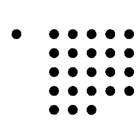

Fachhochschule Köln
University of Applied Sciences Cologne

Algorithmische Anwendungen

WS 2005 / 2006

Praktikum

Prof. Dr. Heiner Klocke

Thema: Erweiterungen und Verbesserungen des Boyer-Moore-String-Searching-Algorithmus

Sabrina Hackl
110 275 30
Gruppe D gelb

Inhaltsverzeichnis

1 Einleitung

Die grundsätzliche Aufgabenstellung besteht darin, eine Zeichenkette – ein so genanntes Muster – in einem (längeren) Text zu finden. Sowohl Text als auch Muster können auf einem beliebigen, aber natürlich gemeinsamen Alphabet basieren.
Es handelt sich hierbei um eine typische Funktion der Textverarbeitung und ein entsprechendes Softwaresystem sollte daher eine effiziente Lösung bereitstellen.
Als Maß der Effizienz können wir einfach die Anzahl der notwendigen Vergleiche zwischen den Zeichen betrachten.

1.1 Grundproblem Textsuche

Ein naives Verfahren zur Suche eines bestimmten Musters in einem Text könnte darin bestehen, das Muster am ersten Buchstaben des Textes anzulegen, es mit dem entsprechenden Textstück zu vergleichen und, wenn beide nicht übereinstimmen, das Muster Buchstabe um Buchstabe im Text weiter zu verschieben, solange bis entweder das Muster im Text gefunden wurde oder der Text zu Ende ist.

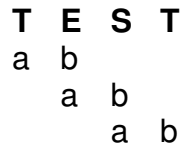

Abb. 1 Naives Suchverfahren

Die zu suchende Zeichnfolge, hier *ab* wird mit jeder Position im Text verglichen. Stimmt das erste Zeichen mit der momentanen Position im zu durchsuchenden Text nicht überein, wird die gesuchte Zeichnfolge einfach um eine Position weiter geschoben, solange bis eine Übereinstimmung des ersten Zeichens vorliegt. Erst nach dieser ersten Übereinstimmung kann das zweite Zeichen verglichen werden. Falls aber das zweite Zeichen nicht mit der nächsten Position gleich ist, wird das gesamte Wort wieder im Text um eine Position nach rechts geschoben.

Folgender Pseudocode soll die Funktionsweise dieser Lösungsmöglichkeit veranschaulichen:

text[0…n-1]	→	*der zu durchsuchende Text*
pat[0…m-1]	→	*das gesuchte Muster*
i	→	*Position im Text*
n	→	*Länge des zu durchsuchenden Textes*
m	→	*Länge des gesuchten Musters*

```
for i:=0 to n-m do
        j := 0;
```

```
        while j < m /\ pat[j] = text[i+j] do
                j := j+1;
        od;
        if j >= m then return i fi
od;
return -1;
```

Wie unschwer zu erkennen ist, handelt es sich dabei nicht gerade um eine effektive Suche im Sinne der Laufzeit.

1.2 Laufzeit

In Abschnitt 1.1 wird ein Pseudocode für die einfachste Lösung vorgestellt. Dieser besteht aus zwei ineinander verschachtelten Schleifen. Eine äußere for-Schleife und eine inner while-Schleife.

Bei geschachtelten Schleifen wird für jede Iteration der äußeren Schleife ein kompletter Durchlauf der inneren Schleife durchgeführt. Demnach ergibt sich die Gesamtlaufzeit wie folgt:

$$O((n-m)*m) = O(nm)$$

Aus dieser Laufzeitanalyse ergibt sich folglich, dass im ungünstigsten Fall fast jedes Mal das gesamte Muster verglichen werden müsste.

Würde nun eine solche Suche in längeren Mustern stattfinden, wäre der Zeitaufwand viel zu hoch. Eine Verbesserung der Laufzeitkomplexität wäre hier enorm von Vorteil.

1.3 Erster Lösungsansatz

1976 erschien unter *Xerox, Palo alto Research Center*, ein Artikel mit dem Titel *A fast String searching Algorithm,* verfasst von Robert S. Boyer und J. Strother Moore.
Wie der Titel schon sagt, verspricht dieser Algorithmus eine schnellere Suche in Texten als die bisher vorgestellte Lösung.

2 Boyer-Moore-Algorithmus

R. S. Boyer ist zur Zeit Professor für Informatik, Mathematik und Philosophie an der Universität Texas.
Abbildung 1 zeigt ein Bild aus dem Jahr 2004.

Abb. 1 R. S. Boyer

J. Strother Moore ist ebenfalls an der Universität in Texas als Admiral B.R. Inman Centennial Chair in Computing Theory.
Abbildung 2 zeigt J. Strother Moore.

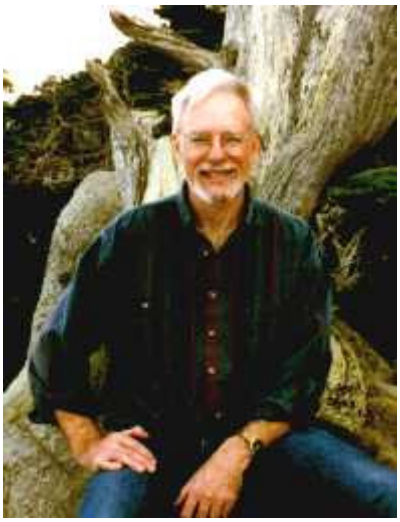

Abb. 2 J. Strother Moore

R. S. Boyer und J. Strother Moore entwickelten einen Algorithmus (Boyer-Moore-Algorithmus) welcher es ermöglicht die Verschiebedistanz der Zeichen zu vregrößern.

2.1 Grundidee

Die Grundidee die sich hinter diesem Algorithmus verbirgt ist die, dass die Zeichen von rechts nach links verglichen werden. Stimmt bereits das letzte Zeichen nicht überein, kann um die gesamte Länge des gesuchten Strings verschoben werden. Liegt so ein günstiger Fall vor, können die benötigten Vergleiche der in Abschnitt 1 vorgestellten Lösung um ein vielfaches reduziert werden.

2.2 Funktionsweise

Gestartet wird bei dieser Suche ebenfalls, wie in Abschnitt 1 beschrieben, mit dem „Untereinanderlegen“ des zu durchsuchenden Textes und des zu suchenden Musters. Wie bereits erwähnt wird beim ersten Vergleich nicht mit dem linken Zeichen, sondern mit dem rechten Zeichen auf Übereinstimmung geprüft.
Tritt bei dem rechten Zeichen bereits eine Nichtübereinstimmung ein, so kann um die ganze Länge des zu suchenden Musters verschoben werden.

```
T E S T
A B
    A B
```

Abb. 2 Gefundene Verschiebedistanz

Zur Erinnerung hier noch einmal das gleiche Beispiel mit der Lösung aus Abschnitt 1:

```
T E S T
A B
  A B
```

Abb. 3 Gefundene Verschiebedistanz nach Lösung aus Abschnitt 1

Anhand der beiden Tabellen kann leicht erkannt werden, dass der Boyer-Moore-Algorithmus große Einsparungen bei den Vergleichen erzielen kann. Diese Art zu Vergleichen nennt sich *"Schlechtes Zeichen"-Strategie* (*bad character heuristics*)“. Wie der Name schon sagt, handelt es sich hier um Vergleiche von einzelnen Zeichen die zu einer Nichtübereinstimmung führen.

