

Proseminar

**Suche in Zeichenketten
(Pattern Matching)**

Michael Denk

05.07.2001

Oft stellt sich folgendes Problem in der Informatik: Gegeben ist ein Text und man will darin ein bestimmtes Wort oder ein Muster (Pattern) finden. Ich werde im folgenden das Problem näher charakterisieren und Lösungen dafür in Form von vier Algorithmen angeben. Zuletzt werde ich noch Verbesserungen der hier vorgestellten Verfahren beschreiben sowie einen Ausblick auf weitere Lösungen des sog. „**Pattern Matching Problems**“ geben.

Anwendungen des Pattern Matching

Das Problem, einen Begriff in einer Folge von Zeichen zu finden, hat seine Anwendung vor allem im Bereich der Textverarbeitung. Dort stellt sich häufig die Aufgabe des Suchens bzw. Ersetzens von Textabschnitten. Hat man beispielsweise einen Text über einen Herrn „Weisser“ verfaßt und stellt am Ende fest, daß sich besagter Herr eigentlich „Weißer“ schreibt, so kann man mit der Funktion „Suchen/Ersetzen“ in jedem (guten) Textverarbeitungsprogramm sehr komfortabel sämtliche Vorkommen von „Weisser“ durch „Weißer“ ersetzen lassen.

Eine wichtige Anwendung des Pattern Matching ist auch die Suche von Mustern in DNA-Sequenzen, z.B. bei Vaterschaftstests oder neuerdings auch immer häufiger bei DNA-Tests zur Ermittlung der Täter von Sexualdelikten.

Auch in einfachen Archivierungs- und Verwaltungsprogrammen wird das Pattern Matching eingesetzt. Ein Beispiel wäre ein Archiv einer kleinen Bücherei, in dem man dann per Volltextsuche nach dem Titel, Autor etc. ein bestimmtes Buch finden kann.

Nun stellt sich die Frage, wie man das oben beschriebene Problem (effizient) lösen kann.

Das Pattern Matching Problem

Man kann dies allgemein so formulieren: Gegeben sei ein Text T der Länge n und ein Pattern P der Länge m . Beide liegen als Strings vor, die man als char-Arrays interpretiert. Wenn nun das Zeichen auf Index i in P mit dem Zeichen auf Index j in T übereinstimmt, so schreibt man: $P[i] = T[j]$. Stimmen alle Zeichen von P mit einer zusammenhängenden Folge von Zeichen in T , die bei Index s beginnt, überein, so sagt man, **P kommt in T mit Shift s vor**, also $P[0] = T[s]$, $P[1] = T[s+1]$, ..., $P[m-1] = T[s+m-1]$, d.h. $P[0..m-1] = T[s..s+m-1]$ mit $0 \leq s \leq n-m$. s heißt dann ein **gültiger Shift**.

Gesucht sind nun alle gültigen Shifts s .

Bemerkung

Die im folgenden vorgestellten Algorithmen beschränken sich alle (mit Ausnahme des Brute Force Verfahrens) darauf, nur das erste Vorkommen des Patterns im Text zu suchen. Dies ist von Vorteil, wenn man nur überprüfen will, ob das Pattern überhaupt vorkommt. Hat man es gefunden, hat der Algorithmus eine Stop-Bedingung und man kann dadurch in den meisten Fällen die Laufzeit verkürzen. Aber man kann durch das gleiche Vorgehen wie beim hier vorgestellten Brute Force Algorithmus, nämlich die eigentliche Suche in eine Subroutine zu packen und mit den möglichen Shifts als Parameter aufzurufen, dieses „Manko“ leicht beheben. Den übergebenen Shift macht man dabei abhängig vom letzten gefundenen Shift und ob man zuläßt, daß Teile des Patterns in sich selbst wieder gefunden werden, etwa bei der Suche von „aa“ im Text „aaaa“. Kommt „aa“ jetzt zwei Mal oder drei Mal in „aaaa“ vor? Im folgenden wird immer von letzterem Fall ausgegangen, d.h. es werden auch Vorkommen des Patterns in sich selbst gefunden.

