6.6 Pattern Matching und Editierdistanz

Für viele Anwendungen gilt:

- nicht die Suche nach einem vollständigen spezifizierten Muster
- sondern die Suche nach einem nur *teilweise bekannten Muster*

ist relevant. Die Suche nach einer wohldefinierten, teilweisen Übereinstimmung nennt man **Pattern Matching**.

Wir brauchen zunächst eine **Methode zur Musterbeschreibung**, durch eine in Syntax und Semantik wohldefinierte Sprache.

Musterbeschreibung

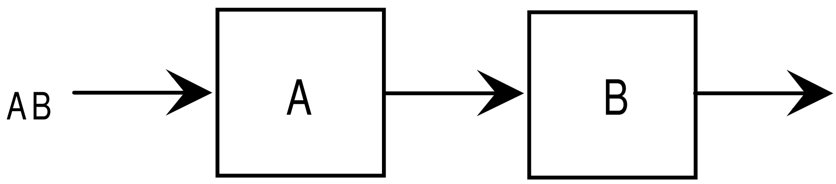
ElementareSymbole

Zeichen aus einem endlichen Alphabet Σ

Auf diesen Symbolen definieren wir **drei Operationen:**

1. **Verkettung**

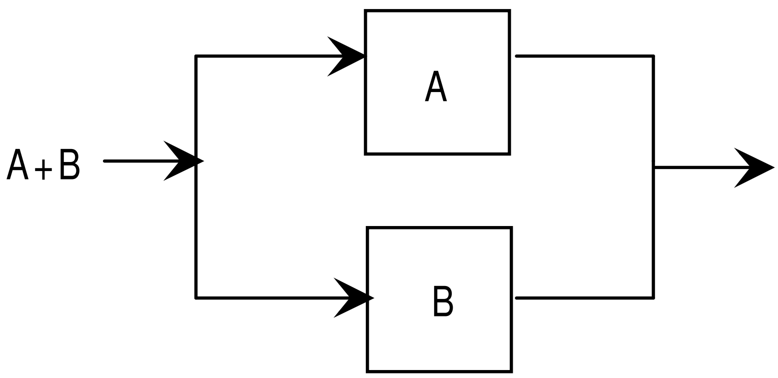
   (Konkatenation, Produkt) zweier Symbole A, B zu einem neuen Symbol



$A \mathrel{\hat{=}} abc,\ B \mathrel{\hat{=}} aa,\ AB \mathrel{\hat{=}} abcaa$

2. **Oder** (+, OR)

   beschreibt die Möglichkeit, Alternativen im Muster vorzugeben:



$(AC+B)\ D \mathrel{\hat{=}} ACD$ oder BD

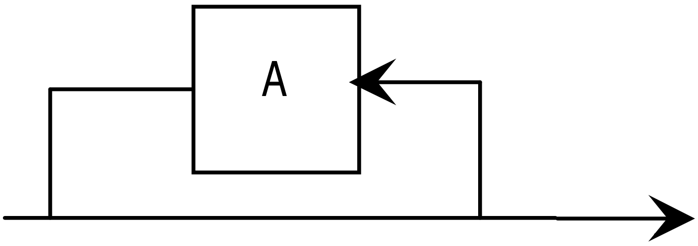
$(A+C)((B+C)D) \mathrel{\hat{=}} (A+C)(BD+CD)$

$\mathrel{\hat{=}} ABD$ oder CBD oder ACD oder CCD

3. **Hüllenbildung** (*, Closure)

zur Wiederholung von Musterteilen, 0-mal, 1-mal, n-mal

A*



$A^* \,\hat{=}\, \{(), A, AA, AAA, ...\}$
$AB^* \,\hat{=}\, \{(A), AB, ABB, ABBB, ...\}$
$(AB)^* \,\hat{=}\, \{(), AB; ABAB, ABABAB, ..\}$

Eine Folge von Symbolen, die mit Hilfe der drei obigen Operationen über dem ausgewählten Alphabet aufgebaut wurde, heißt **ein regulärer Ausdruck** bzw. **ein reguläres Muster.**

Aufgabe des PATTERN MATCHING:

Man finde einen Algorithmus, der eine Entscheidung darüber trifft, ob eine gegebene Zeichenfolge als reguläres Muster akzeptabel ist.

Bei der Realisierung eines solchen Algorithmus entsteht eine
kritische Situation:

Die Oder-Verknüpfungen bedingen *Verzweigungen*. An diesen müssen *eindeutige Entscheidungen* getroffen werden. Dies geht nur dann ohne Vorausschau, wenn alle *Pfade mit verschiedenen Zeichen* beginnen.

Auflösung:
Durch Transformation der regulären Muster in eine deterministische Form.

Regeln:

1. $aA+aB = a(A+B)$, $a \in$ Alphabet
2. $(aA)^*aB = a(Aa)^*B$
3. $(aA)^*(aB+C) = (aA)^*aB+(aA)^*C$
   $= a(Aa)^*B+(aA)^*C$

Beispiel zur Anwendung

A = abc

B = abd

C = efgf

Muster M = (A+B)C*ef

Einsetzen der Definitionen von A, B, C:

M = (abc+abd)(efgf)*ef

Regel1

M = ab(c+d)(efgf)*ef

Regel2

M = ab(c+d)ef(gfef)* (c+d):verschiedene Zeichen

(gfef)*: Hüllenbildung am

Ende

⇨ deterministische Form

Weitere, vereinfachte Schreibweisen

In der Praxis werden aus Gründen der Zweckmäßigkeit gewöhnlich noch weitere Operatoren verwendet. Zum Beispiel könnte -A “Übereinstimmung mit jedem Zeichen *außer* A“ bedeuten. Der Operator *nicht* (not) entspricht einem *oder*, das alle Zeichen außer A umfaßt, läßt sich jedoch viel leichter handhaben.

Analog könnte „?“ „Übereinstimmung mit einem beliebigen Zeichen“ bedeuten. Auch dies ist offensichtlich wesentlich kompakter als ein umfangreiches *oder*.

Beispiel 1

?*(ie+ei)?*	:	alle Wörter, in denen ie oder ei vorkommt
(1+01)*(0+1)	:	alle binären Ziffernfolgen, in denen keine zwei aufeinanderfolgenden Nullen auftreten

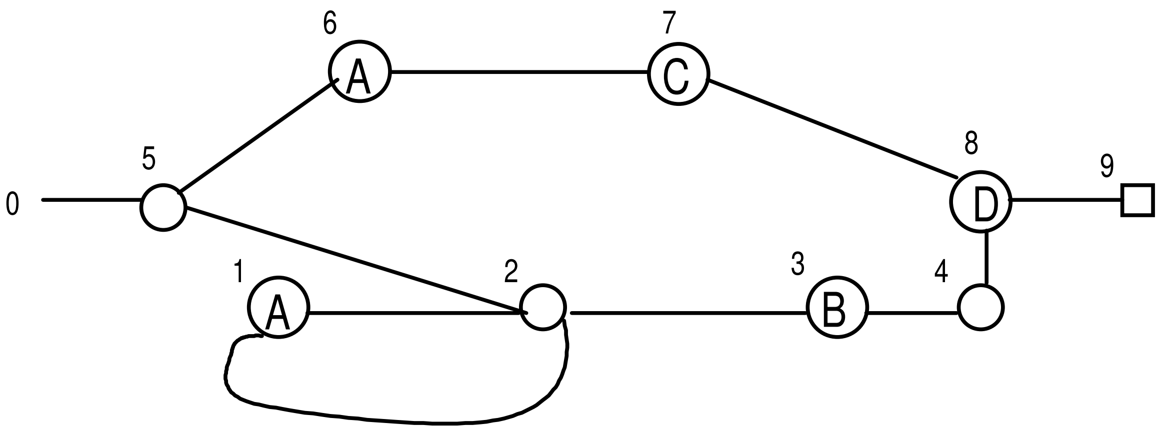
Erkennung regulärer Muster durch endliche Automaten

Endliche Automaten eignen sich hervorragend als **Akzeptoren** für reguläre Muster. Wegen der OR-Verzweigung müssen sie allerdings nichtdeterministisch sein.

Nichtdeterministischer Akzeptor, Beispiel

Musterbeschreibung: (A*B+AC)D

Im Unterschied zum deterministischen Automaten gibt es **Nullzustände**, die nichtdeterministisch zu einem Nachfolgezustand führen, unabhängig von der Eingabe. Der Automat „errät“, wie er weiter vorgehen muß.