Diese Strategie beinhaltet zwei wesentliche Unterscheidungsfälle:

Unterscheidungsfall 1 *(„Schlechtes Zeichen“-Strategie)**:***
Befindet sich das Zeichen das zur Nichtübereinstimmung führt noch einmal im Muster, so kann das Muster nur bis zu diesem Zeichen im Muster verschoben werden.

```
I C H S U C H E
T E S T
  T E S T
```

Abb. 4 Fall 1

Unterscheidungsfall 2 *(„Schlechtes Zeichen“-Strategie)***:**
Das Zeichen das zur Nichtübereinstimmung führt, tritt nicht weiter im Muster auf, dann kann das Muster um seine komplette Länge verschoben werden.

I C H S U C H E
T E E
T E E

Abb. 5 Fall 2

Um die Verschiebedistanz der *bad character heuristics* zu berechnen wird eine *last*-Tabelle benötigt.

In der *last*-Tabelle wird die für jedes Zeichen des Textes (oder praktischerweise des zugrunde liegenden Alphabets) die sichere Verschiebedistanz gespeichert.
Die erste Frage die sich hier sicherlich stellt ist: „Wie wird die Verschiebedistanz ermittelt?“.

Diese Frage beantwortet sich einfacher mit dem Quellcode, welcher in Abbildung 3 dargestellt wird.

```
private static int[] initLast(String pat){

    int[] last = new int[256];
    int i;

    for(i = 0; i<256; i++)
        last[i] = -1;

    for (i=0; i<pat.length(); i++)
        last[pat.charAt(i)] = i;

    return last;
}
```

Abb. 3 Quellcode last-Tabelle initialisieren

Die Mehtode *initLast(String pat)* bekommt als Parameter den zu suchenden Text übergeben (Im Folgenden wird dafür der Begriff *pattern*, kurz *pat,* benutzt).
Was macht nun diese Mehtode?
In der ersten Zeile wird ein Integer-Array der Größe 256 erstellt und alle 256 Felder mit dem Wert *-1* initialisiert.

256 entspricht der Anzahl des ASCII-Zeichensatzes und wurde deswegen gewählt, da angenommen wird, dass nur solche Zeichen in Muster und Text verwendet werden. Alternativ kann auch der komplette Unicode-Zeichensatz verwendet werden, was allerdings zu einem Array der Größe 2^{16} führen würde und hier nicht vonnöten ist.
Mit der festgelegten Größe 256 ist es also möglich nach allen Zeichen des ASCII-Zeichensatzes zu suchen.
Da es sehr wahrscheinlich ist, dass nicht alle 256 Zeichen in dem *pattern* vorkommen, wird als erstes jedes Zeichen mit dem Wert *-1* initialisiert. Die *-1* steht für „*nicht vorkommen im pattern*“.

In der zweiten for-Schleife wird für jedes Zeichen im pattern seine letzte Position bestimmt und im Array gespeichert.
Wird zum Beispiel das *pattern „acbab cbayjr“* an initLast(String pat) übergeben, ergibt sich folgende Belegung der Felder in der Tabelle:

x: 0-->-1	x: 1-->-1	x: 2-->-1	x: 3-->-1
x: 8-->-1	x: 9-->-1	x: 10-->-1	x: 11-->-1
x: 16-->-1	x: 17-->-1	x: 18-->-1	x: 19-->-1
x: 24-->-1	x: 25-->-1	x: 26-->-1	x: 27-->-1
x: 32-->5	x: 33-->-1	x: 34-->-1	x: 35-->-1
x: 40-->-1	x: 41-->-1	x: 42-->-1	x: 43-->-1
x: 48-->-1	x: 49-->-1	x: 50-->-1	x: 51-->-1
x: 56-->-1	x: 57-->-1	x: 58-->-1	x: 59-->-1
x: 64-->-1	x: 65-->-1	x: 66-->-1	x: 67-->-1
x: 72-->-1	x: 73-->-1	x: 74-->-1	x: 75-->-1
x: 80-->-1	x: 81-->-1	x: 82-->-1	x: 83-->-1
x: 88-->-1	x: 89-->-1	x: 90-->-1	x: 91-->-1
x: 96-->-1	x: 97-->8	x: 98-->7	x: 99-->6
x: 104-->-1	x: 105-->-1	x: 106-->10	x: 107-->-1
x: 112-->-1	x: 113-->-1	x: 114-->11	x: 115-->-1
x: 120-->-1	x: 121-->9	x: 122-->-1	x: 123-->-1

Abb. 4 Ausschnitt last-Tabelle

Wie bereits erwähnt steht *-1* für „*nicht vorkommen in dem pattern*“. Das heißt die rot gefärbten Werte müssen im Zusammenhang mit pattern stehen.
Wirft man einen Blick auf den ASCII-Zeichensatz, wird schnell deutlich was hier gespeichert wurde.
Der Wert 32 steht laut ASCII-Zeichensatz für ein Leerzeichen. In Feld 32 steht wiederum der Wert 5. Laut Definition sollte das die letzte Position des Zeichens sein, an welcher es vorkommt.

a	**c**	**b**	**a**	**b**		**c**	**b**	**a**	**y**	**i**	**r**
0	1	2	3	4	5	6	7	8	9	10	11

Abb. 6 Übergebener String als char-Array dargestellt

Betrachtet man den übergebenen String als char-Array erkennt man deutlich dass in Feld 5 ein Leerzeichen eingetragen ist. Da es das einzige Leerzeichen in dem *pattern* ist, ist es auch seine letzte Position.

Anders sieht es nun mit Zeichen aus, die mehrfach vorkommen. Hier wird jedes Mal nach erneutem Auffinden des Zeichens der bereits gesetzte Wert überschrieben, was das komplette Durchlaufen des patterns mittels der for-Schleife garantiert.

Ein Beispiel hierfür ist das Zeichen ’a’. Zuerst wird der Wert 0 in Feld 97 geschrieben. Steht die Schleife an Position 3, an welcher erneut ein ’a’ zu finden ist, wird 3 in Feld 97 geschrieben. Letztendlich findet sich ’a’ noch mal an Stelle 8, was der tatsächlichen letzten Position entspricht und endgültig in Feld 97 gespeichert wird.

Nach demselben Prinzip wird auch die letzte Position der anderen Zeichen bestimmt und gespeichert.

Da die „Schlechtes Zeichen“-Strategie nicht immer ein zufrieden stellendes Ergebnis liefert, bedient sich der Boyer-Moore-Algorithmus einer weiteren Strategie, der *"Gutes Ende"-Strategie* (*good suffix heuristics*).

Stimmt am Ende mehr als ein Zeichen überein, jedoch nicht das ganze Muster, und kommt dieses Präfix in dem Muster noch mal vor, so wird das Muster soweit verschoben, bis das Präfix mit der Zeichenfolge übereinstimmt.