Letzte Anmerkung: Nicht immer wird der triviale Fall berücksichtigt, daß das Pattern länger als der Text ist. Es wird also o.B.d.A. vorausgesetzt, daß Länge des Patterns kleiner gleich Länge des Textes.

Lösungen des Pattern Matching Problems

1.) Brute Force Algorithmus

Beim einfachsten aller Algorithmen geht man intuitiv an das Problem heran: Man stellt sich vor, man legt eine Schablone, eine Art „Suchfenster“, beginnend am Textanfang über den Text und vergleicht dann Zeichen für Zeichen Pattern mit Text. Stimmen alle Zeichen überein, so hat man einen **Match** (Übereinstimmung) gefunden. Andernfalls schiebt man das Fenster um eine Position nach rechts und vergleicht wiederum. Dies wiederholt man so lange, wie das Fenster noch vollständig über dem Text liegt, also Verschiebeposition plus Patternlänge kleiner gleich Textlänge. Die Implementierung ist „straight forward“, könnte einfacher nicht sein. Der zugehörige Algorithmus findet sich weiter hinten als **Algorithmus_1**.

Die **Komplexität** von Algorithmus_1 läßt sich ebenfalls sehr leicht bestimmen: Man hat zwei geschachtelte for-Schleifen (nested loop), die äußere läuft von $[0:n-m]$, die innere von $[0:m-1]$. Somit ergibt sich eine Gesamtkomplexität von $O((n-m+1)*m) = O(n*m)$.

Der Brute Force Algorithmus ist zwar effektiv (und auch korrekt, da er mit dem Durchprobieren aller möglichen Verschiebepositionen eine erschöpfende Suche durchführt), jedoch nicht sehr effizient.

Nun fragt man sich: Geht es denn nicht besser? Die Antwort ist ja, es geht besser. Die nächsten drei Algorithmen erreichen wesentlich bessere Laufzeiten, weil sie nicht mehr jedes Zeichen vergleichen müssen, sondern durch geschickte Verschiebestrategien das Vergleichsfenster um mehr als ein Zeichen (teilweise sogar um die gesamte Patternlänge) verschieben können. Dafür muß man aber ein endliches Alphabet für Text und Pattern voraussetzen, wie man beim PM mit endlichem Automaten leicht erkennen kann.

2.) Pattern Matching mittels endlichem Automaten

In der Informatik sehr wichtig sind sog. „**endliche Automaten**“. Ein solcher Automat M besteht dabei aus einer Zustandsmenge Q , einem Startzustand q_0 , einer Menge von akzeptierenden Zuständen A (A ist Teilmenge von Q), einem Eingabealphabet Σ und einer Übergangsbzw. **Transitionsfunktion** δ , die angibt, in welchen Zustand z_1 der Automat wechselt, wenn er sich gerade im Zustand z_0 befindet und als nächstes Zeichen das Zeichen ξ liest, also $\delta(z_0, \xi) = z_1$. Ein solcher Automat „erkennt“ dann beliebige Zeichenketten, indem er angibt, ob er die Zeichenfolge **akzeptiert** oder nicht. Der Automat ist am Anfang im Zustand 0 (default), liest dann das erste Zeichen der Zeichenkette $T \in \Sigma^*$ und geht in den Zustand $z = \delta(0, T[0])$ über. Im Zustand z liest er $T[1]$ und geht in Zustand $z' = \delta(z, T[1])$ über. Wenn der Automat das letzte Zeichen von T gelesen hat und sich dann in einem akzeptierenden Zustand befindet, dann hat er das Wort erkannt, ansonsten hat er das Wort nicht erkannt, d.h. das Wort war ungültig.

