String-Matching-Algorithmen

von

Thomas Kramer

20.11.2011

1. Problemstellung und Allgemeines

String-Matching-Algorithmen sind Algorithmen, die das Finden von Textfragmenten innerhalb einer Zeichenkette ermöglichen. Ziel ist es, eine erfolgreiche Suche des Musters P innerhalb des Textes T mit dem geringsten Zeitaufwand zu realisieren.

Speziell bei String-Matching-Algorithmen sollte darauf hingewiesen werden dass es eine Menge an älterer Literatur gibt, die auf Delphi/Pascal setzen – im Vergleich zu anderen Programmiersprachen fängt bei Delphi der Index eines Strings bei 1 statt 0 an, bei der Adaption sollte dies bedacht werden.

Mittels Pseudocode wird die Arbeitsweise eines Algorithmus verdeutlicht, dazu werden allgemein umgangssprachliche Metaphern statt Programmbefehlen verwendet. Es gibt aber keine verbindlichen Konventionen dazu, in diesem[1] Skript der Universität Wuppertal beispielsweise werden trotzdem Programmbefehle im Pseudocode benutzt.

Allgemein wird bei Algorithmen zwischen einem Präprozessing und der allgemeinen Verarbeitungszeit unterschieden. Performancetechnisch relevant ist nur die eigentliche Suchzeit, weil das Präprozessing lediglich einmal stattfindet.

Die Bewertung von Algorithmen erfolgt mittels der O-Notation/Landau-Symbole. Bezüglich einer einführenden Übersicht in das Thema möchte ich auf die Präsentationsfolien der Universität Münster[2] verweisen – sowie diese Zusammenfassung[3].

1 http://www2.math.uni-wuppertal.de/~schaefer/Sammlung/String-Matching-Folien.pdf
2 http://cvpr.uni-muenster.de/teaching/ws08/info1WS08/script/Kapitel6-Complexity-1.pdf
3 http://tilman.de/uni/ws03/alp/o-notation.php

1.1 Definitionen und Notationen

- Ein Alphabet ∑ ist eine endliche Menge an Buchstaben, |∑| kennzeichnet den Betrag/Kardinalität von ∑.
- Der Text wird als T[1..n] bezeichnet, das Suchpattern/Muster dagegen als P[1..m] – programmiertechnisch folglich T[0..n-1] und P[0..m-1].
- In n=|T| wird die Länge des Textes gespeichert, in m=|P| die des Suchpatterns.
- q für eine Primzahl beim Rabin-Karp-Algorithmus
- N[m+1] für das next-Array beim Knuth-Morris-Pratt-Algorithmus
- Die Position im gesamten Text wird mit der Variablen i gezählt, die im Muster mit der Variablen j.
- Mit einem Match wird eine Übereinstimmung zwischen dem Muster und dem Text innerhalb des aktuellen Suchfensters bezeichnet, ein Mismatch kennzeichnet einen Unterschied.
- „false matches“ sind Übereinstimmungen zwischen Hashwerten des Textes und des Musters, bei gleichzeitigen Unterschieden zwischen den tatsächlichen Texten.
- Das Präprozessing ist die Vorbereitungszeit, die Suchzeit die allgemeine Verarbeitungszeit.

1.2 Quellcode in Processing

Bezüglich des Quellcodes wurde duchgängig PascalCase für Klassennamen und CamelCase für sonstige Bezeichner eingehalten, gemäß den Java-Coding Style Guidelines[4]. Bei den Einrückungsstilen wurde der Allman-Stil bevorzugt.

4 http://www.iwombat.com/standards/JavaStyleGuide.html#Class and Interface Names

2. Die naive Suche

Die naheliegendste Möglichkeit für die Implementierung einer Suche ist der naive Algorithmus, dabei wird der Text T[1..n] Zeichen für Zeichen mit dem Suchpattern P[1..m] verglichen.

Eine äußere Schleife durchläuft den Text ab dem ersten bis zum letzten Zeichen – abzüglich der Länge des Musters, folglich T[1..n-m+1]. In der inneren Schleife werden dann die nachfolgenden Zeichen (ab der aktuellen Position der äußeren Schleife) mit dem Muster verglichen.