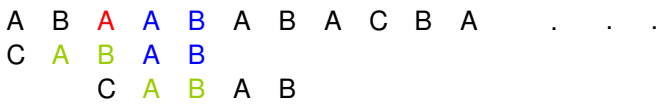

Abb. 7 Fall 3

Für die Berechnung der Verschiebedistanz wird eine sogenannte *shift*-Tabelle erstellt.
Die *shift*-Tabelle ist etwas komplizierter zu berechnen und zu implementieren als die *last*-Tabelle.

Zunächst wird die Frage geklärt was die *shift*-Tabelle macht:
In der *shift*-Tabelle wird, die zu jedem Suffix des Musters, das eventuell zu einer Nichtübereinstimmung führen kann, die sichere Verschiebedistanz gespeichert. Konkret bedeutet dies, dass nach der Übereinstimmung zwischen dem Teilmuster *pat[j+1…m-1]* und dem Text und einer Nichtübereinstimmung von *pat[j]* der Wert *shift[j]* die mögliche Distanz zur Verschiebung liefert.

Damit die richtige Verschiebedistanz berechnet werden kann, müssen zwei Fälle in betracht gezogen werden:

Fall 1
Das übereinstimmende Suffix kommt noch an anderer Stelle im Muster vor.

Fall 2
Nur ein Teil des übereinstimmenden Suffixes kommt am Anfang des Musters vor.

Um die beiden Fälle genauer zu beschreiben und zu erklären sind einige Begriffsbestimmungen nötig.

Angenommen es liegt ein Text folgender Form vor:

a b a c a b

Echte Präfixe sind:

ε, a, ab, aba, abac, abaca

Echte Suffixe sind:

ε, b, ab, cab, acab, bacab

Ränder sind:

ε, ab

Der Rand *ab* hat die *Breite* 2

Ein Rand ist also eine Zeichenreihe die wiederholt in einem Muster vorkommt. Anhand dieser Grundlagen kann die *shift*-Tabelle nun erklärt werden.

Die Tabelle enthält also für jedes *j* die Distanz der Verschiebung, wenn das an Position *j* beginnende Suffix des Musters übereinstimmt und an Position *j-1* eine Nichtübereinstimmung auftritt.

Beispiel:
Gegeben ist ein pattern der Form:

a b b a b a b

Suffixe sind:

ε, b, ab, bab, abab, babab, bbabab

In folgendem ersten Teil-Algorithmus wird ein Array *suffix* berechnet. Für ein an Position *i* beginnendes Suffix des Musters enthält der Eintrag *suffix*[*i*] die Anfangsposition seines breitesten Randes. Für das Suffix ε, das an Position *m* beginnt, wird *suffix*[*m*] = *m*+1 gesetzt.

Des weiteren wird ein Array *shift* benötigt, der für jedes *i* angibt, um wie viel das Muster geschoben werden kann, wenn ein Mismatch an Position *i* – 1 auftritt, d.h. wenn das an Position *i* beginnende Suffix übereingestimmt hat.

Folgender Quellcode deckt Fall 1 der *Gutes-Ende-Strategie* ab.

```
private static int[] initShift(String pat){
    int m = pat.length();
    int i=m, j=m+1;

    int suffix [] = new int[m+1];
    int shift [] = new int[m+1];

    suffix[i]=j;

    while (i>0){
        while (j<=m && pat.charAt(i-1)!= pat.charAt(j-1)){
            if (shift[j]==0) shift[j]=j-i;
            j=suffix[j];
        }
        i--; j--;
        suffix[i]=j;
    }
```

Abb. 5 initShift Abdeckung von Fall 1

Wird an die Methode initShift das pattern aus vorherigem Beispiel übergeben, erhält man wie folgt eine erste Belegung der Felder:

0	1	2	3	4	5	6	7
pattern-->a	pattern-->b	pattern-->b	pattern-->a	pattern-->b	pattern-->a	pattern-->b	
suffix-->5	suffix-->6	suffix-->4	suffix-->5	suffix-->6	suffix-->7	suffix-->7	suffix-->8
shift-->0	shift-->0	shift-->0	shift-->0	shift-->2	shift-->0	shift-->4	shift-->1

Abb. 6 Ausgabe initShift Fall 1

Noch mal zur Erinnerung die Suffixe des patterns:

ε, b, ab, bab, abab, babab, bbabab

Was steht also jetzt in der Tabelle?
Im Array s*uffix* steht wie vorher beschrieben, die Anfangsposition seines breitesten Randes. In *shift* steht die vorläufig ermittelte Verschiebedistanz.

Als Erstes wird das Suffix *b* betrachtet, als Einzelnes Zeichen hat es als Rand ε.

Das an Position 5 beginnende Suffix ab hat als breitesten Rand ε, dieser beginnt an Position 7. Daher ist *suffix*[5] = 7.

Drittes Suffix ist bab, breitester Rand ist b das an Stelle 6 steht was in die Suffixtabelle eingetragen wird.

Das an Position 2 beginnende Suffix *babab* hat als breitesten Rand *bab*, dieser beginnt an Position 4. Daher ist *suffixt*[2] = 4.

Dieser Rand lässt sich nicht fortsetzen, denn es ist $p[1] \neq p[3]$. Die Differenz 4 - 2 = 2, j-i entspricht der Distanz der beiden gefundenen Suffixen, ist daher die Schiebedistanz, wenn *bab* übereingestimmt hat und dann ein Mismatch auftritt. Somit ist *shift*[4] = 2.

Somit wäre Fall 1 geklärt, bleibt noch Fall 2 (Nur ein Teil des übereinstimmenden Suffixes kommt am Anfang des Musters vor).

Der Eintrag *suffix[0]* enthält die Anfangsposition des breitesten Randes des ganzen Musters. In obigem Beispiel also 5, da der Rand *ab* an Position 5 beginnt. Tritt das "*gute Ende*", also das übereinstimmende Suffix des Musters nicht an anderer Stelle im Muster auf, wie eben in Fall 1 dargestellt, so kann das Muster so weit geschoben werden, wie es sein Rand zulässt. Maßgebend ist dabei jeweils der breiteste Rand, sofern er nicht breiter als das übereinstimmende Suffix ist.
Im Folgenden zweiten Teil des Algorithmus werden alle noch freien Einträge des Arrays *shift* belegt.

