschicht für den Datenbankzugriff sich im Hinblick auf den Durchsatz gelohnt hat. Experimente mit dem O/R-Mapping-Werkzeug Castor haben bestätigt, dass damit solche Zeiten auf keinen Fall möglich gewesen wären.

Die Werte mit Verzögerungen zeigen, dass das System gut in der Anzahl der Worker-Threads skaliert. Abbildung 6.2 veranschaulicht, dass der Einsatz von n Threads die benötigte Zeit durch n teilt. Der Versatz der Kurven von etwa 5 Sekunden lässt sich durch die Bestandteile des Programmablaufs erklären, die nicht nebenläufig durch den WorkerPool bearbeitet werden.

6.2.4 Performanceprobleme durch Verwendung von CLOBs

Als erste Maßnahme zur Leistungssteigerung wäre zu prüfen, inwiefern die Speicherung der Seiteninhalte als CLOBs (*character large objects*, d. h. größere Textstücke als Attribute von Datenbanktabellen) die Gesamteistung beeinträchtigt.

[8] Die Gesamtzeit gilt also für die Bearbeitung von 99 Seiten.

Anzahl Worker	Zeit ohne Verzögerung	Zeit mit 2s Verzögerung	Zeit mit 5s Verzögerung
1	20.0	210.0	508.9
5	14.9	46.7	100.7
10	14.0	26.2	51.0
15	13.5	19.5	36.1
20	14.7	17.5	27.3
30	14.8	14.5	22.5

Tabelle 6.3: Zeitvergleich je nach Anzahl der Worker-Threads

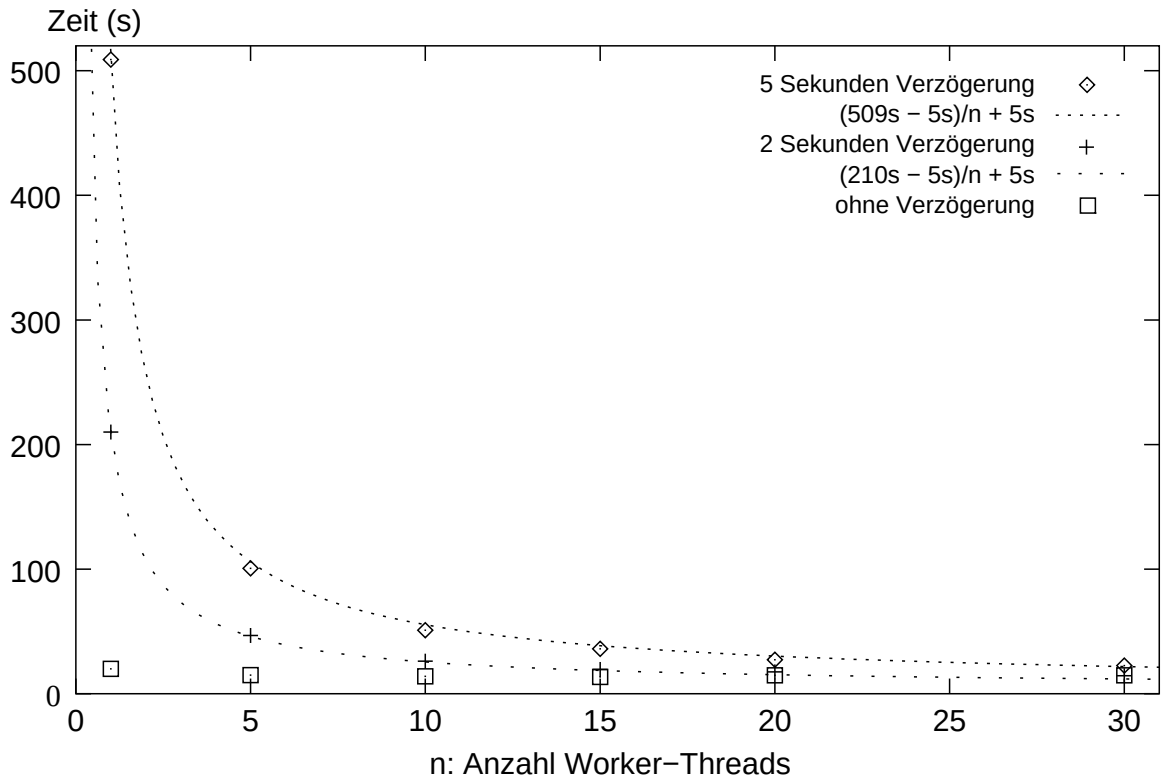


Abbildung 6.2: Skalierung in der Anzahl der Worker

Dass die Benutzung von CLOBs teuer ist, zeigt Abbildung 6.3. Hier werden 20000 Datensätze, bestehend jeweils aus einer Integer-ID und einem 10 kB großen Textstück, auf zwei verschiedene Weisen durch ein Java-Programm via JDBC in einer Datenbank gespeichert:

- Die erste Variante benutzt CLOBs um die Daten zu speichern, also eine Tabelle der Form

```
create table test_clob (
   id integer not null primary key,
   text clob(2000000) compact not logged
)
```

- Die zweite Variante speichert die Textdaten in Dateien; ID und Dateiname werden in der Datenbank gespeichert:

```
create table test_files (
    id integer not null primary key,
    filename varchar(100)
)
```

Wie Abbildung 6.3 verdeutlicht, ist die Speicherung in CLOBs deutlich teurer als die Benutzung von Dateien. Vor allem fällt auf, dass nach 20000 Datensätzen das Einfügen neuer Daten sich noch nicht wesentlich verteuert hat, wenn man Dateien benutzt. Im Gegensatz dazu ist die Variante mit CLOBs schon um den Faktor fünf langsamer geworden, eine einzelne Einfügung dauert dann schon zehn Mal so lange wie bei der Speicherung in Dateien.

Schlussbemerkungen

Insgesamt bieten sich sicherlich noch viele Stellen, an denen die Performance des Crawlers verbessert werden könnte. Neben der angesprochenen Problematik mit der Speicherung von Textdaten in CLOBs zeigt die Diskussion von Mercator in Abschnitt 2.3.2 noch eine Reihe von Punkten auf, die durch eigene Datenstrukturen und Verbesserung der Java-Bibliotheken beschleunigt werden können.