Sobald eine komplette Übereinstimmung gefunden wurde wird ein Vorkommen gemeldet und die Suchroutine abgebrochen - sofern nur der erste Match gesucht wird. Beim ersten Mismatch dagegen wird die innere Schleife vorzeitig beendet und mit der Abarbeitung der äußeren Schleife fortgefahren.

Nachfolgend zwei Beispiele zur Verdeutlichung; mit grün werden die Matches gekennzeichnet, mit rot dagegen die Mismatches. Die Zeichen nach dem ersten Mismatch im Muster sind nicht mehr relevant und daher in der normalen Schriftfarbe gekennzeichnet.

a)
```
aaaaabaaaaaaab
aaab
 aaab
  aaab
```

b)
```
aabaababaabaaa
aabaaa
 aabaaa
  aabaaa
   aabaaa
```

Pseudocode für die naive Suche:

```
i=0;
solange (i <= n - m)
{
   j = 0;
   solange ((j < m) && (P[j] == T[i + j]))
   {
      j = j + 1;
   }

   falls (j == m) dann
   {
      Ausgabe: „Muster kommt an Stelle „ i „ vor.“;
   }

   i = i + 1;
}
```

Die äußere Schleife wird (n-m+1)-mal durchlaufen, die innere Schleife maximal m-mal. Ein Präprozessing existiert bei dieser Suchvariante nicht, die obere Schranke der eigentlichen Suchzeit beträgt O(n*m).

Jedoch hat der Algorithmus die Eigenschaft bei einem frühen Mismatch weniger Vergleiche zu machen, daher kann er bei großen Texten mit wenig Mustern oder nicht-natürlichsprachigen Mustern eine deutlich bessere Laufzeit erzielen. Bei einem Laufzeitvergleich dieses Algorithmus sollte daher kein unnatürliches Muster in einem Text gesucht werden, weil so keine repräsentative Aussage zu der Laufzeit gemacht werden kann.

Da der naive Algorithmus am einfachsten zu implementieren ist, ist auch seine Relevanz am höchsten – im Durchschnitt. Jedoch nutzt er nicht das Wissen aus vorherigen Vergleichen und die Abarbeitung ist nicht rein sequentiell – in der inneren Schleife werden Zeichen gelesen, die in der äußeren Schleife erst später durchlaufen werden.

Wenn in sehr großen Datenmengen nicht hin- und hergesprungen werden soll – etwa bei externen Speichermedien – existieren daher bessere Algorithmen.

Auch bezüglich der durchschnittlichen Laufzeit gibt es heute bessere Varianten – zumindest bei Verwendung längerer Suchpatterns. In meinen Laufzeittests mit sehr kurzen Suchpatterns ist der naive Algorithmus stets am schnellsten gewesen.

Die O-Notation bewertet eben nur die obere Schranke der Anzahl Operationen und berücksichtigt bspw. nicht dass manche Operationen wie Modulo (beim Rabin-Karp-Algorithmus) mehr Rechenzeit als andere Operationen (wie reine Stringvergleiche) benötigen.

	Vorbereitungszeit	**Suchzeit**
Naiver Algorithmus	0 (keine)	O(n*m)

2. Der Rabin-Karp-Algorithmus

Der Rabin-Karp-Algorithmus wurde von Michael O. Rabin und Richard M. Karp 1987 entwickelt und basiert auf Transformation, bei der Zeichenketten durch Zahlen repräsentiert werden.

Eine Ausarbeitung speziell zu dem RK-Algorithmus befindet sich in diesem PDF[5] der Universität Bielefeld, als vergleichende Übersicht kann dieses Skript[6] der Universität Paderborn gewertet werden.

Der Algorithmus berechnet an jeder Position in T[1…n] mit den nachfolgenden m Zeichen einen Hashwert und vergleicht sie mit dem Hashwert des Musters P[1..m]. Bei einer Übereinstimmung wird zusätzlich der Text an der aktuellen Position T[i..i+m] mit dem Suchpattern verglichen, um Sicherheit zu gewährleisten.

Das Verfahren benötigt eine Laufzeit von O(m) für das Präprozessing, weil initial ein Hashwert für das Muster P und der ersten m-Zeichen des Textes T berechnet werden müssen.

Bei den verwendeten Hash-Verfahren für den Rabin-Karp-Algorithmus gibt es durchaus Unterschiede bei der Implementierung, bei diesem Skript[7] der Fachhochschule Flensburg bspw. wird zunächst die Möglichkeit einer Quersumme vorgeschlagen. Vorschläge für weitere allgemeine Hash-Funktionen finden sich auch in dieser[8] Schrift.