Eingetragen wird zunächst überall die Anfangsposition des breitesten Randes des Musters, diese ist *j = suffix[0].* Ab Position *i = j* wird auf den nächst schmaleren Rand suffix[*j*] umgeschaltet usw.

```
        j = suffix[0];
        for (i=0; i<=m; i++){
            if (shift[i]==0) shift[i]=j;
            if (i==j) j=suffix[j];
        }

        return shift;
    }
```

Abb. 7 initShift Abdeckung von Fall 2

Eine Testausgabe sieht wie folgt aus (Eingabe war *abbabab*):

```
i=7 j=8
setze suffix an stelle 6 auf 7
i=6 j=7
vergleiche a an der Stelle 5 mit b an der stelle 6
```

a	b	b	a	b	a	b

```
setze shift an der stelle 7 auf 1
setze j auf 8
setze suffix an stelle 5 auf 7
i=5 j=7

vergleiche b an der Stelle 4 mit b an der stelle 6
```

a	b	b	a	b	a	b

```
setze suffix an stelle 4 auf 6
i=4 j=6

vergleiche a an der Stelle 3 mit a an der stelle 5
```

a	b	b	a	b	a	b

```
setze suffix an stelle 3 auf 5
i=3 j=5

vergleiche b an der Stelle 2 mit b an der stelle 4
```

a	b	b	a	b	a	b

```
setze suffix an stelle 2 auf 4
i=2 j=4

vergleiche b an der Stelle 1 mit a an der stelle 3
```

a	b	b	a	b	a	b

```
setze shift an der stelle 4 auf 2
setze j auf 6
setze shift an der stelle 6 auf 4
setze j auf 7
setze suffix an stelle 1 auf 6
```

```
i=1 j=6

vergleiche a an der Stelle 0 mit a an der stelle 5
```

a	b	b	a	b	a	b

```
setze suffix an stelle 0 auf 5
```

Die endgültige shift-Tabelle hat damit folgenden Inhalt:

```
0            1            2            3            4            5            6            7
pattern-->a  pattern-->b  pattern-->b  pattern-->a  pattern-->b  pattern-->a  pattern-->b
-------------------------------------------------------------------------------------------------------------------

suffix-->5   suffix-->6   suffix-->4   suffix-->5   suffix-->6   suffix-->7   suffix-->7   suffix-->8
shift-->5    shift-->5    shift-->5    shift-->5    shift-->2    shift-->5    shift-->4    shift-->1
```

Abb. 8 Ausgabe initShift Fall 2

Hier noch einmal der vollständige Quellcode von initShift:

```
private static int[] initShift(String pat){
    int m = pat.length();
    int i=m, j=m+1;

    int suffix [] = new int[m+1];
    int shift [] = new int[m+1];

    suffix[i]=j;
    while (i>0){
        while (j<=m && pat.charAt(i-1)!=pat.charAt(j-1)) {
            if (shift[j]==0) shift[j]=j-i;
            j=suffix[j];
        }
        i--; j--;
        suffix[i]=j;
    }

    j = suffix[0];
    for (i=0; i<=m; i++){
        if (shift[i]==0) shift[i]=j;
        if (i==j) j=suffix[j];
    }

    return shift;
}
```

Abb. 9 initShift

Zu guter Letzt fehlt noch der eigentlich Boyer-Moore-Algorithmus, welcher aufgrund der vorangegangenen Tabellen einfach zu implementieren ist.

```
public static int bmSearch(String text, String pat){
    int [] last = initLast(pat);
    int [] shift = initShift(pat);

    int i=0;
    while(i <= text.length()-pat.length()){
        int j = pat.length() - 1;
        while(j>=0 && pat.charAt(j) == text.charAt(i+j))
            j--;
        if(j<0)
            return i;
        else
            i += Math.max(shift[j], j-last[text.charAt(i+j)]);
    }

    return -1;
}
```

Abb. 10 Implementierung des Boyer-Moore-Algorithmus

Als Parameter werden der zu durchsuchende Text und das zu suchende *pattern* übergeben. In den ersten beiden Zeilen werden die shift- und die last-Tabelle erstellt.
Danach erfolgt der Vergleich der einzelnen Zeichen. Wird eine Übereinstimmung festgestellt, wird *j* solange inkrementiert bis entweder *j* null ist, damit ist die Stelle an der das pattern vorhanden ist, gefunden, oder ein Mismatch erfolgt. In diesem Fall wird anhand der *shift*- bzw. *last*-Tabelle die größtmögliche Verschiebedistanz ermittelt.
Wurde das pattern gefunden, wird die Startposition in dem Text, an welcher das gesuchte pattern beginnt, zurückgegeben. Ist der Rückgabewert *-1*, wurde das pattern nicht gefunden.

Mittels der hier vollzogenen Vergleiche kann die Laufzeit, im Vergleich zu dem naiven Ansatz, wesentlich verbessert werden.

2.3 Laufzeit

Der Aufwand des Boyer-Moore-Algorithmus ergibt sich aus dem Aufwand der Vorbereitungsphase und dem eigentlichen Suchalgorithmus.

Zuerst soll die Laufzeit zur Initialisierung der lastTabelle ermittelt werden.
Hier noch mal der Quellcode:

```
private static int[] initLast(String pat){

    int[] last = new int[256];
    int i;

    for(i = 0; i<256; i++)
        last[i] = -1;

    for (i=0; i<pat.length(); i++)
        last[pat.charAt(i)] = i;

    return last;
}
```

Abb. 11 Quellcode initLast

In der ersten *for-Schleife* wird ein Array mit der Größe des Alphabets erstellt und mit -1 initialisiert. Das bedeutet, dass hier eine Laufzeit von *O(n)* vorliegt, wobei *n* die Größe des Alphabets ist.
In der zweiten *for-Schleife* werden die Werte der jeweiligen Zeichen im Array belegt. Daraus folgt dass eine Laufzeit von *O(m)* entsteht, wobei *m* die Länge des pattern darstellt.
Diese beiden Laufzeiten müssen nun addiert werden um die gesamt Laufzeit ermitteln zu können. Daraus ergibt sich

$$O(n) + O(m) = O(n+m)$$

```
private static int[] initShift(String pat){
    int m = pat.length();
    int i=m, j=m+1;

    int suffix [] = new int[m+1];
    int shift [] = new int[m+1];

    suffix[i]=j;
    while (i>0){
        while (j<=m && pat.charAt(i-1)!=pat.charAt(j-1)) {
            if (shift[j]==0) shift[j]=j-i;
            j=suffix[j];
        }
        i--; j--;
        suffix[i]=j;
    }

    j = suffix[0];
    for (i=0; i<=m; i++){
        if (shift[i]==0) shift[i]=j;
        if (i==j) j=suffix[j];
    }

    return shift;
}
```

Abb. 12 Quellcode initShift

Die Initialisierung der shift-Tabelle erfolgt in zwei Schritten.

