Fast Pattern Matching in Strings Knuth-Morris-Pratt

Seminararbeit
zum Abschluss des Seminars
Datenstrukturen, Algorithmen und deren effiziente Implementierung

vorgelegt von
Malte Laukötter
geboren in Hamburg

Institut für Eingebettete Systeme
Technische Universität Hamburg

Januar 2019

Prüfer : Prof. Dr.-Ing. Görschwin Fey
Rezensent : Torben Lehmann

Kurzfassung

Diese Ausarbeitung erklärt den Knuth-Morris-Pratt Algorithmus zum finden von Mustern in Texten. Der Algorithmus nutzt hierfür eine Vorberechnung die Wiederholungen im Muster erkennt und somit eine Laufzeit von **O**(Musterlänge + Textlänge) erreicht. Zur Anwendung kommt dieser Algorithmus trotz der geringen Komplexität nur selten, da der Boyer-Moore Algorithmus eine im Durchschnitt schnellere Laufzeit hat und es zusätzlich viele Algorithmen für spezielle Textarten gibt.

Abstract

This essay explains the Knuth-Morris-Pratt algorithm for fast pattern matching in strings. The algorithm uses a precomputation to detect repeated sections of the pattern to reach a runtime of **O**(patternlength + textlength). Eventhough the algorithm has this short runtime it isn't used that ofthen as the Boyer-Moore algorithm has a shorter runtime in the average case and a lot of other algorithms for special cases exist.

Inhaltsverzeichnis

Abkürzungsverzeichnis

KMP Knuth-Morris-Pratt / Algorithmus von Knuth-Morris-Pratt

DFA Deterministic finite automaton / Deterministischer endlicher Automat

1 Das Problem

Das Problem, welches vom Algorithmus gelöst wird, ist das finden eines Muster in einem Text. So findet der Algorithmus in dem Text "*Dies ist ein Text*"bei der Suche nach dem Muster "*in*"das Vorkommen dieses an der 11. Stelle ("*Dies ist e**in** Text*"). Der Algorithmus kann modifiziert werden um zum Beispiel nicht nur das erste Aufkommen zu finden, sondern alle Aufkommen eines Musters, oder auch um nach dem längsten Präfix eines Musters im Text zu suchen.

2 Anwendungen

Zur Anwendung kommen kann KMP in allen Fällen in denen ein Text in einem anderen Text gesucht wird, so zum Beispiel in der DNA Analyse, Textbearbeitungsprogrammen, Suchmaschinen oder auch in Angriffserkennungssystem von Computern, die durch den Vergleich von Angriffssignaturen versuchen Angriffe zu erkennen [1]. Allerdings ist er nicht immer der schnellste Algorithmus für alle Anwendungen, so wird insbesondere bei Textbearbeitungsprogrammen und der Angriffserkenneung hauptsächlich auf den Boyer-Moore Algorithmus gesetzt [2]. Besonders geeignet ist KMP für das Finden von längsten Präfixen eines Musters in einem Text, wofür der Algorithmus zwar etwas modifiziert werden muss, aber aufgrund der Art des Überprüfens dann schneller als viele andere Algorithmen ist [2].

3 Entstehungsgeschichte

Der Algorithmus wurde 1970 von Donald Knuth und Vaughan Pratt basierend auf Cook's Theorem über Zweiwege-DFA's entwickelt. Ursprünglich entwickelte Donald Knuth mithilfe des Theorems einen Algorithmus um konkatenierte Palindrome zu erkennen und stellte dann fest, dass er auf die selbe Art auch einen einen Textsuchalgorithmus entwickeln konnte, welcher dann hauptsächlich von Vaughan Pratt umgesetzt und von Donald Knuth aufgeschrieben wurde. Wie sich dann später herausstellte hatte James Morris bereits 1969 den selben Algorithmus, ohne Cook's Theorem zu verwenden, entwickelt und bereits in einem Programm verwendet, aus welchem dieser, da verschiedene Reparaturen ihn unlesbar gemacht hatten und auch kein anderer Entwickler außer Morris diesen überhaupt verstand, wieder entfernt wurde. [3]

4 Der Algorithmus

In diesem Abschnitt wird darauf eingegangen wie der Algorithmus genau Funktioniert, wofür als erstes ein simpler naiver Suchalgorithmus vorgestellt wird, auf welchem der KMP Algorithmus basiert. Daraufhin werden dann der eigentliche Algorithmus und einige mögliche Erweiterung erklärt.

