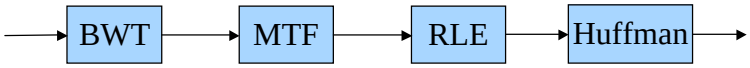
Fachhochschule Köln
University of Applied Sciences Cologne

Verlustfreie Datenkompression mit Hilfe der Burrows Wheeler Transformation

Bjoern Deppe
Frank Fuest
Ingo Kaulbach

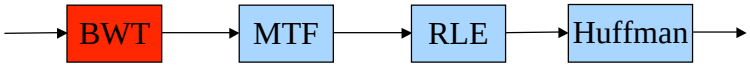
Ablauf:



BWT = Burrows Wheeler Tranformation

MTF = Move-to-front-coding

RLE = Run-length-encoding



Burrows Wheeler Tranformation

- Der BWT ist kein richtiger Kompressionsalgorithmus, sondern ordnet vielmehr die zu komprimierenden Daten nur um.
- Die umgeordneten Daten lassen sich anschliessend effizienter und unkomplizierter komprimieren.
- Dazu werden die Daten nicht sequentiell gelesen, sondern in Bloecken der Groesse N.
- Als Erstes wird aus dem Block eine quadratische Matrix erstellt, wobei die erste Zeile den Original-String enthält und jede weitere Zeile den String aus der Zeile darüber, nur um eine Stelle nach links rotiert.
- Da die Matrix genauso viele Zeilen wie Spalten hat, kommt jede mögliche Rotation des Original-Strings genau einmal vor.

Burrows Wheeler Transformation - Rotation

S0	**V**	**O**	**L**	**L**	**T**	**O**	**L**	**L**
S1	O	L	L	T	O	L	L	V
S2	L	L	T	O	L	L	V	O
S3	L	T	O	L	L	V	O	L
S4	T	O	L	L	V	O	L	L
S5	O	L	L	V	O	L	L	T
S6	L	L	V	O	L	L	T	O
S7	L	V	O	L	L	T	O	L

Burrows Wheeler Transformation - Sortierung

- Im zweiten Schritt werden die Zeilen der Matrix alphabetisch sortiert.
- Alle Spalten sind Permutationen des Ausgangsstrings, das heißt, sie enthalten genau die gleichen Buchstaben wie S0, nur sind diese gegenüber S0 verschoben.
- In der ersten Spalte F stehen dann alle Zeichen des Original-Strings in aufsteigender Reihenfolge sortiert.

	F							L
S2	L	L	T	O	L	L	V	O
S6	L	L	V	O	L	L	T	O
S3	L	T	O	L	L	V	O	L
S7	L	V	O	L	L	T	O	L
S1	O	L	L	T	O	L	L	V
S5	O	L	L	V	O	L	L	T
S4	T	O	L	L	V	O	L	L
S0	**V**	**O**	**L**	**L**	**T**	**O**	**L**	**L**

- In der letzten Spalte L steht der transformierte String S0.
- Im Index I steht die Position des Ausgangsstrings S0 in der sortierten Matrix.

Burrows Wheeler Transformation - Sortierung

	F							L	I
S2	L	L	T	O	L	L	V	O	0
S6	L	L	V	O	L	L	T	O	1
S3	L	T	O	L	L	V	O	L	2
S7	L	V	O	L	L	T	O	L	3
S1	O	L	L	T	O	L	L	V	4
S5	O	L	L	V	O	L	L	T	5
S4	T	O	L	L	V	O	L	L	6
S0	**V**	**O**	**L**	**L**	**T**	**O**	**L**	**L**	7

Burrows Wheeler Transformation - Ausgabe

- Die eigentliche Ausgabe der BWT besteht aus dem transformierten String L und dem Index I.
- L = „OOLLVTLL“ und
- I = 7

Burrows Wheeler Transformation - Ausgabe

- Es fällt auf, dass obwohl F der sortierte String ist, auch L zur Gruppenbildung neigt.
- Dieser Effekt wird deutlicher, wenn N vergrößert wird.

Burrows Wheeler Transformation - Retransformation

- Zur Zurückgewinnung des Originalstrings bauen wir die Matrix mit L wieder auf.
- Aus dem String L läßt sich problemlos der sortierte String F gewinnen – so erhält man die erste und letzte Spalte der Matrix.

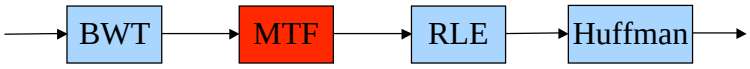
F							L
L	?	?	?	?	?	?	O
L	?	?	?	?	?	?	O
L	?	?	?	?	?	?	L
L	?	?	?	?	?	?	L
O	?	?	?	?	?	?	V
O	?	?	?	?	?	?	T
T	?	?	?	?	?	?	L
V	?	?	?	?	?	?	L

- Die letzte Spalte der Matrix L ist der Vorgänger der ersten Spalte F.
- Nun wird der Transformationsvektor T ermittelt, in welchem die Indexe I der Buchstaben in L in der Reihenfolge ihres Vorkommens in F gespeichert werden.
- Der Transformationsvektor beschreibt die Beziehung zwischen Vorgänger- und Nachfolgerspalte.

