Der Algorithmus von Boyer & Moore

Thomas Pajor

8. August 2006

Der Algorithmus von Boyer & Moore (1977) ist einer der schnellsten Algorithmen um das Pattern–Matching Problem zu lösen.

Im Internet und in Lehrbüchern finden sich relativ viele verschiedene Erklärungen des Algorithmus. Oftmals unterscheiden sie sich in Details bzw. der Betrachtungsweise, so dass es relativ verwirrend sein kann einen Zugang zu finden – zumal der Algorithmus, insbesondere die Vorverarbeitung, nicht ganz einfach ist.

Dies soll der Versuch sein den Algorithmus (besonders die Berechnung der Sprungtabelle[1]) möglichst so zu erklären, dass man die Funktionsweise versteht. Deshalb werde ich die einzelnen Berechnungsverfahren der Sprungtabelle so exakt wie möglich, und trotzdem anschaulich herleiten. Die „Variante“ des Algorithmus sowie die Bezeichnungen, die ich in diesem Papier verwende, lehnen sich an die Vorlesung „Informatik IV“ (Sommersemester 2006) an der Universität Karlsruhe [2] an.

1 Das Pattern–Matching Problem

Der Boyer & Moore Algorithmus löst das Pattern–Matching Problem: Gegeben sei ein Alphabet Σ sowie ein Text $T \in \Sigma^+$. Wir definieren außerdem $n := |T|$ als die Länge des Textes. Des Weiteren haben wir ein Muster (engl. *Pattern*) $M \in \Sigma^+$ der Länge m. Das Problem ist nun alle Vorkommen von M in T zu finden. Die Ausgabe könnte also eine Menge I von Indizes sein, so dass für jedes $i \in I$ gilt $T[i, \ldots, i+m] = M$.

Ein paar Bemerkungen zur Notation. Um in einem Wort — zum Beispiel T — ein Teilwort von der Stelle i bis j zu beschreiben, schreiben wir kurz $T[i, \ldots, j]$. Außerdem indizieren wir Wörter beginnend bei 1 (nicht, wie es in Programmiersprachen üblich ist, beginnend bei 0). Damit ist also $T = T[1, \ldots, |T|]$ und für $T := Hallo$ ist $T[2, \ldots, 3] = al$. Für ein einzelnes Zeichen in T, das sich an Position i befindet, schreiben wir kurz $T[i]$.

[1] später mehr dazu

2 Ein naiver Ansatz

Möchte man das Problem lösen, so ist die erste Idee wohlmöglich eine Variante der Folgenden

Algorithmus 1 : NaivFind

Eingabe : Text T und Muster M
Ausgabe : Alle Vorkommen von M in T

```
1 für i ← 1 ... n − m tue
2 |   wenn ∀j ∈ {1, ..., m} : M[j] = T[i + j − 1] dann
3 |   |   Muster gefunden an Stelle i
```

Wir setzen das Muster M beginnend bei Index 1 an den Text T und vergleichen zeichenweise ob $T[1, \ldots, m] = M$. Gibt es einen Index i mit $T[i] \neq M[i]$, so schieben wir das Muster um eine Stelle nach rechts und vergleichen nun ob $T[2, \ldots, m+1] = M$, und so weiter. Offensichtlich finden wir durch diese Vorgehensweise jedes Vorkommen von M in T. Es fällt jedoch schnell auf, dass die Laufzeit dieses Algorithmus in $\mathcal{O}(mn)$ ist, was noch verbesserungswürdig ist.

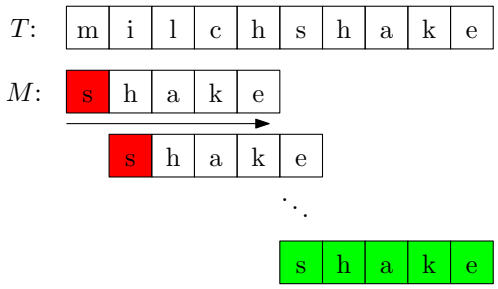

Abbildung 1: Das naive Vorgehen. Das Muser M wird an jede Stelle im Text angesetzt. Bei einem Mismatch wird M um eine Position nach rechts verschoben.