4.1 Naive Suche

Bei der Naiven Suche handelt es sich um einen sehr einfachen Suchalgorithmus, welcher das Problem auf eine einfache Variante löst ohne auf Laufzeit oder Speichereffizenz zu achten.

Die Idee des Algorithmus ist es, beginnend bei jedem Zeichen des zu durchsuchenden Textes, zu überprüfen ob dieser mit dem ersten Zeichen des Musters übereinstimmt und dann jedes darauf folgenden Zeichen mit dem jeweils nächsten Zeichen des Musters zu vergleichen. Sollten an einer Stelle alle Zeichen des Musters mit denen des zu durchsuchenden Textes übereinstimmen wird dann die Position des ersten Zeichens zurückgegeben. Dies kann man sich so vorstellen, dass das Muster an dem Text vorbeigeschoben wird. Dies ist in Tab. 4.1, für die 3. Startposition, zu sehen.

Text	A	B	A	A	B	A	A	B	A	C
Muster			A	**B**	A	A	B	A	C	

Tabelle 4.1: Ein Zustand des Naiven Algorithmus

Um dies in einem Programm umzusetzen werden 2 Zähler benötigt welche die aktuelle Startposition im zu durchsuchenden Text (j) und die aktuell zu überprüfende Stelle im Muster (k) speichern. In der Schleife werden dann alle Zeichen des Textes durchgegangen und für jedes dann überprüft ob diese äquivalent sind mit den entsprechenden Zeichen des Textes, hierfür wird j in jedem Durchgang um 1 erhöht, solange das Ende des Musters noch nicht erreicht wurde und die Zeichen übereinstimmen. Am Ende des Durchgangs für ein Zeichen des Textes wird dann in der Bedingung der äußeren Schleife überprüft, ob weder das gesamte Muster gefunden wurde ($k \leq n$) noch alle Stellen des Textes überprüft worden sind (j $\leq$ m) und solange beide Bedingungen erfüllt sind dann mit dem nächste möglichen Startzeichen fortgefahren und k wieder auf 1 gesetzt, damit wieder am Anfang des Musters weiter verglichen wird. In der darauf folgenden If-Bedingung wird dann nochmal überprüft, welcher der beiden Fälle für das Ende der Schleife verantwortlich war und dann entsprechend die Startposition des Auftreten des Musters (k - m) oder, wenn das Muster nicht gefunden wurde, *-1* ausgegeben.

Input:
text: zu durchsuchender Text
pattern: zu findendes Muster
Result: Position des ersten Zeichen des Musters, -1 wenn das Muster nicht gefunden wurde

```
1  j := 1;
2  k := 1;
3  m := length(pattern);
4  n := length(text);
5  while j ≤ m and k ≤ n do
6  |  j := 1;
7  |  while j ≤ m and text[k+j-1] = pattern[j] do
8  |  |  j := j + 1;
9  |  end
10 |  k := k + 1;
11 end
12 if j < m then
13 |  return k - m;
14 end
15 return -1;
```

Algorithm 1: Naive Suche

4.2 KMP

Der KMP Algorithmus basiert auf der Naiven Suche wobei in diesem, anstatt jeden möglichen Startbuchstaben auszuprobieren, einige überspringt, wenn bekannt ist, dass bei diesem keine Übereinstimmung entstehen kann. Dies erkennt der Algorithmus anhand der Stelle im Muster in der beim da vorigen Startbuchstaben keine Übereinstimmung mehr vorhanden war. Hierfür wird vor der Suche eine Tabelle berechnet die für jedes Zeichen des Musters angibt wie viele Startzeichen übersprungen werden können.