Der worst-Case würde eintreten wenn alle Zeichen unterschiedlich sind, also keine Zeichen doppelt in dem pattern vorkommen. Das würde für die erste while-Schleife bedeuten dass diese bereits, zusammen mit der inneren while-Schleife, einen Aufwand von $O(m^2)$ benötigen würde. Dazu kommen noch 2xm Vergleiche. Also ein Gesamtaufwand von

$$O(m^2) + O(m) = O(m^2)$$

Für die darauf folgende for-Schleife würde das bedeuten, dass die shift-Tabelle bereits vollständig belegt ist, und daher ein Aufwand von

$$O(m) + O(m) + O(m) = O(m)$$

Insgesamt ergibt sich im worst-Case eine Laufzeit von

$$O(m^2) + O(m)$$

Ein best-Case liegt vor wenn alle Zeichen gleich sind.
Die while-Schleife würde einen Aufwand von ***O(m)*** verursachen, da die zweite Schleife aufgrund der Gleichheit der Zeichen, nicht betreten werden kann. Allerdings bedeutet das für die nächste for-Schleife, dass die shift-Tabelle mit keinen Werten, außer 0, belegt ist, und somit 2xm Vergleiche plus 2xm Zuweisungen. Was zu einer Laufzeit von

$$O(m)$$

Im worst-Case entsteht in der Vorbereitungsphase also ein Aufwand von

$$O(m^2) + O(m) + O(m+n) = O(m^2+n)$$

und im best-Case

$$O(m) + O(m+n) = O(m+n)$$

Der eigentliche Algorithmus benötigt im besten Fall nur einen Aufwand von ***O(m/n),*** wobei n die Länge des pattern darstellt und m die Länge des zu durchsuchenden Textes. Diese Herleitung ist einfach nachzuvollziehen, wenn man bedenkt dass der beste Fall dann vorliegt, wenn das pattern nicht in dem Muster vorkommt, denn dann kann immer um die komplette Länge des Musters verschoben werden.

Der schlechteste Fall würde eintreten wenn ein pattern sehr häufig in einem Muster vorkommt, denn dann würde immer nur um ein Zeichen verschoben werden, was zu demselben Laufzeitaufwand führen würde, wie der erste naive Lösungsansatz, nämlich ***O(nm).***

Der average-Case wurde anhand eines Beispiels getestet.

Boyer-Moore			
Wort	**Wortlänge**	**Zugriffe**	**Durchschnitt**
ein	3	323	
die	3	339	
der	3	647	
als	3	343	
sie	3	531	
von	3	967	**525**
Byte	4	499	
sich	4	275	
alle	4	328	
beim	4	812	
sind	4	699	
eine	4	317	**488.3333333…**
Datei	5	481	
nicht	5	1045	
durch	5	943	
liegt	5	1018	
immer	5	1308	
ASCII	5	506	**883,5**
lesbar	6	426	
können	6	618	
einem	6	764	
Bilder	6	818	
solche	6	909	
Format	6	819	**725,666666…**

Für den Boyer-Moore-Algorithmus wurde ein einziger deutscher Text verwendet und nach bestimmten Wörtern unterschiedlicher Länge gesucht.

2.4 Verbesserung

Wie an der Laufzeitanalyse des Boyer-Moore-Algorithmus erkannt werden kann, benötigt die Initialisierung der shift-Tabelle einen relativen hohen Aufwand. Deswegen gibt es auch Such-Algorithmen, die ohne diese Tabelle arbeiten. Grund dafür ist, wie bereits erwähnt, die Komplexität. Zudem soll sie laut einiger Experten nicht die gewünschten Erfolge erreichen.
Ein Algorithmus der ohne diese shift-Tabelle auskommt und eine allgemeine Vereinfachung des Boyer-Moore-Algorithmus darstellt, wurde von Horspool entwickelt.

3 Boyer-Moore-Horspool-Algorithmus

Im Dezember 1979 erschien in der Zeitschrift *Software-Practice and Experience* ein Artikel mit der Überschrift *Practical fast Searching in Strings*, geschrieben von Nigel Horspool.
Nigel Horspool ist Professor für Informatik an der Universität von Victoria, Kanada.

Abb. 13 Nigel Horspool

3.1 Grundidee

Während der Boyer-Moore-Algorithmus zwei Verfahren verwendet um die größtmögliche Verschiebedistanz zu finden, verwendet Horspool nur die „*Schlechtes-Zeichen*“ Strategie. Einziger Unterschied dazu ist, dass er nicht das Zeichen heranzieht das zu einem Mismatch geführt hat, sondern immer das ganz rechte Zeichen des zu durchsuchenden Textes benutzt.
Horspool begründet seine Vereinfachung damit, dass bei Benutzung eines „normalen“ Alphabets die shift-Tabelle nicht sonderlich von Vorteil ist, sondern nur dann, wenn nach ausgefallenen Alphabeten z.B. „xabcyyabc“ gesucht wird. In diesem Fall kann mit der shift-Tabelle ein worst-Case verhindert werden.

Zum Vergleich:

Boyer-Moore-Algorithmus:

a b c a b d a a c b a
b c a a b
b c a a b

Abb. 8 Boyer-Moore-Algorithmus

Boyer-Moore-Horspool-Algorithmus:

a b c a b d a a c b a
b c a a b

b c a a b

Abb. 9 Boyer-Moore-Horspool-Algorithmus

3.2 *Funktionsweise*

Anhand des vorherigen Beispiels soll die Funktionsweise verdeutlicht werden:

a b c a b d a a c b a
b c a a b
b c a a b

Abb. 10 Beispiel zu Horspool

Hier verursacht der Vergleich zwischen *c* und *a* einen Mismatch. Würde nach Boyer-Moore die Verschiebedistanz berechnet, würde hier nur um ein Zeichen nach rechts verschoben, bis das *c* in pattern unter dem *c* im zu durchsuchenden Text steht.
Horspool vernachlässigt aber dieses weitere Vorkommen und nimmt dafür das äußere rechte Zeichen aus dem zu durchsuchenden Text und vergleicht ob es noch einmal in pattern auftaucht. In diesem Beispiel ist das der Fall und es wird nicht um ein Zeichen verschoben, sondern um vier.

Um diesen Algorithmus (Boyer-Moore-Horspool-Algorithmus) zu implementieren, muss die *last*-Tabelle geringfügig abgeändert werden. Horspool ignoriert beim Erstellen der last-Tabelle das letzte Vorkommen des Zeichens.

```
private static int[] initLast(String pat)
{
    int j, a;
    int last[] = new int [256];
    int m = pat.length();

    for(a=0; a<256; a++)
        last[a]=-1;

    for(j=0; j<m-1; j++){
        a=pat.charAt(j);
        last[a]=j;
    }

    return last;
}
```

Abb. 14 initLast nach Horspool

Ein Ausschnitt der last-Tabelle ist folgender:

```
x: 0-->-1       x: 1-->-1       x: 2-->-1       x: 3-->-1
x: 8-->-1       x: 9-->-1       x: 10-->-1      x: 11-->-1
x: 16-->-1      x: 17-->-1      x: 18-->-1      x: 19-->-1
x: 24-->-1      x: 25-->-1      x: 26-->-1      x: 27-->-1
x: 32-->-1      x: 33-->-1      x: 34-->-1      x: 35-->-1
x: 40-->-1      x: 41-->-1      x: 42-->-1      x: 43-->-1
x: 48-->-1      x: 49-->-1      x: 50-->-1      x: 51-->-1
x: 56-->-1      x: 57-->-1      x: 58-->-1      x: 59-->-1
x: 64-->-1      x: 65-->-1      x: 66-->-1      x: 67-->-1
x: 72-->-1      x: 73-->-1      x: 74-->-1      x: 75-->-1
x: 80-->-1      x: 81-->-1      x: 82-->-1      x: 83-->-1
x: 88-->-1      x: 89-->-1      x: 90-->-1      x: 91-->-1
x: 96-->-1      x. 97-->3       x: 98-->6       x: 99-->5
x: 104-->-1     x: 105-->-1     x: 106-->-1     x: 107-->-1
```

Abb. 15 Ausschnitt Horspool last-Tabelle

Betrachtet man das Vorkommen der Zeichen, kann man sehr gut erkennen, dass das letzte Vorkommen ignoriert wird.

