ADS: Algorithmen und Datenstrukturen 2

Teil V

Peter F. Stadler & Konstantin Klemm

Bioinformatics Group, Dept. of Computer Science & Interdisciplinary Center for Bioinformatics, **University of Leipzig**

05. Mai 2010

Datenkomprimierung

Bei den meisten bisher betrachteten Algorithmen wurde vor allem das Ziel verfolgt, möglichst wenig Zeit aufzuwenden, und erst in zweiter Linie ging es darum, Platz zu sparen. Jetzt werden Methoden untersucht, die primär dazu bestimmt sind, den benötigten Platz zu reduzieren, ohne zu viel Zeit zu verbrauchen.

Lauflängencodierung I

Der einfachste Typ einer Redundanz in einer Datei sind lange Folgen sich wiederholender Zeichen, die man Läufe (runs) nennt. Betrachtet man z. B. die folgende Zeichenkette
AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD.
Diese Zeichenfolge kann in einer kompakten Form codiert werden, indem jede Serie sich wiederholender Zeichen durch eine einmalige Angabe des sich wiederholenden Zeichens und einer Angabe der Anzahl der Wiederholungen ersetzt wird. Wenn die Zeichenfolge nur aus Buchstaben besteht, ergibt sich folgende Darstellung:
4A3B2A5B8CDABCB3A4B3CD
Es lohnt sich nicht, Läufe der Länge eins oder zwei zu codieren, da für die Codierung zwei Zeichen benötigt werden.

Lauflängencodierung II

Für binäre Dateien wird gewöhnlich eine verfeinerte Variante dieser Methode benutzt, mit der drastische Einsparungen erzielt werden können. Die Idee besteht einfach darin, die Lauflängen zu speichern und dabei die Tatsache auszunutzen, dass sich Läufe zwischen 0 und 1 abwechseln, so dass auf das Speichern der Werte 0 und 1 selbst verzichtet werden kann. Die Lauflängencodierung erfordert unterschiedliche Darstellung für die Datei und ihre kodierte Variante, so dass sie nicht für alle Dateien möglich ist. Dies kann sehr störend sein: Beispielsweise ist die oben vorgeschlagene Methode zur Komprimierung von Zeichendaten nicht für Zeichenfolgen geeignet, die Ziffern enthalten.

Lauflängencodierung III

Wie kann man erreichen, dass einige Buchstaben Ziffern und andere Buchstaben darstellen? Eine Lösung besteht darin, ein Zeichen, das im Text wahrscheinlich nur selten erscheint, als so genanntes Escape-Zeichen zu verwenden. Jedes Auftreten dieses Zeichens besagt, dass die folgenden beiden Buchstaben ein Paar (Zähler, Zeichen) bilden, wobei Zähler dargestellt werden, indem der i-te Buchstabe des Alphabets zur Darstellung der Zahl i benutzt wird. Demzufolge könnte das angegebene Beispiel einer Zeichenfolge mit Q als Escape- Zeichen wie folgt dargestellt werden:
QDABBBAAQEBQHCDABCBAAAQDBCCCD
Die Kombination des Escape-Zeichens, des Zählers und der einen Kopie des sich wiederholenden Zeichens wird Escape-Sequenzgenannt. Hier ist erst eine Verschlüsselung ab 4 Zeichen sinnvoll.

Lauflängencodierung IV

Was ist zu tun, wenn das Escape-Zeichen selbst in den Eingabedaten auftritt? Diese Möglichkeit darf man nicht ignorieren, da es kaum zu gewährleisten ist, dass irgendein spezielles Zeichen nicht auftreten kann (z. B. wenn jemand versucht eine Zeichenfolge zu codieren, die schon codiert ist).

Eine Lösung für dieses Problem besteht in der Benutzung einer Escape-Sequenz mit Zähler Null zur Darstellung des Escape-Zeichens. Demzufolge könnte im obigen Beispiel das Leerzeichen die Null darstellen, und die Escape-Sequenz “Q<Leerzeichen>” würde jedes Auftreten von Q in den Eingabedaten bezeichnen. Es ist anzumerken, dass nur Dateien, die Q enthalten, durch dieses Komprimierungs-Verfahren verlängert werden. Wenn eine komprimierte Datei nochmals komprimiert wird, vergrößert sie sich um eine Anzahl von Zeichen, die wenigstens gleich der Anzahl der benutzten Escape-Sequenzen ist. Sehr lange Läufe können mit Hilfe mehrerer Escape-Sequenzen codiert werden. Zum Beispiel würde ein aus 51 As bestehender Lauf unter Verwendung der obigen Vereinbarung als

QZAQYA

codiert. Wenn viele sehr lange Läufe zu erwarten sind, könnte es lohnenswert sein, für die Codierung der Zähler mehr als ein Zeichen zu reservieren.