Solche möglichen Verschiebungen treten im Muster auf, wenn dieses Wiederholungen, von Teilen von sich, in sich hat. So erkennt der KMP Algorithmus in Tabelle 4.2, dass es nachdem es keine Übereinstimmung beim C gab, dass es keinen Sinn macht als nächstes beim ersten B des Textes weiter zu überprüfen, sondern es möglich ist das Muster um 3 Felder weiter zu verschieben (siehe Tabelle 4.3) und dann an der aktuellen Stelle fortzufahren.

Text	A	B	A	A	B	A	A	B	A	C
Muster	A	B	A	A	B	A	**C**			

Tabelle 4.2: Zustand vom KMP bei der ersten fehlende Übereinstimmung

Text	A	B	A	A	B	A	A	B	A	C
Muster				A	B	A	**A**	B	A	C

Tabelle 4.3: Zustand vom KMP direkt nach der Verschiebung aufgrund der ersten fehlenden Übereinstimmung

Im folgenden werden jetzt die beiden Teile des Algorithmus einzeln betrachtet und erst erklärt wie die Suche funktioniert, wenn die Tabelle der Vorberechnung bereits existiert und dann erklärt wie diese Tabelle mithilfe des Musters berechnet wird. Die Zusammensetzung dieser beiden Teile des Algorithmus ist in Algorithmus 2 zu sehen.

Input:
text: zu durchsuchender Text
pattern: zu findendes Muster
Result: Position des ersten Zeichen des Musters, -1 wenn das Muster nicht gefunden wurde

```
1 next := kmpPrecomputation(pattern);
2 return kmpSearch(text, pattern, next);
```

Algorithm 2: KMP

4.2.1 Suche

Beim Suchalgorithmus handelt es sich um eine modifizierte Version der naiven Suche die den Text nicht immer nur um ein Zeichen verschiebt.
Dieser speichert in der Variable j nicht mehr die aktuelle Startposition des Überprüfungsschrittes des Textes, sondern die Position im Text an der sich der Algorithmus gerade befindet. k repräsentiert weiterhin die aktuell zu überprüfende Stelle im Muster. Sollten Text und Muster an einer Stelle nicht mehr übereinstimmen wird aus der Tabelle *next* das nächste zu überprüfende Feld des Musters, anhand der zuvor überprüften Stelle des Musters, ausgelesen und dann erneut überprüft ob diese übereinstimmen. Dies geschieht solange bis Text und Muster an dieser Stelle des Textes übereinstimmen. (siehe Zeilen 6 - 8, Algo. 3) Daraufhin werden Text und Muster um eine Position weiter verschoben um dann zu Testen ob diese an der nächsten Stelle übereinstimmen.
Sobald der Zähler k die letzte Stelle des Musters überschritten hat ist bekannt, dass das Muster eine Musterlänge vor der aktuellen Stelle des Textes anfängt und dieser Wert wird ausgegeben. Sollte zu einem Zeitpunkt der Zähler j die Länge des Textes überschreiten und gleichzeitig nicht das Ende des Muster erreicht ist ist bekannt, dass das Muster nicht im Text auftritt und es wird *-1* ausgegeben (siehe Zeilen 12 - 15, Algo. 3).
Wichtig für die Geschwindigkeit des Algorithmus ist insbesondere die Reihenfolge der Bedingungen in der 2. While-Schleife (Algo. 3 Zeile 6), da ein Vergleich der Elemente der beiden Felder *text[k]* und *pattern[j]* deutlich langsamer ist als der Test ob j größer als 0, beziehungsweise, da j zwingend positiv ist, ungleich 0, ist.