0	1	2	3	4	5	6	7
a	c	b	a	b	c	b	a

```
private static int hpSearch(String text, String pat)
{
    int [] last = initLast(pat);
    int n = text.length();
    int m = pat.length();
    int i=0, j;
    while (i<=n-m){
        j=m-1;
        while (j>=0 && pat.charAt(j)==text.charAt(i+j))
            j--;
        if (j<0)
            return i;
        i+=m-1;
        i-=last[text.charAt(i)];
    }

    return -1;
}
```

Abb. 16 Boyer-Moore-Horspool-Algorithmus Implementierung

3.3 Laufzeit

In der Vorbereitungsphase des Horspool-Algorithmus wird ebenfalls, wie beim Boyer-Moore-Algorithmus, eine last-Tabelle erstellt. Hier noch mal der Quellcode:

```
private static int[] initLast(String pat)
{
    int j, a;
    int last[] = new int [256];
    int m = pat.length();

    for(a=0; a<256; a++)
        last[a]=-1;

    for(j=0; j<m-1; j++){
        a=pat.charAt(j);
        last[a]=j;
    }

    return last;
}
```

Abb. 17 Quellcode initLast von Horspool

Auch hier wird ein Aufwand von

$$O(n) + O(m) = O(n+m)$$

benötigt.

Die Initialisierung der shift-Tabelle fällt hier allerdings weg.

Um den eigentlichen Suchalgorithmus zu analysieren, wird der Quellecode noch einmal an dieser Stelle gezeigt:

Auch im Horspool-Algorithmus tritt der günstigste Fall ein, wenn jedes Mal beim ersten Vergleich ein Textsymbol gefunden wird, das im Muster überhaupt nicht vorkommt. Dann benötigt der Algorithmus nur $O(n/m)$ Vergleiche.

Der worst-Case liegt hier vor, wenn das das gesuchte Muster häufig in dem Text auftritt. Ist dies der Fall liegt eine Laufzeitkomplexität von

$$O(nm)$$

auf.

Um den average-Case zu ermitteln, wurde, genau wie beim Boyer-Moore-Algorithmus, ein deutscher Text benutzt und nach verschiedenen Wörtern mit unterschiedlicher Länge gesucht.

Horspool			
Wort	**Wortlänge**	**Zugriffe**	**Durchschnitt**
ein	3	321	
die	3	335	
der	3	305	
als	3	336	
sie	3	537	

von	3	405	**373,16666…**
Byte	4	508	
sich	4	272	
alle	4	321	
beim	4	780	
sind	4	334	
eine	4	314	**421,5**
Datei	5	270	
nicht	5	404	
durch	5	416	
liegt	5	1100	
immer	5	1215	
ASCII	5	490	**649,166666…**
lesbar	6	421	
können	6	650	
einem	6	808	
Bilder	6	787	
solche	6	357	
Format	6	769	**632**

Interessant ist hier die Beobachtung, dass in den meisten Fällen der Algorithmus von Horspool mit geringfügig weniger Zugriffen auskommt als der Algorithmus von Boyer-Moore. Das liegt hauptsächlich, vor allem bei gesuchten Wörtern mit gängigem Alphabet und kurzer Länge, daran, dass die shift-Tabelle nicht erstellt wird.
Wie Horspool selber schreibt, ist diese nicht nötig wenn mit einem für die Sprache gängigen Alphabet gearbeitet wird. Beim Boyer-Moore-Algorithmus liefert die shift-Tabelle nur einen Vorteil wenn nach ausgefallenen Mustern gesucht wird. In diesem Beispiel auch gut zuerkennen an dem gesuchten Wort „Byte“. In diesem Fall benötigt der Boyer-Moore-Algorithmus weniger Zugriffe als Horspool.

Allgemein lässt sich sagen, je länger ein gesuchtes Pattern ist, desto größer werden logischerweise auch die Zugriffe.
Dies gilt sowohl für Boyer-Moore als auch für Horspool.

4 Praktisches Beispiel

Als praktisches Beispiel wurde ein Texteditor in Java erstellt der unter anderem eine Suchefunktion unterstützt.