Codierung mit variabler Länge I

Hier wird ein Datenkompressions-Verfahren betrachtet, das in Textdateien (und vielen anderen Arten von Dateien) eine beträchtliche Platzeinsparung ermöglichen kann. Die Idee besteht darin, von der Methode abzuweichen, mit der Textdateien gewöhnlich gespeichert werden: Anstatt die üblichen sieben oder acht Bits für jedes Zeichen zu benutzen, werden für Zeichen, die häufig auftreten, nur wenige Bits verwendet, und mehr Bits für die, die selten vorkommen.
Beispiel-Zeichenfolge:
ABRACADABRA
Eine Codierung mittels des standardmäßigen binären Codes, bei dem die Binärdarstellung von i mit Hilfe von 5 Bits zur Darstellung des i-ten Buchstabens des Alphabets benutzt wird (0 für Leerzeichen), ergibt folgende Bitfolge:
00001000101001000001000110000100100000010001010010000001

Codierung mit variabler Länge II

Um diese Meldung zu decodieren, lese man jeweils fünf Bits und wandle diese entsprechend dem oben definierten binären Code um. In diesem Standard-Code benötigt das nur einmal vorkommende D die gleiche Anzahl Bits wie A,welches fünfmal auftritt. Bei Verwendung eines Codes mit variabler Länge könnte man eine Platzeinsparung erzielen, indem man häufig verwendete Zeichen mittels möglichst weniger Bits verschlüsselt, so dass die Gesamtzahl der für die Zeichenfolge benutzten Bits minimiert wird. Man kann versuchen, den am häufigsten verwendeten Buchstaben die kürzesten Bitfolgen zuzuweisen, indem man A mit 0 kodiert, B mit 1, R mit 01, C mit 10 und D mit 11. So wird ABRACADABRA als
0 1 01 0 10 0 11 0 1 01 0
codiert. Hier werden nur 15 Bits im Vergleich zu den oben erforderlichen 55 Bits benutzt, doch es ist kein wirklicher Code, da er von den Leerzeichen abhängt, die die Zeichen voneinander abgrenzen.

Codierung mit variabler Länge III

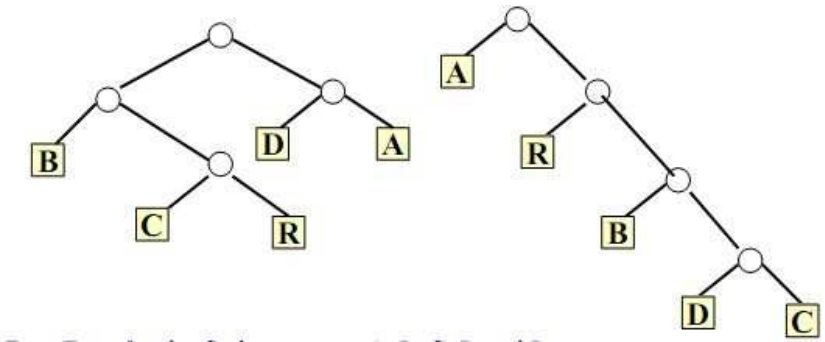
Ohne die Leerzeichen könnte die Zeichenfolge 010101001101010 als RRRARBRRA oder eine Reihe weiterer Zeichenfolgen decodiert werden. Trotzdem ist die Zahl von 15 Bits plus 10 Begrenzernviel kompakter als der Standard-Code, vor allem weil keine Bits verwendet werden um Buchstaben zu codieren, die in der Meldung nicht vorkommen. Die Bits im Code müßtenauch mitgezählt werden, außerdem hängt der Code von der Zeichenfolge ab, da andere Zeichenfolgen andere Buchstabenhäufigkeiten besitzen. Es werden keine Begrenzer benötigt, wenn kein Zeichencode mit dem Anfang eines anderen übereinstimmt.

Codierung mit variabler Länge IV

Wenn z.B. A mit 11 verschlüsselt wird, B mit 00, C mit 010, D mit 10 und R mit 011, so gibt es nur eine Möglichkeit, die aus 25 Bits bestehende Zeichenfolge 1100011110101110110001111 zu decodieren. 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 codieren. Die nachfolgende Abbildung zeigt zwei Codes, die für ABRACADABRA verwendet werden können. Der Code für jedes Zeichen wird durch den Pfad von der Wurzel zu diesem Zeichen bestimmt, mit 0 für “nach links” gehen und mit 1 für “nach rechts” gehen, wie gewöhnlich in einem Trie.