Input:
text: zu durchsuchender Text
pattern: zu findendes Muster
next: Tabelle aus der Vorberechnung
Result: Position des ersten Zeichen des Musters, -1 wenn das Muster nicht gefunden wurde

```
1  j := 1;
2  k := 1;
3  m := length(pattern);
4  n := length(text);
5  while j ≤ m and k ≤ n do
6  |  while j > 0 and text[k] ≠ pattern[j] do
7  |  |  j := next[j];
8  |  end
9  |  j := j + 1;
10 |  k := k + 1;
11 end
12 if j > m then
13 |  return k - m;
14 end
15 return -1;
```

Algorithm 3: KMP Suche

4.2.2 Vorberechnung

Die Tabelle die für die Verschiebungen benötigt wird beinhaltet für jede Stelle des Musters, welche Stelle des Musters als nächstes überprüft werden muss, wenn an der entsprechenden Stelle keine Übereinstimmung existiert aber alle vorherigen Stellen mit dem Text übereinstimmten. Für das auch schon weiter oben gezeigte Beispiel Muster ergeben sich daraus die Werte aus Tab. 4.4.

Muster	A	B	A	A	B	A	C
Tabelle	0	1	0	2	1	0	4

Tabelle 4.4: Beispiel für eine Tabelle aus der Vorberechung

Der Wert beim C ergibt sich so zum Beispiel daraus, dass die Zeichen zwischen dem 4. Zeichen und dem C (ABA) so auch am Anfang des Musters auftreten und dadurch bereits bekannt ist, dass die ersten 3 Zeichen des Musters bereits gefunden wurden, wenn erst beim C keine Übereinstimmung mehr gefunden wird. Deswegen müssen diese 3 Zeichen nicht nochmal überprüft werden sondern es kann direkt das 4. Zeichen überprüft werden.

Eine Besonderheit in der Tabelle betrifft die *0*, da es kein 0. Zeichen vom Muster gibt, diese bedeutet, dass es bei dem aktuellen Zeichen des Textes keine Möglichkeit mehr gibt, dass diese Teil des Musters ist und somit direkt das nächste Zeichen des Textes überprüft werden kann. Dementsprechend bekommt das erste Zeichen des Musters auch immer den Wert *0* zugeordnet. Hier betrifft dies alle A's denen nicht direkt ein weiters A folgt.

Um diese Tabelle zu berechnen wird im Muster nach Übereinstimmungen gesucht, genauer gesagt wird für jeden Präfix des Musters nach einem echten Suffix[1] der gleichzeitig auch ein Präfix des Präfix des Musters ist gesucht. Sprich es wird im obigen Beispiel zuerst nur das A betrachtet, dann AB, dann ABA und so weiter.

Um dies in einem Programm umzusetzen werden wieder 2 Zähler benötigt, zum einen *pos* welcher die letzte Stelle des aktuellen Präfix des Musters speichert und zum anderen *cnd* in welchem den nächsten Wert für die Tabelle gespeichert wird. Zusätzlich wird auch bereits das erste Element der *next* Tabelle auf 0 gesetzt, da dies immer der Wert des 1. Zeichens ist.
Innerhalb der Schleife wird zuerst der *cnd* solange verringert, indem jeweils der Wert aus der Tabelle, von dem Zeichen das mit dem aktuellen *cnd* Wert als nächstes im Suchalgorithmus angesehen geworden wäre, als neuer wert von *cnd* übernommen wird, bis dieser Entweder 0 ist oder ein Zeichen gefunden wurde in welches mit dem aktuelle Letzten Zeichen des Musterteils übereinstimmt (vgl. Algo. 4.4 Zeilen 5-7). Daraufhin wird dann, wenn auch noch das nächste Zeichen beider Stellen übereinstimmen der *next*-Wert von *cnd* für die aktuelle Position übernommen und andernfalls *cnd* als *next*-Wert verwendet.
Vorstellen kann man sich das zu gewissen Teilen so als wenn der Suchalgorithmus auf jeden Präfix vom Muster mit sich selbst als Text und Muster angewendet werden würde.