T	L	F							L	I
2	O	L	?	?	?	?	?	?	O	0
3	O	L	?	?	?	?	?	?	O	1
6	L	L	?	?	?	?	?	?	L	2
7	L	L	?	?	?	?	?	?	L	3
0	V	O	?	?	?	?	?	?	V	4
1	T	O	?	?	?	?	?	?	T	5
5	L	T	?	?	?	?	?	?	L	6
4	L	V	?	?	?	?	?	?	L	7

- Unserer Transformationsvektor lautet: T = „2, 3, 6, 7, 0, 1, 5, 4“
- Mit seiner Hilfe kann nun die Matrix von F bis L spaltenweise gefüllt werden.

I	F							L	T
0	L	L	T	O	L	L	V	O	2
1	L	L	V	O	L	L	T	O	3
2	L	T	O	L	L	V	O	L	6
3	L	V	O	L	L	T	O	L	7
4	O	L	L	T	O	L	L	V	0
5	O	L	L	V	O	L	L	T	1
6	T	O	L	L	V	O	L	L	5
7	V	O	L	L	T	O	L	L	4



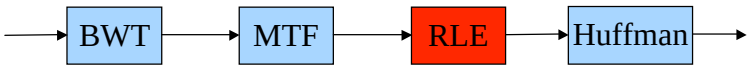
Move-to-front-coding

- ordne alle Zeichen in eine Liste L
  - z.B. L = [A, B, C, D, E, F]
- wiederhole für jedes zu kodierende Zeichen x:
  - c(x) = Position von x in L
  - bewege (move) x unmittelbar nach Benutzung an das linke Ende (to front) von L (andere Zeichen rutschen nach rechts)

- Beispiel: OOLVTLL
- L = [L, O, T, V]

Symbol	Code	I
O	1	OLTV
O	0	OLTV
L	1	LOTV
L	0	LOTV
V	3	VLOT
T	3	TVLO
L	2	LTVO
L	0	LTVO

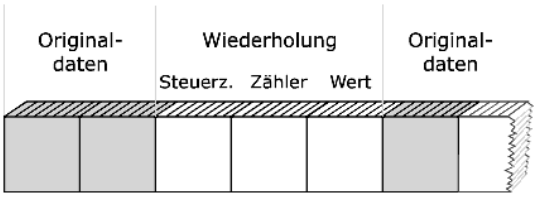
- Ausgabe: 10103320, L = [L, O, T, V]
- Bei Eingaben mit vielen Wiederholungen (wie nach der BWT) entstehen lange Reihen von Nullen



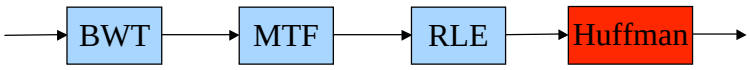
Run-length-encoding

- speichert alle Zeichen in ihrer Reihenfolge
- allerdings ohne ihre Wiederholungen
- Zeichen, die sich wiederholen werden zusammen mit der Anzahl ihrer Wiederholungen gespeichert
- ist erst bei Wiederholungen mit mehr als zwei Zeichen effizient

- Sollen auch Ziffern codiert werden (Normalfall), wird ein Symbol benötigt, das anzeigt, daß eine Wiederholung kommt, und keine Ziffer.



- Beispiel:
- Ausgabe MTF: 10103320, L = [L, O, T, V], I = 7
- Ausgabe RLE: 10103320, L = [L, O, T, V], I = 7
- Keine Verbesserung möglich, da höchstens 2 aufeinanderfolgende Zeichen vorkommen.



Huffman-encoding

- Die Huffmankodierung ist eine Entropiekodierung, d.h. dass Zeichen mit einer unterschiedlichen Anzahl von Bits kodiert werden, um eine speichersparende Darstellung zu erhalten.
- Zeichen die häufig vorkommen, sollen mit einer kleinen Bitfolge, Zeichen die weniger häufig vorkommen sollen mit einer groesseren Bitfolge kodiert werden.
- Um Mehrdeutigkeit zu verhindern, müssen die den Zeichen zugewiesenen Bitfolgen die Eigenschaft besitzen, präfixfrei zu sein, d. h. keine Bitfolge, die für ein einzelnes Zeichen steht, darf am Anfang eines anderen Zeichens stehen.