Decodierung I

Bei der Wurzel beginnend bewegt man sich entsprechend den Bits der Zeichenfolge im Trie abwärts. Jedesmal, wenn ein äußerer Knoten angetroffen wird, gebe man das zu diesem Knoten gehörige Zeichen aus und beginne erneut bei der Wurzel.



Zwei Tries für die Codierung von A, B, C, D und R.

Der links dargestellt Trie entspricht dem oben angegebenen Code, und der rechts dargestellte Trieentspricht einem Code, der aus ABRACADABRA die Zeichenfolge
01101001111011100110100
erzeugt, die um 2 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 decodieren lässt.

Erzeugung des Huffman-Codes

Das allgemeine Verfahren zur Bestimmung dieses Codes wird Huffman-Codierung genannt. Der erste Schritt bei der Erzeugung des Huffman-Codes besteht darin, durch Zählen die Häufigkeit jedes Zeichens innerhalb der zu codierenden Zeichenfolge zu ermitteln. Das folgende Programm ermittelt die Buchstaben-Häufigkeiten einer Zeichenfolge a und trägt diese in ein Feld count[26] ein. Die Funktion index dient hier dazu, dass der Häufigkeitswert für den i-ten Buchstaben des Alphabets in dem Eintrag count[i] eingetragen wird, wobei wie üblich der Index 0 für das Leerzeichen verwendet wird.

```
for ( i = 0 ; i <= 26 ; i++ ) count[i] = 0;
  for ( i = 0 ; i < M ; i++ ) count[index(a[i])]++;
```

Beispiel und Algorithmus

A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”

Die dazugehörige Häufigkeits-Tabelle

	_	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
count[k]	11	3	3	1	2	5	1	2	0	6	0	0	2	4	5	3	1	0	2	4	3	2	0	0	0	0	0

Es sind 11 Leerzeichen, drei A, drei B usw. Der nächste Schritt ist der Aufbau des Codierungs-Tries entsprechend den Häufigkeiten. Während der Erzeugung des Trie betrachtet man ihn als binären Baum mit Häufigkeiten, die in den Knoten gespeichert sind; nach seiner Erzeugung betrachtet man ihn dann als einen Trie für die Codierung in der oben beschriebenen Weise. Für jede von Null verschiedene Häufigkeit wird ein Knoten des Baumes erzeugt.

Algorithmus II

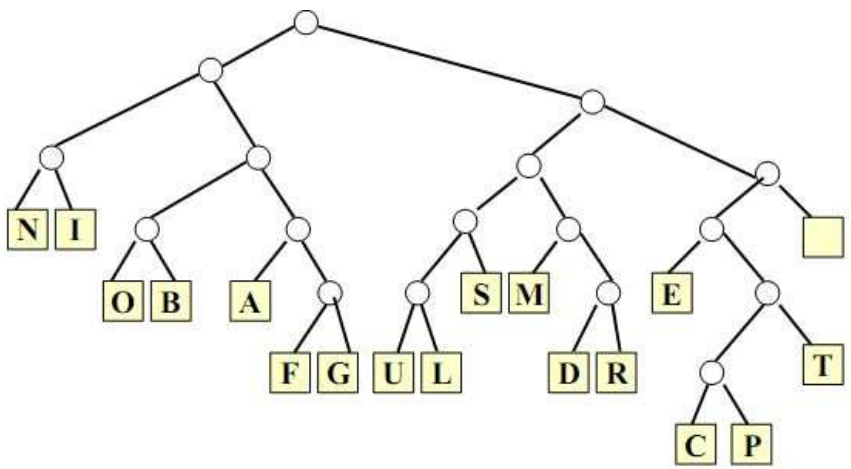
Dann werden die beiden Knoten mit den kleinsten Häufigkeiten ausgewählt und es wird ein neuer Knoten erzeugt, der diese beiden Knoten als Nachfolger hat und dessen Häufigkeit einen Wert hat, der gleich der Summe der Werte für seine Nachfolger ist. Danach werden die beiden Knoten mit der kleinsten Häufigkeit in diesem Wald ermittelt, und ein neuer Knoten wird auf die gleiche Weise erzeugt. Am Schluss sind alle Knoten miteinander zu einem einzigen Baum verbunden. Man beachte, dass sich am Ende Knoten mit geringen Häufigkeiten weit unten im Baum befinden, Knoten mit großen Häufigkeiten in der Nähe der Wurzel des Baumes.