3 Der Algorithmus von Boyer & Moore

Kommen wir nun zum Algorithmus von Boyer & Moore. Die grundlegende Idee das Muster M als sogenanntes *Suchfenster* an den Text zu legen und zeichenweise zu vergleichen ist im Prinzip garnicht so schlecht. Das Schlechte an dem „naiven“ Algorithmus ist bloß, dass wir das Suchmuster jedes Mal um nur eine einzige Stelle nach rechts verschieben. Der Boyer & Moore Algorithmus versucht nun, unter Ausnutzung der Struktur des Musters, das Muster um möglichst viele Stellen weiterzuschieben. Diese Informatio-

nen werden im Voraus berechnet und in sogenannten *Sprungtabellen*[2] gespeichert.

Zunächst gibt es jedoch noch einen weiteren Unterschied zum vorherigen Ansatz. War es bis jetzt nicht spezifiziert in welcher Reihenfolge wir beim Anlegen von M den zeichenweisen Vergleich vornehmen, so fordern wir ab jetzt dass wir beginnend vom rechten Ende des Musters uns nach links durcharbeiten, wie in Abbildung 2 illustriert.

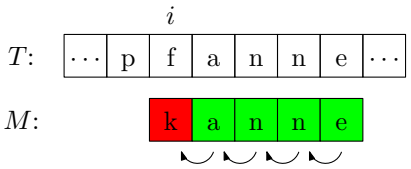

Abbildung 2: Vergleichen von rechts nach links, bis ein Mismatch stattfindet.

Liegt das Muster also an Stelle i im Text an, so vergleichen wir die Positionen $i + m - 1, \ldots, i$ im Text mit den Stellen $j = m, \ldots, 1$ im Muster.

Kommen wir nun zur Konstruktion unserer Sprungtabellen.

3.1 Die „Schlechtes–Zeichen“ Strategie

Die Schlechtes–Zeichen Strategie definiert eine Abbildung $\delta_1 : \Sigma \to \mathbb{N}$, die wir in einer Tabelle speichern können. Findet im Text T ein Mismatch mit dem Zeichen σ (Das Zeichen von T!) statt, so soll uns der Wert von $\delta_1(\sigma)$ angeben, um wieviel Stellen man in T weiterspringen darf. Diese Stelle definiert die Position wo wir das Muster rechtsbündig anlegen, also die Stelle ab der wir das zeichenweise Vergleichen beginnen werden.

Betrachte zunächst das Beispiel aus Abbildung 3. Der erste Mismatch tritt beim Vergleich des 'b' mit dem 'e' auf. Weiterhin sehen wir, dass das 'b' in unserem Muster M überhaupt nicht vorkommt. Es bringt also nichts, wenn wir M so an T anlegen, dass sich M mit dem 'b' überlappt, da es an dieser Stelle auf jeden Fall wieder zu einem Mismatch kommen würde.

Wollen wir das mal etwas formaler festhalten. Findet ein Mismatch an der Position j an einem Zeichen $\sigma \notin M$ statt, so dürfen wir das Muster maximal verschieben, also um genau j Stellen, so dass der Anfang des Musters an der Position $i + j$ im Text steht. Die Sprungweite, die angibt, wo wir wieder das Vergleichen mit dem Muster beginnen ist demnach $j + (m - j) = m$. Wir springen also im Text um genau m Stellen nach rechts! Wir halten fest

$$\delta_1(\sigma) = m = |M| \qquad \sigma \notin M$$

[2] oder auch *Lookuptabellen*

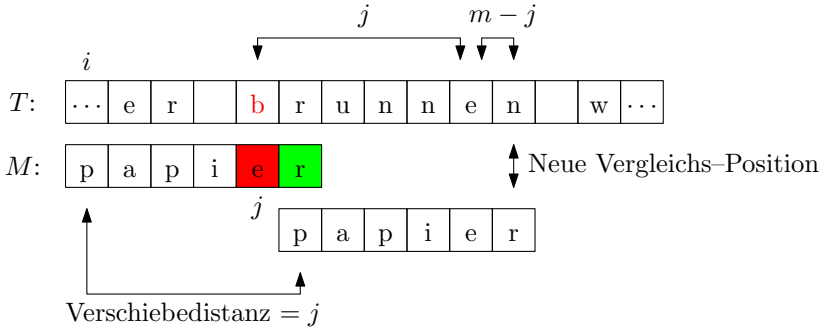

Abbildung 3: Schlechtes–Zeichen Strategie für ein Zeichen das *nicht* im Muster vorkommt. Der nächste Vergleich ist m Stellen weiter rechts im Text.

Was ist aber wenn ein Mismatch an einem Zeichen σ in T stattfindet, dass in M vorkommt? Betrachte dazu Abbildung 4. Offensichtlich können wir M maximal soweit nach rechts schieben, bis das p im Text mit dem *rechtesten* Vorkommen von p in M übereinstimmt, da wir sonst – würden wir weiter verschieben – eventuell einen Treffer übergehen könnten.