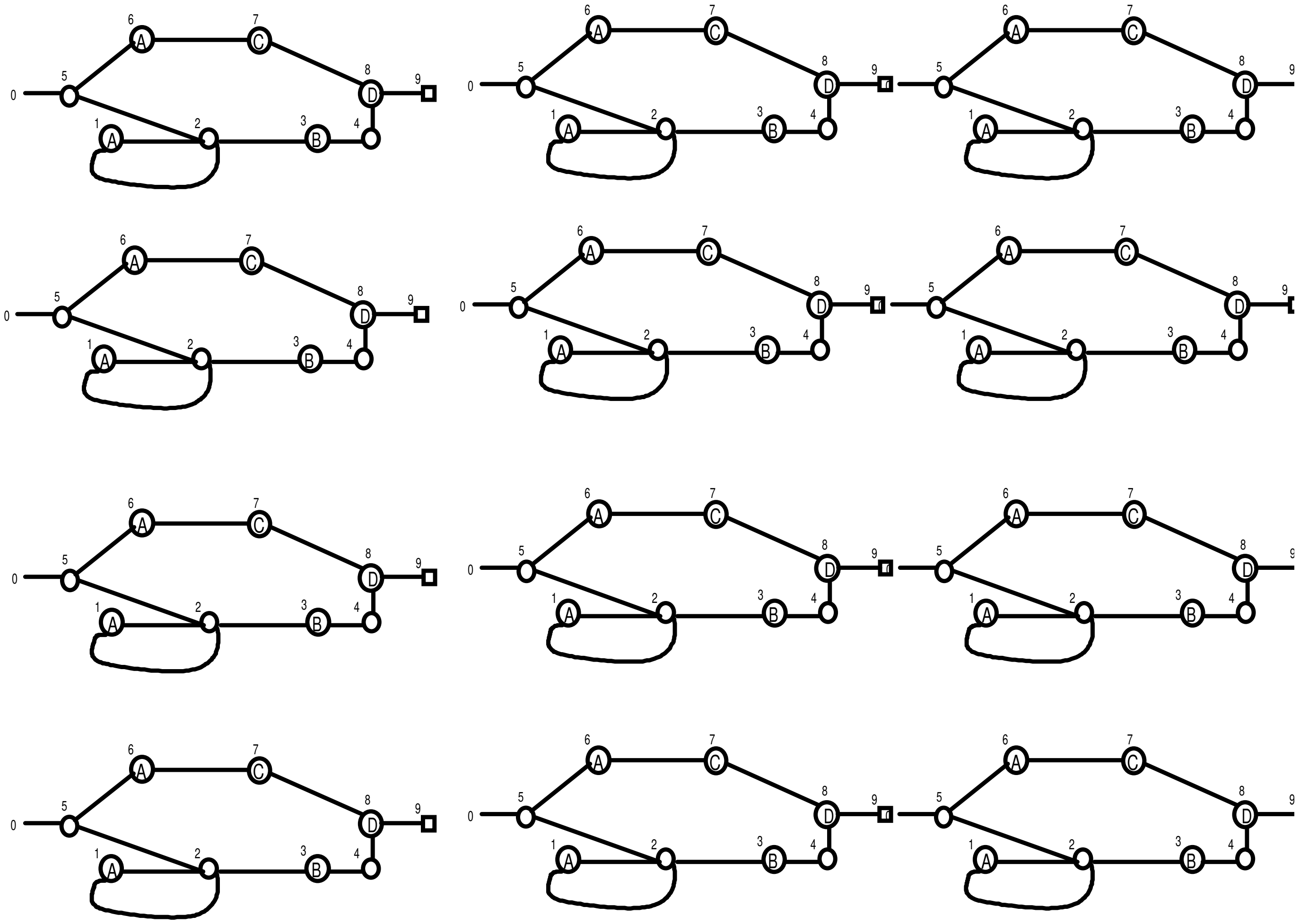
Akzeptor-Beispiel in Aktion

Text: CDAABCAAABDDACDAAC

Muster: (A*B+AC)D

Start mit Zeichen 1:	Mismatch (sofort)
Start mit Zeichen 2:	Mismatch (sofort)
Start mit Zeichen 3:	Mismatch (nach etwas Arbeit)
Start mit Zeichen 4:	Mismatch (nach etwas Arbeit)
Start mit Zeichen 5:	Mismatch (sofort)
Start mit Zeichen 6:	Mismatch (sofort)
Start mit Zeichen 7:	Match mit „erratenen“ Zustandsfolgen

Arbeitsweise des Akzeptors ab Zeichen 7



Automatische Erzeugung eines Programms

Erste Version: deterministisch

Halten wir das Muster fest, etwa

p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]	p[7]
1	0	1	0	0	1	1	1

dann können wir mit Hilfe der failure-Tabelle folgendes Programm erzeugen:

```
int kmpsearch(char *a)
  {
    int i = -1;
sm: i++;
s0: if (a [i] != '1') goto sm; i++;
s1: if (a [i] != '0') goto s0; i++;
s2: if (a [i] != '1') goto s0; i++;
s3: if (a [i] != '0') goto s1; i++;
s4: if (a [i] != '0') goto s2; i++;
s5: if (a [i] != '1') goto s0; i++;
s6: if (a [i] != '1') goto s1; i++;
s7: if (a [i] != '1') goto s1; i++;
    return i-8;
  }
```

wobei die goto-Marken der Tabelle failure+1 entsprechen.

Der Vorteil diese Programmes liegt in seinen wenigen, **elementaren** Operationen. Es ist leicht automatisch generierbar. Die **Äquivalenz zum einem endlichen Automaten ist offensichtlich**.

Regeln für die Übergänge

- aktueller Zustand durch die Marke *si* markiert
- *Gleichheit* bedeutet der Automat nimmt den Übereinstimmungsübergang(wg), im anderen Fall wird der Nichtübereinstimmungsübergang genommen
- im Zustand *sm* wird immer „Übereinstimmung“ gewählt => i++, d.h. Textdurchlauf
- Endzustand: Musterübereinstimmung

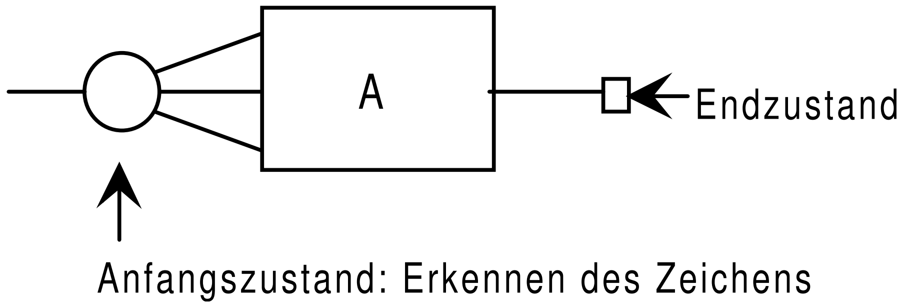
Insgesamt:

Jeder Übergang wird vollständig durch das nächste eingegebene Zeichen festgelegt

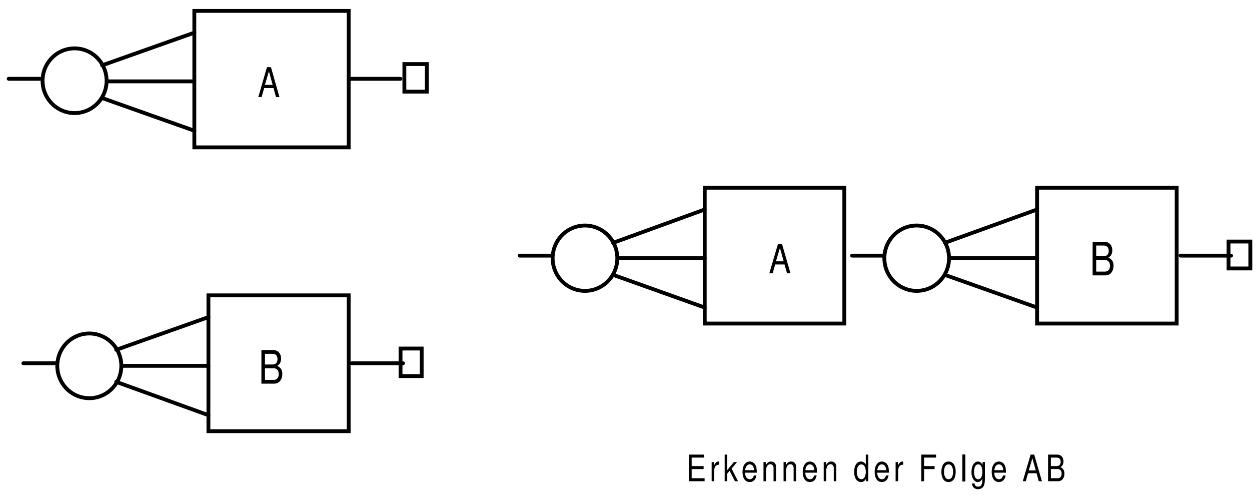
Automatische Konstruktion eines endlichen Automaten

Aus der Spezifikation des Muster läßt sich ein Akzeptor **automatisch** generieren!