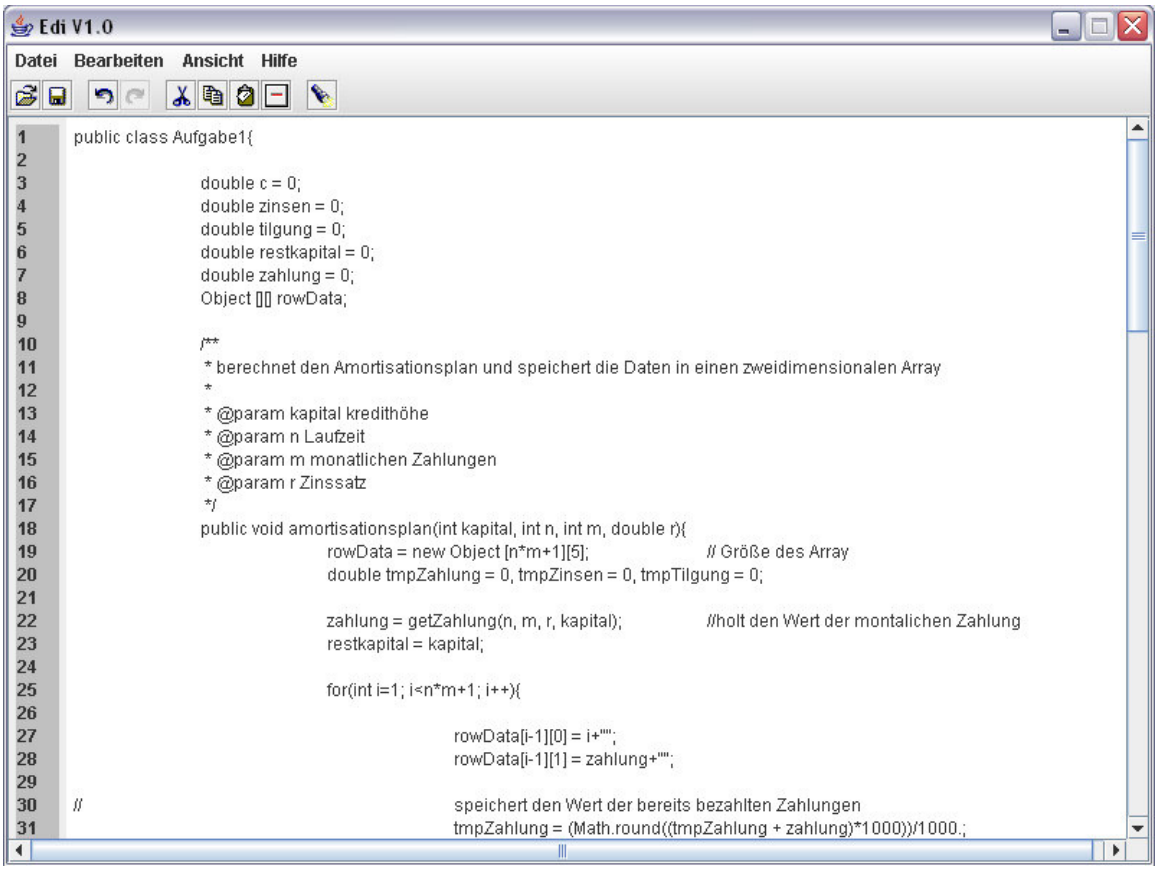

Abb. 18 Screenshot des Editors

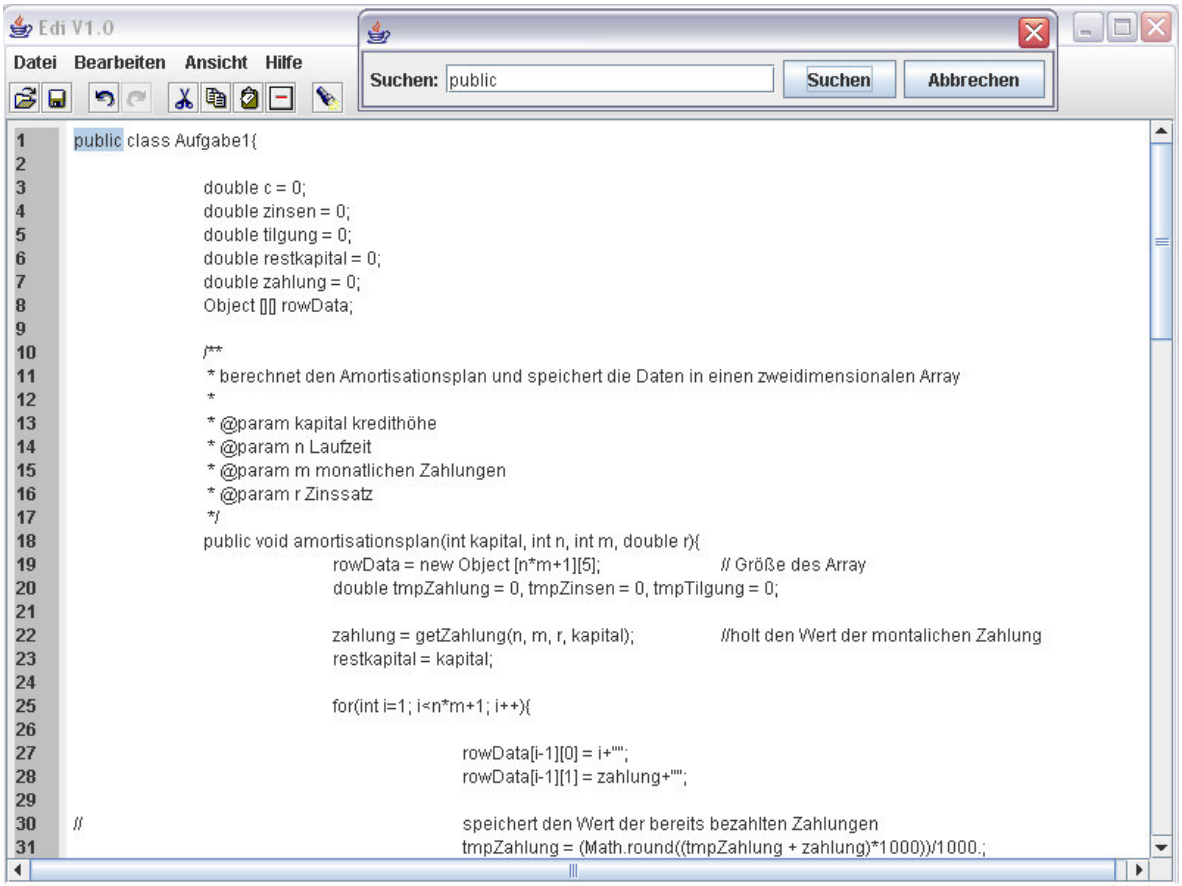

Abb. 19 Screenshot des Editors während einer Suche

5 Fazit

Durch die Laufzeitanalysen wird erkannt, dass der einzige Unterschied zwischen den beiden Algorithmen in der Vorbereitungsphase liegt.
Soll in Texten gesucht werden, sollte vorher überlegt werden, um welche Art von Texten es sich handelt und welche Alphabete oder Sprachen eingesetzt werden. Je nach Art der Texte und Vorkommen der zu suchenden Zeichen, sollte abgeschätzt werden, welcher Algorithmus benutzt wird.

Der Boyer-Moore-Algorithmus eignet sich vor allem dann gut, wenn in einer Sprache nach eher außergewöhnlichen Zeichen gesucht werden soll.
Wird ein Suchalgorithmus nur für beispielsweise einen Texteditor benötigt, eignet sich der Horspool-Algorithmus. Diese Vereinfachung kann bei solcher Art zu suchen durch den Wegfall der Initialisierung der shift-Tabelle zu einem Geschwindigkeitsvorteil führen. Bei großem Alphabet und hierzu eher kleineren zu suchenden Mustern, wird nur eine geringe Auswirkung auf die Laufzeit erreicht.

Abbildungsverzeichnis

Quellenverzeichnis

R. Nigel Horspool, “Practical Fast Searching in Strings”, Software-Practice and Experience, Vol. 10, 501-506 (1980)

R. S. Boyer and J.S. Moore, “A fast String searching algorithm”, CACM, 20 (10), 762-772 (1977)

www.wikipedia.org

http://www-igm.univ-mlv.fr/~lecroq/string/node18.html

http://wwwmayr.informatik.tu-muenchen.de/lehre/2002SS/cb/lecturenotes/chapter2.pdf

http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/bm.htm

http://www.thillm.th.schule.de/pages/schule/faecher/informatik/lpif/programm/otext.htm

http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/horse.htm

http://www.dcc.uchile.cl/~rbaeza/handbook/algs/7/713b.srch.c.html

Einfacher ASCII-Zeichensatz (7 bit)

Scan-code	ASCII hex	dez	Zeichen	Scan-code	ASCII hex	dez	Zeichen	Scan-code	ASCII hex	dez	Zeichen	Scan-code	ASCII hex	dez	Zeichen
	00	0	NUL		20	32	SP		40	64	@	0D	60	96	`
	01	1	SOH ^A	02	21	33	!	1E	41	65	A	1E	61	97	a
	02	2	STX ^B	03	22	34	"	30	42	66	B	30	62	98	b
	03	3	ETX ^C	29	23	35	#	2E	43	67	C	2E	63	99	c
	04	4	EOT ^D	05	24	36	$	20	44	68	D	20	64	100	d
	05	5	ENQ ^E	06	25	37	%	12	45	69	E	12	65	101	e
	06	6	ACK ^F	07	26	38	&	21	46	70	F	21	66	102	f
	07	7	BEL ^G	0D	27	39	'	22	47	71	G	22	67	103	g
0E	08	8	BS ^H	09	28	40	(	23	48	72	H	23	68	104	h
0F	09	9	TAB ^I	0A	29	41	)	17	49	73	I	17	69	105	i
	0A	10	LF ^J	1B	2A	42	*	24	4A	74	J	24	6A	106	j
	0B	11	VT ^K	1B	2B	43	+	25	4B	75	K	25	6B	107	k
	0C	12	FF ^L	33	2C	44	,	26	4C	76	L	26	6C	108	l
1C	0D	13	CR ^M	35	2D	45	-	32	4D	77	M	32	6D	109	m
	0E	14	SO ^N	34	2E	46	.	31	4E	78	N	31	6E	110	n
	0F	15	SI ^O	08	2F	47	/	18	4F	79	O	18	6F	111	o
	10	16	DLE ^P	0B	30	48	0	19	50	80	P	19	70	112	p
	11	17	DC1 ^Q	02	31	49	1	10	51	81	Q	10	71	113	q
	12	18	DC2 ^R	03	32	50	2	13	52	82	R	13	72	114	r
	13	19	DC3 ^S	04	33	51	3	1F	53	83	S	1F	73	115	s
	14	20	DC4 ^T	05	34	52	4	14	54	84	T	14	74	116	t
	15	21	NAK ^U	06	35	53	5	16	55	85	U	16	75	117	u
	16	22	SYN ^V	07	36	54	6	2F	56	86	V	2F	76	118	v
	17	23	ETB ^W	08	37	55	7	11	57	87	W	11	77	119	w
	18	24	CAN ^X	09	38	56	8	2D	58	88	X	2D	78	120	x
	19	25	EM ^Y	0A	39	57	9	2C	59	89	Y	2C	79	121	y
	1A	26	SUB ^Z	34	3A	58	:	15	5A	90	Z	15	7A	122	z
01	1B	27	Esc	33	3B	59	;		5B	91	[		7B	123	{
	1C	28	FS	2B	3C	60	<		5C	92	\		7C	124	|
	1D	29	GS	0B	3D	61	=		5D	93	]		7D	125	}
	1E	30	RS	2B	3E	62	>	29	5E	94	^		7E	126	~
	1F	31	US	0C	3F	63	?	35	5F	95	_	53	7F	127	DEL

Erweiteter ASCII-Zeichensatz (8 bit)

HTML-code	ANSI hex	dez	Zeichen	HTML-code	ANSI hex	dez	Zeichen	HTML-code	ANSI hex	dez	Zeichen	HTML-code	ANSI hex	dez	Zeichen
&#x80;	80	128	€		A0	160		&Agrave;	C0	192	À	&agrave;	E0	224	à
&#x81;	81	129	?	&iexcl;	A1	161	¡	&Aacute;	C1	193	Á	&aacute;	E1	225	á
&#x82;	82	130	‚	&cent;	A2	162	¢	&Acirc;	C2	194	Â	&acirc;	E2	226	â
&#x83;	83	131	ƒ	&pound;	A3	163	£	&Atilde;	C3	195	Ã	&atilde;	E3	227	ã
&#x84;	84	132	„	&curren;	A4	164	¤	&Auml;	C4	196	Ä	&auml;	E4	228	ä
&#x85;	85	133	…	&yen;	A5	165	¥	&Aring;	C5	197	Å	&aring;	E5	229	å
&#x86;	86	134	†	&brvbar;	A6	166	¦	&AElig;	C6	198	Æ	&aelig;	E6	230	æ
&#x87;	87	135	‡	&sect;	A7	167	§	&Ccedil;	C7	199	Ç	&ccedil;	E7	231	ç
&#x88;	88	136	ˆ	&uml;	A8	168	¨	&Egrave;	C8	200	È	&egrave;	E8	232	è
&#x89;	89	137	‰	&copy;	A9	169	©	&Eacute;	C9	201	É	&eacute;	E9	233	é
&#x8A;	8A	138	Š	&ordf;	AA	170	ª	&Ecirc;	CA	202	Ê	&ecirc;	EA	234	ê
&#x8B;	8B	139	‹	&laquo;	AB	171	«	&Euml;	CB	203	Ë	&euml;	EB	235	ë
&#x8C;	8C	140	Œ	&not;	AC	172	¬	&Igrave;	CC	204	Ì	&igrave;	EC	236	ì
&#x8D;	8D	141	?	&shy;	AD	173		&Iacute;	CD	205	Í	&iacute;	ED	237	í
&#x8E;	8E	142	Ž	&reg;	AE	174	®	&Icirc;	CE	206	Î	&icirc;	EE	238	î
&#x8F;	8F	143	?	&macr;	AF	175	¯	&Iuml;	CF	207	Ï	&iuml;	EF	239	ï
&#x90;	90	144	?	&deg;	B0	176	°	&ETH;	D0	208	Ð	&eth;	F0	240	ð
&#x91;	91	145	‘	&plusmn;	B1	177	±	&Ntilde;	D1	209	Ñ	&ntilde;	F1	241	ñ
&#x92;	92	146	’	&sup2;	B2	178	²	&Ograve;	D2	210	Ò	&ograve;	F2	242	ò
&#x93;	93	147	“	&sup3;	B3	179	³	&Oacute;	D3	211	Ó	&oacute;	F3	243	ó
&#x94;	94	148	”	&acute;	B4	180	´	&Ocirc;	D4	212	Ô	&ocirc;	F4	244	ô
&#x95;	95	149	•	&micro;	B5	181	µ	&Otilde;	D5	213	Õ	&otilde;	F5	245	õ
&#x96;	96	150	–	&para;	B6	182	¶	&Ouml;	D6	214	Ö	&ouml;	F6	246	ö
&#x97;	97	151	—	&middot;	B7	183	·	&times;	D7	215	×	&divide;	F7	247	÷
&#x98;	98	152	˜	&cedil;	B8	184	¸	&Oslash;	D8	216	Ø	&oslash;	F8	248	ø
&#x99;	99	153	™	&sup1;	B9	185	¹	&Ugrave;	D9	217	Ù	&ugrave;	F9	249	ù
&#x9A;	9A	154	š	&ordm;	BA	186	º	&Uacute;	DA	218	Ú	&uacute;	FA	250	ú
&#x9B;	9B	155	›	&raquo;	BB	187	»	&Ucirc;	DB	219	Û	&ucirc;	FB	251	û
&#x9C;	9C	156	œ	&frac14;	BC	188	¼	&Uuml;	DC	220	Ü	&uuml;	FC	252	ü
&#x9D;	9D	157	?	&frac12;	BD	189	½	&Yacute;	DD	221	Ý	&yacute;	FD	253	ý
&#x9E;	9E	158	ž	&frac34;	BE	190	¾	&THORN;	DE	222	Þ	&thorn;	FE	254	þ
&#x9F;	9F	159	Ÿ	&iquest;	BF	191	¿	&szlig;	DF	223	ß	&yuml;	FF	255	ÿ

Abb. 20 ASCII-Zeichensatz