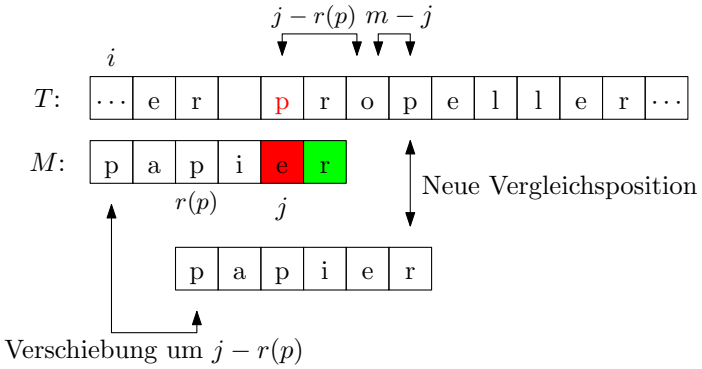

Abbildung 4: Schlechtes–Zeichen Strategie für ein Zeichen das im Muster vorkommt. $r(\sigma)$ bezeichnet den Index in M an der das Zeichen σ am weitesten rechts in M vorkommt.

Bezeichne also $r(\sigma)$ für ein $\sigma \in M$ das rechteste Vorkommen von σ in M. Befinden wir uns an Position j im Muster, so können wir das Muster um $j - r(\sigma)$ nach rechts verschieben, damit σ korrekt an dem Text anliegt. Wie auch im letzten Fall wollen wir für $\delta_1(\sigma)$ die Sprungweite berechnen. Da wir uns an Position j befanden, müssen wir $m-j$ zu der Verschiebedistanz dazuaddieren was zu $j-r(\sigma)+(m-j) = m-r(\sigma)$ führt. Wir halten somit fest

$$\delta_1(\sigma) = m - r(\sigma) \qquad \sigma \in M$$

In Worten ist das äquivalent zu „$\delta_1(\sigma)$ ist der Abstand von rechts des rechtesten Vor-

kommens von σ in M“. So steht es auch in den Vorlesungsfolien.

Fazit Zusammenfassend gilt für die Berechnung von δ_1:

$$\delta_1(\sigma) := \begin{cases} m - r(\sigma) & \text{falls } \sigma \in M \\ m & \text{sonst} \end{cases}$$

Um das Ganze mit unserem Beispiel „papier“ abzuschließen ist hier die komplette Tabelle für δ_1:

σ:	p	a	p	i	e	r	sonstige
δ_1:	3	4	3	2	1	0	6

Ein möglicher Algorithmus zur Berechnung von δ_1 in Pseudocode könnte wie folgt aussehen:

Algorithmus 2 : SchlechtesZeichen

Eingabe : Muster M
Ausgabe : Sprungfunktion $\delta_1 : \Sigma \to \mathbb{N}$

```
1 für alle σ ∈ Σ tue                    /* Alle Werte mit |M| initialisieren */
2   δ1(σ) ← |M|
3 für j = |M| ... 1 tue                 /* Rechteste Vorkommen in M ermitteln */
4   wenn δ1(M[j]) = |M| dann
5     δ1(M[j]) ← m − j
```

3.2 Die „Gutes–Ende“ Strategie

Die „Schlechtes–Zeichen“ Strategie liefert nicht immer gute Ergebnisse. Man kann der Tabelle aus dem obigen Beispiel schon entnehmen, dass ein Mismatch mit einem r einen Sprung um 0 verursachen würde. Es kann sogar passieren, dass dadurch das Muster nach links zurückgeschoben wird. Garnicht gut!

Einen Ausweg verleiht uns die Gutes–Ende Strategie. Sie soll uns eine Abbildung $\delta_2 : \{1, \ldots, m\} \to \mathbb{N}$ liefern, die uns die Sprungweite in Abhängigkeit der aktuellen Position j in M, an der ein Mismatch stattfand, angibt. Wir betrachten dabei das Ende $M[j+1, \ldots, m]$ des Musters, das bereits gematcht hat.