Algorithmus III

Nunmehr kann der Huffman-Code abgeleitet werden, indem die Häufigkeiten an den unteren Knoten einfach durch die zugehörigen Buchstaben ersetzt werden und der Baum dann als ein Trie für die Codierung angesehen wird, wobei genau wie oben “links” einem Bit 0 und “rechts” einem Bit 1 im Code entspricht. Der Code für N ist 000, der Code für I ist 001, der Code für C ist 110100 usw. Die kleine Zahl oberhalb jedes Knotens in diesem Baum ist der Index für das Feld count, der angibt, wo die Häufigkeit gespeichert ist. Diese Angabe benötigt man, um sich bei der Untersuchung des Programms, das den untenstehenden Baum erzeugt, darauf beziehen zu können. Folglich ist für dieses Beispiel count[33] gleich 11, der Summe der Häufigkeitszähler für N und I.

Trie für die Huffman-Codierung von “A SIMPLE STRING ...”

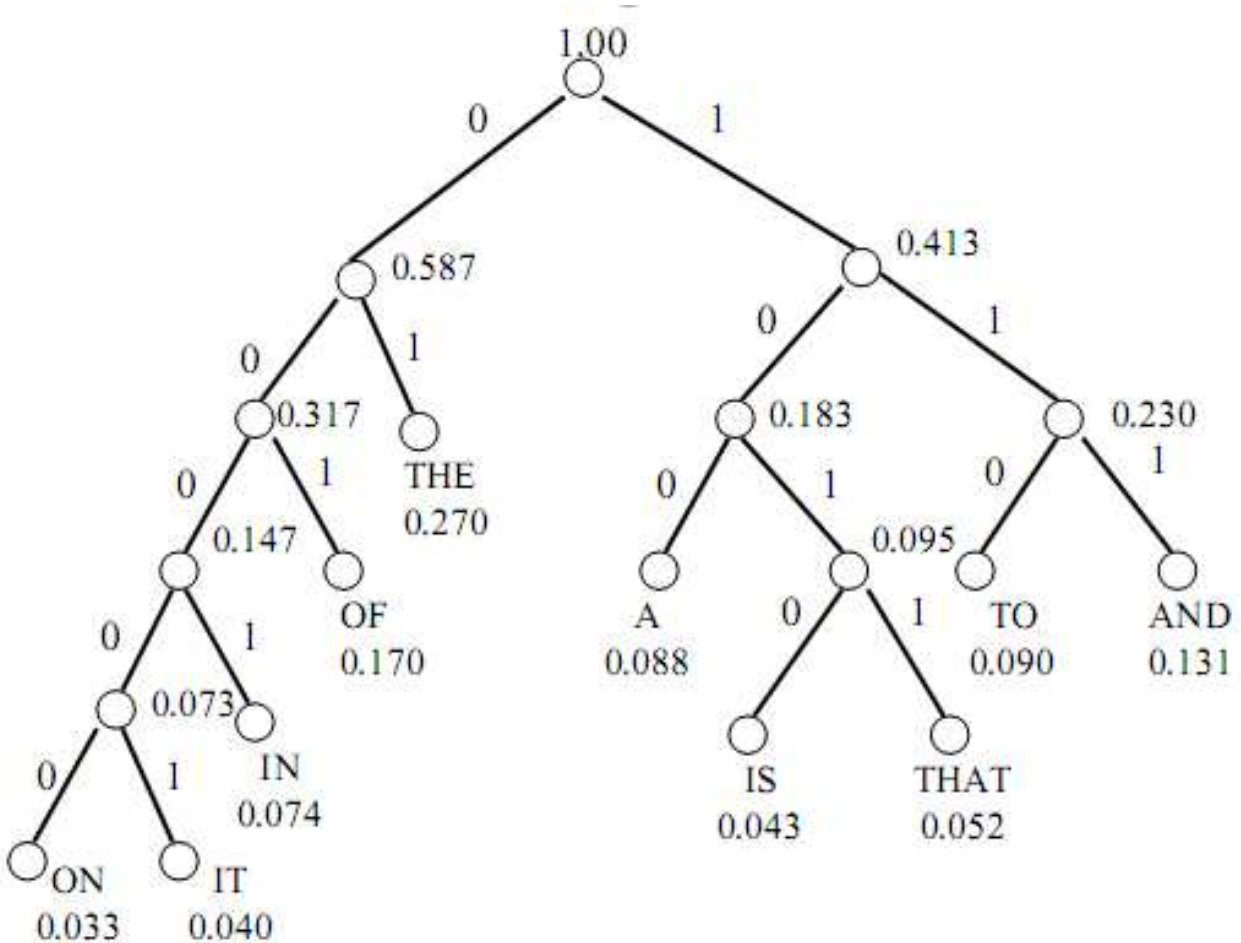
	␣	A	B	C	D	E	F	G	I	L	M	N	O	P	R	S	T	U
k	0	1	2	3	4	5	6	7	9	12	13	14	15	16	18	19	20	21
count[k]	11	3	3	1	2	5	1	2	6	2	4	5	3	1	2	4	3	2