Was hat dieses Konzept jetzt mit dem Pattern Matching Problem zu tun? Sei wieder Pattern $P[0...m-1]$ und $T[0...n-1]$ gegeben. Man kann sich sehr leicht einen Automaten konstruieren, der aus m Zuständen besteht, wobei der Zustand m der **einzige akzeptierende Zustand** ist. Der Übergang von Zustand i nach $i+1$ geschieht durch das Lesen des Zeichens $T[i]$, $0 \leq i \leq m-1$. Dieser Automat akzeptiert dann das Pattern P .

Zum Vergleichen von Text und Pattern liest der Automat den Text zeichenweise ein, und wechselt nach jedem Zeichen mittels δ in den entsprechenden neuen Zustand über. Bei einem „**Mismatch**“ (Nicht-Übereinstimmung) von Text und erwartetem Zeichen im Automaten, wechselt der Automat wieder in den Zustand 0, d.h. er verwendet schon einmal gewonnene Informationen über das Pattern nicht wieder. Man versucht nun, diesen „**Skelettautomaten**“ durch Hinzufügen von Transitionen so zu ergänzen, daß man bei einem Mismatch nicht wie-

der im Zustand 0 landet, sondern seine Suche in einem möglichst hohen Zustand fortsetzen kann.

Ob dies möglich ist oder nicht, hängt alleine vom Pattern ab. Taucht etwa ein Teil P' des Patterns nochmal als Präfix des Patterns auf, so kann man bei einem Mismatch nach P' die schon gemachten Vergleiche dazu benutzen, um erst in einem Zustand einzusteigen, in dem das Präfix bereits gefunden wurde, und ab dort weiterzuvergleichen.

Formal wird der Automat mit folgender Funktion ergänzt: $\delta(i, \xi) = \max\{k \leq i \mid P[0..k-1] \text{ ist Suffix von } P[0..i-1]^{\xi}\}$, falls das Maximum ex. Ansonsten wird der Wert auf 0 gesetzt. Der Automat geht also in den neuen Zustand k über, wobei k die **Länge des längsten Präfix von P ist, das zugleich Suffix der bisher übereinstimmenden Zeichen des Pattern konkateniert mit dem gelesenen Zeichen ξ** ist.

Vervollständigt man den Automaten auf diese Weise, so kann man mit **Algorithmus_2** einen Pattern Matching Algorithmus in $O(n + m \cdot |\Sigma|)$ realisieren. Dabei wurde allerdings schon ein recht ausgeklügelter Algorithmus zur Konstruktion des Automaten eingesetzt, der in $O(m \cdot |\Sigma|)$ läuft (naiver Ansatz läuft in $O(m^3 \cdot |\Sigma|)$). Aus der Komplexität von **Algorithmus_2** kann man leicht erkennen, daß dieses Verfahren eine große Schwachstelle besitzt: Es hängt sehr stark von der Größe des Alphabets ab. Je größer das Alphabet ist, umso länger braucht man zum Erstellen der Übergangstabelle für den Automaten. Dieses Problem löst der **KMP-Algorithmus** von **Knuth, Morris und Pratt**, der in linearer Laufzeit arbeitet.

3.) KMP-Algorithmus

Die Grundidee hierbei ist, die Übergangstabelle des Automaten so zu vereinfachen, daß man nicht mehr alle Zeichen des Alphabets berücksichtigen muß, aber dennoch beim Vergleich Text-Pattern nicht jedes Zeichen des Pattern „in die Hand nehmen muß“. Durch ein **Pre-Processing** des Patterns erhält man eine Verschiebetabelle, die angibt, **ab welchem Zeichen im Pattern man bei einem Mismatch weitervergleichen kann** (die sog. **failure-Funktion**). Es gilt: $\text{failure}(j)$ ist die **Länge des längsten Präfix von P , das zugleich Suffix von $P[1..j]$ ist**. $\text{failure}(0)$ ist dabei per default gleich 0. Vergleicht man nun Text und Pattern und stellt beim Vergleich von $P[i]$ mit $T[i]$ einen Mismatch fest, so wird ab der Stelle $\text{failure}(i-1)$ weitervergleichen, weil ja vorher schon i Zeichen übereinstimmten, falls $i > 0$. Ansonsten hat man den Fehler beim Index 0 und kann gleich um eine Position nach rechts schieben. Der zugehörige Algorithmus findet sich hinten als **Algorithmus_3.1**.