Im allgemeinen wird als Hash-Funktion innerhalb des Rabin-Karp-Algorithmus nachfolgende Variante genommen:

a) Für die Zahl 1234 = (1*|∑|^3)+(2*|∑|^2)+(3*|∑|^1)+(4*|∑|^0) = 1234
b) Für den Text „hi“ = (104*|∑|^1)+(105*|∑|^0) = 26729

Wobei als Basis der Betrag |∑| = 10 für a) gewählt wird und |∑| = 256 im Fall von b) für die Anzahl möglicher Buchstaben im ASCII-Alphabet. 104 in diesem Beispiel ist der ASCII-Wert des Buchstabens „h“ und 105 für den Buchstaben „i“. Das Ergebnis dieser Rechnung wird dann modulo einer hohen Primzahl q genommen; ausgehend von dieser Zahl finden dann die Vergleiche statt.

Die Primzahl q sollte kleiner als q<=8m^2*log(e,m^2) betragen. Diese Schrift[9] der Technischen Universität München legt auf Seite 77 außerdem einen Beweis vor, dass q gleichzeitig größer als n*m liegen sollte. q kann außerdem bei zuvielen „false matches“ - wenn die Hashwerte übereinstimmen, aber die tatsächlichen Texte nicht – zufällig neu gewählt werden; diese Variante nennt sich dann „randomisierter Rabin-Karp-Algorithmus“.

---

5 http://www.techfak.uni-bielefeld.de/~jreeder/lehre/algorithmen_auf_zk/ausarbeitungen/RabinKarp.pdf
6 http://www.cs.uni-paderborn.de/fileadmin/Informatik/FG-TI/GA09/GA7-Strings.pdf
7 http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/karp.htm
8 http://www.ntecs.de/projects/guugelhupf/doc/html/x435.html
9 http://www14.informatik.tu-muenchen.de/lehre/2002SS/cb/lecturenotes/chapter2.pdf

Veranschaulichung der Vorgehensweise:

```
Text:                   1413921012
Muster:                 2101
Basis:                  10
Primzahl:               13
Hashwert des Musters: 2101 mod 13 = 8.
```

Vorgang:

```
1413921012
1413                        mod 13 = 9
1413921012
 4139                       mod 13 = 5
1413921012
  1392                      mod 13 = 1
1413921012
   3921                     mod 13 = 8, Match! aber: 3921 != 2101
1413921012
    9210                    mod 13 = 6
1413921012
     2101                   mod 13 = 8, Match!
1413921012
      1012                  mod 13 = 11
```

Um die Hash-Funktion zu verschnellern wird sowohl eine initiale Hash-Funktion, als auch eine „rollende“ Hash-Funktion implementiert. Die initiale Variante berechnet den Hashwert anhand des zuerst genannten Verfahrens - um Überläufe der üblichen 32/64-Bit-Arithmetik an dieser Stelle zu vermeiden wird nach jeder Rechenoperation bereits modulo genommen.

Die rollende Hash-Funktion berechnet den Wert nicht jedes Mal komplett neu sondern geht anders vor: Das Zeichen ganz links wird herausgerechnet und das neu hinzukommende Zeichen hinzugerechnet. Für diese Aktion wird ein Verschiebefaktor benötigt, dieser kann bereits in der initialen Hash-Funktion berechnet werden.

Angenommen es existiert ein Text „314152“ mit fünfstelligen Patterns, dann wird bei einer Basis von 10 als Verschiebefaktor der Wert 3 berechnet. Als Primzahl nehmen wir q=13, der erste Hashwert betrug 31415 modulo 13=7. Für den nächsten Hashwert wird dann ausgehend vom alten

h(14152) = ((7 – 3 * Verschiebefaktor) * Basis + 2) mod 13 = 8.

berechnet.

Neben der randomisierten Variante besteht eine weitere Möglichkeit den Algorithmus zu verschnellern, indem als Basis eine Primzahl und für den Divisor q eine Zweierpotenz genommen wird – auf Basis dessen kann die Modulo-Operation bitweise durchgeführt werden. Diese Schrift[10] der Harvard-Universität verdeutlicht es.