Huffman-encoding

	a	b	c	d	e	f
Häufigkeit (in Tausend)	45	13	12	16	9	5
Codewort fester Länge	000	001	010	011	100	101
Codewort variabler Länge	0	101	100	111	1101	1100

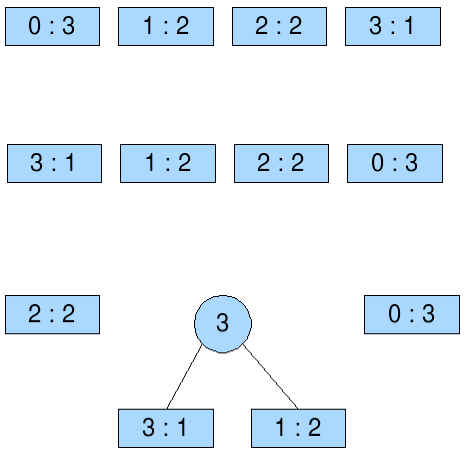
- Angenommen wir möchten ein File mit 100000 Zeichen platzsparend speichern. Es kommen nur die Zeichen a-f mit den oben genannten Häufigkeiten vor.
- Bei einem Code fester Länge benötigen wir 3 Bits, um diese 6 Zeichen darzustellen, d.h. das File ist 300000 Bits gross.
- Ein Code mit variabler Länge löst diese Aufgabe viel besser. In unserem Fall benötigen wir nur noch 224000 Bits um das File darzustellen.
- (45*1+13*3+12*3+16*3+9*4+5*4)*1000=224000
- Dies ist eine Ersparnis von 25%.

Huffman-encoding

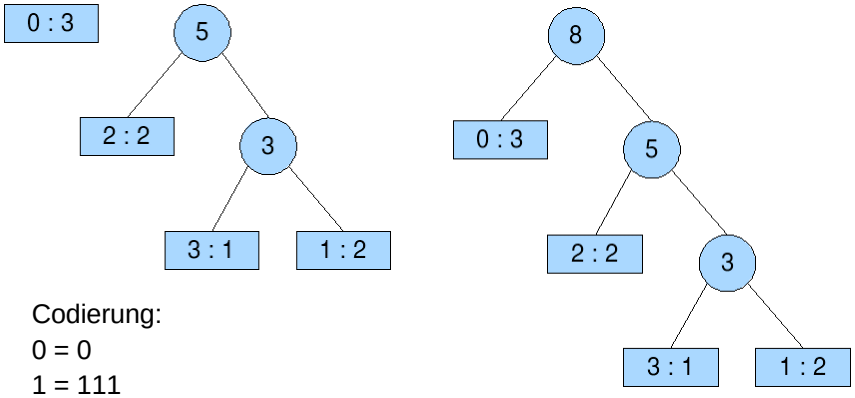
- Die Huffmankodierung funktioniert folgendermassen:

- Erstelle einen Wald mit Bäumen für jedes Zeichen. Diese Bäume erhalten nur einen Knoten, nämlich das Zeichen.
- Entferne die beiden Bäume mit der geringsten Häufigkeit aus dem Wald, erstelle aus ihnen einen neuen Baum und füge ihn wieder in den Wald ein.
- Wiederholen, bis nur noch ein Baum übrig ist.

- Beispiel:
- Ausgabe RLE: 10103320, L = [L, O, T, V], I = 7



Huffman-encoding



Codierung:
0 = 0
1 = 111
2 = 10
3 = 110

Die Huffmankodierung liefert beweisbar immer optimalen reduzierten präfixfreien Code!

Huffman-encoding

- Das RLE lieferte uns 10103320
- Ausgabe Huffman
  - 111 0 111 0 110 110 10 0
  - Codierung: 0 = 0, 1 = 111, 2 = 10, 3 = 110
  - L = [L, O, T, V], I = 7

- Hinweis: Die Zeichen des RLE haben 8Bit, die Zeichen der Huffmancodierung hingegen nur 1Bit, außer L und I (auch 8Bit).

- Kompressionsrate
  - Originalstring: VOLLTOLL = 8x8Bit = 64 Bit
  - Ausgabe: (17 Bit + 41 Bit + 40 Bit) = 98 Bit
    - Bitfolge(Huffman): 111 0 111 0 110 110 10 0
      - 17 Bit
    - Codierung: 0 = 0, 1 = 111, 2 = 10, 3 = 110
      - 4x8Bit + 9x1Bit = 41 Bit
    - L = [L, O, T, V], I = 7
      - 5x8Bit = 40 Bit
  - Kompressionsrate: 98/64 Bit = - 53%

- Unser Beispiel hatte eine „Expansion“ von 53%
- Dies liegt an der geringen Blockgröße von 8 Zeichen
- Die Blockgröße bei bzip2 kann von 100kByte bis 900kByte eingestellt werden.