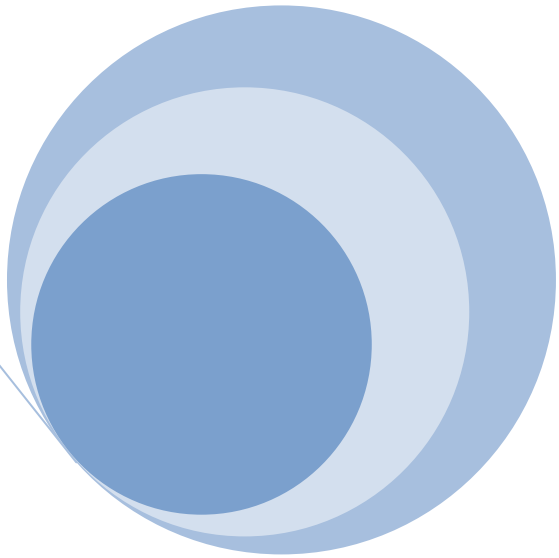
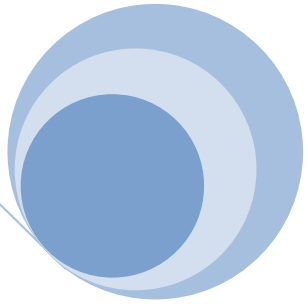
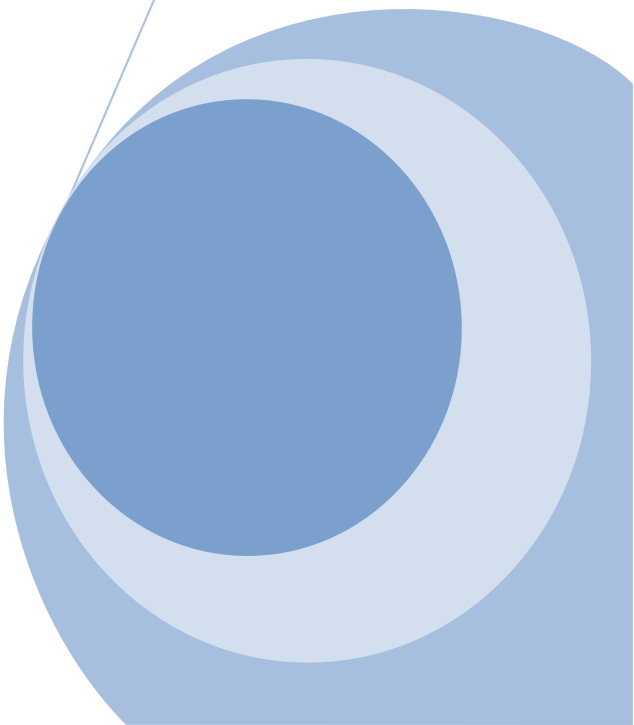






Zeit vs. Platz I

Thema 6

Comparison Counting, Distribution Counting, Horspool, Boyer Moore

Marco Brakmann TiNF8548
12.06.2008

Inhaltsverzeichnis

Einleitung

Zeit und Platz sind zwei Wörter, die beim Entwerfen von Algorithmen häufig konkurrieren. Man stelle sich eine beliebig komplexe Gleichung in einem Programm vor, für die für sehr viele Werte die Lösung ausgerechnet werden soll. Je komplexer die Gleichung, und je mehr Lösungen ausgerechnet werden sollen, desto länger dauert dies.
Bei der Implementierung muss man sich also die Frage stellen, was man will. Eine denkbare Lösung wäre sicher, für alle oder bestimmte Werte der Funktion die Lösung schon im Vorwege zu bestimmen, diese in einer Tabelle abzuspeichern und dann zu Laufzeit nur noch dort nachzusehen. Das würde das Auswerten deutlich beschleunigen, allerdings mit dem Kompromiss, dass extra Platz für die Tabelle benötigt wird. Entscheidet man sich gegen die Tabelle und rechnet jede Lösung immer neu aus, dauert dies zwar länger, aber es wird kein extra Platz benötigt.

Ein weiteres Beispiel für den Zeit-Platz-Kompromiss bietet das Spiel TicTacToe. Wollte man dieses Spiel programmieren, wäre es möglich für jede Spielsituation eine bestmöglichen Zug in einer Tabelle abzuspeichern, den der Computer dann ausführt. Auch hier wird wieder extra Platz benötigt, um diese Daten abzuspeichern. Dafür kann der Zug des Computers aber sehr schnell ausgeführt werden. Die andere Möglichkeit wäre die Realisierung einer Spiele-KI, die aufgrund der Spielsituation einen bestmöglichen Spielzug errechnet. Hierfür wird kein extra Platz benötigt, aber die Zeit die benötigt wird, um den Spielzug zu ermitteln, ist größer.

Sortieralgorithmen

Comparison Counting

Bei Comparison Counting handelt es sich um einen Sortieralgorithmus, der die Preprocessing-Technik nutzt. Der Sortieralgorithmus baut auf einer sehr einfachen Idee auf. Zu jedem Element im unsortierten Array wird genau die Anzahl der Elemente ermittelt, die kleiner sind. So steht für jedes Element fest, an welcher Stelle es im sortierten Array stehen muss.

Beispiel:

Gegeben sei das unsortierte Array *A* mit den Elementen 66, 21, 80, 22, 72 und 48. Das Array *C* hat die gleiche Länge wie *A* und ist mit dem Wert 0 initialisiert. Im ersten Schritt vergleicht man nun die Zahl 66 mit allen folgenden Zahlen. Sollte die 66 größer sein als die Zahl, mit der verglichen wird, wird der Wert für 66 im Array *C* um eins erhöht. Andernfalls wird der Wert der anderen Zahl um eins erhöht. Die gleiche Prozedur wird mit den Zahlen 21, 80, 22 und 72 wiederholt. Dann wurde jedes Element mit jedem anderen verglichen und als Resultat erhält man eine Information für jedes Element, die angibt, wie viele kleinere Elemente existieren.

Sämtliche Schritte können in der folgenden Tabelle nachvollzogen werden.

i	**66**	**21**	**80**	**22**	**72**	**48**
	0	0	0	0	0	0
0	3	0	1	0	1	0
1		0	2	1	2	1
2			5	1	2	1
3				1	3	2
4					4	2

	3	0	5	1	4	2
	21	22	48	66	72	80

Die Implementierung dieses Algorithmus gestaltet sich sehr einfach.

```
for i := 0 to n-2 do
  for j := i+1 to n-1 do
    if A[i] < A[j] then
      inc(C[j])
    else
      inc(C[i]);
for i := 0 to n-1 do
  S[ C[i] ] := A[i];
```

Wie sich bereits aus dem Pseudocode für den Algorithmus erahnen lässt, besitzt er eine quadratische Laufzeit. Der Vergleich zwischen zwei Elementen findet innerhalb von zwei Schleifen statt. Mathematisch lässt sich dieses Verhalten wie folgt nachweisen.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} \left[n-1 - i+1 +1 \right] = \frac{n\ n-1}{2}$$

Der Algorithmus benötigt die gleiche Anzahl an Vergleichen wie Selection Sort, und zusätzlich benötigt er noch linear ansteigenden Platz. Daher kann man ihn nicht für den praktischen Gebrauch empfehlen.

Distribution Counting

Der erste Ansatz, durch Zählen ein Array zu sortieren, lässt sich weiterverfolgen. Dafür setzt man voraus, dass der kleinste Wert *l* und der größte Wert *u* im unsortierten Array bekannt sind. Im ersten Schritt wird beim Distribution Counting die Häufigkeit eines jeden vorkommenden Elementes ermittelt. Diese Information muss separat in einem Array *F*[0..*u*-*l*] abgespeichert werden und kann für den nächsten Schritt weiterverwendet werden. Da jetzt die Häufigkeiten bekannt sind, wird daraus die eigentliche, für das Sortieren wichtige, Information gebildet: der Verteil-Wert *D*[0..*u*-*l*]. Dieser errechnet sich aus der Häufigkeit für das Element *i* und dem Verteil-Wert des vorherigen Elementes. *D*[*i*] = *D*[*i*-1] + *F*[*i*].
Der Verteil-Wert gibt für jede Zahl die letzte, am weitesten rechts liegende Position im sortierten Array an.

In einem kleinen Beispiel soll das Vorgehen verdeutlicht werden. Gegeben sei das Array *A* mit den Werten 6, 5, 6, 5, 4, 3, 4 und 5. Offensichtlich ist die kleinste Zahl 3, und die größte 6.

6	5	6	5	4	3	4	5

1. Ermitteln der Häufigkeiten für jede mögliche Zahl im Bereich von *l* (hier 3) bis *u* (hier 6).

Zahlenwerte	3	4	5	6
Häufigkeit F	1	2	3	2

2. Errechnen der Verteil-Werte für jede Zahl.

Zahlenwerte	3	4	5	6
Häufigkeit F	1	2	3	2
Verteil-Wert D	1	3	6	8

3. Sortieren des Arrays. Dafür wird das unsortierte Array von rechts nach links durchlaufen. Für jede Zahl wird der Verteil-Wert nachgeguckt und als Index für diese Zahl im Ziel-Array verwendet. Gegebenenfalls muss vom Index noch 1 abgezogen werden, wenn man wie hier damit ein Array indiziert. Anschließend muss der Verteil-Wert um 1 verringert werden, damit die nächste gleiche Zahl eine Position davor eingefügt wird. Nachdem dies für jede Zahl geschehen ist, liegen die Zahlen im Zielarray in sortierter Reihenfolge vor.

	D[0..3]					**S[0..7]**							
A[7] = 5	1	3	**6**	8							5		
A[6] = 4	1	**3**	5	8				4					
A[5] = 3	**1**	2	5	8		3							
A[4] = 4	0	**2**	5	8			4						
A[3] = 5	0	1	**5**	8						5			
A[2] = 6	0	1	4	**8**									6
A[1] = 5	0	1	**4**	7					5				
A[0] = 6	0	1	3	**7**								6	
						3	4	4	5	5	5	6	6

Da das Array beim Sortiervorgang von rechts nach links durchlaufen wird, ist dieser Sortieralgorithmus stabil. Stabil bedeutet, dass gleich große Elemente im sortierten Array genau die gleiche Reihenfolge haben wie im unsortierten Array.

Unter der Voraussetzung, dass der Bereich der Zahlen bekannt ist, kann mit diesem Algorithmus in linearer Zeit sortiert werden.

String-Matching-Algorithmen

Ein String-Matching-Algorithmus befasst sich mit dem Problem, eine Zeichenkette in einer anderen, mindestens gleich langen Zeichenkette zu finden. Ein trivialer Ansatz ist sicherlich die Brute-Force Variante, bei der das Suchwort von links nach rechts mit dem Text verglichen wird und, sobald es zu einem Mismatch kommt, um eine Position nach rechts verschoben wird.

Beispiel:

```
schokolade                    0000000001
lade                          0001
 lade                          0001
  lade                          0001
   lade                          0001
    lade                          0001
     lade                          0001
      lade                          0001
       lade                          0001
```

Im linken Beispiel sind zehn Vergleiche nötig, um das Suchwort im Text zu finden. Es handelt sich dabei um den Idealfall, da jedes Mal nach dem ersten Vergleich keine Übereinstimmung festgestellt wird und so das Suchwort um eins nach rechts verschoben werden kann.
Im rechten Beispiel hingegen sind 28 Vergleiche nötig. Das liegt daran, dass es jedesmal zu drei Übereinstimmungen kommt und erst danach ein Mismatch festgestellt wird.

Die Laufzeitkomplexität des Brute-Force-Algorithmus ist im Worst-Case $O(nm)$. Bei Texten mit natürlicher Sprache wird im Durchschnitt eine Laufzeit von $O(n)$ erreicht.

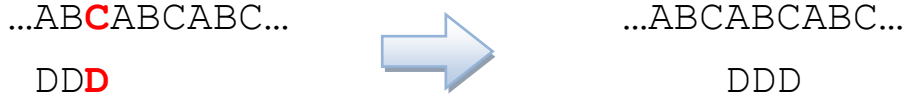
Eine lineare Laufzeit ist in der Praxis durchaus brauchbar, allerdings gibt es viele andere Algorithmen, die ein besseres Laufzeitverhalten aufweisen. Zwei von ihnen werden im Folgenden angesprochen.

Horspool Algorithmus

Der String-Matching-Algorithmus von Horspool wurde 1980 veröffentlicht. Es handelt sich dabei um eine vereinfachte Form des Boyer-Moore-Algorithmus, welcher im Anschluss beschrieben wird.

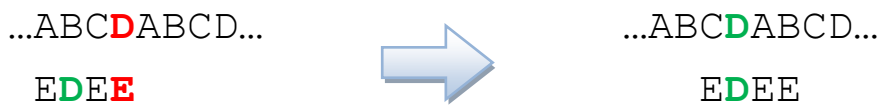
Der grundlegende Ansatz eines intelligenten String-Matching-Algorithmus besteht darin herauszufinden, wann das Suchwort um mehr als eine Position nach einem Mismatch nach rechts verschoben werden kann. Der Horspool-Algorithmus vergleicht das Suchwort mit dem Text nicht wie der Brute-Force-Algorithmus von links nach rechts, sondern von rechts nach links. Dies hat einen entscheidenden Vorteil beim Weiterschieben nach einem Mismatch.
Zuerst wollen wir analysieren, zu welchen Situationen es während des Vergleichs kommen kann.

1. Beim ersten Vergleich des Suchwortes mit dem Text kommt es zu keiner Übereinstimmung. Das Zeichen *c*, das im Text zum Mismatch führte, kommt nicht im Suchwort vor.



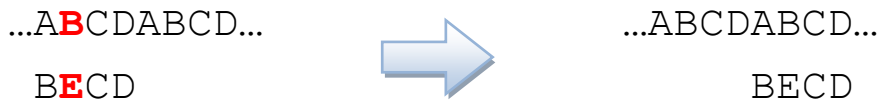
   Das Suchwort kann jetzt um die eigene Länge nach rechts verschoben werden.

2. Beim ersten Vergleich des Suchwortes mit dem Text kommt es zu keiner Übereinstimmung. Das Zeichen *c*, das im Text zum Mismatch führte, kommt an einer Stelle im Suchwort vor.



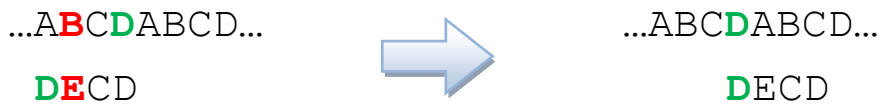
Jetzt kann das Suchwort soweit verschoben werden, dass das Zeichen *c* mit dem am weitesten rechts vorkommenden Zeichen *c* übereinstimmt.

3. Der erste Vergleich geht auf. Bei einem der darauffolgenden Vergleiche kommt es zu einem Mismatch. Das letzte Zeichen des Suchwortes kommt an keiner anderen Stelle im Suchwort nochmals vor.



Genau wie bei Fall 1 ist es nun möglich, das Suchwort um die eigene Länge nach rechts zu verschieben.

4. Der erste Vergleich geht auf. Bei einem der darauffolgenden Vergleiche kommt es zu einem Mismatch. Das letzte Zeichen *c* des Suchwortes kommt noch an einer anderen Stelle im Suchwort vor.



Genau wie bei Fall 2 kann das Suchwort nun so ausgerichtet werden, dass das am weitesten rechts stehende Zeichen *c* der ersten *m*-1 (*m* = Länge des Suchwort) Zeichen mit dem Zeichen *c* aus dem Text übereinanderliegen.

Anhand dieser Beispiele kann deutlich gezeigt werden, dass eine Überprüfung des Suchwortes von rechts nach links zu größeren Sprüngen führen kann. Aufgrund des Preprocessing sind doppelte Vergleiche unnötig, wodurch eine Menge Zeit gegenüber der Brute-Force-Variante eingespart werden kann.

Um nun die maximale Schiebedistanz für die Fälle 2 und 4 zu ermitteln, benötigt man eine Tabelle, die diese für jedes Zeichen parat hält. Dazu muss von jedem Zeichen, das in dem zum Text und Suchwort gehörendem Alphabet vorkommen kann, die Distanz des am weitesten rechts stehenden Vorkommens und dem letzten Zeichen des Suchwortes abgespeichert werden. Kommt ein Zeichen nicht im Suchwort vor, so wird für dieses Zeichen die maximale Schiebedistanz, nämlich die Länge des Suchwortes, eingetragen.

Beispiel

Suchwort: `abyxazbg`

a	b	…	g	…	x	y	z	Alle weiteren Zeichen
3	1	8	8	8	4	5	2	8

Bad-Character Tabelle

Die Implementierung für die Erstellung der Bad-Character Tabelle sieht wie folgt aus. Zuerst wird jedem Zeichen die maximale Schiebedistanz zugewiesen. Danach wird das Suchwort von links nach rechts durchlaufen und jeweils die Distanz zum letzten Zeichen ausgerechnet. Dadurch ist sichergestellt, dass bei Zeichen, die mehrmals vorkommen, nur die Distanz des am weitesten rechts Stehenden in die Tabelle übernommen wird.

//Input: Suchwort der Länge *m* P[1..m]
//Output: Bad-Character Tabelle A[0..255]

```
for i := 0 to 255 do
  A[i] := m;
for i := 1 to m - 1 do
  A[ord(P[i])] := m - i;
```

Zusammenfassung des Algorithmus

1. Schritt: Erstellen der Bad-Character Tabelle für das Suchwort

2. Schritt: Das Suchwort linksbündig unter dem Text ausrichten.

3. Schritt: Das Suchwort wird jetzt solange von rechts nach links mit dem Text verglichen, bis es zu einem Mismatch kommt oder das Suchwort gefunden wurde. Sollte es zu einem Mismatch kommen, wird die Bad-Character Tabelle mit dem Zeichen, das gegen das letzte Zeichen des Suchwortes ausgerichtet ist, indiziert, und um diesen Betrag nach rechts verschoben. Schritt 3 wird solange wiederholt, bis entweder das Suchwort gefunden wurde oder das Suchwort außerhalb des Textes ausgerichtet wird. Im letzten Fall ist das Suchwort nicht im Text vorhanden.

Beispiel

Suchwort: bcaab

1. Bad-Character Tabelle erstellen

a	b	c
1	4	3

2. Text: abcabcaadadebcaab
   bcaab
   bcaab
   bcaab
   bcaab

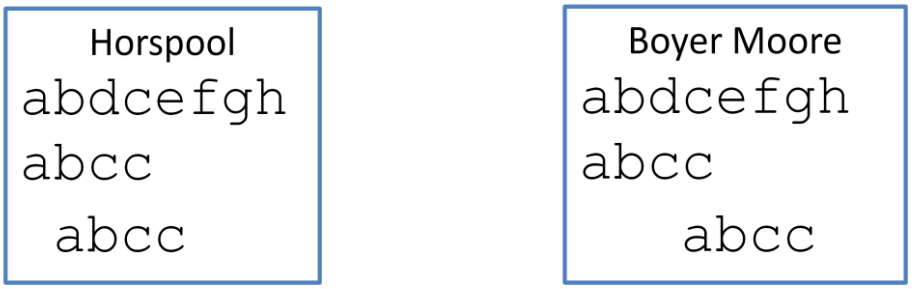
Das Laufzeitverhalten ist im Worst-Case $O(nm)$ und im Average-Case $O(n)$. Der benötigte Platzbedarf für die Tabelle ist linear, abhängig vom verwendeten Alphabet. Es fällt auf, dass der Algorithmus von Horspool das gleiche Laufzeitverhalten wie der Brute-Force Algorithmus hat. In der Praxis ist der Algorithmus von Horspool jedoch deutlich schneller. In den meisten Fällen genauso gut wie der Algorithmus von Boyer-Moore, der im Folgenden erläutert wird.

Boyer-Moore Algorithmus

Der Algorithmus von Bob Boyer und J. Strother Moore wurde 1977 veröffentlicht. Die Vorgehensweise des Algorithmus ist ähnlich der von Horspool, unterscheidet sich aber in einem wichtigen Punkt. Zusätzlich zur Bad-Character Tabelle wird noch eine Good-Suffix Tabelle erstellt. In dieser Tabelle steht die maximale Schiebedistanz, wenn nur ein bestimmter Teil des Suchwortes (Suffix) mit dem Text übereinstimmt. Dann wird in beiden Tabellen nach der Schiebedistanz geguckt und um den größeren Wert nach rechts verschoben.

Auch die Verwendung der Bad-Character Tabelle unterscheidet sich minimal. Wurde die Tabelle beim Horspool Algorithmus immer mit dem Zeichen, dass dem letzten Zeichen des Suchwortes gegenübersteht, indiziert, wird beim Boyer-Moore Algorithmus das aktuelle Zeichen im Text, das mit dem Suchwort verglichen wird, verwendet. Von diesem Wert muss dann noch die Anzahl der bisherigen Übereinstimmungen subtrahiert werden.

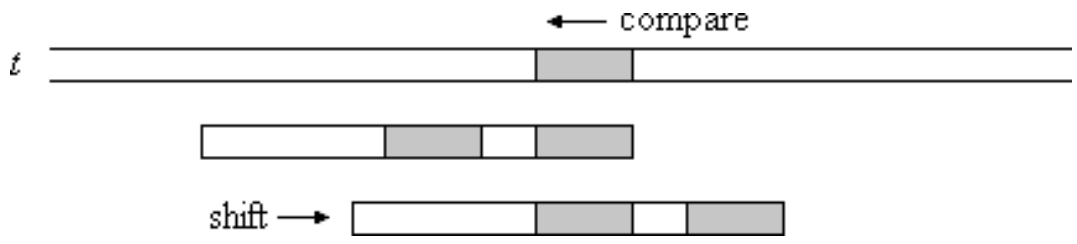
Beispiel



In dem Beispiel darf unter Verwendung des Horspool Algorithmus nur um eine Position verschoben werden, denn die Distanz von ‚c' bis zum letzten Zeichen ist eins. Der Boyer-Moore Algorithmus erlaubt eine Verschiebung um 3 Positionen nach rechts, da die Schiebedistanz für das Zeichen ‚d' vier beträgt. Da es zu einer Übereinstimmung kam, wird noch 1 abgezogen.

Für die Good-Suffix Tabelle müssen wieder zwei unterschiedliche Fälle betrachtet werden, die beim Vergleichen auftreten können.

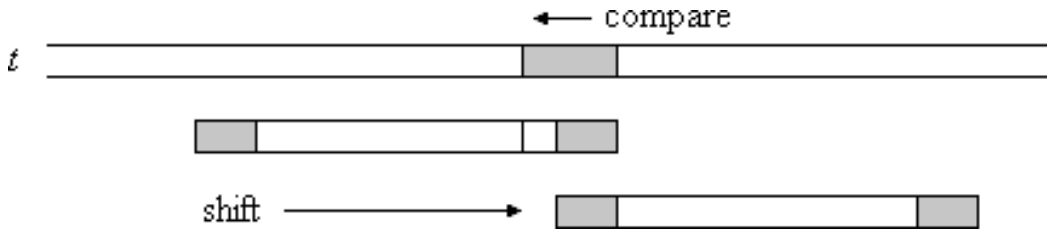
1. Fall:



Das übereinstimmende Suffix (grau) kommt noch an anderer Stelle im Muster vor

Für den Fall, dass das übereinstimmende Suffix noch an anderer Stelle im Suchwort vorkommt, und zusätzlich ein anderes Zeichen davor steht, kann das Suchwort so weit nach rechts verschoben werden, dass das zweite Vorkommen des Suffix unter dem des Textes liegt.

2. Fall:



Ein Teil des übereinstimmenden Suffixes kommt am Anfang des Musters vor

Wenn das Suffix nicht mehr an anderer Stelle im Suffix vorhanden ist, darf man nicht, wie man fälschlicherweise annehmen könnte, um die komplette Länge nach rechts verschieben. Es könnte nämlich sein, dass ein Teil des Suffixes am Anfang des Suchwortes vorkommt. Dann darf nur so weit verschoben werden, dass der Anfang des Suchwortes unter dem letzten Teil des eben verglichenen Textes steht.

In der Tabelle steht also für jedes mögliche Suffix die maximal erlaubte Schiebedistanz. Um die Tabelle zu erstellen, kann so vorgegangen werden, dass beim kleinsten Suffix angefangen wird und dann zuerst Fall 1 und dann Fall 2 geprüft werden. Anschließend betrachtet man das nächstgrößere Suffix und wiederholt die Prozedur.

Beispiel

Suchwort: `abbabab`

Suffix ‚b‘: Das nächste, am weitesten rechts stehende Vorkommen von ‚b‘ ist an Position 5. Vor diesem Zeichen steht aber ein ‚a‘, genau wie vor dem Suffix. Es würde also keinen Sinn machen, nur um 2 Positionen nach rechts zu verschieben, denn dann würde es erneut beim ‚a‘ zu einem Mismatch kommen. Es wird also das nächste Vorkommen von ‚b‘ gesucht. Dies befindet sich an Position 3. Davor steht kein ‚a‘. Darum kann als Schiebedistanz für das Suffix ‚b‘ der Wert 4 eingetragen werden.

Suffix ‚ab‘: Der Substring ‚ab‘ existiert nochmal an Position 4. Aber auch hier befindet sich davor das gleiche Zeichen wie vor dem Suffix. Das nächste Vorkommen ist an Position 1. Als Schiebedistanz wird 5 eingetragen.

Suffix ‚bab‘: Die Zeichenkette ‚bab‘ tritt nur ein weiteres Mal auf und hat davor ein anderes Zeichen als das Suffix. Als Schiebedistanz kann der Wert 2 eingetragen werden.

Suffix ‚abab‘: Das Suffix ‚abab‘ existiert an keiner weiteren Stelle im Suchwort. Hier tritt zum ersten Mal der zweite Fall ein. Ein Teil des Suffixes, nämlich ‚ab‘, kommt am Anfang des Suchwortes vor. Als Schiebedistanz kann deshalb nur 5 eingetragen werden.

Die komplette Tabelle sieht so aus:

a	b	b	a	b	a	b
-	5	5	5	2	5	4

Zusammenfassung des Algorithmus:

1. Schritt: Erstellen der Bad-Character Tabelle für das Suchwort.

2. Schritt: Erstellen der Good-Suffix Tabelle für das Suchwort.

3. Schritt: Das Suchwort linksbündig unter den Text ausrichten.

4. Schritt: Das Suchwort wird von rechts nach links mit dem Text verglichen. Kommt es beim ersten Vergleich zu einem Mismatch, wird die Schiebedistanz aus der Bad-Character Tabelle ermittelt. Wenn allerdings das erste Zeichen übereinstimmt, und es erst danach zu einem Mismatch kommt, dann wird jeweils die Schiebedistanz für das übereinstimmende Suffix aus der Good-Suffix Tabelle, und für das Zeichen, das zum Mismatch führte aus der Bad-Character Tabelle, ermittelt. Das Suchwort wird um den größeren von beiden Werten verschoben. Dieser Schritt wird solange wiederholt, bis entweder das Suchwort gefunden wurde, oder das Suchwort außerhalb des Textes geschoben wird. Im letzteren Fall kommt das Suchwort nicht im Text vor.

Beispiel

Suchwort: `EXAMPLE`
Text: `HERE_IS_A_SIMPLE_EXAMPLE`

1. Bad-Character Tabelle

A	E	L	M	P	X	übrige Zeichen
4	6	1	3	2	5	7

2. Good-Suffix Tabelle

E	X	A	M	P	L	E
-	6	6	6	6	6	6

3. HERE_IS_A_SIMPLE_EXAMPLE
EXAMPLE

Da das ‚S' nicht im Suchwort vorkommt, darf um 7 Positionen nach rechts verschoben werden.

HERE_IS_A_SIMPLE_EXAMPLE
EXAMPLE

‚P' kommt noch an anderer Stelle im Suchwort vor. Aus der Bad-Character Tabelle wird die Schiebedistanz von 2 für ‚P' ermittelt.

HERE_IS_A_SIMPLE_EXAMPLE
EXAMPLE

In dieser Situation können die Schiebedistanzen aus beiden Tabellen ermittelt werden. Das Zeichen ‚i' erlaubt eine Verschiebung um sieben. Da es bereits vier Übereinstimmungen gab, darf nur um die Differenz verschoben werden, in diesem Fall drei.
Für das Suffix ‚MPLE' wird eine Schiebedistanz von sechs aus der Good-Suffix Tabelle ermittelt. Da diese größer ist, wird das Suchwort um sechs Positionen nach rechts verschoben.

HERE_IS_A_SIMPLE_EXAMPLE
EXAMPLE

Erneut kann um zwei Positionen nach rechts verschoben werden.

HERE_IS_A_SIMPLE_EXAMPLE
EXAMPLE

Das Laufzeitverhalten für den Boyer-Moore Algorithmus ist im schlechtesten Fall $O(n)$. Im Durchschnitt wird eine Laufzeit von $O(n/m)$ erreicht.