10 http://www.eecs.harvard.edu/~ellard/Q-97/HTML/root/node43.html#rkFast

Im Skript[11] der Technischen Universität München steht auf Seite 75 zum Rabin-Karp-Algorithmus:

„Die Zahl, durch welche modulo gerechnet wird, sollte teilerfremd zu |∑| sein, da ansonsten Anomalien auftreten können. Wählt man beispielsweise eine Zweierpotenz, so werden nur die ersten Zeichen des Suchwortes berücksichtigt, wo hingegen die letzten Zeichen des Suchwortes bei dieser Suche überhaupt keine Relevanz haben. Um unabhängig von der Alphabetgröße die Teilerfremdheit garantieren zu können, wählt man p als eine Primzahl.“

Eine Einschränkung, die natürlich auch für die andere Variante mit q als Zweierpotenz von der Harvard-Universität gelten muss – als Basis wird dann zu der Zahl 257 geraten, demnach eins mehr als das Alphabet.

Die Laufzeit der normalen Variante beträgt im Average Case O(n+m), im Worst Case dagegen wie beim naiven Algorithmus O(n*m). Der randomisierte Rabin-Karp-Algorithmus hat eine Laufzeit von O(m + n + 1/m * m * n).

Der Rabin-Karp-Algorithmus hat den einzigartigen Vorteil, beliebig viele Zeichenketten in einem Durchlauf im Durchschnitt in O(*n*) oder weniger zu finden.

Pseudo-Code für den Rabin-Karp-Algorithmus:

```
hs   = hash (T[0..m-1]);
hsub = hash (P);

i = 0;
solange (i <= (n - m))
{
   falls (hs == hsub) dann
   {
      falls (T[i..i+m] == P) dann
      {
         Ausgabe: „Muster kommt an Stelle „  i „ vor.“;
      }
   }
   hs = hash(T[i+1..i+m]);

   i = i + 1;
}
```

	Vorbereitungszeit	**Suchzeit**
Rabin-Karp-Algorithmus	O(m)	average O(n+m), worst O(n*m)

11 http://www14.informatik.tu-muenchen.de/lehre/2002SS/cb/lecturenotes/chapter2.pdf

3. Der Knuth-Morris-Pratt-Algorithmus

Der Knuth-Morris-Pratt-Algorithmus ist eine Weiterentwicklung des naiven Suchalgorithmus. Wesentlicher Unterschied ist, das das Vergleichsfenster nicht immer nur um eine Position weitergerückt wird, sondern eventuell um mehr als eine Position.

Ausarbeitungen zu diesem Algorithmus sind bspw. bei der Fachhochschule Köln[12] oder der Technischen Universität Dresden[13] zu finden.

Das Verfahren unterscheidet zwischen Präprozessing und eigentlichem Suchverfahren. Im Präprozessing wird zunächst das Muster P untersucht und das Ergebnis in einem Array N gespeichert.

Angenommen es gibt ein Muster „ababcabab“, dann wird die Präfix-Analyse folgende wiederkehrende Muster erkennen:

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

Programmiertechnisch ergeben sich darauf folgende Werte, die Abarbeitung erfolgt eigentlich erst ab dem Index 2, die Indizes 0-1 werden immer mit den gleichen Werten gefüllt. vZ steht für vorangegangene Zeichen.

i = Index	Muster	j = Wert	Bemerkung	Verschiebung [=i - j]
0	a	-1	Initialisierungswert	1
1	b	0		1
2	a	0	vZ: b	2
3	b	1	vZ: ba, Treffer: a	2
4	c	2	vZ: bab, Treffer: ab	2
5	a	0	vZ: babc	5
6	b	1	vZ: babca, Treffer: a	5
7	a	2	vZ: babcab, Treffer: ab	5
8	b	3	vZ: babcaba, Treffer: aba	5
9		4	vZ: babcabab, Treffer: abab	5

12 http://www.gm.fh-koeln.de/~hk/lehre/ala/ws0506/Praktikum/Projekt/A_lila/String-Searching_KMP_Boyer-Moore.pdf

13 http://www1.inf.tu-dresden.de/~s0368252/downloads/KMP.pdf

Nach der Erstellung des N-Arrays im Präprozessing erfolgt die eigentliche Suche folgendermaßen:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	b	a	b	a	b	c	b	a	b	a	b	c	a	b	a	b	c	a	b
a	b	a	b	c	a	b	a	b											
		a	b	a	b	c	a	b	a	b									
							a	b	a	b	c	a	b	a	b				
								a	b	a	b	c	a	b	a	b			

Der erste Mismatch ist beim 5. Zeichen mit dem Index 4. Die ersten 4 Zeichen waren übereinstimmend, also muss im N-Array an Indexposition 4 der Wert 2 ausgelesen werden - der Verschiebefaktor beträgt an der Stelle 4-2=2, dass Muster wird demnach um 2 Zeichen weiterbewegt.