Die Laufzeit des Algorithmus ist nicht gleich offensichtlich. Setzt man $k := i - j$, so kennzeichnet k gerade die Verschiebung von Pattern und Text. Dabei ist i durch n begrenzt, daher kann auch k maximal n groß werden.

Im Fall, daß $P[i]$ gleich $T[i]$, erhöhen sich i und j um jeweils 1, k bleibt gleich.

Im zweiten Fall $P[i]$ ungleich $T[i]$ und $j > 0$, d.h. man hatte vorher schon mind. eine Übereinstimmung von Text und Pattern, bleibt i gleich, j wird mind. eins kleiner, da man aus der failure-Funktion leicht ersehen kann, daß $\text{failure}(j-1) < j$. Das heißt k wird um mindestens eins größer.

Im letzten Fall $P[i]$ ungleich $T[i]$ und j gleich 0, d.h. Fehler am ersten Index des Pattern, erhöht sich i um eins, d.h. k erhöht sich um eins.

Daraus folgt, daß k durch $2n$ begrenzt ist: $k < 2n$. Damit läuft die Schleife max. $2n$ mal.

Analog folgt, daß die Schleife bei der failure-Funktion im **Algorithmus_3.2**, der wie **Algorithmus_3.1** aufgebaut ist mit einem Unterschied, nämlich daß statt Text mit Pattern Pattern mit Pattern verglichen wird, maximal $2m$ mal läuft: $k < 2m$.

Hieraus folgt, daß der KMP-Algorithmus eine **Komplexität** von **$O(n+m)$** hat.

4.) Boyer-Moore-Algorithmus

Dieser Algorithmus beruht auf der cleveren Taktik, bei den Vergleichen nicht von links nach rechts sondern umgekehrt von rechts nach links vorzugehen (**looking-glass-Technik**). Der große Vorteil dabei ist, daß man so zusammen mit der **bad-character-Strategie** sehr große Sprünge beim Vergleichen machen kann. Man kann im günstigsten Fall nach einem Vergleich das Pattern relativ zum Text um die gesamte Patternlänge verschieben.

Die bad-character-Strategie besagt Folgendes: Tritt beim Vergleichen von Text und Pattern (von rechts nach links) ein Mismatch beim Zeichen $T[i]$ auf, so sucht man das erste Vorkommen des Zeichens $T[i]$ im Pattern links von der Mismatch-Stelle (Suche von rechts beginnend). Anschaulich gesprochen kann man nun das Pattern so weit nach rechts verschieben, daß das rechteste Auftreten von $T[i]$ im Pattern (links von Mismatch-Stelle) unter $T[i]$ zu liegen kommt bzw. das Zeichen auf Index 0 in P hinter $T[i]$, wenn $T[i]$ nicht in P vorkommt. Der günstigste Fall ergibt sich also, wenn beim ersten Vergleich ein Fehler auftritt und das betreffende Zeichen gar nicht im Pattern vorkommt. Dann kann um die volle Patternlänge verschoben werden. Der JAVA-Code findet sich hinten als **Algorithmus_4**.