Huffman-Codierungs-Baum für Wörter aus der Englischen Sprache

Codierungs-Einheit	Wahrscheinlichkeit des Auftretens	Code-Wert	Code-Länge
the	0.270	01	2
of	0.170	001	3
and	0.137	111	3
to	0.099	110	3
a	0.088	100	3
in	0.074	0001	4
that	0.052	1011	4
is	0.043	1010	4
it	0.040	00001	5
on	0.033	00000	5

Huffman-Codierungs-Baum II



Fragen zur Kompression

- Verschiedene Kompressionsverfahren arbeiten unterschiedlich gut bei Daten unterschiedlichen Typs (Bilder, numerische Daten, Text).
- Welches Verfahren sollte wann benutzt werden?
- Welches Verfahren ist gut für Text?
- Sind Kombinationen, d.h. die Hintereinanderausführung mehrerer solcher Verfahren sinnvoll?
- Schlussfolgerung: Wir interessieren uns für mehr Verfahren.

Geschichte

1977	Der Kompressionsalgorithmus LZ77 wird von Abrahem Lempel und Jacop Ziv erfunden
1983	(20. Juni) Terry A. Welch (Sperry Corporation - später Unisys) patentiert das LZW-Kompressionsverfahren (Variante von LZ78)
1987	(15. Juni) CompuServe veröffentlicht GIF als freie und offene Spezifikation (Version 87a)
1989	GIF 89a wird vorgestellt

Geschichte

1993	Unisys informiert CompuServe über die Verwendung ihres patentierten LZW-Algorithmus in GIF
1994	(29. Dez.) Unisys gibt öffentlich bekannt, Gebühren für die Verwendung des LZW-Algorithmus einzufordern
1995	(4. Jan.) die PNG Gruppe wird gegründet (7. März) erste PNG Bilder werden ins Netz gestellt (8.Dez.) die PNG-Spezifikation 0.92 steht im W3C
1997	Netscape 4.04 und Internet Explorer 4.0 erscheinen mit PNG Unterstützung

Lempel-Ziv-Welch(LZW)

- Erfunden 1978 (nach dem heute üblichen LZ77), siehe später.
- Erlaubt Kompression, ohne Initialisierungsdaten zu übertragen.
- Arbeitet mit Zeichensatz variabler Länge l. Die Zeichen selbst bestehen aus einem oder mehreren Buchstaben.

Algorithmus:

- Zunächst wird der Zeichensatz mit den 256 Byte-Zeichen und einem Ende-Zeichen initialisiert.
- In der Kodierungsschleife wird das längste (ein-oder mehrbuchstabige) Zeichen aus dem Zeichensatz ermittelt, das mit der Buchstabenfolge des Eingabestroms übereinstimmt.
  - Die Nummer dieses Zeichens wird in die Ausgabe geschrieben.
  - Zusätzlich wird ein neues Zeichen definiert: Die Verlängerung der eben gefundenen Buchstabenfolge um den nächsten Buchstaben.
- Der Zeichensatz wird so immer größer. Bei einer maximalen Größe wird der Zeichensatz wieder auf 257 Zeichen reduziert und der Vorgang wiederholt sich so lange, bis der Eingabestrom vollständig codiert ist.