Input: pattern: zu findendes Muster
Result: Tabelle mit den Verschiebungen für die Suche

```
1  pos := 1;
2  cnd := 0;
3  next[1] := 0;
4  while pos < length(pattern) do
5  |  while cnd > 0 and pattern[pos] ≠ pattern[cnd] do
6  |  |  cnd := next[cnd];
7  |  end
8  |  cnd := cnd + 1;
9  |  pos := pos + 1;
10 |  if pattern[pos] = pattern[cnd] then
11 |  |  next[pos] := next[cnd];
12 |  else
13 |  |  next[pos] := cnd;
14 |  end
15 end
16 return next;
```

Algorithm 4: KMP Vorberechung

4.3 Erweiterungen

Der Algorithmus kann erweitert werden um zum Beispiel mehr als nur das erste Vorkommen des Musters im Text zu finden, indem, als einfache Variante, in jedem Durchgang der äußeren Schleife überprüft wird ob das Ende des Musters erreicht ist und dann die Startposition des Musters gespeichert

[1] der Suffix hat weniger Zeichen als der eigentliche Text, sprich ABA ist kein echter Suffix von ABA aber von $CABA$

wird. Daraufhin müsste dann die Position im Text auf die vorherige Position gesetzt werden, wodurch die Laufzeit des Algorithmus deutlich verschlechtert werden würde. In ihrer Veröffentlichung stellen Knuth, Morris und Pratt außerdem noch verschiedene andere Möglichkeiten vor alle Vorkommen eines Musters zu finden, welche ohne diese deutliche Verschlechterung funktionieren, dafür allerdings deutlichere Modifikationen des Algorithmus benötigen um alle Aufkommen in einer Liste zu speichern, an derer jeweils die ersten j Zeichen übereinstimmen.

Eine weitere Erweiterung die Knuth, Morris und Pratt, für lange Texte, vorschlagen ist es, da viele Systeme gleich ganze Worte aus dem Speicher auslesen und vergleichen können, alle mögliche Worte als Zeichen des Alphabets zu behandeln und dann gleich ganze Worte zu vergleichen. Hierfür ist es außerdem nötig für jede mögliche Start Position des Musters in einem Wort eine einzelne Suche durchzuführen, da ansonsten nur aufkommen des Musters gefunden werden können, die am Anfang eines Wortes anfangen. Diese Modifikation ist im KMP Algorithmus effizient möglich, da die Laufzeit, anders als bei Boyer-Moore, nicht von der Anzahl Zeichen im Alphabet abhängig ist.

Auch ist es möglich in einem Text nach mehreren Mustern gleichzeitig zu suchen, indem eine j Variable und *next* Tabelle für jedes Muster angelegt und in jedem Schleifendurchlauf angepasst wird. Dies würde erst einmal die Laufzeit Verschlechtern kann aber theoretisch wieder zu $\mathbf{O}$(#Zeichen im Text) verbessert werden, der Beweis und die Anpassungen die hierfür notwendig sind wurden allerdings nie veröffentlicht..

5 Komplexität und Speicherbedarf

Die Komplexität von KMP setzt sich zusammen aus der Laufzeit der Vorberechnung und der eigentlichen Suche. Die Vorberechnung hat eine Komplexität von $\mathbf{O}$(#Zeichen im Muster). Dies ergibt sich daraus, dass die Verschiebung in der inneren Schleife (siehe Algo. 4.4, Zeile 6) maximal so oft ausgeführt wird wie das Muster Zeichen hat, da die Verschiebung eine Kopie des Musters immer nach rechts weiter verschiebt. Die eigentliche Suche hat, mit fast der selben Begründung, eine Komplexität von $\mathbf{O}$(#Zeichen im Text).

Im Vergleich mit anderen Algorithmen für die selbe Aufgabe (siehe Tab. 5.1) zeigt sich, dass KMP ein Algorithmus ist der im schlechtesten Fall deutlich schneller als andere Algorithmen ist, Boyer-Moore aber auch schneller als KMP sein kann.

Algorithmus	Vorberechnung	Suche
Naive Suche		$\mathbf{O}(n * m)$
KMP	$\mathbf{O}(m)$	$\mathbf{O}(n)$
Boyer-Moore	$\mathbf{O}(m + k)$	bester Fall: $\mathbf{\Omega}(n/m)$
		schlechtester Fall: $\mathbf{O}(m * n)$
Rabin-Karp	$\mathbf{O}(m)$	$\mathbf{O}(n * m)$

Tabelle 5.1: Komplexität von Suchalgorithmen [2][4]
Länge des Musters: m
Länge des Textes: n
Elemente im Alphabet: k

Aufgrund der immer zugesicherten linearen Laufzeit, ist es möglich den KMP Algorithmus so zu modifizieren, dass er die Anforderungen an ein Echtzeitsystem erfüllt[5] und bietet sich für solche besser an als Boyer-Moore, da für ein solches System die schlechteste Laufzeit ausschlaggebend ist und dieser bei Boyer-Moore deutlich schlechter ist als bei KMP.

Besonders bemerkenswert ist, dass KMP nur ein Zeichen des zu durchsuchenden Textes zur Zeit im Speicher halten muss und nie ein Zeichen nochmals benötigt, welches bereits wieder aus dem Speicher entfernt wurde. Dies ist der Fall, da nie ein vorheriges Zeichen des Textes nochmals überprüft werden muss und kann bei Systemen mit wenig Speicher hilfreich sein, da der Speicher für den Text sofort wiederverwendet werden kann und dennoch nie darauf gewartet werden muss, ein bereits einmal gelesenes Zeichen nochmal zu lesen.

6 Laufzeit bei verschiedenen Textarten

Um die praktische Laufzeit von KMP im Vergleich zur naiven Suche zu untersuchen habe ich beide Algorithmen in Javascript Implementiert (Listing A.1, A.2 und A.3) und deren Laufzeit für 100 Suchen des selben Musters im selben Text mithilfe der Javascript Console API erfasst (Listing A.4) und zusätzlich die Anzahl der Schleifendurchläufe, für jeweils die erste Suche, gezählt. Da die Ausführungszeiten des Javascriptcodes stark von verschiedenen externen Faktoren abhängt schwanken die angegebenen Laufzeiten, trotz der 100 Wiederholungen, immer noch je nach Ausführung, sie sollten aber dennoch bei groß genügenden Unterschieden, die in den meisten Fällen vorhanden sind, erlauben Rückschlüsse über die Laufzeit von KMP und der naiven Suche für die verschiedenen Textarten zu ziehen.

Text	Muster	KMP	Naive Suche
DNA (Permutation von *atcg*)	aaaaaataaaataacaaaagccagaa	300.368	1.558.057
Die Bibel	Psalm 8:53	3.638.328	3.742.657
100.000 a's und ein b	1.000 a's und ein b	99,000	99.100.001
Zahlen von 0 bis 999	879, 880	879	881

Tabelle 6.1: Anzahl an Schleifeniterationen bei der Ausführung von KMP und der naiven Suche

Text	Muster	KMP	Naive Suche
DNA (Permutation von *atcg*)	aaaaaataaaataacaaaagccagaa	0,745s	1,309s
Die Bibel	Psalm 8:53	3,239s	2,227s
100.000 a's und ein b	1.000 a's und ein b	0,182s	120,894s
Zahlen von 0 bis 999	879, 880	0,013s	0,011s

Tabelle 6.2: Dauer von 100 Ausführung von KMP und der naiven Suche

Bei der Auswertung der Tabellen 6.1 und 6.2 zeigt sich, dass KMP besonders gut funktioniert wenn es viele Wiederholungen im Muster gibt, wie bei der DNA und den vielen a's. Wohingegen KMP bei "normalen"Texten, wie der Bibel, zwar weniger Vergleiche als die naive Suche benötigt allerdings dennoch merklich langsamer ist. Das Beispiel mit den vielen a's Zeigt außerdem besonders gut wie KMP funktioniert, da dieser hier nur jedes 1000te Zeichen überprüfen braucht da von allen vorherigen bereits bekannt ist, dass diese a's sind, wodurch die Laufzeit hier deutlich schneller ist als bei der naiven Suche.

7 Fazit

Der KMP Algorithmus ist ein sehr schneller Algorithmus der in der Praxis fast nie schneller ist als Boyer-Moore oder spezielleren Algorithmen, die zusätzlich Informationen über die zu durchsuchenden Daten nutzen. Entsprechend ist auch KMP meist nur in speziellen Fällen empfehlenswert, in denen zum Beispiel nach längsten Präfixen des Musters gesucht wird.

Aufgrund der nur vom Muster abhängigen Vorberechnung kann der Algorithmus außerdem auch bei Suchen nach bekannten Mustern in vielen verschiedenen, sogar noch nicht bekannten, Texten sinnvoll sein, da die *next*-Tabelle in einem solchen Fall nur einmal berechnet werden muss und dann zusammen mit dem Muster gespeichert werden kann, sodass alle zukünftigen Suchen nach diesem Muster in **O**(#Anzahl Zeichen im Text) geschehen und überhaupt nicht mehr vom Muster abhängen. Dies eigenet sich zum Beispiel für Blacklisten zur Schimpfwort Erkennung, da hierbei viele unterschiedliche Texte nach immer wieder den selben Worten durchsucht wird und somit die Vorberechnung vernachlässigbar ist.

A Anhang

```
function naive (text, pattern, logIterations) {
  let j = 0
  let k = 0
  let iterations = 0

  const m = pattern.length
  const n = text.length

  while (k < n) {
    if (text[k + j] !== pattern[j]) {
      j = 0
      k++
    } else {
      j++
    }

    iterations++

    if (j === m) {
      if (logIterations) {
        console.log(`Iterations : ${iterations.toLocaleString()}`)
      }
      return k
    }
  }
}
```

Listing A.1: Javascript Implementation der naiven Suche

```
function kmp (text, pattern, logIterations) {
  const next = kmpPreCompute(pattern)

  let j = 0
  let k = 0
  let iterations = 0

  const m = pattern.length
  const n = text.length

  while (k < n) {
    while (j >= 0 && text[k] !== pattern[j]) {
      iterations++
      j = next[j]
    }

    k++
    j++

    if (j === m) {
      if (logIterations) {
        console.log(`Iterations : ${new Number(iterations).toLocaleString()}`)
      }
      return k - m
    }
  }
}
```

Listing A.2: Javascript Implementation der KMP Suche

```
function kmpPreCompute (pattern) {
  let pos = 0
  let cnd = -1
  let next = [-1]

  const m = pattern.length

  while (pos <= m) {
    while (cnd >= 0 && pattern[pos] !== pattern[cnd]) {
      cnd = next[cnd]
    }

    cnd++
    pos++

    if (pattern[pos] === pattern[cnd]) {
      next[pos] = next[cnd]
    } else {
      next[pos] = cnd
    }
  }

  return next
}
```

Listing A.3: Javascript Implementation der Vorberechnung von KMP

```
const fs = require('fs')

let text = fs.readFileSync('./dna.txt').toLocaleString()
let search = 'aaaaaataaaataacaaaagccagaa'
let tries = 100

if (process.argv[2] === 'kmp') {
  console.time('Time')

  kmp(text, search, true)

  for (let i = 1; i <= tries; i++) {
    kmp(text, search, false)
  }

  console.timeEnd('Time')
} else {
  console.time('Time')

  naive(text, search, true)

  for (let i = 1; i <= tries; i++) {
    naive(text, search, false)
  }

  console.timeEnd('Time')
}
```

Listing A.4: Zeitmessung in Javascript von KMP und der naiven Suche

Literaturverzeichnis

[1] Brandon Lokesak. A comparison between signature based and anomaly based intrusion detection systems. `www.iup.edu/WorkArea/DownloadAsset.aspx?id=81109`, Dec 2008.

[2] Nimisha Singla and Deepak Garg. String matching algorithms and their applicability in various applications. *International journal of soft computing and engineering*, 1(6):218–222, 2012.

[3] WEB of STORIES. Donald knuth - the knuth-morris-pratt algorithm. `https://www.webofstories.com/play/donald.knuth/92`.

[4] Richard Cole. Tight bounds on the complexity of the boyer-moore string matching algorithm. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '91, pages 224–233, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[5] John R. Rose. Bioinformatics algorithms and data structures chapter 2: Kmp algorithm. `https://cse.sc.edu/~rose/590I/PPT/KMP.ppt`, Jan 2003.