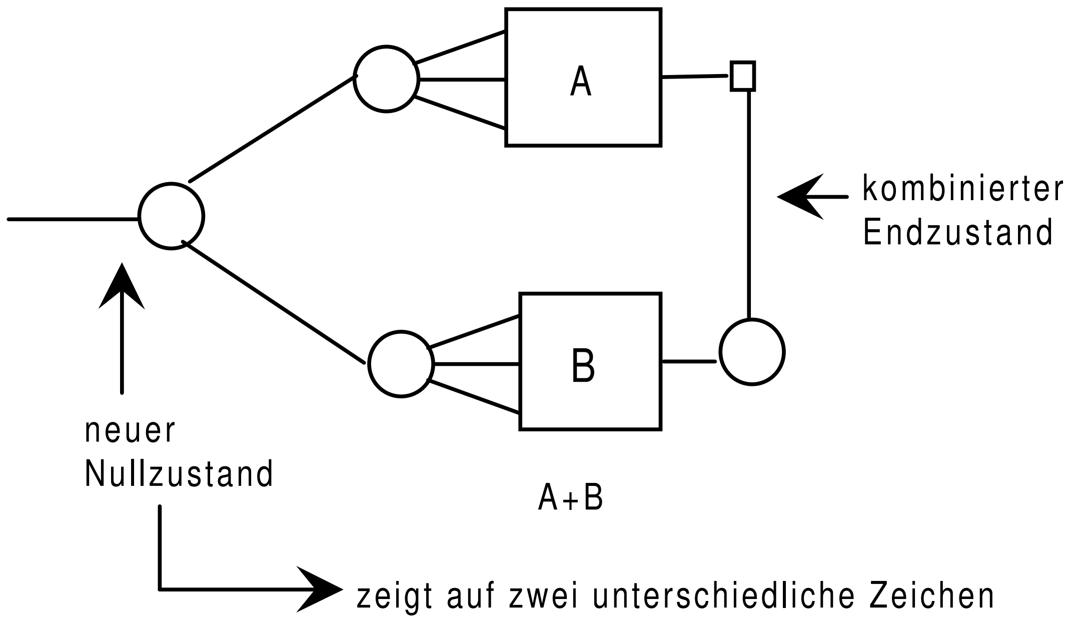
1. Erkennen eines Zeichens A



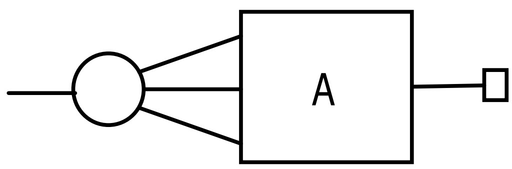
2. Verkettung

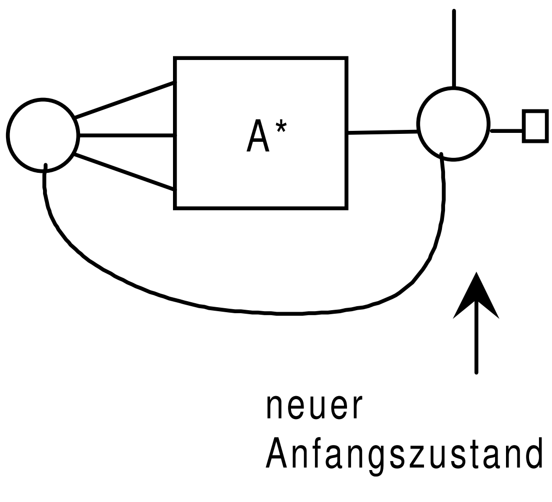


3. Oder-Verknüpfung



4. Hüllenbildung

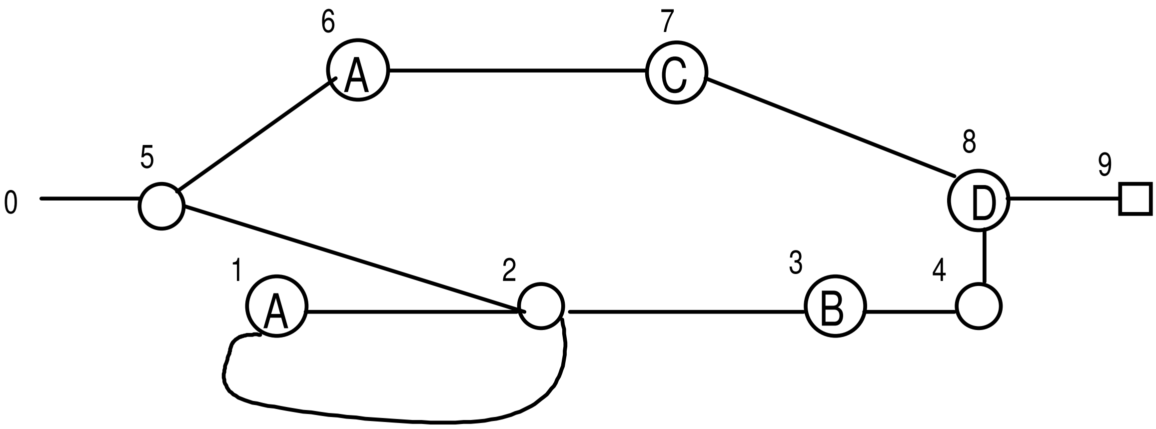




Automatische Konstruktion des Akzeptors, Beispiel

Muster: (A*B+AC)D

Konstruktion von links nach rechts



Die Ziffern geben die Reihenfolge der Konstruktion an.

Zusammenfassung der Vorgehensweise

beliebiges reguläres Muster

$\downarrow$

Anwendung der Regeln (1)-(4)

$\downarrow$

Konstruktion eines dem Muster äquivalenten Automaten

$\downarrow$

Automatische Generierung eines Programms zur Simulation das Automaten und damit zur Mustererkennung

Zustands-/Übergangstabelle

Der endliche Automat läßt sich auch in Form einer Zustands-/Übergangstabelle beschreiben.

Eingabe-zeichen	Zustand									
	0	1	2	3	4	5	6	7	8	9
A			1			2/6				
B			3							
C							7			
D					8			8		
(spontan)	5	2		4					9	

Der Eintrag in der Tabelle gibt den Folgezustand an.

Die Tabelle beschreibt die ***Dynamik*** unseres Beispielautomaten.

Die Darstellungen als Diagramm und als Zustands-/Übergangstabelle sind äquivalent.

Vom Automaten zum Programm

Letzter Schritt ist die Programmerstellung für den Mustererkennungs-automaten mit den Möglichkeiten:

a) rekursive Implementierung, die jeweils in den Nullzuständen aufsetzt

b) Warteschlangenimplementierung
- vom Typ „double ended queue“ (deque)
- da nämlich die Nullzustände jeweils einen Neubeginn charakterisieren => **Stapel (stack)**
- da die anderen Zustände dadurch charakterisiert sind, daß keine neuen möglichen Zustände analysiert werden, bevor man das aktuelle Zeichen nicht beendet hat => **Warteschlange**

Kombination von Stapel und Warteschlange => **deque**

deque, Definition

Eine deque (double ended queue) ist eine verkettet Liste, auf die an beiden Enden zugegriffen werden kann:

stapel-orientiert
- push/ ablegen
- pop/ entnehmen

schlangen-orientiert
- put/ Einfügen am Anfang
- get/ Entnahme am Ende

Für das vorliegende Problem wird eine

ausgabebeschränkte deque

implementiert, da die *Elemente* der deque immer am Anfang entnommen werden (mit pop).