LZW Algorithmus

Encode:
Codetabelle initialisieren (jedes Zeichen erhält einen Code);
präfix= “ “;
while *Ende des Eingabedatenstroms noch nicht erreicht* **do**
    suffix:= nächstes Zeichen aus dem Eingabedatenstrom;
    muster:= präfix+ suffix;
    if *muster* $\in$ *Codetabelle* **then** präfix:= muster;
    else
        muster in Codetabelle eintragen;
        LZW-Code von präfix ausgeben;
        präfix:= suffix;
    end
end
if *präfix* $\neq \emptyset$ **then**
    LZW-Code von präfix ausgeben;
end

LZW Algorithmus

Beispiel: Codetabelle: 0:A1:B2:C3:D

	präfix	muster	suffix	Eingabedatenstrom	LZW-Code	Ausgabe
0		A	A	ABCABCABCD		
1	A	AB	B	BCABCABCD	4:AB	0 (A)
2	B	BC	C	CABCABCD	5:BC	1 (B)
3	C	CA	A	ABCABCD	6:CA	2 (C)
4	A	AB	B	BCABCD		
5	AB	ABC	C	CABCD	7:ABC	4 (AB)
6	C	CA	A	ABCD		
7	CA	CAB	B	BCD	8:CAB	6 (CA)
8	B	BC	C	CD		
9	BC	BCD	D	D	9:BCD	5 (BC)
10	D					3 (D)

LZW Algorithmus

Decode:
Codetabelle initialisieren (jedes Zeichen erhält einen Code);
präfix= “ “;
while *Ende der Daten noch nicht erreicht* **do**
    lese LZW-Code;
    muster:= dekodiere (LZW-Code);
    gebe muster aus;
    neuer LZW-Code := präfix + erstes Zeichen von muster;
    präfix:= muster;
end

LZW Algorithmus

Beispiel: (Dekodierung) Codetabelle: 0:A1:B2:C3:D

Code	präfix	muster	neuer Code	Ausgabe
0		A		A
1	A	B	4 = AB	B
2	B	C	5 = BC	C
4	C	AB	6 = CA	AB
6	AB	CA	7 = ABC	CA
5	CA	BC	8 = CAB	BC
3	ABC	D	9 = BCD	D

Vorteile von LZW:

- Zeichensatz und Codierung werden häufig gewechselt: LZW kann sich einem Kontextwechsel im Eingabestrom gut anpassen.
- Um den damit verbundenen “Gedächtnisverlust” zu beschränken, kann man die Zeichen aus dem Zeichensatz (nach Anzahl der Benutzung und Länge) bewerten und die besten n Zeichen behalten.

Lempel-Ziv Algorithmen

LZ77 (Sliding Window)

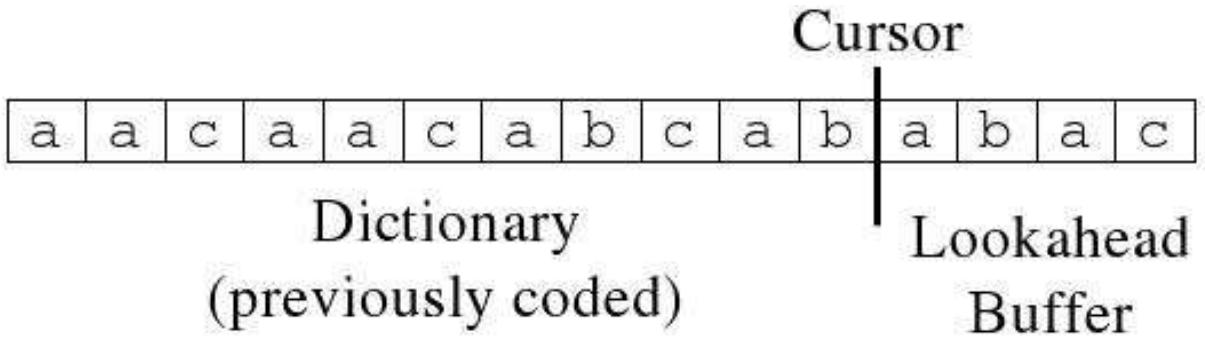
- Varianten: LZSS (Lempel-Ziv-Storer-Szymanski)
- Applications: `gzip`, Squeeze, LHA, PKZIP, ZOO

LZ78 (Dictionary Based)

- Variants: LZW (Lempel-Ziv-Welch), LZC (Lempel-Ziv-Compress)
- Applications: `compress`, GIF, CCITT (modems), ARC, PAK

Normalerweise wurde LZ77 als besser und langsamer als LZ78 betrachtet, aber auf leistungsfähigeren Rechnern ist LZ77 auch schnell.

LZ77: Sliding Window Lempel-Ziv

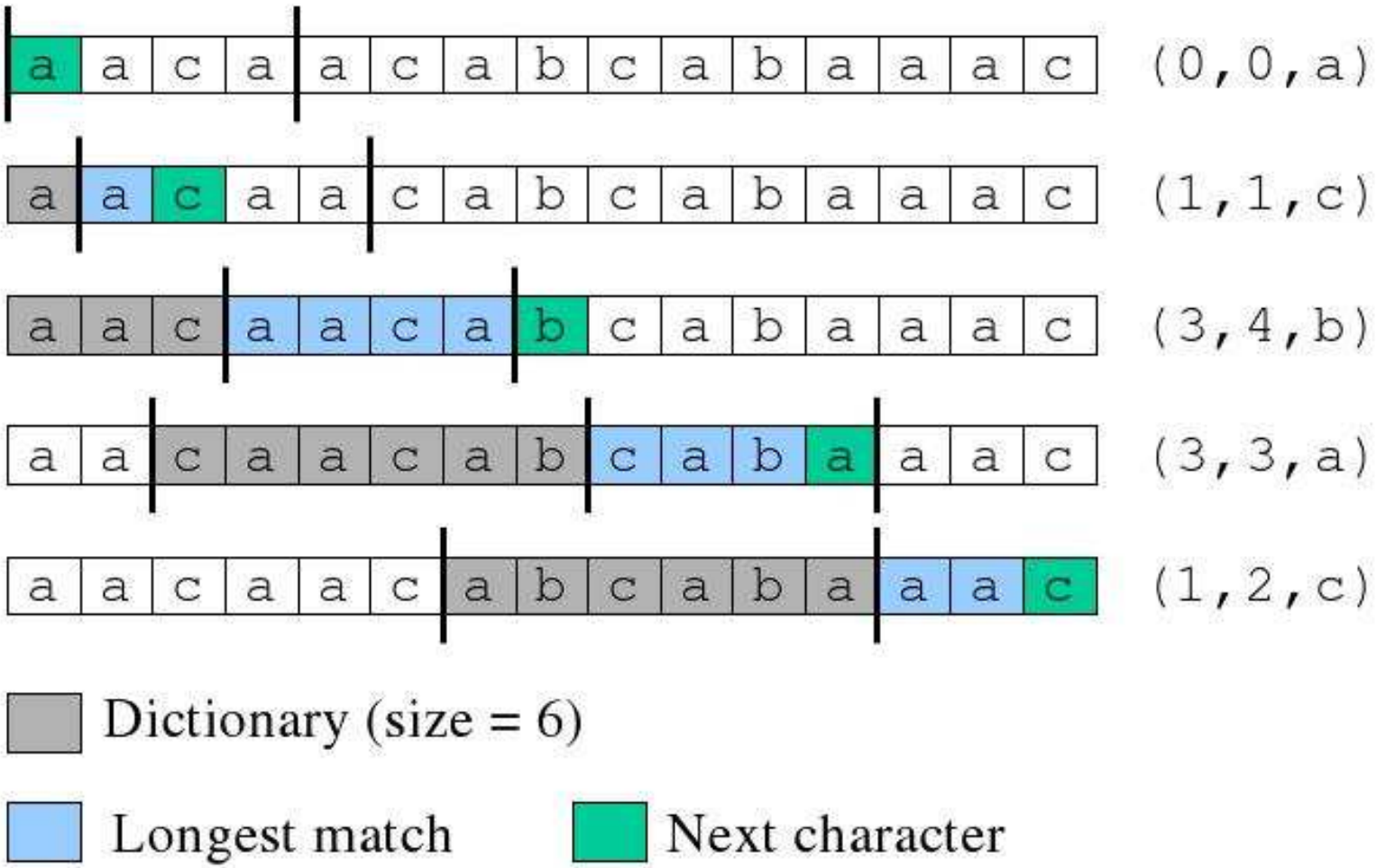


Dictionary- und Buffer-Windows haben feste Länge und verschieben sich zusammen mit dem Cursor.

An jeder Cursor-Position passiert folgendes:

- Ausgabe des Tripels `(p,l,c)`
  - `p` = relative Position des longest match im Dictionary
  - `l` = Länge des longest match
  - `c` = nächstes Zeichen rechts vom longest match
- Verschiebe das Window um `l+1`

LZ77: Example



LZ77 Decoding

Der Decodierer arbeitet mit dem selben Dictionary-Window wie der Codierer

- Im Falle des Tripels `(p,l,c)` geht er `p` Schritte zurück, liest die nächsten `l` Zeichen und kopiert diese nach hinten. Dann wird noch `c` angefügt.