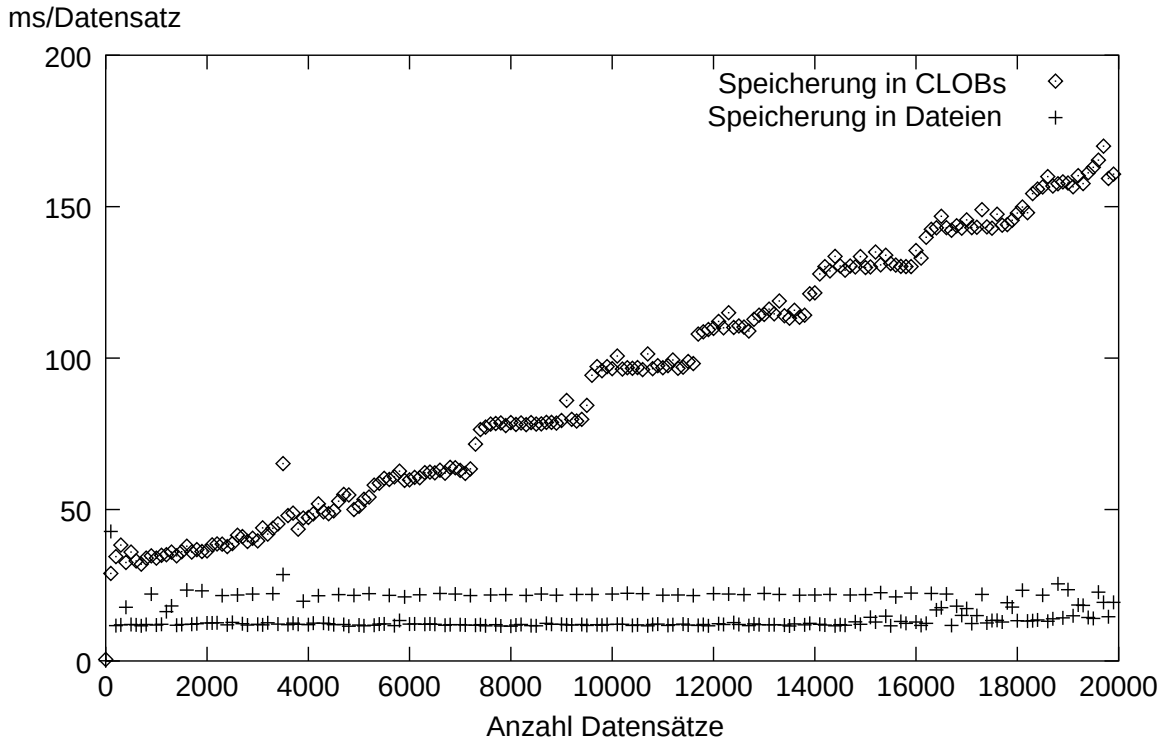


Abbildung 6.3: Durchsatz des Crawlers

Allerdings liegt bei der vorliegenden Arbeit die Priorität nicht auf der Erhöhung des Durchsatzes um jeden Preis. Abgesehen davon, dass ein Optimierungsaufwand wie bei Mercator im gegebenen zeitlichen Rahmen nicht zu leisten ist, liegt das Hauptaugenmerk bei der Entwicklung des Crawlers darauf, durch Verwendung fertiger Komponenten wie z. B. des XML-Parsers oder eben des Datenbanksystems schnell zu einem benutzbaren Systems zu kommen. Außerdem bietet die Verwendung von Standardkomponenten auch im Betrieb größeren Komfort – so können z. B. die kompletten Daten mit den Hilfsmitteln des DBMS[9] bearbeitet werden, etwa für ad-hoc-Anfragen oder ein einfach durchzuführendes Backup.[10]

6.3 Erfahrungen aus der Implementierung

> Plan to throw one away; you will, anyhow. [Bro95, S. 116]

Bei der Erstellung von Software sind in der Regel Kompromisse zu machen, und es muss oft zwischen mehreren Möglichkeiten abgewogen werden, ein bestimmtes Problem zu lösen. Beurteilen lassen sich solche Entscheidungen manchmal erst im Nachhinein.

Dieser Abschnitt führt einige Erfahrungen auf, die sich im Verlauf der Arbeit an der vorgestellten Software ergeben haben.

6.3.1 Logging und Debugging

Ein Programm wie der hier entwickelte Crawler produziert wenige Ausgaben, anhand derer sein Ablauf nachvollzogen werden könnte. Auch die großen Datenmengen, die in der Datenbank abgelegt werden, können nicht unmittelbar betrachtet und auf Richtigkeit überprüft werden.

[9] Database Management System

[10] Tatsächlich kamen während des Crawler-Laufs und der Experimente mit den Daten zwei Mal Komplettbackups einer Datenbank zum Einsatz, die mit je einem einzigen Befehl erstellt und zurückgespielt werden können. Eine Sicherung der Datenbank und *zusätzlich* von Hunderttausenden von Dateien wäre vermutlich angesichts des Aufwands gar nicht erfolgt.

Als wichtiges Instrument zur Fehlersuche und zur Dokumentation des Programmablaufs wird daher *Logging* benutzt, d. h. das Protokollieren von Statusinformationen zur Laufzeit des Programms. Auch dazu gibt es wieder umfangreiche Open-Source-Pakete; hier wird *Log4j* [Apa01] aus dem Apache-Projekt eingesetzt. (Die neue Version 1.4 des JDK, die bei der Erstellung dieser Arbeit noch nicht vorlag, wird ein ähnliches Logging-Paket beinhalten.)

Log4j hat sich als sehr flexibles Werkzeug für diese Aufgabe erwiesen. Es bestehen viele Möglichkeiten, ohne Änderungen am Sourcecode der Anwendung über eine Konfigurationsdatei zu steuern, welche Ausgaben in welche Ausgabemedien (Konsole, Dateien, Unix-Syslog uvm.) protokolliert werden.

Unterstützend dazu sind gelegentlich die eingebauten Möglichkeiten der Java-Ausführungsumgebung hilfreich, insbesondere der eingebaute *Profiler* des JDK. Dieser ermöglicht es, beispielsweise über jedes einzelne instanziierte Objekt Auskunft zu erhalten und so z. B. Speicherlecks zu finden.

6.3.2 Beurteilung der O/R-Abbildung

Rückblickend lässt sich sagen, dass die Wahl der O/R-Abbildung, die in Abschnitt 47 getroffen wurde, durchaus aufrecht erhalten werden kann. Die geringe Anzahl von Änderungen, die im Laufe der Entwicklung noch am Datenmodell gemacht worden sind, war gut zu bewältigen, obwohl die Modifikationen sowohl am SQL- als auch am Java-Code nachgezogen werden mussten.

Dennoch hat sich abgezeichnet, dass solche nachträglichen Änderungen tatsächlich einen sehr großen Arbeitsaufwand nach sich ziehen könnten, wenn das Datenmodell und der Programmcode umfangreicher wären. In einem solchen Fall wäre abzuwägen, ob nicht doch für die Flexibilität und Zeitersparnis bei der Entwicklung mit einer generischen Zugriffsschicht deren Nachteile in Kauf zu nehmen wären.