Grundidee für das Programm

- Die Textzeichenfolge a wird von links nach rechts linear durchlaufen: Schleife über j = 0 ..N-1
- Bei jeder Textposition wird geprüft, ob eine erlaubte (nach Musterbeschreibung) Teilzeichenfolge beginnt: while (state) –Schleife
- Wegen der möglichen Alternativen müssen in Abhängigkeit von der Ausgangsposition neue Prüfzustände kreiert werden - diese werden durch put [scan] vom Ausgangszustand unterschieden.
- Anzeige der kürzeste Teilfolge in a, die mit dem Muster übereinstimmt, sonst *return = 0*

Zu jedem Zeitpunkt speichert die deque alle Zustände, in denen sich der nichtdeterministische Automat an der Stelle i im Text befinden könnte.

scan = -1 trennt die Elemente der deque, die das aktuelle Zeichen betreffen, von denen, die das nächste Zeichen betreffen.

Die Zustands-/Übergangstabelle schreiben wir platzsparend wie folgt um:

state	0	1	2	3	4	5	6	7	8	9
ch[state]		A		B			A	C	D	
next1[state]	5	2	3	4	8	6	7	8	9	0
next2[state]	5	2	1	4	8	2	7	8	9	0

„state“ ist die Nummer des aktuellen Zustandes, ch[state] das zu durchlaufende Zeichen, next1/2 die Nummer des jeweiligen Folgezustandes.

Nichtdeterministischer Automat, Programm

```
# define scan -1
int match(char *a)
  {int n1, n2;
   int j=0, N=strlen(a), state=next1[0];
   initateque(); put(scan);
   while(state)
  {if(state==scan){j++; put[scan];}
  else if(ch[state]==a[j])put(next1[state]);
  else if(ch[state]==’’)
    {n1=next1[state]; n2=next2[state];
     push(n1); if(n1!=n2) push(n2);}
  if(deque empty() || j==N) return 0;
  state=pop();
  }
  return j;
  }
```

Nachteile diese Programms

- keine Prüfung, ob sich ein Zustand bereits in der deque befindet
  => Ansammlung von Kopien des gleichen Zustandes, die alle verarbeitet werden müssen
- Laufzeitverhalten hängt damit vom anzupassenden Muster ab und ist somit **schlechter** als O(N*M)

Pattern Matching, Zusammenfassung

- Verläßt man den sicheren Boden von festen Mustern, wird schon der theoretische Aufwand zur Entwicklung von Algorithmen erheblich aufwendiger.

- Das Laufzeitverhalten der Suchprogramme ist oft $O(M*N^2)$ bestimmt

- Das Problem des Pattern Matching ist dem des Parsing in einem Compiler eng verwandt.

Editierdistanzen

In vielen Anwendungen ist es von Interesse festzustellen, wie weit das Muster von einem Teilstring des Textes „entfernt“ ist, oder auch Teilstrings des Textes zu suchen, die eine bestimmte maximale Entfernung vom Muster haben. Dazu lassen wir neben dem **Ändern** eines Zeichens auch das **Einfügen** und **Löschen** eines Zeichens zu.

Editierdistanz, Definition

Jeder Editier-Operation $\alpha \rightarrow \beta$ werden nichtnegative Kosten $c(\alpha \rightarrow \beta)$ zugeordnet. Man interessiert sich insbesondere für den Fall, daß die Kosten jeder Editier-Operation einheitlich gleich 1 gewählt werden (Einheitskosten-Modell). Im Einheitskosten-Modell gilt also für zwei beliebige Zeichen $a, b \in \Sigma$, $a \neq b$: $c(a \rightarrow b) = c(\in \rightarrow b) = c(a \rightarrow \in) = 1$ und natürlich $c(a \rightarrow a) = 0$.

Sind nun zwei Zeichenketten $A = a_1 ... a_m$ und $B = b_1 ... b_n$ gegeben, so definieren wird als **Editierdistanz** *D(A,B)* die minimalen Kosten, die eine Folge von Editier-Operationen hat, die A in B überführt.

Editierdistanz, Beispiel

Beispiel für eine Folge von Editier-Operationen, die `auto` in `rad` überführt:

`auto`	Operation *Lösche* (u$\rightarrow\in$) an Position 2 liefert
`ato`	Operation *Ändere* (t$\rightarrow$d) an Position 2 liefert
`ado`	Operation *Lösche* (o$\rightarrow\in$) an Position 3 liefert
`ad`	Operation *Einfüge* ($\in\rightarrow$r) an Position 0 liefert
`rad`	

Im Einheitskosten-Modell hat diese Folge von Editier-Operationen die Kosten 4. Es ist nicht schwer zu sehen, daß es keine Folge von Editier-Operationen mit geringeren Kosten gibt, die `auto` in `rad` überführt. Daher ist D(`auto`,`rad`)=4.

Editierdistanz, Anwendungen

Zwei Probleme sind im Zusammenhang mit Editierdistanzen von besonderem Interesse:

Problem 1: (Berechnung der Editierdistanz)

Berechne für zwei gegebene Zeichenketten A und B möglichst effizient die Editierdistanz D(A,B) und eine kostenminimale Folge von Editier-Operationen, die A in B überführt.

Problem 2: (Approximative Zeichenkettensuche)

Gegeben seien ein Text A und ein Muster B sowie eine Zahl $k \geq 0$. Gesucht sind alle Vorkommen von Zeichenreihen B′ in A, so daß $D(B, B') \leq k$ ist.

Für $k = 0$ ist Problem 2 natürlich wieder das gewöhnliche Zeichenketten-Suchproblem. Das k-Mismatch-Problem kann als Spezialfall von Problem 2 aufgefaßt werden, wenn man nur Änderungen von Zeichen, also weder Einfügen noch Löschen zuläßt.

6.7 Datenkomprimierung

Ziel: Speicherung von Zeichenketten auf eine möglichst platzsparende Weise, ohne daß Information verlorengeht.

Anwendung:
- für Dateien auf Speichermedien
- für Dateien, die übertragen werden sollen

Die Lauflängenkodierung

Englisch: run-length encoding

Prinzip

Ersetze Wiederholungen desselben Zeichens im Text („runs“) jeweils durch einen Zähler und das Zeichen.

Beispiel

Text: AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD

Kodierung: 4A3B2A5B8C1D1A1B1C1B3A4B3C1D

Wie wir sehen, ist mit einer guten Kompressionsrate nur zu rechnen, wenn es häufig lange „runs“ gibt, zum Beispiel lange Folgen von Leerzeichen oder führenden Nullen oder auch identischen Graustufen in Bildern.

Lauflängenkodierung für Binärdateien

Bei Binärdateien sind wir sicher, daß ein Run von Einsen stets von einem Run von Nullen gefolgt ist und umgekehrt. Es genügt deshalb, nur die Länge der Runs zu speichern!

Beispiel

```
000000000000000000000000000011111111111111000000000   28 14  9
000000000000000000000000001111111111111111110000000   26 18  7
000000000000000000000001111111111111111111111110000   23 24  4
000000000000000000000011111111111111111111111111000   22 26  3
000000000000000000001111111111111111111111111111110   20 30  1
000000000000000000011111110000000000000000001111111   19  7 18  7
000000000000000000011111000000000000000000000011111   19  5 22  5
000000000000000000011100000000000000000000000000111   19  3 26  3
000000000000000000011100000000000000000000000000111   19  3 26  3
000000000000000000011100000000000000000000000000111   19  3 26  3
000000000000000000011100000000000000000000000000111   19  3 26  3
000000000000000000001111000000000000000000000001110   20  4 23  3  1
000000000000000000000011100000000000000000000111000   22  3 20  3  3
011111111111111111111111111111111111111111111111111    1 50
011111111111111111111111111111111111111111111111111    1 50
011111111111111111111111111111111111111111111111111    1 50
011111111111111111111111111111111111111111111111111    1 50
011111111111111111111111111111111111111111111111111    1 50
011000000000000000000000000000000000000000000000011    1  2 46  2
```

Kodierung mit variabler Länge

Die klassischen Zeichencodes verwenden gleich viele Bits für jedes Zeichen. Wenn nun die verschiedenen Zeichen mit unterschiedlichen Häufigkeiten vorkommen, so kann man für häufige Zeichen wenige Bits und für seltene Zeichen mehr Bits vorsehen.

Beispiel

Kode 1:	A	B	C	D	E ...
	1	2	3	4	5 ...

Kodierung von ABRACADABRA mit **konstanter** Zeichengröße (=5 Bits):

0000100010100100000100011000010010000001000101001000001

Kode 2:	A	B	R	C	D
	0	1	01	10	11

kodierter Text: 0 1 01 0 10 0 11 0 1 01 0

Kodierung ohne explizite Begrenzer

Kode 2 kann nur eindeutig dekodiert werden, wenn die Zeichenbegrenzer mitgespeichert werden. Das vergrößert die Datenmenge erheblich.