Was ist im Falle `l>p`? (d.h. nur ein Teil der zu copierenden Nachricht ist im Dictionary)

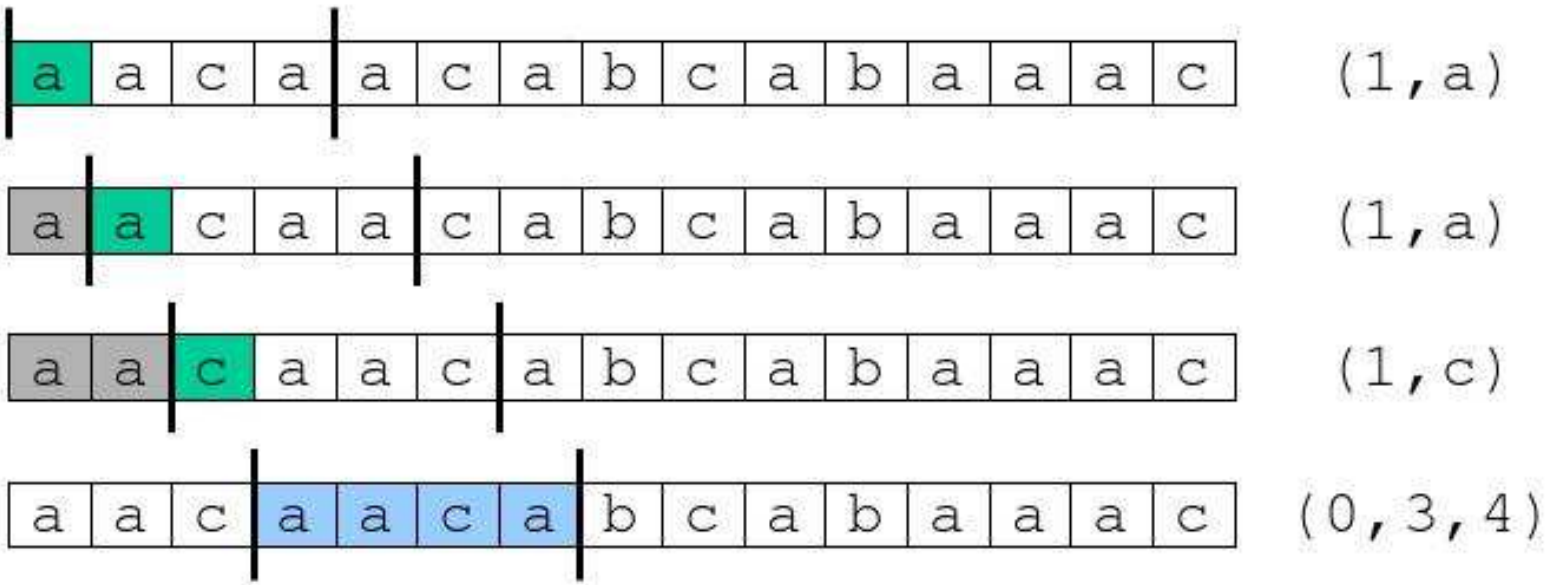
- Beispiel `dict = abcd, codeword = (2,9,e)`
- Lösung: Kopiere einfach zeichenweise:

```
for (i = 0; i < length; i++)
out[cursor+i] = out[cursor-offset+i]
```

- `Out = abcdcdcdcdcdce`

LZ77 Optimierungen bei `gzip` I

LZSS: Der Output hat eins der zwei Formate
`(0, position, length)` oder `(1,char)`
Benutze das zweite Format, falls `length < 3`.



Optimierungen bei `gzip` II

- Nachträgliche Huffman-Codierung der Ausgabe
- Clevere Strategie bei der Codierung: Möglicherweise erlaubt ein kürzerer Match in diesem Schritt einen viel längeren Match im nächsten Schritt
- Benutze eine Hash-Tabelle für das Wörterbuch.
  - Hash-Funktion für Strings der Länge drei.
  - Suche für längere Strings im entsprechenden Überlaufbereich die längste Übereinstimmung.

Theorie zu LZ77

LZ77 ist asymptotisch optimal [Wyner-Ziv,94]
LZ77 komprimiert hinreichend lange Strings entsprechend seiner Entropie, falls die Fenstergröße gegen unendlich geht.

$$H_n = \sum_{X \in A^n} p(X) \log \frac{1}{p(X)}$$

$$H = \lim_{n \to \infty} H_n$$

Achtung, hier ist wirklich eine sehr große Fenstergröße nötig.
In der Praxis wird meist ein Puffer von 216 Zeichen verwendet.