Der zweite Mismatch ist beim 8. Zeichen mit dem Index 7. Die ersten 5 Zeichen waren übereinstimmend, also muss im N-Array an Indexposition 5 der Wert 0 ausgelesen werden – der Verschiebefaktor beträgt an der Stelle 5-0=5, dass Muster wird demnach um 5 Zeichen weiterbewegt.

Die Laufzeit des Knuth-Morris-Pratt-Algorithmus beträgt O(m) für das Präprozessing und O(n) für die eigentliche Suche.

Als besonderer Vorteil des KMP-Algorithmus wird immer wieder genannt dass mit ihm der Text ausschließlich sequentiell durchlaufen wird, der Index niemals zurückgesetzt wird[14].

Als weiterer Vorteil des KMP gilt dass die Größe des Alphabets bei der Laufzeit kaum noch eine Rolle spielt.

14 https://vowi.fsinf.at/images/c/ce/TU_Wien-Algorithmen_und_Datenstrukturen_2_VO_%28Raidl%29_-_Theoriefragen_ausgearbeitet_SS11.pdf

Pseudocode für Knuth-Morris-Pratt-Algorithmus:

Präprozessing:

```
i=0;
j=-1;
N[i] := j;

solange (i < m)
{
  solange ((j >= 0) && (P[j] != P[i]))
  {
    j = N[j];
  }

  i = i + 1;
  j = j + 1;
  N[i] = j;
}
```

Eigentliche Suche:

```
i = 0;
j = 0;

solange (i < n)
{
   solange ((j >= 0) && (T[i] != P[j]))
   {
      j = N[j];
   }

   i = i + 1;
   j = j + 1;

   falls (j == m) dann
   {
     Ausgabe: „Muster kommt an Stelle „ (i-j) „ vor.“;
     j = N[j];
   }
}
```

	Vorbereitungszeit	Suchzeit
Knuth-Morris-Pratt-Algorithmus	O(m)	O(n)

4. Laufzeitvergleich

Nachfolgend die Laufzeitanalysen. Das Testsystem ist ein Pentium DualCore-T4200-Notebook @ 2 GHZ mit Windows 7 64-Bit und 2 GB RAM.

Zur Laufzeitmessung wurde eine Textdatei mit einer Länge von ca. 64 MB erstellt - ein Suchpattern mit einer Länge von 10.000 Zeichen wurde unten angefügt so dass die Suche immer weitgehend bis zum Ende läuft.

Von diesem Suchpattern wurden zuerst die ersten 1.000 Zeichen gesucht, danach die ersten 2.000 Zeichen, danach die ersten 3.000 Zeichen etc.

Jeder Durchlauf wurde automatisch 10x intern durchgeführt, und aus den Ergebnissen jeweils der Median[15] berechnet, welcher Extremwerte minimiert. Die Millisekunden-Ergebnisse dürften teilweise bereits im Bereich der Messungenauigkeit liegen, wichtiger ist jedoch der Laufzeitvergleich der verschiedenen Algorithmen untereinander.

Für eine wirklich repräsentative Aussage müsste mit noch deutlich größeren Dateien getestet werden, jedoch konnte bei der Dateigröße nicht wesentlich weiter gegangen werden weil der Java-Prozess etwa 10x soviel RAM verbraucht wie die jeweilige Testdatei.

Die naive Suche ist allgemein bei kurzen Suchpatterns im Vorteil, da sie die Eigenschaft hat bei frühen Mismatches weniger Vergleiche zu machen.

„Somit ist ersichtlich dass er bei einem großen zugrundeliegenden Alphabet und Texten die wenig Muster aufweisen, welche den Präfixen des Suchwortes s entsprechen, eine deutlich bessere Laufzeit aufweist als O(n*m). [...] Der Knuth-Morris-Pratt-Algorithmus ist laut [Gu99 S.23] der beste bekannte Algorithmus für das „exact pattern matching“-Problem. “ schreibt Jan Ackermann in seinem PDF[16] der Universität Münster.

