Praktikum Algorithmische Anwendungen WS 2006/07
Ausarbeitung:
Schnelle Stringsuchalgorithmen
Boyer-Moore und Knuth-Morris-Pratt

Team A Rot
Daniel Baldes (Nr. 11041002, ai688@gm.fh-koeln.de)
Holger Pontius (Nr. 11036561, ai892@gm.fh-koeln.de)

31. Januar 2007

Inhaltsverzeichnis

1 Einleitung

Der Boyer-Moore- und der Knuth-Morris-Pratt-Algorithmus sind Stringsuchalgorithmen. Die Algorithmen werden dazu genutzt, um in einem Text die Anfangsposition eines bestimmten Teiltextes (Suchmuster) zu finden.

2 Der Boyer-Moore-Algorithmus

Der Algorithmus wurde im Jahre 1977 von Robert Stephen "Bob"Boyer und J Strother Moore an der Universität von Austin, Texas entwickelt. Der Algorithmus vergleicht immer vom Ende des Suchmusters zum Anfang desselben (der Einfachheit halber für Links-nach-rechts-Schreibung im Folgenden auch als ,,rechts nach links" beschrieben). Die Idee dahinter ist, dass man, wenn das Ende des Musters nicht zum Text passt, man am Anfang des Musters garnicht erst vergleichen muss, sondern direkt weiterspringen kann. Um dies bewerkstelligen zu können, bedient er sich zweier Tabellen, die vor der tatsächlichen Suche in einer Vorverarbeitung des Suchmusters angelegt werden. Diese Tabellen enthalten Sprungpositionen. Die zwei Heuristiken, die der Algorithmus benutzt, bewerten bei jedem fehlgeschlagenen Zeichenvergleich die aktuelle Situation, und ermessen anhand der Sprungtabelle, wieviele Zeichen übersprungen werden können. Auf diese Art und Weise ist der Algorithmus in der Lage, einen Text sehr schnell zu durchsuchen.

2.1 Die Funktionsweise der Heuristiken im Einzelnen

2.1.1 Bad-Character Heuristik

Der Algorithmus vergleicht immer Muster und Text von rechts nach links. Bei einer Abweichung (Mismatch) wird der entsprechende ,,schlechte Buchstabe" (bad charakter) in der entsprechenden Tabelle nachgeschlagen. Falls der Buchstabe laut Tabelle im Suchmuster vorkommt, schlägt die Heuristik einen Sprung um die passende Anzahl Buchstaben vor. Kommt der Buchstabe nicht im Suchmuster vor, schlägt die Heuristik vor, das Muster um seine gesamte Länge zu verschieben, da er innerhalb des Musters ja nicht mehr passen kann.

2.1.2 Good-Suffix Heuristik

Wenn beim finden einer Abweichung der Teil des Suchmusters, der schon erfolgreich verglichen wurde - also ein Suffix des Musters -, gleichzeitig im Muster an anderer Stelle vorkommt, schlägt die Heuristik vor, das Muster entsprechend zu verschieben, um das andere Vorkommen des Suffixes passend unter den Text anzuordnen. Falls das Suffix kein weiteres Mal im Suchmuster vorkommt, schlägt die Heuristik vor, das Muster um seine gesamte Länge zu verschieben.

2.2 Preprocessing des Suchmusters

Für die beiden Heuristiken ist es jeweils nötig, vor dem eigentlichen Suchlauf Sprungtabellen vorzubereiten. Dies wird im Folgenden für die beiden Heuristiken näher erläutert.

2.2.1 Bad-Character Heuristik

Die von der Bad-Character-Heuristik genutzte Sprungtabelle wird über eine Occurence-Funktion occ() erstellt. Diese Funktion gibt für jedes Alphabetsymbol die Position seines letzten (rechtesten) Vorkommens im Muster zurück, oder -1, falls das Symbol im Muster überhaupt nicht vorkommt.

Anhand des Beispielmusters ,,abcababca" und einem Alphabet a, b, c, d ergeben sich dann

```
Position:       0 1 2 3 4 5 6 7 8
Muster:         a b c a b a b c a
```

folgende Rückgabewerte für occ():

```
occ(muster,a) = 8
occ(muster,b) = 6
occ(muster,c) = 7
occ(muster,d) = -1
```

a	0
b	2
c	1

Hierfür wird die Tabelle zunächst mit dem Wert -1 initialisiert. Danach wird das Muster von rechts nach links durchlaufen, und immer der Abstand des Buchstabens vom Ende des Musters in der Tabelle gespeichert (also Länge − occ()).

2.2.2 Good-Suffix Heuristik

Die Sprungtabelle für die Good-Suffix-Heuristik ist komplizierter. Für jedes N größer oder gleich 0 und kleiner als die Länge des Suchmusters werden Sprungwerte berechnet. Hierzu werden die N rechtesten Zeichen des Suchmusters angeschaut, und das Zeichen links davon negiert davorgesetzt. Dieses Testmuster (mittlere Spalte der unten stehenden Tabelle) wird nach links geschoben, bis es wieder unter das Suchmuster passt. Die Anzahl der nötigen Verschiebungen wird in S abgelegt. Falls das Testmuster dabei links aus dem Suchmuster ,,herausragt", werden nur die Zeichen ,,innerhalb" des Suchmusters überprüft. Im Beispiel muss also das Testmuster für N=1 um 3 Zeichen nach links verschoben werden, um unter ,,ba" zu passen. Das Teytmuster für N=3 muss um 8 Zeichen verschoben werden, es wird dann nurnoch das ,,a" des Testmusters mit dem ersten ,,a" des Suchmusters verglichen und als Übereinstimmung gefunden.

```
N       abcababca  S
0              a   1
1           ca     3
2    bca           8
3   abca           8
4     babca        5
5    ababca        5
6   cababca        5
7  bcababca        5
8 abcababca        5
```

2.3 Suche anhand eines Beispiels

Das Suchmuster wird linksbündig unter dem Text angeordnet, es wird von rechts nach links verglichen:

```
Position: 0123456789…
Text:     abcabababcababcabc
Pattern:  abcababca
```

Mismatch, die Bad-Charakter Heuristik greift und schlägt vor, das Pattern um 2 Zeichen nach rechts zu verschieben:

```
Position: 0123456789…
Text:     abcababababcabcabc
Pattern:    abcababca
```

Mismatch, die Good-Suffix-Heuristik erkennt, dass abca auch weiter links im Pattern vorkommt, und schlägt vor, das Pattern entsprechend um 5 Zeichen zu verschieben. Die Bad-Character-Heuristik hätte um 1 Zeichen nach rechts verschoben, und das linke b des Patterns unter dem des Textes angeordnet.

```
Position: 0123456789…
Text:     abcababababcabcabc
Pattern:          abcababca
```

Das Muster wurde gefunden.

3 Der Knuth-Morris-Pratt-Algorithmus

Der Algorithmus wurde 1977 von Donald Knuth, Vaughan Pratt und unabhängig davon von James H. Morris erfunden, und daraufhin von allen gemeinsam publiziert.

3.1 Preprocessing des Suchmusters

Das Suchmuster wird auf Zeichenfolgen untersucht, die ein möglichst langes Präfix des Suchpatterns selbst sind. Beispiel anhand des Suchmusters ,,abcababca":

```
Position:       0 1 2 3 4 5 6 7 8
Muster:         a b c a b a b c a
1. Präfix:            a
2. Präfix:            a b
3. Präfix:                a
…
letztes Präfix:           a b c a
```

Damit entsprechend sinnvoll ,,gesprungen" werden kann, wird hierfür eine Tabelle angelegt, in der unter jedes Zeichen die Länge des längstmöglichen vorangehenden Vorkommens eines Präfixes geschrieben wird. Mit Ausnahme des ersten Buchstabens, also des tatsächlichen Präfixes. Unter den Ersten Buchstaben wird als Sonderfall -1 geschrieben.

```
Position:       0 1 2 3 4 5 6 7 8
Muster:         a b c a b a b c a
Präfixlänge:   -1 0 0 1 2 1 2 3 4
```

Mit diesen Informationen lassen sich während der Suche doppelte Vergleiche vermeiden, da bei einem Sprung das Präfix, falls passend, nicht noch einmal mit dem zu durchsuchenden Text verglichen werden muss.

