Köln, den 30. August 2002

Studiengang Informationsverarbeitung
Sommersemester 2002
Proseminar: Softwaretechnologie
(Java für Geisteswissenschaftler)
bei Dr. J.-Y. Lalande

SearchEngy

Crawler-Indexer-Suchmachine
- Globale Dokumentation -

vorgelegt von

Fabian Steeg
Matrikelnummer: 3598900
e-mail: steeg@netcologne.de
Liebigstr. 43
50823 Köln

Inhaltsverzeichnis

1. Grundlegender Aufbau der Suchmaschine

SearchEngy ist eine Suchmaschine mit der geläufigen, für web-basierte Anwendung jedoch nicht unproblematischen Crawler-Indexer-Architektur (dazu und zu alternativen Entwürfen vgl. Lam 2001).

Hierbei übergibt der `Crawler` die Pfade zu den zu durchsuchenden Verzeichnissen an den `Parser`, dieser liefert die Wörter an den `Indexer`, der alle suchrelevanten Wörter der gelesenen Dateien auf eine Direktzugriffsstruktur speichert und diese binär auf die Festplatte speichert.

Die Index-Datei wird von der `QueryEngine` eingelesen und ermöglicht Suchabfragen auf dem Index, als Ergebnis liefert sie den Pfad zu dem Dokument, sowie, wenn gewünscht, der Kontext des gefundenen Wortes in der Datei (d.h. inklusive nicht-suchrelevanter Wörter).

Die Steuerung der Suchmaschine erfolgt durch eine graphische Benutzerschnittstelle (GUI, Graphical User Interface).

2. Komponenten des Programms

2.1 `Crawler`

Der `Crawler` ermittelt die Pfade zu allen Dateien in allen Unterverzeichnissen des gewählten Ausgangsverzeichnis. Welche Dateien dabei berücksichtigt werden, kann durch eine `CheckBox` in der `GUI` bestimmt werden. Seine Funktion `public String[] getFiles()` liefert diese an den `Parser`.

2.2 `Parser`

Der `Parser` liefert alle Wörter aller durch den `Crawler` ermittleten Dokumente und liefert sie in einem Objekt vom Typ `KeyData` durch einen `Zeichenfilter` an den `Indexer`.

2.2.1 `Data`

`Data` ist die abstrakte Basisklasse, die durch `KeyData` und `DocumentData` erweitert wird.

2.2.2 `KeyData`

Ein `KeyData` ist eine Erweiterung einer abstrakten Basisklasse `Data` und speichert das Wort sowie eine Direktzugriffsstruktur, die Objekte vom Typ `DocumentData` enthält (ebenfalls eine Erweiterung von `Data`). Soll beim Indizieren ein neues Wort ergänzt werden, das zwar schon vorhanden ist, allerdings nur in anderen Dokumenten, wird das `DocumentData` des neuen Wertes in dieser Direktzugriffsstruktur akkumuliert.

2.2.3 `DocumentData`

Ein `DocumentData` speichert eine Dokumenten-Kennziffer sowie sämtliche Fundstellen sowie deren Abstand zum vorigen Wort in zwei parallel verwalteten Arrays. Soll beim Indizieren ein neues Wort ergänzt werden, das schon in dem gleichen Dokument enthalten war, werden die Positionen und Abstände

in diesen Arrays akkumuliert. Alternativ wäre eine Verwaltung in einem speziellen Objekttyp, der Position und Abstand speichert, denkbar. Problematisch wäre hier die Tatsache, daß zu jeder Position schon mindestens 4 byte durch die Klasseninformationen belegt würden, während die implementierte Variante mit zwei Objekten (die Arrays) pro Dokument für alle Fundstellen auskommt, d.h. eine Lösung mit gesondertem Objekttyp hätte mit zunehmender Fundstellenhäufigkeit mehr, bei nur einer Fundstelle aber weniger Speicherplatzbedarf als die implementierte.