Fall 1 Betrachte Abbildung 5. Es findet ein Mismatch am Zeichen ’n’ in M an Position j statt. Da das Suffix $M[j+1, \ldots, m]$ ($= M[m] =$ ’a’ in diesem Fall) mit T gematcht hat,

können wir unser Muster soweit nach rechts verschieben bis dieses Suffix in M nochmals vorkommt und dann an dieser Stelle anliegt. Das nächste Vorkommen dieses Suffix in M ist an der Stelle banana. Hier würde jedoch wieder ein Mismatch stattfinden, weil wir ja schon wissen dass das 'n' vor dem 'a' zu einem Mismatch führt.

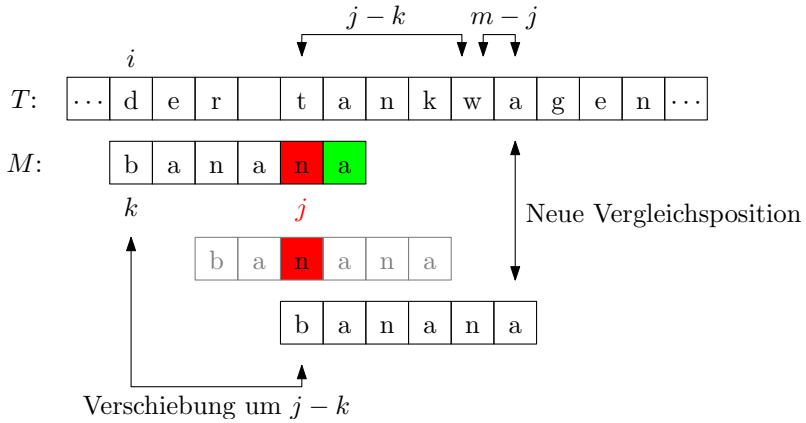

Abbildung 5: Gutes–Ende Strategie, wenn das Suffix das gematcht hat nochmals in M vorkommt, aber mit einem anderen Zeichen davor als das, an dem der Mismatch in M stattfand.

Wir suchen also das rechteste Vorkommen des Suffix $M[j+1,\ldots,m]$ in M, so dass „das Zeichen davor“ nicht mit $M[j]$ übereinstimmt. Nochmal formal: Finde ein größtmögliches k, so dass $M[k+1,\ldots,k+m-j] = M[j+1,\ldots,m]$ wobei $M[k] \neq M[j]$. Sowas muss natürlich nicht existieren, dazu später mehr. Wenn wir so ein k gefunden haben, so können wir das Muster um $j-k$ Stellen nach rechts verschieben[3]. Mit der gewohnten Addition von $m-j$ ergibt sich somit unsere Sprungdistanz durch

$$\delta_2(j) = m - k$$

Fall 2 Wir haben schon gesehen dass die Forderung das Suffix in M nochmals zu finden möglicherweise nicht erfüllt werden kann. In diesem Fall wird es jetzt etwas haarig.

Betrachten wir Abbildung 6. Das Muster M hat ein Mismatch an Stelle j. Das Suffix $M[j+1,\ldots,m] =$ '$taste$' kommt nicht nochmal in M vor. Wir dürfen trotzdem nicht maximal verschieben, denn der Anfang 'te' des Musters M ist im Ende des Suffix '$taste$' enthalten. Wir dürfen also nur so viel verschieben, bis das 'te' am Musteranfang bündig an dem 'te' im Text anliegt.

Formal lautet das dann folgendermaßen: Wir suchen ein möglichst großes k derart, dass $M[m-k+1,\ldots,m]$ (das Ende vom Suffix – im Beispiel 'te') gleich $M[1,\ldots,k]$ (der Anfang von M) übereinstimmt. Beim graphischen Ermitteln dieses k kann man sich

[3] Es gilt hier stets $j-k>0$, wir verschieben also immer mindestens um eine Stelle!

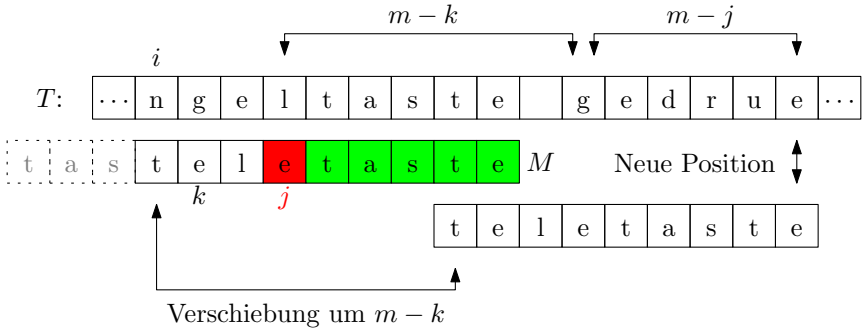

Abbildung 6: Gutes–Ende Strategie, wenn das Suffix das gematcht hat nicht nochmal in M vorkommt. Wir versuchen das Suffix so weit wie möglich mit dem Anfang des Musters zu Matchen.

dadurch behelfen, dass wir uns das Suffix $M[j+1, \ldots, m]$ imaginär vor das Muster stellen, und so weit wie möglich nach rechts schieben ohne dass eins der „überlappten“ Zeichen verschieden ist. Falls das überhaupt nicht möglich ist, so folgt $k = 0$!