3.2 Suche anhand eines Beispiels

Das Suchmuster wird linksbündig unter den Text geschrieben, es wird von links nach rechts laufend verglichen, bis eine Abweichung (oder im Erfolgsfall keine Abweichung) gefunden wird:

```
Position: 0123456789…
Text:     abcabababcababcabc
Pattern:  abcababca
```

Da bei Position 3 ein Präfix auftaucht, wird das Suchmuster bis dorthin verschoben, und wieder mit dem Text verglichen bis zur nächsten Abweichung, usw, usf.:

```
Position: 0123456789…
Text:     abcabababcababcabc
Pattern:     abcababca
```

```
Position: 0123456789…
Text:     abcababababcababcabc
Pattern:       abcababca
```

```
Position: 0123456789…
Text:     abcababababcababcabc
Pattern:         abcababca
```

Wenn alle Zeichen übereinstimmen, wird ein Treffer ausgegeben. Der Algorithmus ist beendet, sobald das Textende erreicht ist, oder das Suchmuster im Text gefunden wurde.

4 Empirische Analyse

Um das Verhalten des Algorithmus empirisch zu untersuchen, haben wir ihn in Java implementiert. In Anlehnung an [1] messen wir die Kosten des Algorithmus anhand der Anzahl der Zugriffe auf den zu durchsuchenden Text. Die Kosten für die Berechnung der Sprungtabellen werden hierbei außer acht gelassen. Zu diesem Zweck wurde eine Proxyklasse entwickelt, die Zugriffe auf die ,,charAt(index)"-Methode einer Zeichensequenz während der Abarbeitung des Algorithmus zählt. Das gezählte Ergebnis wird dann durch die Anzahl der Zeichen bis zur Fundposition dividiert und mit der Länge des Suchmusters in Relation dargestellt.

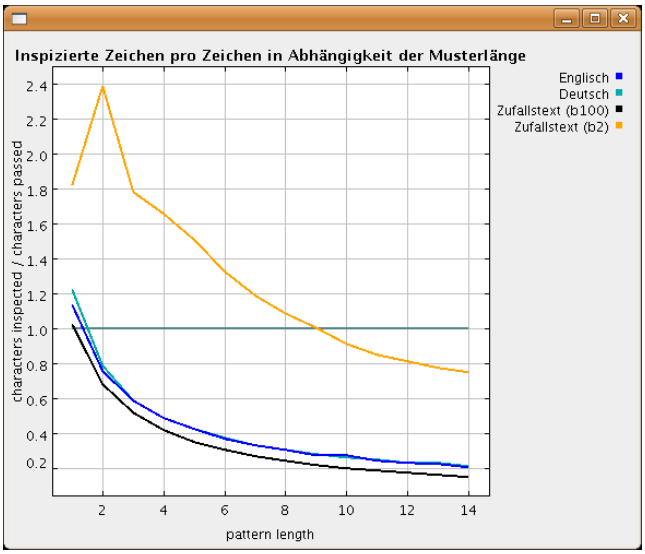

Abbildung 1: Zugriffe pro Zeichen (BM)

Wie in [1] wählen wir aus einem Text zufällig je 300 Suchmuster der Längen 1-14 aus und bilden den Durchschnittswert für alle Muster pro Musterlänge.

Die waagerechte Linie in Abbildung 1 markiert das Verhältnis 1.0; hier wurde auf jedes Zeichen des zu durchsuchenden Textes einmal zugegriffen. Wie zu sehen ist, fällt das Verhältnis beim Boyer-Moore Algorithmus mit zunehmender Suchmusterlänge deutlich unter diesen Wert. Die güne und die blaue Linie zeigen den Verlauf für englischen und deutschen Text an. Die schwarze Linie beschreibt den Verlauf für eine zufällig generierte Zeichenfolge aus einem Alphabet von 100 verschiedenen Zeichen; die gelbe Linie für eine zufällig generierte Folge aus einem Alphabet von zwei Zeichen.