Idee

Keine Kodierung eines Zeichens darf mit dem Anfang der Kodierung eines anderen Zeichens übereinstimmen. Dann können wir auf Begrenzer verzichten.

Kode 3:	A	B	R	C	D
	11	00	011	010	10

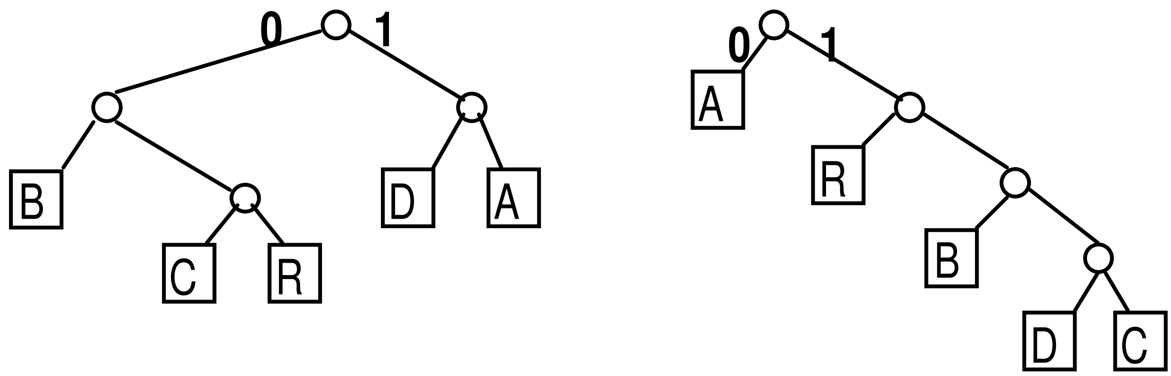
kodierter Text: 1100011110101110110001111

Darstellung als Trie

Eine einfache Methode zur Darstellung des Codes ist die Verwendung eines Trie. Tatsächlich kann jeder beliebige Trie mit *M* äußeren Knoten benutzt werden, um jede beliebige Zeichenfolge mit *M* verschiedenen Zeichen zu kodieren. Als Beispiel zeigt die folgende Abbildung zwei Codes, die für ABRACADABRA verwendet werden könnten. Der Code für jedes Zeichen wird durch den Pfad von der Wurzel zu diesem Zeichen bestimmt, mit 0 für „nach links gehen“ und 1 für „nach rechts gehen“, wie in einem Trie üblich. Demzufolge entspricht der links dargestellte Trie dem oben angegebenen Code, und der rechts dargestellte Trie entspricht einem Code, der die Zeichenfolge

01101001111011100110100

erzeugt, die um zwei Bits kürzer ist. Die Darstellung als Trie garantiert, daß kein Code für ein Zeichen mit dem Anfang eines anderen übereinstimmt, so daß sich die Zeichenfolge unter Benutzung des Trie auf eindeutige Weise dekodieren läßt.



Huffman - Code

Die Frage ist nun, wie man für eine Kodierung mit variabler Länge für gegeben Zeichenhäufigkeiten (oder Zeichenwahrscheinlichkeiten) im Text die optimale Bitkodierung findet. Der Algorithmus dazu wurde von D. Huffman (1952) angegeben.

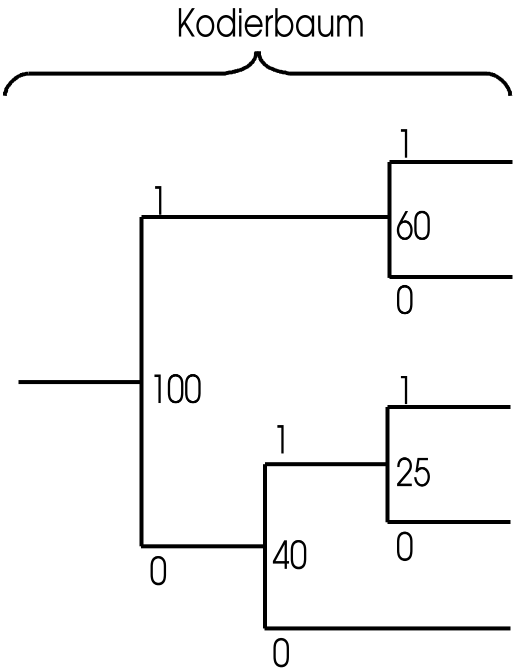
Algorithmus Erzeuge-Huffman-Code

1. Bestimme die Auftrittshäufigkeiten der Zeichen und schreibe sie an die Blattknoten eines entstehenden Binärbaums an.
2. Nimm die zwei Knoten mit den geringsten Häufigkeiten und berechne deren Summe.
3. Erzeuge einen Elternknoten für diese beiden und beschrifte ihn mit der Summe. Markiere die Verzweigung zum linken Sohn mit 0, die zum rechten Sohn mit 1.
4. Markiere die beiden bearbeiteten Knoten als erledigt. Wenn es nur noch einen nicht erledigten Knoten gibt, sind wir fertig. Sonst weiter mit Schritt 2.

Huffman - Code, Beispiel 1

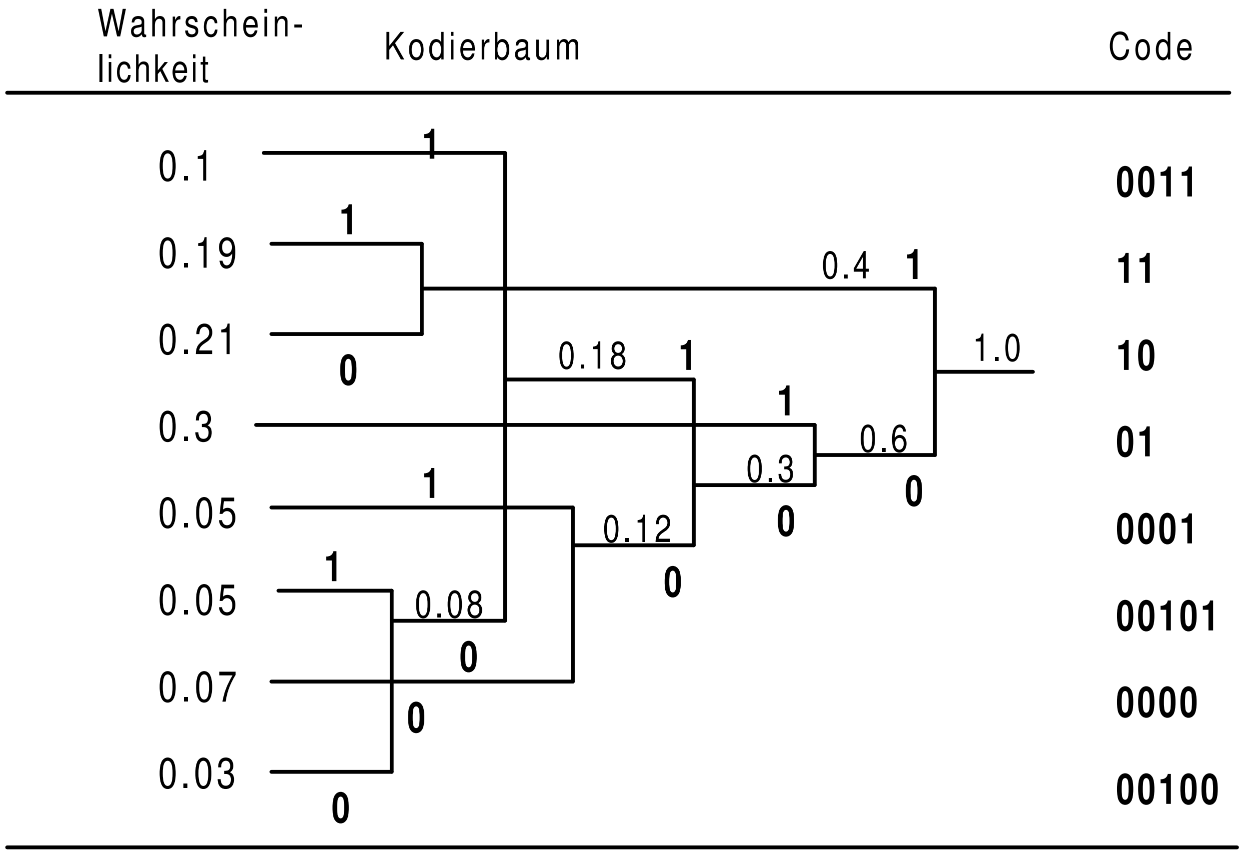
Wahrscheinlichkeiten der Zeichen:

p(A) = 0.3; p(B) = 0.3; p(C) = 0.1; p(D) = 0.15; p(E) = 0.15



Wahrschein-lichkeit	Zeichen	Code
30 %	A	1 1
30 %	B	1 0
10 %	C	0 1 1
15 %	D	0 1 0
15 %	E	0 0

Huffman - Code, Beispiel 2



Huffman - Code, Optimalität

Die Zeichen mit großen Häufigkeiten sind näher an der Wurzel des Baumes und haben somit eine kürzere Codewortlänge; deshalb ist es ein guter Code. Es ist sogar der bestmögliche Code!