Die **Laufzeitanalyse** ist bei diesem Algorithmus wie beim KMP etwas schwieriger. Geht man von oben beschriebenem Idealfall aus, so kann man nach $O(1)$ Vergleichen das Pattern um $m - o(m)$ Positionen verschieben. Der Durchschnittsfall ist wohl nicht weit vom Idealfall entfernt, aber schwieriger zu analysieren. Daher ergibt sich eine **Komplexität** von $O(1) * n / (m - o(m)) = O(n/m)$. Die worst-case Komplexität ist aber $O(n * m)$ wie beim Brute Force Algorithmus, wie man sich leicht beim Suchen des Patterns ba^{m-1} im Text a^n überlegen kann.

Ausblick

Verbesserung des Boyer-Moore-Algorithmus

Die schlechte worst-case Komplexität kann man mit einer zweiten Strategie, der sog. **good-suffix-Strategie** etwas abfangen. Dabei wendet man im Prinzip den KMP-Algorithmus auf das gespiegelte Pattern an, weil ja von rechts nach links verglichen wird. Man sucht nach einem Mismatch, ob die schon verglichenen Zeichen noch mal im Pattern auftreten und wählt wiederum das rechteste Auftreten. Bis dahin kann man das Pattern dann gegen den Text verschieben, weil vorher sicher kein Match auftreten kann.

Der BM-Algorithmus wählt dabei diejenige Strategie (bad-character oder good-suffix), die die größere Verschiebung des Patterns liefert.

Weitere Algorithmen

Für das Pattern Matching Problem gibt es wie für Sortierprobleme unzählige Algorithmen. Stellvertretend sei hier noch kurz der **Rabin-Karp-Algorithmus** erwähnt. Dieser beruht darauf, daß man von Text und Pattern, die o.B.d.A. aus einem Alphabet $\{0,1,\dots,9\}$ sind, eine Art „**Fingerabdruck**“ errechnet, der das Pattern bzw. einen Textabschnitt der Länge des Patterns schon ziemlich gut identifiziert. Setzt man nun voraus, daß man den Abdruck des Patterns in $O(m)$ und die Abdrücke der möglichen Textabschnitte in $O(n+m)$ berechnen kann und wählt man seine Funktion für den Abdruck so, daß das Ergebnis in den Speicher paßt und mit $O(1)$ verarbeitet werden kann, so kann man in $O(n+m)$ einen String Matcher realisieren, der einfach die Fingerabdrücke von Pattern und den möglichen Substrings im Text vergleicht. Besteht keine Gleichheit, kann kein Match vorliegen. Bei Gleichheit muß man Pattern und den betreffenden Textabschnitt jedoch auch wieder zeichenweise vergleichen, weil die Fingerabdruckfunktion (ähnlich wie beim Hashing) nicht eindeutig ist, also für zwei unterschiedliche Strings den gleichen Wert liefern kann. Solche zufälligen Übereinstimmungen kommen aber glücklicherweise nur selten vor.

Literaturverzeichnis

Bücher

- **Cormen, Leiserson, Rivest** – Introduction to Algorithms
- **Goodrich, Tamassia** – Data Structures and Algorithms in Java
- **Schöning** – Algorithmen - kurz gefaßt

Internet

- http://ls4-www.cs.uni-dortmund.de/~Misch/applets/StringMatching/english_version_smaa.html
- <http://sunburn.informatik.uni-tuebingen.de/~buehler/BM/BM.html>
- <http://www-igm.univ-mlv.fr/~lecroq/string/>

Algorithmus_1

```

/**
 * Findet alle Vorkommen von pattern in text und gibt die entsprechenden
 * Shifts aus.
 * @param text Suchtext
 * @param pattern Pattern
 */
public static void naivePM(String text, String pattern) {

    int n = text.length();
    int m = pattern.length();
    for (int s = 0; s <= (n-m); s++) {
        if (test(text, pattern, s)) {
            System.out.println("Match bei Shift" + s);
        }
    }
}

/**
 * Prüft, ob pattern in text mit Shift s vorkommt.
 * @param text Suchtext
 * @param pattern Pattern
 * @param s zu prüfender Shift
 * @return true wenn pattern in text mit Shift s vorkommt; false sonst.
 */
public static boolean test(String text, String pattern, int s) {

    for (int j = 0; j < pattern.length(); j++) {
        if (pattern.charAt(j) != text.charAt(s + j)) {
            return false;
        }
    }
    return true;
}