Ist das k nun gefunden, so können wir das Muster um $m - k$ nach rechts verschieben, und erhalten somit als Sprungweite für diesen Fall

$$\delta_2(j) = (m - k) + (m - j) = 2m - k - j$$

Fazit Die Gutes–Ende Strategie ist etwas schwieriger als die Schlechtes–Zeichen Strategie. Sie garantiert uns jedoch, dass wir immer um mindestens ein Zeichen nach rechts verschieben. Insgesamt ergeben sich folgende Fälle:

(1) $j = m$

Für diesen Sonderfall (wir erhielten ein leeres Suffix), definieren wir uns die Sprungweite einfach durch

$$\delta_2(j) = 1$$

(2) Suffix existiert nochmals im Muster.

Wähle größtmögliches k so, dass $M[k+1, \ldots, k+m-j] = M[j+1, \ldots, m]$ aber $M[k] \neq M[j]$. Dann ist

$$\delta_2(j) = m - k$$

der Abstand des erneuten Vorkommens des Suffixes von rechts.

(3) „Sonst Fall“. Das Suffix existiert nicht nochmal im Muster mit der o.g. Bedingung.

Wähle größtmögliches k, so dass $M[1,\ldots,k] = M[m-k+1,\ldots,m]$. Dann gilt

$$\delta_2(j) = 2m - k - j$$

Anschaulich ist das die „Länge“ des Musters plus der imaginäre Teil des Suffix der nicht mit dem Anfang des Musters überlappt.

Ein kleines Beispiel zum Schluss. Für unser Wort „banana“ ergibt sich folgende Tabelle

σ:	b	a	n	a	n	a
j:	1	2	3	4	5	6
$m-k$:	6	6	5	6	5	1
$m-j$:	5	4	3	2	1	0
Fall:	3	3	2	3	2	1
δ_2:	11	10	5	8	5	1

Auch hier noch eine mögliche Formulierung der Strategie in Pseudo–Code

Algorithmus 3 : GutesEnde

Eingabe : Muster M
Ausgabe : Sprungfunktion $\delta_2 : \{1,\ldots,m\} \to \mathbb{N}$

```
/* Fall (1) */
```
1 $\delta_2(m) \leftarrow 1$
2 **für** $j \leftarrow (m-1)\ldots 1$ **tue**
3 $S \leftarrow M[j+1,\ldots,m]$
```
  /* Fall (2) */
```
4 **wenn** $k \leftarrow \max\{\kappa \mid M[\kappa+1,\ldots,\kappa+m-j] = S \wedge M[\kappa] \neq M[j]\} \neq \perp$ **dann**
5 $\delta_2(j) \leftarrow m-k$
```
  /* Fall (3) */
```
6 **sonst**
7 $k \leftarrow \max(\{\kappa \mid M[1,\ldots,k] = M[m-k+1,\ldots,m]\} \cup \{0\})$
8 $\delta_2(j) \leftarrow 2m-k-j$

3.3 Der Algorithmus

Wir wissen jetzt wie wir die Sprungtabellen berechnen. Ist diese Vorverarbeitung getan, so ist der eigentliche Suchalgorithmus erschreckend ähnlich zu unserem naiven Verfahren. Der einzige Unterschied ist, dass wir, vermöge ein Mismatch an der Stelle j mit dem Zeichen σ in T, um $\max\{\delta_1(\sigma), \delta_2(j)\}$ nach rechts verschieben.