Denn:

Die Länge einer kodierten Zeichenfolge ist gleich der gewichteten äußeren Pfadlänge des Huffman-Baumes.
Die „gewichtete äußere Pfadlänge" eines Baumes ist gleich der über alle äußeren Knoten gebildeten Summe der Produkte des „Gewichts" (zugehöriger Häufigkeitszähler) mit der Entfernung von der Wurzel. Dies ist offensichtlich eine Möglichkeit, die Länge der kodierten Zeichenfolge zu berechnen; sie ist äquivalent zu der über alle Buchstaben gebildeten Summe der Produkte der Häufigkeit des Auftretens eines Buchstaben mit der Anzahl der Bits bei jedem Auftreten.

Kein Baum mit den gleichen Häufigkeiten bei den äußeren Knoten hat eine kleinere gewichtete äußere Pfadlänge als der Huffman-Baum.

Beweisidee

Mit Hilfe des gleichen Prozesses kann ein beliebiger anderer Binärbaum konstruiert werden, doch ohne bei jedem Schritt unbedingt die zwei Knoten mit dem kleinsten Gewicht auszuwählen. Mittels Induktion läßt sich bewiesen, daß keine Strategie zu einem besseren Ergebnis führen kann, als die, bei der jeweils zuerst die beiden kleinsten Gewichte ausgewählt werden.

Dekodierung von Huffman-Codes

Naheliegend ist eine Dekodierung unter Verwendung des Tries:

1. Lies den Eingabestrom sequentiell und traversiere den Trie, bis ein Blattknoten erreicht ist.
2. Gib bei Erreichen des Blattknotens das erkannte Zeichen aus.

Beobachtung

Die Eingabe-Bitrate ist konstant, aber die Ausgabe-Zeichenrate ist variabel!

Dekodierung von Huffman-Codes

Als Alternative bietet sich die Verwendung einer Dekodiertabelle an.

Erzeugung der Tabelle

- Hat das längste Codewort L Bits, so hat die Tabelle 2^L Einträge.
- Sei c_i das Codewort für Zeichen s_i; c_i habe l_i Bits. Wir erzeugen 2^{L-li} Einträge in der Tabelle, bei denen jeweils die ersten l_i Bits = c_i sind und die verbleibenden $L-l_i$ Bits alle möglichen Kombinationen annehmen.
- An all diesen Adressen wird s_i als erkanntes Zeichen eingetragen, zugleich wird l_i als Codewortlänge vermerkt.

Dekodierung von Huffman-Codes

Einsatz der Tabelle zur Dekodierung

1. Lies L Bits aus dem Eingabestrom in einen Puffer.
2. Benutze den Puffer als Adresse in der Tabelle und gib das erkannte Zeichen s_i aus.
3. Entferne die ersten l_i Bits aus dem Puffer und ziehe weitere l_i Bits aus dem Eingabestrom nach.
4. Weiter mit Schritt 2.

Beobachtung

- Das Tabellenverfahren ist schnell.
- Die Ausgabe-Zeichenrate ist konstant, aber die Eingabe-Bitrate ist variabel!

Huffman - Code, Kommentare

- Ein sehr guter Kode für viele praktische Zwecke.
- Allerdings nur geeignet, wenn die Häufigkeiten der Zeichen a priori bekannt und immer gleich (oder ähnlich) sind.
- Variante: Ermittle die Häufigkeiten für jeden gegebenen Text neu und speichere/übertrage den Code mit den Daten.
- Ein (durchaus berechenbarer) Verlust entsteht dadurch, daß jedes Zeichen mit einer ganzen Zahl von Bits kodiert werden muß und somit die Codelänge den Häufigkeiten nicht ganz (theoretisch optimal) angepaßt werden kann. Verbesserung: arithmetische Kodierung.

Ziv-Lempel-Kodierung

gehört zu der großen Gruppe der auf einem Dictionary basierenden Kodierungstechniken.

Dictionary:

Sammlung von Zeichenketten, auf die bei der Kodierung Bezug genommen wird.

Beispiel:

„Vorlesung" stehe im Dictionary auf Seite x, Zeile y.
Damit kann es durch (x,y) referenziert werden.
Ein Satz wie z.B. „Heute ist Vorlesung." wird dann kodiert als Sequenz von Referenzen: (x_1,y_1) (x_2,y_2) (x_3,y_3).

Dictionary-basierte Kodierungstechniken

Statische Verfahren:

Das Dictionary wird vor der Kodierung festgelegt und nicht mehr geändert, weder beim Kodieren, noch beim Dekodieren.

Dynamische Verfahren:

Das Dictionary wird basierend auf der zu übertragenden Nachricht beim Sender bzw. Empfänger aufgebaut.

Die Kodierung nach Ziv und Lempel (1977) ist eine dynamische Dictionary-basierte Kodierungstechnik. Eine Weiterentwicklung des Algorithmus wird in vielen Compress-Befehlen (Unix, DOS) eingesetzt: Lempel-Ziv-Welch (LZW).

Das bekannte Bildkomprimierungsformat TIFF (Tag Image File Format) basiert auch auf Ziv-Lempel.

Ziv-Lempel-Kodierung, Prinzip

Idee:

Der aktuelle Teil einer Nachricht kann durch eine Referenz auf einen bereits vorgekommenen identischen Teil der Nachricht durch dynamisches Anpassen des Dictionary ersetzt werden.

LZW-Algorithmus

```
1.)   Initialize (Dictionary); /* empty */
2.)   Add (empty entry, Dictionary);
      n=0;
3.)   Add (alle basic encoding entities, Dictionary);
4.)   while (message not totally transmitted) {
        Read (encoding entity b, message);
        if (word(n)+ b ∈ Dictionary)
            n = index (word(n) + b);
        else {
            write (n);
            add ( word(n)+b, Dictionary);
            n=index(b);
        }
      }
      write(n);
```

LZW, Beispiel Kodierung

Alphabet: X = {a, b, c}

Nachricht: ababcbabab

Dictionary: Initialzustand

Index	Eintrag
0	
1	a
2	b
3	c

Kodierung: 1 2 4 3 5 8

LZW-Algorithmus: Dekodierung

```
1.)Initialize(Dictionary)
2.)Add(empty entity, Dictionary);
3.)Add(all basic encoding entities, Dictionary);
4.)Read(codeword c, message);
    w = word(c, Dictionary);
    while(message not totally decoded) {
      last_c = c;
      last_w = w;
      Read(codeword c, message);
      if(c is in Index (Dictionary)) {
        w = word(c, Dictionary);
      } else {
        w = last_w + first_entity (_w);
      }
      Write(last_w);
      Add( last_w + first_entity(w), Dictionary);
    }
    Write(w);
```

LZW, Eigenschaften

- Dictionary verwaltet die häufigsten vorkommenden Zeichenketten.
- Es wird automatisch beim Dekodieren aufgebaut und muß nicht zusätzlich übertragen werden.
- Kodierung wird in O(N), N= Länge der nichtkodierten Nachricht, Dekodierung in O(M), M= Länge der kodierten Nachricht, durchgeführt. Damit ist die Dekodierung sehr effizient. Da mehrere Symbole in eine Kodierungseinheit kombiniert werden, ist $M \leq N$ und LZW i.d.R. effizienter als Huffman.
- Bei begrenzter Dictionary-Größe wird die Kompression ineffizient, da das Dictionary dann statisch ist.

Ergebnisse im Vergleich:

(Dateigröße nach Kompression in Prozent der Ursprungsgröße)

Art	Huffmann	Ziv-Lempel
C-Dateien	65 %	45 %
Maschinenkode	80 %	55 %
Text	50 %	30 %

Arithmetische Kodierung

Idee:

Eine Nachricht wird als Real-Zahl aus dem Intervall [0,1) kodiert. Dazu wird das Intervall unterteilt nach der Wahrscheinlichkeit eines Symbols.

Arten Arithmetischer Kodierung

Statische Arithmetische Kodierung:

Die Wahrscheinlichkeit des Vorkommens eines Symbols ist fest.

Adaptive Arithmetische Kodierung:

Die Wahrscheinlichkeit des Vorkommens von Symbolen wird dynamisch bestimmt, basierend auf den wechselnden Symbolfrequenzen in den bereits dekodierten Nachrichten.

Arithmetische Codierung, Algorithmus

Kodierung:

1. Das Intervall [0,1) wird in mehrere Teilintervalle zerlegt, wobei jeweils ein Teilintervall für jedes Zeichen des Alphabets verwendet wird. Die Teilung erfolgt nach den Zeichenwahrscheinlichkeiten.
2. Das nächste zu kodierende Zeichen bestimmt das nächste Basisintervall. Dieses Intervall wird wiederum nach den Zeichenwahrscheinlichkeiten unterteilt. Dies wird solange fortgeführt, bis ein Endesymbol erreicht wird oder eine bestimmte Anzahl Zeichen kodiert ist.
3. Eine Zahl aus dem zuletzt entstandenen Intervall wird gewählt. Am Geschicktesten ist eine Zahl, die sich mit möglichst wenigen Bits darstellen läßt.

Arithmetische Codierung, Algorithmus

Dekodierung:

1. Das Intervall [0,1) wird wie beim Kodierungsprozeß unterteilt.
2. Der Code bestimmt das nächste zu unterteilende Basisintervall nach seinem Wert. Dieses Intervall steht für ein bestimmtes Zeichen, das damit dekodiert ist. Dies wird solange fortgeführt bis das Endesymbol dekodiert worden ist oder eine bestimmte vorab festgelegte Anzahl Zeichen.

Statische Arithmetische Codierung, Beispiel

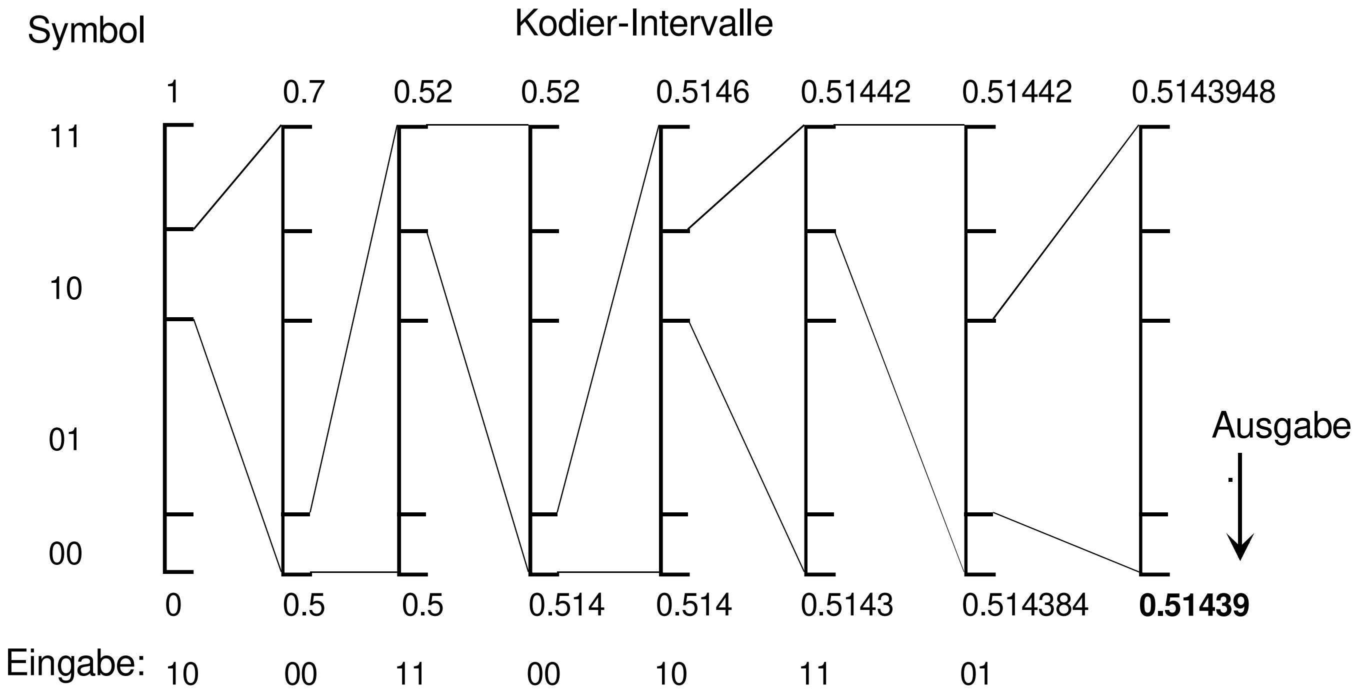
Alphabet = {00,01,10, 11}

Wahrscheinlichkeiten = {0,1;0,4;0,2;0,3}

Zeichen	00	01	10	11
Wahrscheinlichkeiten	0.1	0.4	0.2	0.3
Erste Codierungsintervalle	[0, 0.1)	[0.1, 0.5)	[0.5, 0.7)	[0.7, 1)

Nachricht:

10 00 11 00 10 11 01



Kodierung:

Schritt	Eingabe Zeichen	Codierungsintervall	Codierungsentscheidung
1	10	[0.5, 0.7)	Das Zeichenintervall ist [0.5, 0.7)
2	00	[0.5, 0.52)	Das erste Zehntel von [0.5, 0.7)
3	11	[0.514, 0.52)	Die letzten drei Zehntel von [0.5, 0.52)
4	00	[0.514, 0.5146)	Das erste Zehntel von [0.514, 0.52)
5	10	[0.5143, 0.51442)	Beginnend mit dem fünften Zehntel, zwei Zehntel von [0.514, 0.5146)
6	11	[0.514384, 0.51442)	Die letzten drei Zehntel von [0.5143, 0.51442), beginnend mit dem zweiten Zehntel
7	01	[0.51439, 0.5143948)	Vier Zehntel von [0.514384, 0.51442), beginnend mit dem zweiten Zehntel
8	Wähle eine Zahl aus dem Intervall [0.51439, 0.5143948) als Ausgabe: 0.51439		

Übertragung: 0.51439

Dekodierung:

Schritt	Intervall	dekodiertes Zeichen	Decodierungsentscheidung
1	[0.5, 0.7)	10	0.51439 befindet sich in [0.5, 0.7)
2	[0.5, 0.52)	00	0.51439 befindet sich im ersten Zehntel des Intervalls [0.5, 0.7)
3	[0.514, 0.52)	11	0.51439 befindet sich im 7. Zehntel des Intervalls [0.5, 0.52)
4	[0.514, 0.5146)	00	0.51439 befindet sich im ersten Zehntel des Intervalls [0.514, 0.52)
5	[0.5143, 0.51442)	10	0.51439 befindet sich im 5. Zehntel des Intervalls [0.514, 0.5146)
6	[0.514384, 0.51442)	11	0.51439 befindet sich im 7. Zehntel des Intervalls [0.5143, 0.51442)
7	[0.51439, 0.5143948)	01	0.51439 befindet sich im 4. Zehntel des Intervalls [0.514384, 0.51442)
8	Die decodierte Message ist 10 00 11 00 10 11 01		

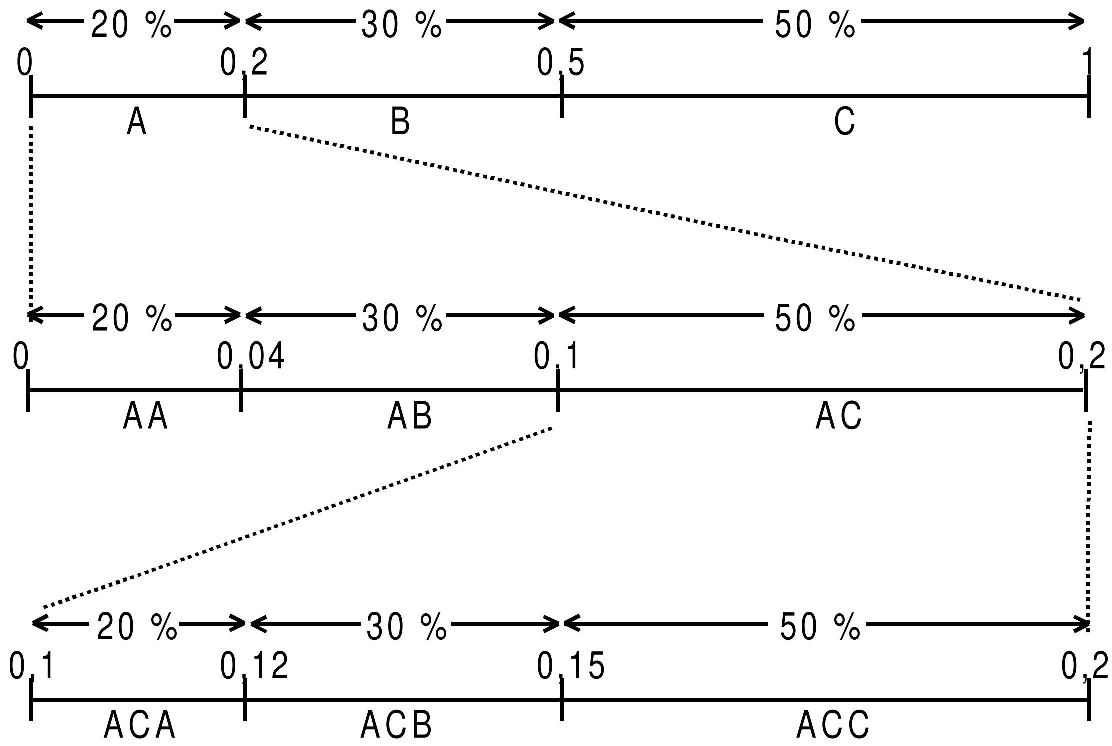
Dynamische Arithmetische Codierung, Beispiel

Alphabet = {A,B,C}

Anfangswahrscheinlichkeiten = {0,2; 0,3; 0,5}

Nachricht: ACBAAB (in 3er Blöcken)

Kodierung:

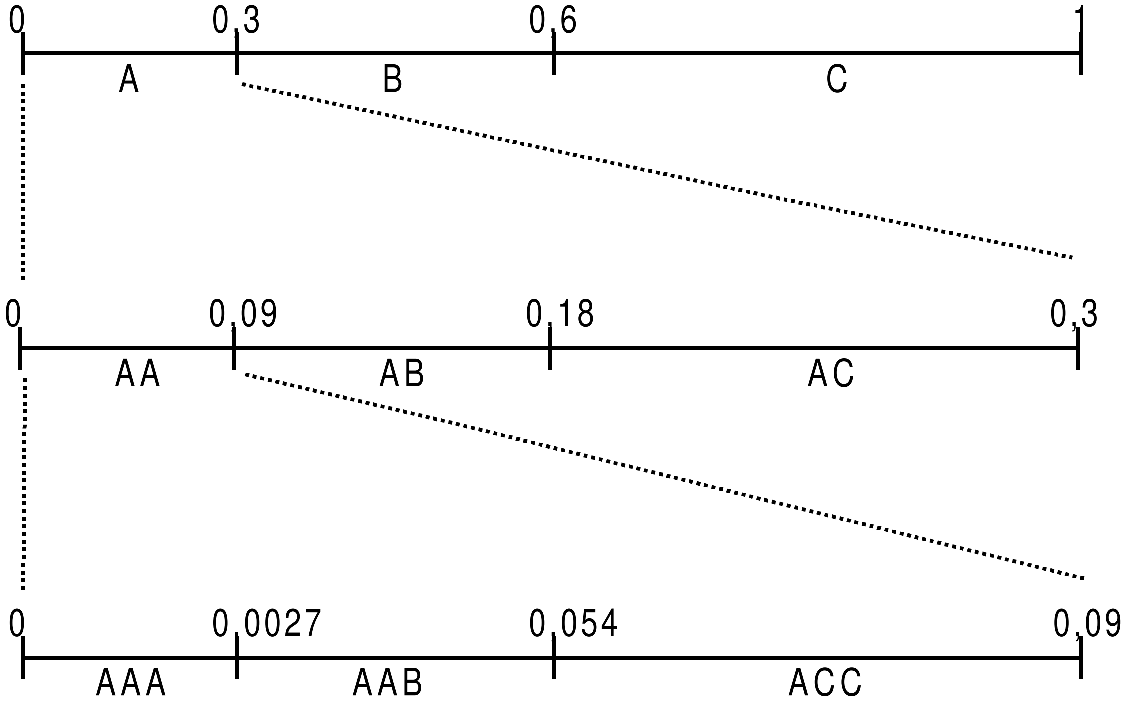


Übertragung: [0,12; 0,15) z.B. 0, 12

$$P_{neu} = \frac{P_{alt} + P_{aktuell}}{2}$$

Neue Wahrscheinlichkeiten = {0,3; 0,3; 0,4}

Kodierung:



Übertragung: [0,0027; 0,054) z.B. 0,05

Arithmetische Codierung, Eigenschaften

- Kodierung hängt von Symbolwahrscheinlichkeit und damit vom Kontext ab.

- Die Wahrscheinlichkeit der Symbole bestimmt die Kompressionseffizienz: wahrscheinlichere Symbole werden in größeren Intervallen kodiert und brauchen so weniger Präzisionsbits.

- Terminierung des Verfahrens beim Dekodieren auf mehrere Arten möglich:
  - Nachrichtenlänge ist Sender und Empfänger bekannt
  - bestimmte Anzahl Nachkommastellen (auch dies ist Sender und Empfänger bekanntzugeben)

Arithmetische Codierung, Probleme:

- Präzision von Maschinen ungenau, so daß „overflow“ oder „underflow“ vorkommen kann

- Dekodierung erst möglich bei vollständigem Erhalt der (Teil-)Nachricht, die aus vielen Nachkommastellen bestehen kann.
  Jedoch ist eine inkrementelle Dekodierung möglich, wenn die Nachkommastellen einzeln übertragen werden.

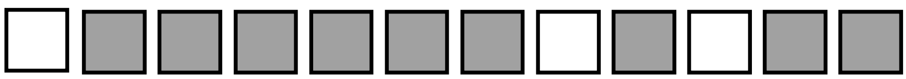
- sehr fehleranfällig: ein Bitfehler zerstört die ganze (Teil-)Nachricht.

- exakte Wahrscheinlichkeiten sind nicht erhältlich, so daß die maximale Effizienz nicht erreicht werden kann

FAX-Kompression

- Standards in der Telekommunikation werden von der Internationalen Telecommunication Union (ITU-T) entwickelt. (früher: CCITT = Commitée Consultatif International de Téléphonie et Télégraphie).
- Der Standard für verlustlose Kompression von Faksimile-Anwendungen war einer der ersten.
- Bilder werden von Faxanwendungen als **bitonal** betrachtete, d.h. jedes Pixel kann entweder schwarz oder weiß sein. So kann jedes Pixel mit 1 Bit repräsentiert werden.
- In jedem bitonalen Bild gibt es größere Regionen (Runs), die entweder vollkommen schwarz oder vollkommen weiß sind.

Beispiel:



Standards für die Fax-Kompression

1.) ITU-T Rec. T.4 (Fax Group 3):

Kodierungsansatz 1: Modified Huffman (MH) Code

- Bild wird als Sequenz von Zeilen betrachtet.
- Für jede Zeile wird eine Lauflängenbeschreibung festgehalten.
- Die (Lauflänge, Wert)-Beschreibung wird dann Huffman-kodiert.
- Dabei werden schwarze und weiße Runs mit separaten Huffman-Codes behandelt, da ihre Eigenschaften recht unterschiedlich sind.
- Die Huffman-Tabelle ist statisch für alle Bilder.
- Zur Fehlererkennung wird nach jeder Zeile ein EOL (end-of-line=) Kodewort eingefügt.

Kodierungsansatz 2: Modified Read (MR) Code

- Pixelwerte der vorangehenden Zeile werden für die Vorhersage der aktuellen Zeile verwendet.
- Anschließend wird eine Lauflängenbeschreibung und ein statischer Huffman-Code wie bei MH eingesetzt.
- Ebenso wird ein EOL Kodewort verwendet.

Die Kodierungsansätze MH und MR werden regelmäßig abgewechselt, um Folgefehler zu vermeiden.

2.) ITU-T Rec. T.6 (Fax Group 4):

Kodierungstechnik: Modified Modified Read (MMR) Code

- Vereinfachung des MR-Codes, in dem keine Fehlererkennungsmechanismen eingesetzt werden, um die Kompressionsrate zu erhöhen

Kompressionsraten:

	Geschäftsdokumente
Group 3:	20:1
Group 4:	50:1

Für Photographien u.ä. naturliche Bilder ist die Kompressionsrate schlecht, weil nur kurze Runs vorkommen. Adaptive Arithmetische Codierung wäre besser geeignet für solche Bilder.