```

Algorithmus_2

```

/**
 * Gibt die erste Stelle zurück, an der pattern in text vorkommt.
 *
 * @param text Suchtext
 * @param pattern Pattern
 * @return 1. Stelle, an der pattern in text vorkommt; -1 wenn kein Match.
 */
public static int finiteAutomatonMatch(String text, String pattern) {

    // int[][] createAutomaton(String s) nicht implementiert (generiert
    // Uebergangstabelle)
    int[][] trans = createAutomaton(pattern);

    int currentState = 0;
    int n = text.length();
    int m = pattern.length();

    for (int i = 0; i < n; i++) {
        currentState = trans[currentState][text.charAt(i)];
        if (currentState == m) {
            return (i-(m-1));
        }
    }
    return -1;
}

```

Algorithmus_3.1

```

/**
 * Gibt die erste Stelle zurück, an der pattern in text vorkommt.
 *
 * @param text Suchtext
 * @param pattern Pattern
 * @return 1. Stelle, an der pattern in text vorkommt; -1 wenn kein Match.
 */
public static int KMPmatch(String text, String pattern) {

    int n = text.length();
    int m = pattern.length();
    int [] fail = computeFailFunction(pattern);

    int i = 0; int j = 0;
    while (i < n) {
        if (pattern.charAt(j) == text.charAt(i)) {
            if (j == (m-1)) {
                return (i-(m-1));
            }
            i++;
            j++;
        } else if (j > 0) {
            j = fail[j-1];
        } else {
            i++;
        }
    }

    return -1;
}

```

Algorithmus_3.2

```

/**
 * Berechnet die Werte der failure-Funktion.
 *
 * @param pattern Pattern
 * @return int-Array mit Werten der failure-Funktion
 */
public static int[] computeFailFunction(String pattern) {

    int [] fail = new int[pattern.length()];
    fail[0] = 0;
    int i = 1;
    int j = 0;
    int m = pattern.length();

    while (i < m) {
        if (pattern.charAt(j) == pattern.charAt(i)) {
            fail[i] = j+1;
            i++;
            j++;
        } else if (j > 0) {
            j = fail[j-1];
        } else {
            fail[i] = 0;
            i++;
        }
    }

    return fail;
}

```

Algorithmus_4

```
/**
 * Gibt die erste Stelle zurück, an der pattern in text vorkommt.
 *
 * @param text Suchtext
 * @param pattern Pattern
 * @return 1. Stelle, an der pattern in text vorkommt; -1 wenn kein Match.
 */
public static int BMmatch(String text, String pattern) {

    int n = text.length();
    int m = pattern.length();
    int [] last = buildLastFunction(pattern);
    int i = (m-1);

    if (i > (n-1)) {
        return -1;
    }
    int j = (m-1);

    do {
        if (pattern.charAt(j) == text.charAt(i)) {
            if (j == 0) {
                return i;
            } else {
                i--;
                j--;
            }
        } else {
            i = i + m - Math.min(j, 1 + last[text.charAt(i)]);
            j = m-1;
        }
    } while (i <= (n-1));

    return -1;
}

/**
 * Berechnet die last-Funktion für die bad-character Strategie.
 *
 * @param pattern Pattern
 * @return int-Array über alle ASCII Zeichen mit Pos. im Pattern
 */
public static int[] buildLastFunction(String pattern) {

    int [] last = new int[128];
    for (int i = 0; i < 128; i++) {
        last[i] = -1;
    }
    for (int i = 0; i < pattern.length(); i++) {
        last[pattern.charAt(i)] = i;
    }
    return last;
}
```