6.3.3 SQLJ

SQLJ erlaubt es, SQL-Anfragen direkt in den Quelltext eines Java-Programmes zu einzubetten. Java heißt dann in diesem Zusammenhang „Gastgebersprache“ (*host language*).

Die Kommunikation zwischen SQLJ und Java läuft über sogenannte Host-Variablen, die Daten zwischen den Java- und SQLJ-Bestandteilen transportieren.

Ein Programm in SQLJ wird von einem Präprozessor in seine SQL- und Java-Bestandteile zerlegt. Der SQL-Teil kann vom Datenbanksystem verarbeitet werden. Der Java-Teil entsteht, indem der eingebettete SQL-Code in entsprechende JDBC-Aufrufe umgesetzt wird.

Die Code-Beispiele in den Abbildungen 6.4 und 6.5 zeigen die Unterschiede.

```
#sql iterator contentIter (long id, String Content);
...
contentIter iter;
int state = WebPage.STATE_FETCHED;
int http_state = 200;
#sql[context] iter = {
    select id, content
    from webpage
    where internal_state = :state
    and http_state = :http_state
};

while (iter.next()) {
   System.out.println (iter.content());
}

iter.close();
```

Abbildung 6.4: Codebeispiel SQLJ

In der tatsächlichen Anwendung haben sich einige Vor- und Nachteile von SQLJ herausgestellt.

Vorteile:

⊕ Wie oben gezeigt, kann SQL-Code in zusammenhängender Form im Java-Code integriert werden. Er ist so wesentlich leichter zu lesen und zu bearbeiten, als wenn er – wie im JDBC-Beispiel – als Java-String zusammengesetzt werden muss.

⊕ Beim Kompilieren der Anwendung kann der SQL-Teil des Codes schon auf Syntaxfehler u. ä. überprüft werden. Es kommt dann nicht erst zur Laufzeit zu SQL-Fehlermeldungen.

⊕ Die SQL-Bestandteile können vom Datenbanksystem bei der Übersetzung schon kompiliert und optimiert werden und liegen dann bei der Ausführung des Programms schon übersetzt vor. Im Gegensatz dazu müssen JDBC-Statements zumindest bei jedem Programmstart neu übersetzt werden.

Nachteile:

⊖ SQLJ erfordert den Einsatz eines Präprozessors, der die SQL-Bestandteile von den Java-Bestandteilen des Codes trennt. Dies verursacht große Probleme, wenn Werk-

```
int state = WebPage.STATE_FETCHED;
int http_state = 200;
PreparedStatement stmt = conn.prepareStatement (
    "select id, content from webpage " +
    " where internal_state = ?" +
    " and http_state = ?";
);
stmt.setInt (1, state);
stmt.setInt (2, http_state);
ResultSet rs = stmt.executeQuery();

while (rs.next()) {
   System.out.println (rs.getString (2));
}

rs.close();
stmt.close();
```

Abbildung 6.5: Codebeispiel JDBC

zeuge zur Unterstützung des Entwicklungsprozesses, wie etwa *make* oder *ant* eingesetzt werden sollen. Diese untersuchen die Abhängigkeiten zwischen Teilen des Quelltextes, was durch den zusätzlichen Übersetzungsschritt von SQLJ nach Java gestört wird.

⊖ Die Syntax von SQLJ wird in der Regel von Entwicklungsumgebungen oder Texteditoren nicht erkannt. Wenn diese die Sprachsyntax von Java berücksichtigen, etwa für Einrückungen, dann wird diese Funktionalität von der zusätzlichen SQLJ-Syntax gestört.

⊖ Fehlermeldungen (SQL-Exceptions), die zur Laufzeit auftreten, sind nur schwer zu verfolgen. Sie nehmen Bezug auf den automatisch generierten Java-Code, der schlecht lesbar ist. Zwar kann dieser generierte Code mit Verweisen auf den Originalcode versehen werden, trotzdem ist die Fehlersuche wesentlich umständlicher als bei „normalem“ Java-Code.

6.3.4 Fehlersuche bei der Datenbankprogrammierung

Die Fehlersuche zwischen Java-Anwendungsprogramm und Datenbanksystem gestaltet sich mitunter recht schwierig. Das Hauptproblem dabei ist, dass mit mehreren Datenbankverbindungen gleichzeitig gearbeitet wird; dabei sind die Verbindungen sowohl auf Java- als auch auf Datenbankseite mit Identifikationsnummern oder -namen versehen, die aber nicht gegenseitig zuzuordnen sind.

So bleibt als einzige Möglichkeit, Zeitstempel der Verbindungen in den Datenbank- und Java-Meldungen mitzuführen und zu vergleichen, was die Suche erheblich erschwert.

6.3.5 Multithreading

Wie schon in Abschnitt 5.1 geschildert, ist das in der Sprache verankerte Multithreading ein wesentlicher Vorteil der Programmierung in Java.

Es hat sich auch tatsächlich gezeigt, dass durch das spracheigene Monitorkonzept die Programmierung von nebenläufigen Anwendungen vergleichsweise einfach und sicher ist.

Dennoch ergeben sich natürlich auch in Java die üblichen Probleme, die nebenläufige Programmierung mit sich bringt. Insbesondere sogenannte „race conditions“, also Ausführungsabläufe, die vom tatsächlichen Scheduling der beteiligten Kontrollflüsse abhängen, sind schwer zu finden. Hier ist lediglich extensives und im Umfang steuerbares Logging (s. Abschnitt 6.3.1) nützlich.

Gelegentlich ist es hilfreich, dass die JVM auf Aufforderung einen sog. *Thread-Dump*, also eine Auflistung aller Kontrollflüsse und ihres aktuellen Zustands, generieren kann. Damit kann z. B. jederzeit geprüft werden, welcher Worker gerade welche Webseite bearbeitet, da die Worker ihren Namen entsprechend dem aktuell ausgeführten Work-Objekt anpassen.

6.3.6 Netzwerkprogrammierung

Wie schon in Abschnitt 5.1 angedeutet, bietet Java auch im Hinblick auf Netzwerkprogrammierung einigen Komfort. So findet sich beispielsweise im Paket java.net eine Reihe von Klassen, die z. B. auch den direkten Zugriff auf Dokumente durch Angabe ihrer URL ermöglichen.

Allerdings kommt für den Abruf von Webseiten ein externes Paket namens *HTTPClient* (s. Anhang A) zum Einsatz. Der Hauptgrund dafür ist, dass *HTTPClient* erlaubt, einen Zeitlimit für eine HTTP-Anfrage zu setzen. Nach Ablauf dieses Limits wird eine Exception ausgelöst, so dass nicht beliebig lange auf einen fehlerhaften Server gewartet wird. Diese Möglichkeit bieten die Standardklassen aus java.net nicht.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die vorliegende Arbeit untersucht einen Teil des WWW aus der Sicht der Graphentheorie, um interessante Strukturen auf einer Menge Informatik-relevanter Webseiten herauszuarbeiten.

Nach einer kurzen Vorstellung der notwendigen Grundbegriffe aus der Graphentheorie und dem WWW-Umfeld (Kapitel 2) werden in Kapitel 3 Strategien und Algorithmen zum Sammeln und Auswerten eines Teils des WWW vorgestellt.

Die zwei vorgestellten Strategien durchmustern das Web in unterschiedlicher Weise: während die einfache Breitensuche alle Seiten in der Nähe der Startseiten abarbeitet, können bei der fokussierten Strategie auch entferntere relevante Seiten gefunden werden. Um dies zu ermöglichen, werden die entreffenden Seiten einer Textklassifikation unterzogen, und Links von relevanten Seiten werden bevorzugt verfolgt.

Auf den so gesammelten Daten werden bekannte Verfahren (HITS (3.6), PageRank (3.5)) – teilweise in einem veränderten Kontext – erprobt sowie neue Algorithmen entworfen (*grow-* … (3.4)).

Dadurch werden verschiedenartige Strukturen auf dem gesamten Graphen (Bowtie-Struktur(3.3), HITS (3.6)) und auf kleinen Subgraphen (3.5, 3.4) sichtbar, die thematische Zusammenhänge innerhalb der Informatik-Community im Web zeigen.

Um diese Untersuchungen zu ermöglichen, wird ein datenbankgestützter Webcrawler entwickelt und implementiert (Kapitel 4 und 5). Dieser Crawler liest die notwendigen Daten aus dem Web und speichert sie lokal. Dabei wird besonders Wert darauf gelegt, dass der Crawler Probleme und Ärgernisse vermeidet, die von Serverbetreibern oft bemängelt werden. Außerdem ist der Crawler so ausgelegt, dass er leicht erweitert werden kann, z. B. durch eine andere Crawl-Strategie.

Schließlich werden noch praktische Aspekte der Entwicklung und des Betriebs der erstellten Software erläutert und Erfahrungen aus der Entwicklung geschildert.

7.2 Ausblick

Die in dieser Arbeit entwickelten Verfahren und Programme könnten sicherlich in verschiedene Richtungen abgeändert oder weiterentwickelt werden. Hier sind einige Punkte aufgeführt, die interessant erscheinen.

Datenbankanbindung Die Datenbankanbindung stellt in der vorliegenden Form einen Kompromiss zwischen leichter Benutzbarkeit und hohem Durchsatz dar.

Um den Durchsatz zu verbessern, könnte zunächst die Verwendung von CLOBs vermieden werden (s. Abschnitt 6.2.4). Im Extremfall könnte ganz auf die Verwendung eines Datenbanksystems verzichtet werden. Die Daten müssten dann in handkodierten Datenstrukturen gehalten werden, was die Weiterverarbeitung erschweren würde.

Auf der anderen Seite erscheint eine noch engere Verbindung von Programmlogik, persistenten Daten und Graphalgorithmen reizvoll. Könnten diese bei ausreichender Leistungsfähigkeit verbunden werden, so wären Online-Algorithmen[1] für die Crawl-Strategie möglich. (Siehe dazu auch den nächsten Punkt.)

[1] Als Online-Algorithmen werden Algorithmen bezeichnet, deren Eingabedaten zum Teil erst während der Berechnung bekannt werden.

Crawl-Strategien Um verschiedene Teilmengen des WWW zu betrachten, ist eine Vielzahl unterschiedlicher Crawl-Strategien denkbar.

Interessant erscheinen z. B. Strategien, die einen größeren Kontext im Graphen zur Steuerung des Crawls heranziehen. So könnte etwa ein PageRank-Gewicht als Relevanzwert für eine fokussierte Strategie dienen. Wie oben aufgeführt, würde dies eine Integration von Datenbank, Graph-Repräsentation und -Algorithmen erfordern.

Steuerung der Netzwerk-, Prozessor- und Datenbank-Auslastung Das Leistungsverhalten des Crawlers wird bestimmt durch

- die Leistungsfähigkeit des Netzes und der angesprochenen Webserver
- die Prozessorleistung
- den Durchsatz des Datenbanksystems

Schwankungen in diesen Größen kann Rechnung getragen werden durch Anpassung der Anzahlen von Worker-Threads und Datenbankverbindungen, die der Crawler benutzt. Wenn beispielsweise die benutzten Webserver nur langsam antworten, kann durch eine höhere Zahl von Threads der Durchsatz erhöht werden. Andererseits kann eine zu hohe Zahl von Threads zu einer Überlastung der Datenbank führen, usw.

Bisher werden diese Parameter manuell der jeweiligen Situation angepasst. Es wäre interessant, automatische Verfahren zum Lastausgleich zwischen Datenbank, Prozessor und Netzwerk zu erproben.

A Übersicht über die benutzte Hard- und Software

Benutzte Rechner Es standen zwei Rechner zur Verfügung:

Rechner 1: Ein Pentium-III mit 500 MHz und 256 MB Hauptspeicher als Datenbankserver und Entwicklungsrechner; auf diesem Rechner lief auch der Crawler.

Rechner 2: Ein Pentium-III-Doppelprozessor mit 500 MHz und 1 GB Hauptspeicher. Dieser Rechner wurde benutzt, weil die große Speicherkapazität für speicherintensive Berechnungen auf dem Web-Graphen erforderlich war.

Betriebssystem Auf beiden Rechnern kam RedHat Linux 6.2 mit Kernel 2.2.16-3 zum Einsatz.

DB2 V7.1 Das benutzte Datenbank-Management-System war DB2 UDB V7.1.0 von IBM. Die Daten wurden in „system managed“-Tablespaces, also in Dateien im normalen Dateisystem, gespeichert.

JDK (Java Development Kit) Es wurden die Java Development Kits 1.3.0 von Sun und IBM eingesetzt.

Bezugsquelle: `http://java.sun.com` und `http://www.ibm.com/java/jdk`

OpenXML OpenXML 1.2 wurde im HTML-Modus als HTML-Parser benutzt.

Bezugsquelle: `ftp://ftp.exolab.org/pub/src/openxml-1.2-src.zip`

Log4j Log4j 1.1b1 dient zur Ausgabe von Logging- und Debugging-Informationen.

Bezugsquelle: http://jakarta.apache.org/log4j/

Apache Jakarta ORO Dieses Paket stellt Textbearbeitungsfunktionen mit regulären Ausdrücken zur Verfügung.

Bezugsquelle: http://jakarta.apache.org/oro/

HTTPClient Diese Bibliothek ersetzt die java.net-Klassen zum Zugriff auf URLs. Sie kommt hier zum Einsatz, weil die Standardklassen keinen Timeout-Mechanismus für HTTP-Zugriffe bieten.

Bezugsquelle: http://www.innovation.ch/java/HTTPClient/

LEDA Die *Library for Efficient Data Types and Algorithms* kam vor allem wegen des enthaltenen Graph-Datentyps zum Einsatz. Auch die Standard-Datentypen wie Warteschlangen, Maps oder Mengen waren nützlich.

Bezugsquelle: http://www.mpi-sb.mpg.de/LEDA/download/application/

Bow Diese Bibliothek bietet eine Reihe von Textklassifizierern, darunter auch einen Naïve-Bayes-Klassifizierer. Neben einem interaktiven Frontend gibt es auch einen Server-Modus, der zur Anbindung an den Crawler genutzt wird.

Bezugsquelle: http://www.cs.cmu.edu/~mccallum/bow

B Übersicht über die beiliegende CD

Datei/Verzeichnis	Erläuterung
auswertung	Programme und Daten der Auswertung
auswertung/programme	C++-Programme für die Auswertung
auswertung/ergebnisse	Ergebnisse der verschiedenen Algorithmen aus Kapitel 3
auswertung/starturls	Startseiten der beiden Crawls
crawler	Hauptverzeichnis des Crawlers mit Konfigurationsdateien, Skripten usw.
crawler/de	Quellcode und Klassen des Crawlers
crawler/de/cs75/assesspages	Programm AssessPages
crawler/de/cs75/crawler	Klassen: HostQueue und Strategien
crawler/de/cs75/database	Klassen: Datenbankzugriff
crawler/de/cs75/pool	Klassen: WorkerPool
crawler/de/cs75/utils	Hilfsklassen
crawler/sql	SQL-Skripte zum Einrichten der Datenbank usw.
text/	dieser Text als PostScript- und PDF-Datei

Tabelle B.1: CD-Inhalt

Literaturverzeichnis

[Alb99] R. Albert, H. Jeong und A.-L. Barabasi. *Diameter of the World Wide Web*. *Nature*, 401:130–131, 1999. 14

[Amb00] Scott Ambler. *Mapping Objects to Relational Databases*, 2000. http://www.ambysoft.com/mappingObjects.html 47, 47

[Apa01] Apache Software Foundation. *Log4j Project*, 2001. http://jakarta.apache.org/log4j/docs/index.html 49, 6.3.1

[Bon76] John Adrian Bondy und U. S. R. Murty. *Graph Theory with Applications*. Macmillan, London, 1976. 2.1

[Bor93] N. Borenstein und N. Freed. *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. Internet Request for Comment RFC 1521, Internet Engineering Task Force, 1993. (document)

[Bri01] BrightPlanet. *The Deep Web: Surfacing Hidden Value*, 2001. http://www.brightplanet.com/deepcontent/tutorials/DeepWeb/index.asp 2.3.1

[Bro95] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley, Reading, MA, 2. Aufl., 1995. 6.3

[Bro00] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins und Janet Wiener. *Graph structure*

in the web: experiments and models. In *Proceedings of the Ninth International World-Wide Web Conference*. 2000. 1, 1, 3.3.1, 3.3.2, 21, 21, 21, 21

[Con87] Jeff Conklin. *Hypertext: An Introduction and Survey*. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society*, 20(9), 1987. 3.2.1

[Con99] The World Wide Web Consortium. *HTML 4.01 Specification*, 1999. `http://www.w3.org/TR/html4/` 39

[Fei97] U. Feige und M. Seltser. *On the densest k-subgraph problem*. Techn. Ber. CS97-16, Department of Applied Math and Computer Science, The Weizman Institute, Rehovot, Israel, 1997. `http://citeseer.nj.nec.com/feige97densest.html` (document), 3.4, 3.4

[Fie97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk und T. Berners-Lee. *RFC 2068: Hypertext Transfer Protocol — HTTP/1.1*, 1997. (document), 2, ⑥, 39

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995. 4.2.4, 4.2.4

[Gra98] Mark Grand. *Patterns in Java*, Bd. 1. Wiley & Sons, 1998. 4.2.4

[Hey99] Allan Heydon und Marc Najork. *Mercator: A Scalable, Extensible Web Crawler*. *World Wide Web*, 2(4):219–229, 1999. 2.3.2, 2.3.2, 3, 42

[Hey00] Allan Heydon und Marc Najork. *Performance limitations of the Java core libraries*. *Concurrency: Practice and Experience*, 12(6):363–373, 2000. 2.3.2

[Hoa74] C. A. R. Hoare. *Monitors: An Operating System Structuring Concept*. *Communications of the ACM*, 17(10):549–557, 1974. 46

[Kap00] Carl S. Kaplan. *Legality of 'Deep Linking' Remains Deeply Complicated*, 2000. `http://www.nytimes.com/library/tech/00/04/cyber/cyberlaw/07law.html` 16

[Kle99] Jon M. Kleinberg. *Authoritative sources in a hyperlinked environment*. *Journal of the ACM*, 46(5):604–632, 1999. (document), 1, 17, 23

[Kol96] Charles P. Kollar, John R. R. Leavitt und Michael Mauldin. *Robot Exclusion Revisited*, 1996.
http://www.kollar.com/robots.html 37

[Kos93] Martijn Koster. *Guidelines for Robot Writers*, 1993.
http://www.robotstxt.org/wc/guidelines.html 4.1.2

[Kos94] Martijn Koster. *A Standard for Robot Exclusion*, 1994.
http://www.robotstxt.org/wc/norobots.html ②, 4.1.2

[Kum99] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan und Andrew Tomkins. *Trawling the Web for emerging cyber-communities*. In *Proceedings of the Eighth International World-Wide Web Conference*. 1999. 3.4.3, 3.4.5

[Law00] Susan Lawson, Richard A. Yevich und Warwick Ford. *DB2 High Performance Design and Tuning*. Prentice Hall, 2000. 47

[Lew98] David D. Lewis. *Naive (Bayes) at forty: The independence assumption in information retrieval.* In Claire Nédellec und Céline Rouveirol (Hg.), *Proceedings of ECML-98, 10th European Conference on Machine Learning*, S. 4–15. Springer Verlag, Heidelberg, DE, Chemnitz, DE, 1998. 3.1.2

[McC96] Andrew Kachites McCallum. *Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering*, 1996.
http://www.cs.cmu.edu/~mccallum/bow 3.1.2

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms*, Bd. 2: Graph Algorithms and NP-Completeness. Springer, Berlin, 1984. (document), (document), 1, 6

[Meh99] Kurt Mehlhorn und Stefan Näher. *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, England, 1999. (document), 3.8

[Naj01] Marc Najork und Janet L. Wiener. *Breadth-first search crawling yields high-quality pages*. In *Proceedings of the Tenth International World-Wide Web Conference*. 2001. 2.3.2, 2.3.2, 9, 14, 42

[Pag98] Lawrence Page, Sergey Brin, Rajeev Motwani und Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Techn. Ber., Computer Science Department, Stanford University, 1998. 1, 3.5.1, 3.5.2

[Riv92] Ronald Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*, 1992. 3.4.6

[Sch92] Peter Schnupp. *Hypertext*, Bd. 10.1 von *Handbuch der Informatik*. Oldenbourg, München, Wien, 1992. 1, 1

[Sil97] Abraham Silberschatz, Henry F. Korth und S. Sudarshan. *Database System Concepts*. McGraw-Hill, New York, 3. Aufl., 1997. 40

[Sti95] Douglas R. Stinson. *Cryptography: Theory and Practice*. CRC Press LLC, Boca Raton, Florida, 1995. 19

[Sul01] Danny Sullivan. *How Search Engines Rank Web Pages*, 2001. `http://searchenginewatch.com/webmasters/rank.html` 33

[Sun00] Sun Microsystems. *Java(TM) 2 Enterprise Edition Developer's Guide (V1.2.1)*, 2000. `http://java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/DevGuideTOC.html` 48

[Wor01] The World Wide Web Consortium. *HyperText Markup Language Homepage*, 1995–2001. `http://www.w3.org/MarkUp/` (document), 2

[Yod98] Joseph Yoder, Ralph Johnson und Quince Wilson. *Connecting Business Objects to Relational Databases*, 1998. `http://www.joeyoder.com/papers/patterns/PersistentObject/Persista.pdf` (document), 47, 48, 5.3.1

[Zim00] Jürgen Zimmermann und Gerd Beneken. *Verteilte Komponenten und Datenbankanbindung*. Addison-Wesley, 2000. 47

Die oben angegebenen URLs wurden zuletzt am 18.10.2001 überprüft.