Algorithmus 4 : BoyerMoore

Eingabe : Text T und Muster M
Ausgabe : Alle Vorkommen von M in T

```
1 δ1 ← SchlechtesZeichen(M)
2 δ2 ← GutesEnde(M)
3 für i ← 1 ... n − m tue
4     für j ← m ... 1 tue
5         wenn T[i + j − 1] ≠ M[j] dann
6             s ← max{δ1(T[i + j − 1]), δ2(j)}
7             i ← s − (m − j)
8             break
```

Der Index i der äußeren Schleife gibt stets die Position vor an der das Muster (linksbündig) im Text anliegt. Daher entspricht ein Neusetzen von i einer Verschiebung des Musters um $i_{\text{neu}} - i_{\text{alt}}$. Unsere δs geben jedoch nicht die Verschiebedistanz an, sondern die Sprungdistanz, also ab wo wieder verglichen wird. Eigentlich brauchen wir diese hier garnicht, und wir müssen das (durch Abziehen von $m - j$ von δ) wieder kompensieren. In der Vorlesung wurden jedoch für die Tabellen stets die Sprungdistanzen berechnet, daher möchte ich mich in diesem Papier auch daran halten.

Durch geschickte Implementierung der Vorverarbeitungsschritte lassen sich die Sprungtabellein in $\mathcal{O}(m)$ Zeit erstellen. Für den Suchvorgang lässt sich eine worst–case Laufzeit von $\mathcal{O}(n)$ beweisen. Im average–case verbessert sie sich sogar auf $\mathcal{O}(n/m)$[4]. Der Boyer & Moore Algorithmus ist also ein *sublinearer* Algorithmus, und stellt eine deutliche Verbesserung unseres ersten „naiven“ Ansatzes dar.

4 Beispiel

Als Abschluss noch ein Beispiel zur Durchführung des Algorithmus von Boyer & Moore.

Vermöge der Text $T := \textit{here is a simple example}$ und das Muster $M := \textit{example}$. Zunächst berechnen wir die Tabellen δ_1 und δ_2:

σ	e	x	a	m	p	l	sonst.
δ_1	0	5	4	3	2	1	7

	e	x	a	m	p	l	e
j	1	2	3	4	5	6	7
δ_2	12	11	10	9	8	7	1

Abbildung 7 verdeutlicht die einzelnen Schritte im Algorithmus. Eine ausfürhlichere und

[4] Die Laufzeitbeweise sind recht kompliziert

interaktive Demonstration dieses Beispiels findet sich außerdem in [3].

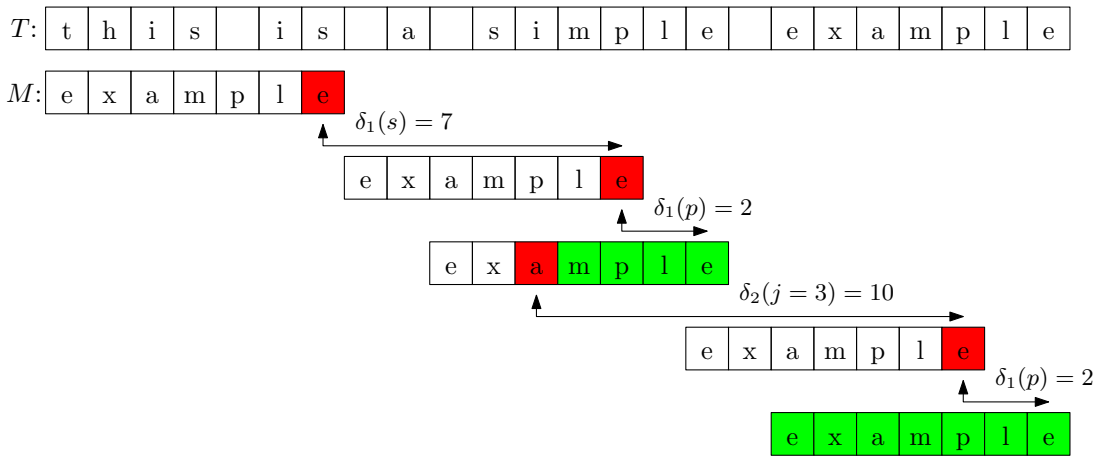

Abbildung 7: Illustration des Boyer & Moore Algorithmus.

Literatur

[1] R.S. Boyer, J.S. Moore, *A Fast String Searching Algorithm*, Communications of the ACM, 20, 10, 762-772 (1977)

[2] Prof. Dr.-Ing. R. Dillmann, Prof. Dr.-Ing. J. Beyerer, Dipl.-Ing. R. Unterhinninghofen, *Vorlesung Informatik IV*, SoSe 2006, Universität Karlsruhe (TH)

[3] J. Strother Moore, *Boyer–Moore Fast String Searching Example*, http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/fstrpos-example.html

[4] Christian Charras, Thierry Lecroq, *Exact String Matching Algorithms*, http://www-igm.univ-mlv.fr/ lecroq/string/