Im Allgemeinen sollte daher von den hier vorgestellten Algorithmen der KMP-Algorithmus bevorzugt werden, mit der besten durchschnittlichen Laufzeit von O(n).

Abgeschlagen in diesem Test der RK-Algorithmus, der zwar eine bessere durchschnittliche Laufzeit von O(n+m) als der naive Algorithmus mit O(n*m) besitzt, diesen Vorteil aber weitgehend durch seine teuren Modulo-Operationen zunichte macht. Seine Überlegenheit gegenüber dem naiven Algorithmus wird er wohl erst bei noch deutlich größeren Suchpatterns ausspielen.

Vom Rabin-Karp-Algorithmus wurde nur noch die 32-Bit-Variante getestet – nach der Änderung dass in der Hash-Funktion nach jeder Potenzierung bereits eine Modulo-Operation erfolgt ist die zusätzliche 64-Bit-Variante unnötig geworden, da keine Gefahr von Überläufen mehr besteht.

15 http://www.mathe-lexikon.at/statistik/lagemasse/median.html
16 http://www.wi.uni-muenster.de/pi/lehre/ss05/seminarSuchen/Ausarbeitungen/JanAckermann.pdf

Für den Laufzeitvergleich wurde bereits die Variante des Rabin-Karp-Algorithmus mit bitweiser Modulo-Operation benutzt, die normale Variante war nochmal ca. 50% langsamer.

Nachfolgend die Laufzeiten für den naiven Algorithmus:

- 1000 Zeichen:
Gesamtzeit 25 sec 135 msec, Median über 10 Durchläufe: 2 sec 571 msec
Maximale Differenz: 00 min, 00 sec, 629 milliseconds

- 2000 Zeichen:
Gesamtzeit 24 sec 909 msec, Median über 10 Durchläufe: 2 sec 516 msec
Maximale Differenz: 00 min, 00 sec, 641 milliseconds

- 3000 Zeichen:
Gesamtzeit 24 sec 116 msec, Median über 10 Durchläufe: 2 sec 616 msec
Maximale Differenz: 00 min, 01 sec, 027 milliseconds

- 4000 Zeichen:
Gesamtzeit 25 sec 878 msec, Median über 10 Durchläufe: 2 sec 662 msec
Maximale Differenz: 00 min, 00 sec, 496 milliseconds

- 5000 Zeichen:
Gesamtzeit 23 sec 174 msec, Median über 10 Durchläufe: 2 sec 522 msec
Maximale Differenz: 00 min, 01 sec, 068 milliseconds

- 6000 Zeichen:
Gesamtzeit 24 sec 519 msec, Median über 10 Durchläufe: 2 sec 748 msec
Maximale Differenz: 00 min, 01 sec, 117 milliseconds

- 7000 Zeichen:
Gesamtzeit 28 sec 132 msec, Median über 10 Durchläufe: 2 sec 881 msec
Maximale Differenz: 00 min, 00 sec, 298 milliseconds

- 8000 Zeichen:
Gesamtzeit 25 sec 557 msec, Median über 10 Durchläufe: 2 sec 658 msec
Maximale Differenz: 00 min, 00 sec, 712 milliseconds

- 9000 Zeichen:
Gesamtzeit 22 sec 610 msec, Median über 10 Durchläufe: 2 sec 254 msec
Maximale Differenz: 00 min, 01 sec, 113 milliseconds

- 10000 Zeichen:
Gesamtzeit 22 sec 957 msec, Median über 10 Durchläufe: 2 sec 448 msec
Maximale Differenz: 00 min, 01 sec, 088 milliseconds

Nachfolgend die Laufzeiten für den (32-Bit-) Rabin-Karp-Algorithmus:

- 1000 Zeichen:
Gesamtzeit 42 sec 806 msec, Median über 10 Durchläufe: 4 sec 421 msec
Maximale Differenz: 00 min, 01 sec, 083 milliseconds

- 2000 Zeichen:
Gesamtzeit 38 sec 379 msec, Median über 10 Durchläufe: 4 sec 297 msec
Maximale Differenz: 00 min, 01 sec, 794 milliseconds