2.2.4 `Zeichenfilter`

`Zeichenfilter` ist eine abstrakte Klasse, die von den Klassen `Textfilter` und `Hypertextfilter` erweitert wird. Instanzen dieser konkreten Klassen werden je nach vorliegendem Dokumententyp zum Lesen durch den `Parser` verwendet.

2.2.5 `Textfilter`

Der `Textfilter` ist liest das nächste Wort der eingelesenen ASCII-256 Datei mit seiner Funktion `nextWord()`, wobei alle Nichtbuchstabenzeichen ein Wort beenden und Sonderzeichen der deutschen Sprache (ä,ö,ü,ß) erkannt und mitgelesen werden.

2.2.6 `Hypertextfilter`

Der `Hypertextfilter` liest im Unterschied zum `Textfilter` HTML-Dateien und verfügt dementsprechend über eine erweiterte Funktionalität: er überliest HTML-Tags und wandelt HTML-kodierte Sonderzeichen, die in einer separaten Textdatei definiert sind, in das darzustellende Zeichen um und ermöglicht damit eine korrekte Wortermittlung. Die HTML-Kodes müssen in der Form "verschlüsseltes Zeichen, Leerstelle, Zeilenumbruch, entschlüsseltes Zeichen" angegeben werden, da sie intern mit einem `LineNumberedReader` eingelesen werden. Die ver- und entschlüsselten Zeichen werden auf eine `Hashtable` eingetragen, auf die beim Auffinden einer Kodierung zugegriffen wird. Wird kein zur Kodierung gehöriges Zeichen gefunden, wird gemeldet, dass eine unbekannte oder ungültige Kodierung vorliegt.

2.3 `Indexer`

Der `Indexer` trägt die Wörter, die der `Parser` liefert, in eine der alternativen Direktzugriffsstrukturen `StorageTree`, `StorageTreeMap` oder `StorageTable` ein. `StorageTree` und `StorageTreeMap` werden mit den Default-Konstruktor erzeugt, die `StorageTable` mit einem Anfangswert, der darauf abzielt, die Table mit einer Größe von 125% der geschätzten Wortzahl zu initialisieren.

2.4 Alternative Direktzugriffsstrukturen

2.4.1 `StorageTree`

Der `StorageTree` ist ein simpler sortierter Binärbaum, er muß bei jedem Einfügevorgang bis zur Einfügestelle traversiert werden.

2.4.2 StorageTreeMap

Die StorageTreeMap ist ein Erweiterung der Klasse java.util.TreeMap, ein ausgewogener und sortierter Binärbaum.

2.4.3 StorageTable

Bei StorageTable handelt es sich um einen geänderten Quelltext der Klasse java.util.Hashtable.

2.5 Zeitmessung

Alle Direktzugriffsstrukturen implementieren die Klasse StorageInterface, was gewährleistet, daß sie alle über eine Funktion asDataArray() verfügen, die die Inhalte der gesamten Struktur auf ein Array of Data kopiert. Ein solches Array wird unter anderem von der QueryEngine benutzt, um mit einer binären Suche nach einem Wort zu suchen. Daher muß dieses Array sortiert sein, was Konsequenzen für die Implementation der asDataArray() in den einzelnen Direktzugriffsstrukturen hat.

So muß bei den Binärbäumen lediglich "pre-order" traversiert werden, da sie bereits sortiert sind. Bei der StorageTable hingegen wird das Array vor der Rückgabe mit einem modifizierten Quicksort-Algorithmus nach Wirth sortiert. Der von mir verwendete Algorithmus verwendet als Anker nicht das erste, sondern das Element in der Mitte und hat sich bei Laufzeitvergleichen als der klassischen Implementation (vgl. Wirth, 1974) als mindestens ebenbürtig erwiesen.

Aufgrund dieses Nachteils der StorageTable haben die Bäume in ihrer Natur als sortierte Struktur einen Vorteil gegenüber einer Hashtable, wenn auf ihnen gesucht werden soll. Da sie jedoch beim Einfügen immer traversiert werden müssen (bei der Suche nach der korrekten Einfügestelle) sind sie strukturbedingt theoretisch hierbei langsamer als eine Hashtable.

Um einen Vergleich der Laufzeitgeschwindigkeiten zu ermöglichen, ist bei der Erstellung der Direktzugriffsstruktur eine Zeitmessung integriert, nicht jedoch bei der Suche, da in jedem Fall mit der gleichen Methode gesucht wird, egal welche Struktur gewählt wurde.

Meine separaten Laufzeitvergleiche haben, je nach Entwicklungsumgebung, auf dem selben Rechner sehr unterschiedliche Ergebnisse geliefert. So ist mein selbst implementierter Baum in einer älteren Umgebung (JDK 1.2) sogar mit dem Umkopieren schneller als StorageTreeMap und StorageTable (letzerer hier am langsamsten) ohne Umkopieren bzw. Umkopieren und Sortieren. In einer neueren (JDK 1.3) jedoch ist StorageTree die langsamste Struktur, StorageTreeMap und StorageTable lieferten annähernd gleiche Ergebnisse, wobei alle schneller als in der alten Umgebung waren. Da in beiden Umgebungen StorageTreeMap trotz der zur Wahrung der Ausgewogenheit notwendigen Rotation von Teilbäumen beim Einfügen gute Ergebnisse lieferte, habe ich diese Struktur als weitere Option implementiert.

Da StorageTreeMap in beiden Umgebungen beim Einfügen ohne Umkopieren schneller ist als die anderen Strukturen beim Einfügen und Umkopieren bzw. Einfügen, Umkopieren und Sortieren und da sowohl Binäres Suchen als auch die Suche auf einem ausgewogenen Baum eine Rechenzeitkomplexität von O(log(n)) haben, zeigen meine Laufzeitvergleiche daß ein effizient arbeitender, ausgewogener und

sortierter Binärbaum, wie `java.util.TreeMap` es ist, für eine Suchmaschine in jedem Fall die überlegene Datenstruktur ist, wenn direkt auf dem Baum gesucht würde.

2.6 `QueryEngine`

Die `QueryEngine` sucht nach Wörtern und gibt die Resultate in ihrem Kontext auf die `GUI` und eine Datei aus. Sie erlaubt beliebige Kombinationen von Suchanfragen der Form >Wort1 Wort2 ... WortN< (und-Verknüpfung) und >"Wort1 Wort2 ... WortN"< (Kette von Worten).

2.7 `Engine`

Die `Engine` steuert den Hauptprogrammablauf: Das Initialisieren aller Komponenten sowie deren Verknüpfung zur Erzeugung des Index und der Suche nach Wörtern.

2.8 `GUI`

Die `GUI` verfügt über zwei unterschiedliche Modi zur Indizierung und zur Suche.

2.8.1 Indizierungsmodus

Im Indizierungsmodus steht folgende Funktionalität zur Verfügung: Auswahl des Verzeichnisses, Auswahl der zu berücksichtigenden Dateien, Wahl der Direktzugriffsstruktur im Index, Wahl der Direktzugriffsstruktur in den `KeyData`, Programmausgabe

2.8.2 Suchmodus

Im Suchmodus steht folgende Funktionalität zur Verfügung: Sucheingabe, Wahl der Ausgabedatei (wird angehangen), Wahl der Anzahl der auszugebenden Fundstellen im Kontext (aufgrund des relativ hohen Aufwandes zur Kontextausgabe), Wahl der Kontextbreite, Programmausgabe

2.9 `Main`

Die `Main` erzeugt die `GUI`.

3. Bibliographie

Lam, Sunny (2001) *The Overview of Web SearchEngy Engines*, Waterloo, Ontario, Kanada, NEC ResearchIndex. 30.8.2002 <http://citeseer.nj.nec.com/lam01overview.html>

Wirth, Niklas (1974) *Systematisches Programmieren*, Stuttgart, Teubner Verlag