4.1 Boyer-Moore und Knuth-Morris-Pratt im Vergleich

Die nebenstehende Abbildung zeigt das Verhalten des KMP-Algorithmus unter denselben Bedingungen wie BM im vorangegangenen Abschnitt. Deutlich zu sehen ist, dass der Algorithmus grundsätzlich jedes Zeichen mindestens einmal untersucht. Dieses Verhalten ist offenbar weitgehend unabhängig von der Länge des Suchmusters; nur bei kurzen Suchmustern wird statistisch häufiger auf den zu durchsuchenden Text zugegriffen, was sich vor allem bei dem aus nur zwei Zeichen bestehenden Alphabet bemerkbar macht. Aus den Diagrammen geht sichtbar hervor, dass der Boyer-Moore Algorithmus bereits ab einer Suchmusterlänge von zwei Zeichen weniger Zugriffe auf den Suchtext benötigt als der KMP-Algorithmus.

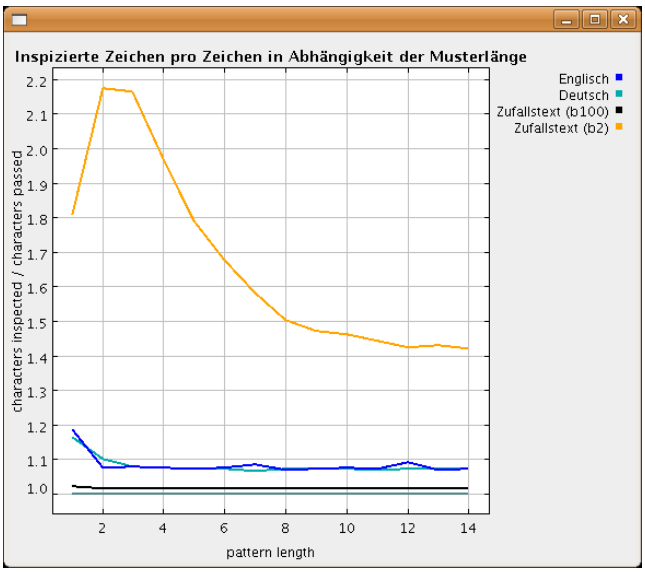

Abbildung 2: Zugriffe pro Zeichen (KMP)

5 Theoretische Analyse

Die Theoretische Analyse des Boyer-Moore Algorithmus stellt sich als nicht trivial heraus. Boyer und Moore ermitteln in [1] eine ,,sublineare" Laufzeit für den average-case. Die durchschnittliche Laufzeit für natürliche Sprachen betrage $\Theta(\frac{n}{m})$. Im worst-case benötigt man nach aktueller Meinung $3N$ Vergleiche, um alle Vorkommen eines Suchmusters in einem Text zu finden; daher ist die worst-case Komplexität $\Theta(n)$, unabhängig davon, ob der Text das Suchmuster enthält oder nicht. Dieser Beweis wurde laut [3] von Richard Cole 1991 geführt [2]. Andere Quellen geben die worst-case Komplexität mit $\Theta(nm)$ an.

Die Laufzeit des KMP-Algorithmus ist garantiert linear [4].

6 BM im praktischen Einsatz

Der Boyer-Moore-Algorithmus und verschiedene Variationen stellen sich in der Praxis als der schnellste bekannte Stringsuchalgorithmen heraus. Sie finden vielfältige Anwendung; zum Beispiel im beliebten RDBMS MySQL [5] und in GNU grep [6]. Oftmals wird für Suche in kürzeren Texten anstelle des BM-Algorithmus ein naiver Suchalgorithmus eingesetzt, der keinen Preprocessing-Schritt benötigt.

Literatur

[1] Robert S. Boyer, J. Strother Moore - A Fast String Searching Algorithm, Communications of the ACM, Oct. 1977, Volume 20, Number 10, Page 762

[2] R. COLE, Tight bounds on the complexity of the Boyer-Moore algorithm, Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, (1991)

[3] http://en.wikipedia.org/wiki/Boyer_moore_algorithm (Stand 25. Januar 2007)

[4] http://de.wikipedia.org/wiki/Knuth-Morris-Pratt-Algorithmus (Stand 25. Januar 2007)

[5] http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html (Stand 25. Januar 2007)

[6] http://directory.fsf.org/grep.html (Stand 25. Januar 2007)