- 3000 Zeichen:
Gesamtzeit 42 sec 753 msec, Median über 10 Durchläufe: 4 sec 370 msec
Maximale Differenz: 00 min, 00 sec, 727 milliseconds

- 4000 Zeichen:
Gesamtzeit 42 sec 182 msec, Median über 10 Durchläufe: 4 sec 384 msec
Maximale Differenz: 00 min, 00 sec, 809 milliseconds

- 5000 Zeichen:
Gesamtzeit 38 sec 671 msec, Median über 10 Durchläufe: 3 sec 801 msec
Maximale Differenz: 00 min, 01 sec, 641 milliseconds

- 6000 Zeichen:
Gesamtzeit 41 sec 200 msec, Median über 10 Durchläufe: 4 sec 338 msec
Maximale Differenz: 00 min, 01 sec, 716 milliseconds

- 7000 Zeichen:
Gesamtzeit 39 sec 77 msec, Median über 10 Durchläufe: 4 sec 135 msec
Maximale Differenz: 00 min, 01 sec, 565 milliseconds

- 8000 Zeichen:
Gesamtzeit 39 sec 841 msec, Median über 10 Durchläufe: 4 sec 257 msec
Maximale Differenz: 00 min, 01 sec, 804 milliseconds

- 9000 Zeichen:
Gesamtzeit 40 sec 394 msec, Median über 10 Durchläufe: 4 sec 277 msec
Maximale Differenz: 00 min, 01 sec, 714 milliseconds

- 10000 Zeichen:
Gesamtzeit 40 sec 908 msec, Median über 10 Durchläufe: 4 sec 340 msec
Maximale Differenz: 00 min, 01 sec, 619 milliseconds

Nachfolgend die Laufzeiten für den Knuth-Morris-Pratt-Algorithmus:

- 1000 Zeichen:
Gesamtzeit 24 sec 180 msec, Median über 10 Durchläufe: 2 sec 571 msec
Maximale Differenz: 00 min, 00 sec, 830 milliseconds

- 2000 Zeichen:
Gesamtzeit 24 sec 875 msec, Median über 10 Durchläufe: 2 sec 709 msec
Maximale Differenz: 00 min, 01 sec, 103 milliseconds

- 3000 Zeichen:
Gesamtzeit 26 sec 889 msec, Median über 10 Durchläufe: 2 sec 718 msec
Maximale Differenz: 00 min, 00 sec, 253 milliseconds

- 4000 Zeichen:
Gesamtzeit 27 sec 102 msec, Median über 10 Durchläufe: 2 sec 741 msec
Maximale Differenz: 00 min, 00 sec, 357 milliseconds

- 5000 Zeichen:
Gesamtzeit 23 sec 250 msec, Median über 10 Durchläufe: 2 sec 273 msec
Maximale Differenz: 00 min, 01 sec, 184 milliseconds

- 6000 Zeichen:
Gesamtzeit 26 sec 349 msec, Median über 10 Durchläufe: 2 sec 697 msec
Maximale Differenz: 00 min, 00 sec, 439 milliseconds

- 7000 Zeichen:
Gesamtzeit 23 sec 760 msec, Median über 10 Durchläufe: 2 sec 400 msec
Maximale Differenz: 00 min, 00 sec, 936 milliseconds

- 8000 Zeichen:
Gesamtzeit 26 sec 375 msec, Median über 10 Durchläufe: 2 sec 704 msec
Maximale Differenz: 00 min, 00 sec, 724 milliseconds

- 9000 Zeichen:
Gesamtzeit 26 sec 349 msec, Median über 10 Durchläufe: 2 sec 684 msec
Maximale Differenz: 00 min, 00 sec, 489 milliseconds

- 10000 Zeichen:
Gesamtzeit 33 sec 767 msec, Median über 10 Durchläufe: 3 sec 424 msec
Maximale Differenz: 00 min, 00 sec, 526 milliseconds

Tabellarische Übersicht:

	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Naive Suche	0,63	0,64	1,03	0,5	1,07	1,12	0,3	0,71	1,11	1,09
RK-Algorithmus	1,08	1,79	0,73	0,81	1,64	1,72	1,57	1,8	1,71	1,62
KMP-Algorithmus	0,83	1,1	0,25	0,36	1,18	0,44	0,94	0,72	0,49	0,53

Graphische Übersicht: