# Inhaltsverzeichnis

**1 Kompressionsarten** **3**
1.1 Verlustfreie Kompression . . . . . 3
1.2 Verlustbehaftete Kompression . . . . . 3

**2 Mathematische Modelle** **4**
2.1 Grundlagen . . . . . 4
2.2 Shannon-Fano Kodierung . . . . . 5
2.3 Huffman Codierung . . . . . 7
2.4 Verlustbehaftete Kompression . . . . . 7

**3 Kompressionsverfahren** **9**
3.1 Lauflängenkodierung . . . . . 9
3.2 Der LZ77 Algorithmus . . . . . 10
3.3 Der LZ78 Algorithmus . . . . . 14

**4 Kompressionsformate** **15**
4.1 Das Grib Format . . . . . 15
4.2 Das NetCDF 4 / HDF 5 Format . . . . . 16

**5 Vergleich von Algorithmen** **17**

**6 Probleme der aktuellen Algorithmen** **17**

# Abbildungsverzeichnis


1 Approximation einer Sinuskurve, mit Gnuplot erstellt . . . . . 3
2 Informationsgehalt für die Basen 2, 10 und 36, erstellt mit Gnuplot . 5
3 Aufbau eines Shannon-Fano Codierbaums
Quelle: https://de.wikipedia.org/wiki/Shannon-Fano-Kodierung 6
4 Aufbau eines Huffman Codierbaums
Quelle: https://en.wikipedia.org/wiki/Huffman_coding . . . . . 8
5 Beispiel für eine LZ77 Kodierung
Quelle: https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf . . . . . 11
6 Beispiel für eine LZ77 Dekodierung Schritt 1
Quelle: https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf . . . . . 13
7 Beispiel für eine LZ77 Dekodierung Schritt 2
Quelle: https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf . . . . . 13
8 Beispiel für eine LZ77 Dekodierung Schritt 3
Quelle: https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf . . . . . 13
9 Beispiel für eine LZ77 Dekodierung Schritt 4
Quelle: https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf . . . . . 13
10 Beispiel für eine LZ77 Dekodierung Schritt 5
Quelle: https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf . . . . . 13
11 Beispiel für eine LZ77 Dekodierung Schritt 6
Quelle: https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf . . . . . 13
12 Beispiel für eine LZ77 Dekodierung Schritt 7
Quelle: https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf . . . . . 14
13 Beispiel für eine LZ78 Kodierung
Quelle: https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf . . . . . 15
14 Vergleich der Algorithmen
In Anlehnung an: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.300.9031&rep=rep1&type=pdf . . . . . 17
15 Vergleich der Performanz der Algorithmen
In Anlehnung an: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.300.9031&rep=rep1&type=pdf . . . . . 18

# 1 Kompressionsarten

## 1.1 Verlustfreie Kompression

Zur Kompression gibt es zwei Unterscheidungen. Die erste Methode ist die verlustfreie Kompression, bei der die Daten komprimiert und wiederhergestellt werden können. Dies ist hauptsächlich für die Kompression von Text notwendig, aus dem Grund, dass hier ein Verlust der Daten nicht sinnvoll wäre. Verlustfreie Komprimierung beschränkt sich aber nicht nur auf Textdaten sondern kann auch für Bilder und andere Daten verwendet werden. Die Grundidee der verlustfreien Kompression ist es, die Redundanten Daten in einer Datenmenge auszunutzen. Häufig wird mit Hilfe eines Wörterbuches dann auf die redundanten Daten verwiesen, so dass diese letztendlich nur auf den einen Wörterbucheintrag zeigen.

## 1.2 Verlustbehaftete Kompression

Zusätzlich existiert noch die verlustbehaftete Kompression. Diese hat den Vorteil, dass zum Teil deutlich besser für den Anwendungsfall komprimiert werden kann, wie im Fall von mp3, da hier nicht benötigte Daten einfach weggelassen werden können. Bei mp3 wären dies Beispielweise für den Menschen nicht hörbare Frequenzen oder bei JPEG 2000 farblich sehr ähnliche Pixel, welche aneinander liegen, so dass diese verschmolzen werden können. Bei der verlustbehafteten Kompression gibt es nicht, wie bei der verlustfreien Methode eine allgemeine Herangehensweise, sondern hier hängen die Kompressionsmöglichkeiten von der Art der Daten ab. Die Grundidee ist es aber, Daten welche sich sehr ähneln, zusammenzufassen. Hierbei ist jeweils bis zu einem Fehler zu komprimieren. Abbildung 1. zeigt hierbei die Approximation einer

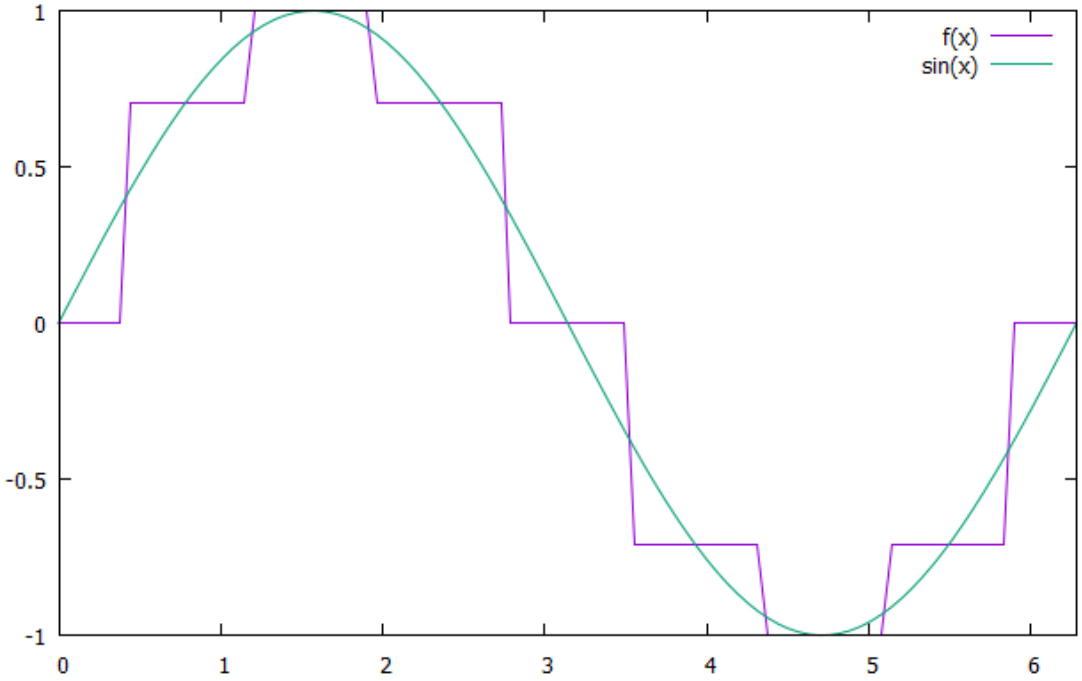


Abbildung 1: Approximation einer Sinuskurve

Sinuskurve durch die Funktion

$$f(x) = \begin{cases} sin(0) & x \in \left[0, \frac{1\pi}{8}\right) \\ sin(\frac{2*pi}{8}) & x \in \left[\frac{1\pi}{8}, \frac{3\pi}{8}\right) \\ sin(\frac{4*pi}{8}) & x \in \left[\frac{3\pi}{8}, \frac{5\pi}{8}\right) \\ sin(\frac{6*pi}{8}) & x \in \left[\frac{5\pi}{8}, \frac{7\pi}{8}\right) \\ sin(\frac{8*pi}{8}) & x \in \left[\frac{7\pi}{8}, \frac{9\pi}{8}\right) \\ sin(\frac{10*pi}{8}) & x \in \left[\frac{9\pi}{8}, \frac{11\pi}{8}\right) \\ sin(\frac{12*pi}{8}) & x \in \left[\frac{11\pi}{8}, \frac{13\pi}{8}\right) \\ sin(\frac{14*pi}{8}) & x \in \left[\frac{13\pi}{8}, \frac{15\pi}{8}\right) \\ sin(0) & sonst \end{cases}$$

durch 3 Bit. Es ist deutlich zu erkennen, dass es hier zu deutlichen Fehlern kommt, aber der Speicherverbrauch im Vergleich zur einer kompletten 32 Bit float Genauigkeit von der Sinus Funktion deutlich besser ist.

# 2 Mathematische Modelle

## 2.1 Grundlagen

Die mathematischen Theorien für die verlustfreie Kompression basieren auf der Arbeit von Claude Shannon "A Mathematical Theory of Communication". Hierbei ist der Informationsgehalt[1] besonders wichtig, da dieser beschreibt, wie wichtig eine Information für ein System ist. Der Informationsgehalt ist hierbei definiert als

$$I(A) = \log_a \left( \frac{1}{P(A)} \right)$$

Hierbei beschreibt $A$ ein Ereignis mit der Eintrittswahrscheinlichkeit $P(A)$ und dem daraus folgenden Informationsgehalt $I(A)$ und $a$ die Basis des Ereignisses, so beispielsweise $a = 2$ für binäre Aussagen. So ist in Abbildung 2 deutlich zu erkennen, dass für eine kleiner Wahrscheinlichkeit der Informationswert deutlich ansteigt. Dies lässt sich einfach durch eine Analogie darstellen. So ist keine Information zu erwarten, wenn ich die binäre Aussage treffe: „Heute trifft kein Meteorit die Erde.", da dies auch für den aktuellen Zeitraum (28.6.2016) nicht zu erwarten ist. Daher wird die Information einem System keinen Mehrwert bringen. Sollte ich jetzt aber die Aussage treffen: „Heute trifft ein Meteorit die Erde.", dann wäre der Informationsgehalt deutlich größer, da die Wahrscheinlichkeit extrem klein und somit kaum zu erwarten war. Daraus folgt dann, dass diese Information sehr wertvoll ist, aber kaum auftritt. Ein mehr technisches Beispiel ist hierbei ein Regensensor. Wenn dieser in der Wüste zur Überwachung stehen würde, so wäre es durchaus sinnvoll kein Regen eine 0 zuzuweisen und Regen eine 1, um hierbei nicht dauerhaft Strom an einer Leitung liegen zu haben. Darauf aufbauend kann für eine ganze Nachricht, welche

[1] de.wikipedia.org/wiki/Informationsgehalt

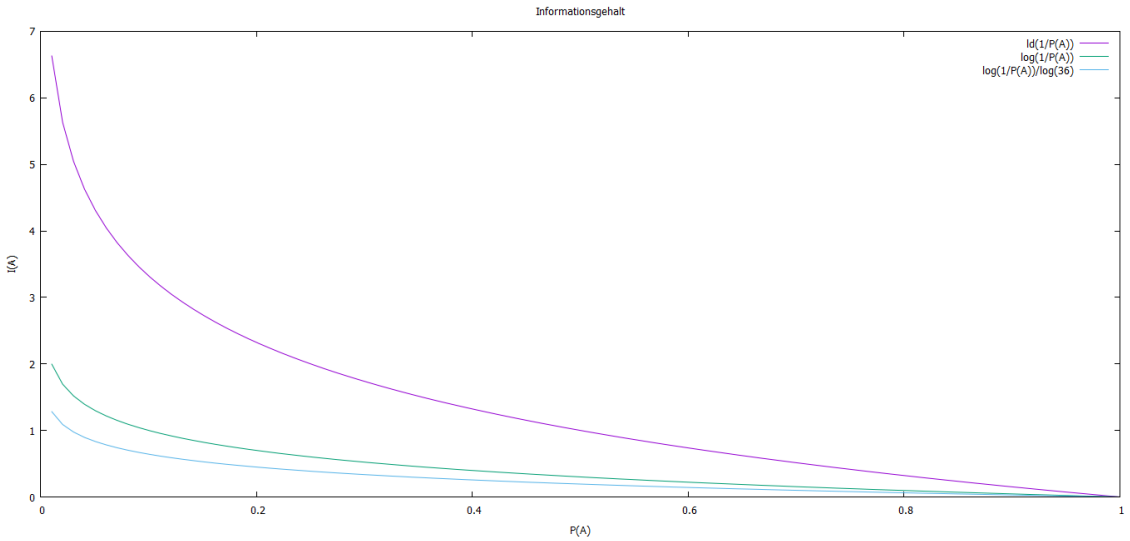


Abbildung 2: Informationsgehalt für die Basen 2, 10 und 36

aus mehreren Ereignissen besteht, der mittlere Informationsgehalt oder Entropie beschrieben werden durch:

$$H(A) = \sum_{A \in S} P(A) \cdot I(A)$$

So ist die Entropie der deutschen Buchstaben bei $H \approx 4.09$[2] (siehe Sektion 2.2). Ein weiterer wichtiger Wert für die Qualität eines Codes ist die Redundanz. Diese beschreibt, wie viel Information mehrfach vorkommt und ohne Informationsverlust weggelassen werden kann. [3]

## 2.2 Shannon-Fano Kodierung

Aus der Entropie und dem Informationsgehalt folgt letztendlich, dass irrelevante Informationen deutlich längere Codes bekommen sollten als sehr häufig vorkommende Informationen. Dies nutzt letztendlich die Shannon-Fano aus, in dem sie häufig vorkommenden Codes einen sehr kurzen Wert zuweist. Die Grundidee ist hierbei, die Codes durch einen Baum Top-Down aufzubauen. Hierbei wird für jedes Ereignis die Auftrittswahrscheinlichkeit berechnet und dann absteigend sortiert. Danach werden die Ereignisse in zwei Mengen geteilt, so dass die Wahrscheinlichkeit der beiden Mengen in etwa gleich ist. Hierbei werden die Mengen zwei Knoten zugewiesen und einem Elternknoten zugewiesen. Die Schritte werden jeweils wiederholt, bis letztendlich keine Ereignisse mehr übrig bleiben. Hierbei ist darauf zu achten, dass die Mengen mit den kleineren Wahrscheinlichkeiten immer in den rechten oder den linken Knoten einsortiert wird und nicht gemischt. Nachdem der Baum erstellt wurde, wird dem Knoten mit den größeren Wahrscheinlichkeiten die 0 zu gewiesen und den mit den kleineren Wahrscheinlichkeiten eine 1, bzw. mathematisch für die direkten

[2] `de.wikipedia.org/wiki/BuchstabenhÃ¤ufigkeit`
[3] `https://de.wikipedia.org/wiki/Redundanz_(Informationstheorie)`

Kindknoten $V_1$ und $V_2$ eines Elternknotens.

$$\forall x \in V_1 : \neg\exists y \in V_2 : y > x$$

dann erhalten alle Elemente in $V_1$ als Suffix eine 1 und alle Elemente in $V_2$ eine 0 als Suffix. In Abbildung 3 sind die Knoten in a) bereits der Wahrscheinlichkeit

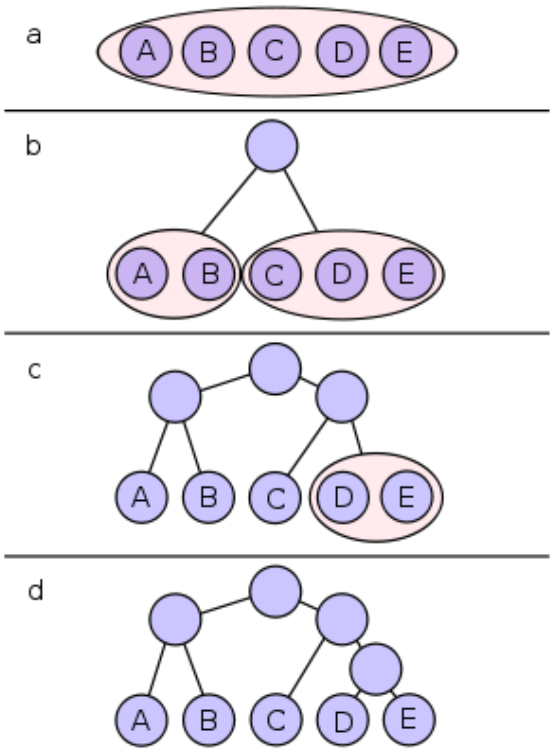


Abbildung 3: Aufbau eines Shannon-Fano Codierbaums

vorsortiert und in diesem Beispiel sind $A$ und $B$ in etwa gleich wahrscheinlich wie $C$, $D$ und $E$. Damit werden dann $A$ und $B$ in einen Knoten und $C$, $D$ und $E$ einem Knoten zugewiesen. Analog sind die Vorgehensweisen für c) und d). Damit folgt dann, dass Codes für die Ereignisse wie folgt sind:

| Ereignis | Code |
|---|---|
| A | 00 |
| B | 01 |
| C | 10 |
| D | 110 |
| E | 111 |

Ein Algorithmus der diese Codierung erzeugt, könnte wie folgt aussehen. Hierbei wird genau die bereits vorgestellte Idee benutzt, so dass in der 2. Zeile die Sortierung vorgenommen wird, dann wenn nur noch ein Element übrig geblieben (Zeile 4.) ist, wird der fertige Code in die Liste eingefügt. Oder wenn noch mehrere Elemente übrig sind (Zeile 5.), dann werden die Elemente geteilt (Zeile 6-7) und der

```
1  def shannon(alphabet, codes, currentCode):
2          sort alphabet descending by there probability
3          if (length of alphabet = 1):
4                  codes append (alphabet.word, currentCode)
5          else:
6                  index = divide alphabet into two parts
7                          with approximately the same probability
8                  shannon(alphabet[start to index], codes, currentCode + '0') +
9                  shannon(alphabet[index+1 to end], codes, currentCode + '1')
```

Algorithmus in Anlehnung an [6]

Algorithmus für die zwei Untermengen erneut durchgeführt (Zeile 8-9). Somit ist bereits eine einfache Kompression der Daten möglich und wird auch häufig für andere Kompressionsalgorithmen als Grundlage benutzt.

## 2.3 Huffman Codierung

Als Alternative zur Shannon-Fano-Codierung existiert die Huffman-Codierung, welche leicht bessere Codes erzeugt und somit die Redundanz minimiert. Die Idee ist hier den Baum nach dem Bottom-up-Verfahren aufzubauen. Hierbei wird jedem Ereignis ein Knoten zugewiesen und dann alle Knoten aufsteigend der Wahrscheinlichkeit sortiert. Danach werden die Knoten mit den zwei kleinsten Wahrscheinlichkeiten zusammengefasst und der Elternknoten erhält dann die summierte Wahrscheinlichkeit der Kindknoten. Abbildung 4. zeigt hierbei den Prozess für die Ereignisse $A$, $B$, $C$, $D$ und $E$ mit den jeweiligen zugewiesen Wahrscheinlichkeiten. Nun werden $D$ und $E$ einem Elternknoten zugewiesen, da diese die kleinste Wahrscheinlichkeit besitzen und der Elternkoten erhält die summierte Wahrscheinlichkeit $5 + 6 = 11$ der beiden Kindknoten. Dieser Prozess wird ebenfalls solange wiederholt bis kein freier Knoten mehr übrig bleibt, so dass nur noch der Baum vorhanden ist. Ein Algorithmus, der den Huffman- Code erzeugt, könnte folgender sein: Hierbei wird in der huffman Funktion, zunächst in Zeile 2, für jedes Element ein Knoten erzeugt und dann daraus der Baum aufgebaut bis nur noch der Rootknoten übrigbleibt. Zur Erzeugung werden zunächst die kleinsten Knoten aus den Knoten gesucht und dann einem Parent hinzugefügt (Zeile 6 - 8). Nachdem der Baum gebaut wurde wird wie beim Shannon-Verfahren aus dem Baum die Codes erzeugt, in dem jeder Seite die Zahl 0 oder 1 zugewiesen wird (Funktion traverseTree).

## 2.4 Verlustbehaftete Kompression

Bei der verlustbehafteten Kompression ist auch wieder kein allgemeines mathematisches Modell vorhanden, so dass im Allgemeinen die Qualität nur mit Hilfe der Quadratischen Abweichung berechnet werden kann.

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (x_n - y_n)^2$$

Das bedeutet auch, dass die Qualität von unterschiedlichen Algorithmen zur verlustbehafteten Komprimierung nicht immer miteinander verglichen werden kann,

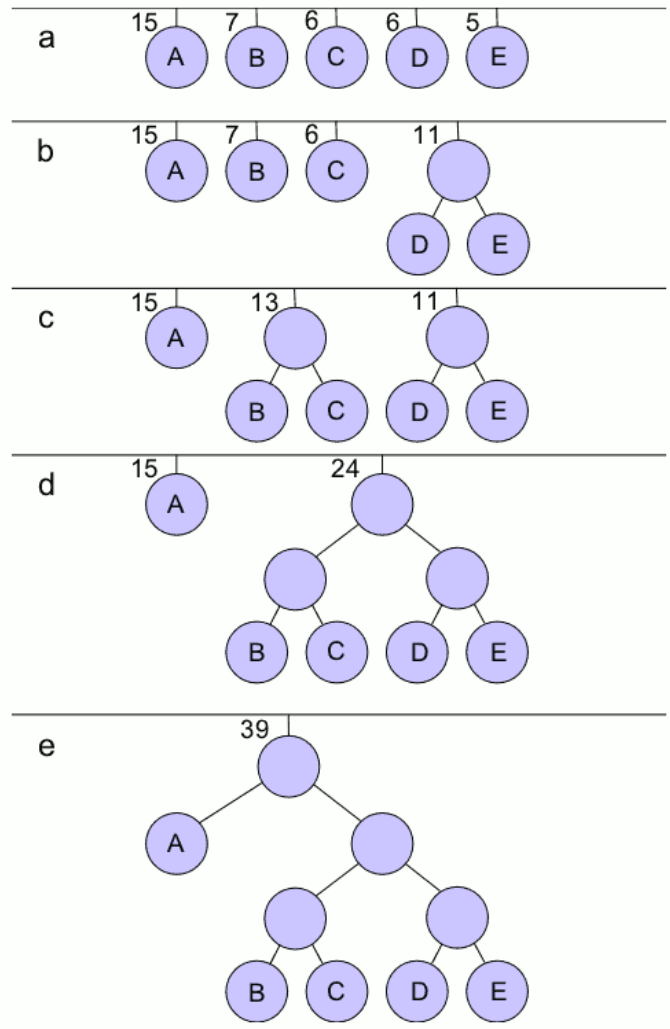


Abbildung 4: Aufbau eines Huffman Codierbaums

```
1  def huffman(alphabet):
2          nodes = create node for every (word, prob) in alphabet
3          # build tree
4          while (nodes count > 1)
5                  sort nodes ascending by there probability
6                  parent = (nodes[0], nodes[1])
7                  remove nodes[0] and nodes[1] from nodes
8                  add parent to nodes
9
10         codes = empty list
11         traverseTree(nodes, codes, '')
12         return codes
13
14 def traverseTree(nodes, codes, currentCode):
15         if leaf node:
16                 codes append (nodes.word, currentCode)
17         else:
18                 traverseTree(nodes.child[0], codes, currentCode + '1') +
19                 traverseTree(nodes.child[1], codes, currentCode + '0')
```

Algorithmus in Anlehnung an [6]

da je nach Problem die Qualität anders gemessen werden kann. Weiter ist es nicht möglich, Codes für eine verlustbehaftete Kompression mit Hilfe von Shannon-Fano- oder Huffman-Codierung zu erzeugen, da diese darauf beruhen, dass seltene aber vielleicht wichtige Ereignisse längere Codes erhalten und wenn diese nun entfernt werden, geht wertvolle Information verloren. Gleiches mit häufigen Codes, da dem Algorithmus nicht bekannt ist, welche Codes in Korrelation zueinander stehen und somit entfernt werden können.

# 3 Kompressionsverfahren

## 3.1 Lauflängenkodierung

Eines der einfachsten Verfahren ist die Lauflängenkodierung. Diese benutzt die sehr grundlegende Idee, dass Daten welche direkt hintereinander stehen, zusammengefasst werden können. Dies hat ebenfalls den Vorteil, dass nur ein einziges Mal über die Daten iteriert werden muss und es auch für Streaming verwendbar ist. Des Weiteren kann es einfach mit anderen Verfahren verbunden werden, da keine besonderen Eigenschaften der Daten, wie beispielsweise die Wahrscheinlichkeiten der Daten nötig sind. Es ist sogar im allgemeinem egal wie die Daten, welche komprimiert werden sollen aussehen, einzig die Möglichkeit des Vergleiches muss gegeben sein. Trotzdem hat das Verfahren einen großen Nachteil, so dass bei sehr zufälligen Daten sich die Datenmenge im schlimmsten Fall verdoppeln kann, da hier vor jedem Bit noch eine 1 geschrieben wird. Der Algorithmus zum Run Length Encoding könnte wie folgt aussehen:

```
1  def rle(data):
2          length = 1
3          symbol = data[0]
4          result = empty list
5          for i = 1 to data.length:
6                  if data[i] == symbol:
7                          length = length + 1
8                  else:
9                          result append tuple (length, symbol)
10                         length = 1
11                         symbol = data[i]
12         result append tuple (length, symbol)
13         return result
```

Algorithmus in Anlehnung an `https://de.wikipedia.org/wiki/LauflÃ¤ngenkodierung`

So wird in den Zeilen 2 und 3 zunächst der erste auftretende Wert als aktueller Laufwert genommen und die aktuelle Länge von eins zugewiesen. Danach wird über die restlichen Daten iteriert (Zeile 5) und solange der Wert gleich bleibt wird einfach nur der Zähler hochgezählt, wie oft das Element hintereinander aufgetaucht ist (Zeile 6 - 7). Sollte das Element nicht gleich des aktuell betrachteten Element sein (Zeile 8), dann wird das Tuple mit der Länge der hintereinander auftauchenden Elemente und dem Element herausgeschrieben (Zeile 9) und die Länge der neuen Elements zurückgesetzt (Zeile 10) sowie das aktuell betrachtete Element neu zugewiesen (Zeile

11). In Zeile 12 werden dann noch die Elemente hinzugefügt, welche noch zum Schluss gezählt wurden. Ein einfaches Beispiel für Lauflängenkodierung ist:

```
rle("00001111111110") = {(4, "0"), (9, "1"), (1, "0")}
```

oder der Worst Case

```
rle("010101010101") = {(1, "0"), (1, "1"), (1, "0"), (1, "1"),
                       (1, "0"), (1, "1"), (1, "0"), (1, "1"),
                       (1, "0"), (1, "1"), (1, "0"), (1, "1")}
```

Die Dekompression erfolgt relativ einfach durch das gezählt Hintereinaderfügen des gelesenen Zeichens. Dies lässt sich Algorithmisch ausdrücken als:

```
1  def uncompressRLE(data):
2          result = ''
3          for length, symbol in data:
4                  # just write length times symbol
5                  for i = 0 to count:
6                          result += symbol
7          return result
```

Algorithmus in Anlehnung an `https://de.wikipedia.org/wiki/LauflÃďngenkodierung`

## 3.2 Der LZ77 Algorithmus

Der LZ77 Algorithmus, welcher von Abraham Lempel und Jacob Ziv entwickelt wurde, nutzt im Gegensatz zu der Huffman- / Shannon-Fano-Kodierung nicht die Wahrscheinlichkeit für ein Ereignis aus sondern Arbeit mit einem Wörterbuch. Aus Performanz / Implementationstechnischen Gründen wird aber meist ein begrenztes Wörterbuch verwendet. Die Idee hinter dem Algorithmus ist es, bereits bearbeitete Präfixe wiederzuverwenden. In der von Lempel und Ziv vorgestellten Version wird ein View benutzt, welches sich in Suchpuffer und einen Kodierpuffer aufteilt. Der Suchpuffer enthält hierbei die bereits verarbeiteten Daten und dient als Wörterbuch für den Kodierpuffer. Die Funktionsweise des Algorithmus ist wie folgt: Zu erst wird ein Kodierpuffer und ein Suchpuffer erzeugt und der Kodierpuffer mit den zu komprimierenden Daten gefüllt. Nun wird für jeden Schritt, bis keine Daten mehr im Kodierpuffer sind, der längste Suffix vom Kodierpuffer aus dem Suchpuffer gesucht. Sollte nun ein Suffix gefunden worden sein, so wird sich die Position in negativer Richtung (vom aktuell gelesenen Zeichen ausgehend rückwärts im array gezählt), die Länge und das nächste Zeichen gemerkt und herausgeschrieben. Hierbei kann es dazu kommen, dass über den Suchpuffer hinaus ins Kodierfenster gelesen wird. Dies ist eine Besonderheit und tritt in der Abbildung 5 beim Kodiertuple $(3, 4, K(b))$ auf. Hierbei bezeichnet $K(x)$ eine Funktion, welche einem Buchstaben eine Kodierung zuweist, es kann aber auch einfach auf die Identität abgebildet werden. Wenn allerdings kein Suffix gefunden wurde so wird einfach ein Tuple $(0, 0, K(x))$ geschrieben, wobei $x$ das aktuell gelesene Zeichen bezeichnet.

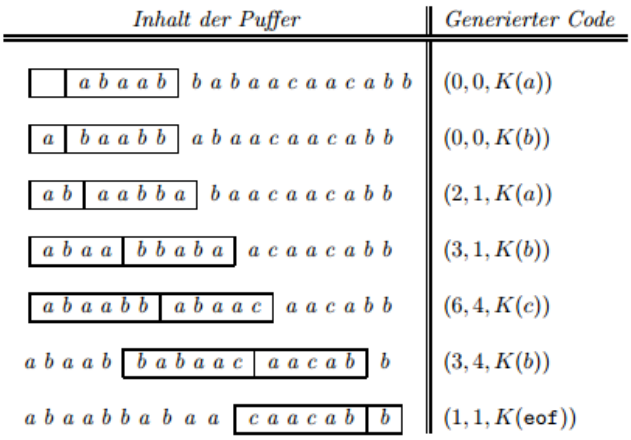


Abbildung 5: Beispiel für eine LZ77 Kodierung

In Abbildung 5 sieht man den Prozess für eine LZ77 Kodierung. So wird im ersten Schritt der Suchpuffer und Kodierpuffer erzeugt, hier dargestellt als schwarzer Kasten um die Buchstaben. Als nächstes wird dann nach dem Buchstaben a im Suchpuffer gesucht, da dieser aber a nicht enthält, wird ein neues Tuple $(0, 0, K(A))$ erzeugt, gleiches gilt für den zweiten Fall. Im dritten Schritt wird dann nach einem $a$ gesucht, welches bereits bearbeitet wurde und deswegen auf den Suchpuffer an Position 2 mit der Länge 1 verwiesen. Der vierte und fünfte Schritt verhält sich wie der dritte Schritt und im sechsten Schritt wird das bereits Erwähnte über den Suchpuffer hinaus lesen vollzogen.
Ein Algorithmus, welcher die LZ77 Kodierung durchführt, könnte folgender sein:

```
1  def compressLZ77(data, dictionarySize, windowSize):
2          dictionaryLength = 0
3          position = 0
4          result = empty list
5          while position < data.length:
6                  length = -1
7                  index = 1
8                  # the view is the compound of the compression view and the search view
9                  start = max(0, position - dictionarySize)
10                 end   = min(data.length, position + windowSize)
11                 view = data from start to end
12                 for i = dictionaryLength + 1 to view.length:
13                         part = view from dictionaryLength to i
14                         if part in view:
15                                 index = view.index(part)
16                                 if index < dictionaryLength:
17                                         length = i - dictionaryLength
18
19                 # if the current symbol is in the dictionary view
20                 if length != -1:
21                         j = dictionaryLength - index
22                         result append tuple (j, length, view[dictionaryLength + length])
23                         position += length
24                 # if its not in the dictionary view
25                 else:
26                         j = 1
27                         result append tuple (0, 0, view[dictionaryLength])
28
29                 # slide the view
30                 position += 1
31                 dictionaryLength = min(dictionaryLength + j, dictionarySize)
32
33         return result
```

Algorithmus in Anlehnung an `https://de.wikipedia.org/wiki/LZ77`

In diesem Algorithmus wird aus technischen Gründen auf zwei Fenster verzichtet, um den Verwaltungsaufwand zu minimieren. Stattdessen wird ein View für beide Puffer in Zeile 11 erzeugt, welcher sich aber dennoch in Suchpuffer und Kodierpuffer aufteilt. Nach der Erzeugung des aktuellen Views wird dann der längste Suffix im Suchpuffer gesucht, welcher nach der oben beschrieben Eigenschaft auch über den Suchpuffer hinauslaufen darf (Zeile 12 - 17). Danach wird, wenn ein Suffix gefunden wurde, die Position j bestimmt und die Länge, sowie das darauf folgende Zeichen in ein Tuple geschrieben (Zeile 20 - 23). Wenn allerdings kein Suffix gefunden wurde (Zeile 25), dann wird das aktuell gelesene Zeichen in die Liste geschrieben (Zeile 26 - 27). Danach wird das Fenster weiter geschoben und der Prozess wiederholt.
Die Dekompression geschieht mit Hilfe der erzeugten Tupel, welche nun angeben ob nur ein neues Zeichen angehängt werden soll (im Falle von **position** = 0) oder ob in den bereits dekomprimierten Daten nachgeschlagen werden soll. Das bedeutet, dass die Dekompression ebenfalls einen Suchpuffer benutzt, welcher jetzt als Wörterbuch dient in dem Suffix nachgeschlagen werden. Im Falle eines Lookups im Wörterbuch wird einfach vom Ende des Arrays rückwärts bis zur Position gezählt und dann die angegebene Länge von Zeichen an das Wörterbuch angehängt. Der Prozess wird in Abbildung 6 - ? noch einmal verdeutlicht.

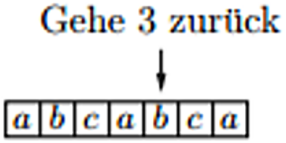


Abbildung 6: Beispiel für eine LZ77 Dekodierung Schritt 1

In diesem Beispiel wird das Tuple $(pos = 3, length = 5, K(d))$ mit dem bereits dekodierten Suchpuffer ABCABCA Dekomprimiert. Hierbei wird zunächst 3 Schritte nach hinten gelaufen und dann die Werte wie in Abbildung 7 - 9 kopiert.

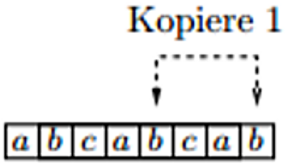


Abbildung 7: Beispiel für eine LZ77 Dekodierung Schritt 2

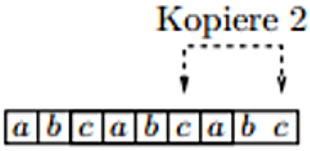


Abbildung 8: Beispiel für eine LZ77 Dekodierung Schritt 3

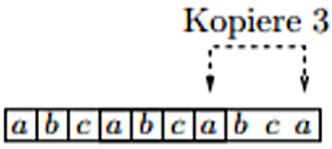


Abbildung 9: Beispiel für eine LZ77 Dekodierung Schritt 4

Jetzt ist es ebenfalls möglich über den Suchpuffer hinaus zu lesen, was in Abbildung 10 und 11 gezeigt wird.

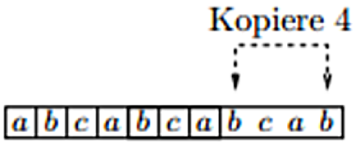


Abbildung 10: Beispiel für eine LZ77 Dekodierung Schritt 5

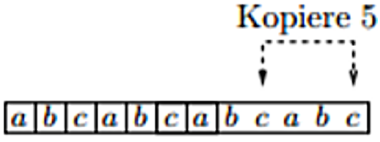


Abbildung 11: Beispiel für eine LZ77 Dekodierung Schritt 6

Im letzten Schritt wird dann der Kodierte Buchstabe aus dem Tuple an den Dekompressionspuffer angehängt.

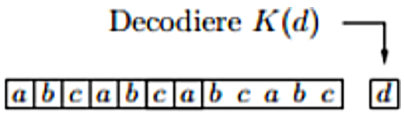


Abbildung 12: Beispiel für eine LZ77 Dekodierung Schritt 7

Algorithmisch sieht dies wie folgt aus:

```
1  def uncompressLZ77(data):
2          view = ''
3          for position, length, symbol in data:
4                  # if the symbol isn't in the search buffer just append it to the view
5                  if position > 0:
6                          start = view.length - position
7                          for i = 0 to length:
8                                  # append values from the back to the buffer
9                                  view += view[start + i]
10                 view += symbol
11         return view
```

Algorithmus in Anlehnung an `https://de.wikipedia.org/wiki/LZ77`

Bei diesem Algorithmus bezeichnet Data die Liste der erzeugten Tupel, welche aus Position, Länge und dem Kodierten Zeichen bestehen. Nun wird im Falle von einer Suche im Suchpuffer die Symbole aus dem Suchpuffer an sich selbst gehängt (Zeile 9).

## 3.3 Der LZ78 Algorithmus

Der LZ78 Algorithmus arbeitet zwar ähnlich wie der LZ77 Algorithmus mit einem Wörterbuch, benutzt allerdings nicht die bereits bearbeiten Daten, sondern nutzt ein globales Dictionary. Hierbei besteht jeder Eintrag aus einem Tupel (index, code) oder (index, index). Der erste Fall tritt ein, wenn kein Eintrag für das aktuell betrachtete Symbol gefunden wurde. Ansonsten werden die Einträge durch zwei Indizes miteinander verknüpft. Beim Alternativverfahren LZW wird statt eines Codes das Symbol direkt in das Dictionary eingefügt.

| Text | Wörterbuch Index | Eintrag | Ausgang Code |
|---|---|---|---|
| $a$ | 1 | $a$ | $(0, K(a))$ |
| $b$ | 2 | $b$ | $(0, K(b))$ |
| $a$ | | | |
| $b$ | 3 | $ab$ | $(1, 2)$ |
| $a$ | | | |
| $b$ | | | |
| $c$ | 4 | $abc$ | $(3, K(c))$ |
| $a$ | | | |
| $b$ | | | |
| $a$ | 5 | $aba$ | $(3, 1)$ |
| $c$ | 6 | $c$ | $(0, K(c))$ |
| eof | | | |

Abbildung 13: Beispiel für eine LZ78 Kodierung

In Abbildung 13 ist ein einfaches Beispiel für die LZ78 Kodierung zu sehen. Hierbei wird im ersten Schritt nach dem Symbol $a$ gesucht. Da dieses nicht im Dictionary ist, wird ein neuer Eintrag im Dictionary erstellt. Das gleiche passiert für das nächste Symbol $b$. Nun wird wieder $a$ gelesen, da dieses bereits einen Eintrag hat wird das nächste Symbol gelesen, für welches ebenfalls ein Eintrag existiert. Aus diesem Grund wird ein Tupel mit den Indices der bereits erzeugten Einträge erstellt. Gleiches passiert beim Lesen der Zeichen $ab$. Da diese bereits einen Eintrag haben, wird auf diesen verwiesen und nur das bisher unbekannte Symbol $c$ hinzugefügt. Dieser Prozess wird solange fortgeführt, bis die Daten zu Ende gelesen wurden.

# 4 Kompressionsformate

## 4.1 Das Grib Format

Das Grib Format ist eines der Standardformate für Wetterdaten und steht für General Regularly-distributed Information in Binary form. Hierbei werden die Werte zur Kompression in Integer umgewandelt, da dies Probleme beim Simple Packing und Complex Packing reduziert. Hierbei wird mit der Formel

$$\text{int\_value} = \textbf{round}(\text{scale} \cdot (\text{float\_value} - \text{offset}))$$

die float Werte in Integer umgerechnet. Zur Kompression bietet Grib folgende Möglichkeiten an:

- **Simple Packing:**
  Hierbei wird die Bitbreite aller Zahlen berechnet und dann alle Zahlen mit der Bitbreite geschrieben.

- **Complex Packing:**
  Beim Complex Packing werden die Daten in Blöcke aufgeteilt und dann die Bitbreite des Maximalen und Minimalen Wertes jedes Blocks berechnet. Danach werden alle Daten des Blocks mit der berechneten Bitbreite geschrieben.

- **Complex Packing (spatial differencing):**
  Dieses Verfahren funktioniert wie das Complex Packing, berechnet allerdings die Differenz der Werte

$$\text{Wert}_i - \text{Wert}_{i-1}$$

  und schreibt dann die Differenzwerte mit der berechneten Bitbreite.

- **JPEG 2000:**
  Bei der JPEG Kompressionsmethode werden die Daten mit Hilfe der verlustbehafteten JPEG Komprimierung geschrieben.

- **PNG:**
  Ähnlich wie das JPEG Verfahren, nur das hier das verlustfreie PNG Verfahren zur Komprimierung verwendet wird.

## 4.2 Das NetCDF 4 / HDF 5 Format

Für das NetCDF 4 bzw. das HDF 5 Format existieren, wie beim Grib Format, ebenfalls bereits Standardkompressionsalgorithmen, wie das LZ77 Verfahren mit Hilfe der gzip Bibliothek. Zusätzlich dazu beherrscht HDF 5 noch eine proprietäres Verfahren namens SZip, welches mit Hilfe der Rice Kompression funktioniert. Außerdem erlaubt das HDF 5 Format die Benutzung von weiteren Kompressionsalgorithmen, durch Drittanbieter über einen Filter, wie Beispielweise den BZip2 Algorithmus.
Es existiert jedoch das grundlegende Problem, dass NetCDF nach [11] selber keine direkte Kompression ermöglicht, was zur Folge hat, dass das HDF 5 Interface benutzt werden muss um die gewünschten Daten zu komprimieren.
Ein Vorteil des NetCDF / HDF 5 Formats ist allerdings die direkte Adressierung der Daten, welche eine direkte Kompression / Dekompression ermöglicht. Zusätzlich abstrahiert die HDF 5 Bibliothek die Kompressionsalgorithmen so, dass der Benutzer des HDF 5 Formats keine Änderungen am aktuellen Programmcode vornehmen müssen um die Daten zu komprimieren. Ebenso werden komprimierte Daten automatisch dekomprimiert, wenn diese mittels der HDF 5 Bibliothek gelesen werden. Es muss jedoch entsprechend der HDF 5 Dokumentation "Chunking" aktiviert sein, damit die Daten komprimiert werden können[5], so dass bestehende Daten eventuell reorganisiert werden müssen. Nach [4] wird als darunterliegende Bibliothek gzip benutzt, so dass die erzeugten Files mittels den gzip Tools inspiziert werden können. Trotzdem unterliegt die Kompression mittels des HDF 5 Formats durch SZip einigen Einschränkungen, wie Beispielweise der Voraussetzung, dass die Daten zu den grundlegenden Datentypen (int, float, char, ...) angehören müssen, was zur Folge hat, dass es nicht möglich Arrays oder zusammengesetzte Datentype zu komprimieren.

# 5 Vergleich von Algorithmen

| Original File | | Table 6: Saving percentages of all selected algorithms | | | | |
|---|---|---|---|---|---|---|
| File | File Size | RLE | LZW | Adaptive | Huffman | Shannon |
| 1 | 22,094 | -0.71 | 38.24 | 39.21 | 37.42 | 36.06 |
| 2 | 44,355 | 1.25 | 43.78 | 39.32 | 38.32 | 37.81 |
| 3 | 11,252 | -0.13 | 30.70 | 35.88 | 32.60 | 31.99 |
| 4 | 15,370 | 11.39 | 47.98 | 44.15 | 41.70 | 40.91 |
| 5 | 78,144 | 11.79 | 69.03 | 42.53 | 41.94 | 40.82 |
| 6 | 39,494 | 3.91 | 44.35 | 42.11 | 41.07 | 40.72 |
| 7 | 118,223 | -0.40 | 50.39 | 37.82 | 37.38 | 36.24 |
| 8 | 180,395 | 0.54 | 55.85 | 42.51 | 42.24 | 40.51 |
| 9 | 242,679 | 0.11 | | 39.38 | 39.15 | 37.85 |
| 10 | 71,575 | 0.53 | 49.31 | 38.38 | 37.71 | 37.40 |

Abbildung 14: Vergleich der Algorithmen

Abbildung 14 zeigt ein Vergleich, der hier vorgestellten Algorithmen. Es ist deutlich zu sehen, dass die RLE-Kompression je nach Daten mit 11,79% gute Kompressionsraten erzielen kann. Dennoch ist der Nachteil der RLE-Kompression deutlich zu sehen, so dass es sogar zu negativen Kompressionsraten kommt und mehr Daten erzeugt werden als ursprünglich vorhanden waren. LZW sticht auf den gegebenen Daten als Favorit hervor, da der Algorithmus die besten Kompressionsergebnisse erzielt. Trotzdem erzielen auch die einfachen Code erzeugenden Verfahren wie der Huffman- / Shannon-Fano-Codierung durchaus solide Kompressionsraten. Bei dem adaptiven Kodierverfahren handelt es sich um ein adaptives Huffman encoding, welches die Probleme der Huffman-Codierung bei großen Codemengen umgeht. Es ist jedoch unterschiedlich wie gut ein Verfahren komprimiert, da dies besonders von den zu komprimierenden Daten abhängt, so kann eine RLE Kodierung für Voxel- / Volumendaten durchaus besser sein als eine Huffman-Codierung, da das RLE Verfahren deutlich schneller ist und bei strukturellen Daten ähnlich gut komprimiert. Bei Textdaten bietet sich RLE aber nicht an sondern LZW oder Huffman- bzw. eine Shannon-Fano Codierung.

# 6 Probleme der aktuellen Algorithmen

Ein nicht zu unterschätzendes Problem bei der Kompression ist immer noch die Datenmenge, da der Komprimierungsprozess meist ein serielles Verfahren ist und somit ein System mehrere Stunden oder Tage in Anspruch nimmt. So ist beispielsweise ein Wikipediadump unkomprimiert mit der gesamten History 10TB groß und komprimiert etwa 100GB. So zeigt Abbildung 15 den Vergleich der Algorithmen RLE, LZW, Adaptive Huffman, Huffman und Shannon-Fano. Es ist deutlich zu erkennen,

das der RLE Algorithmus die beste Kompressionszeit hat, was aus der Linearität des Algorithmus folgt. Deutlich langsamer ist der LZW Algorithmus, was aus dem Lookup der bereits komprimierten Daten aus dem Wörterbuch folgt.

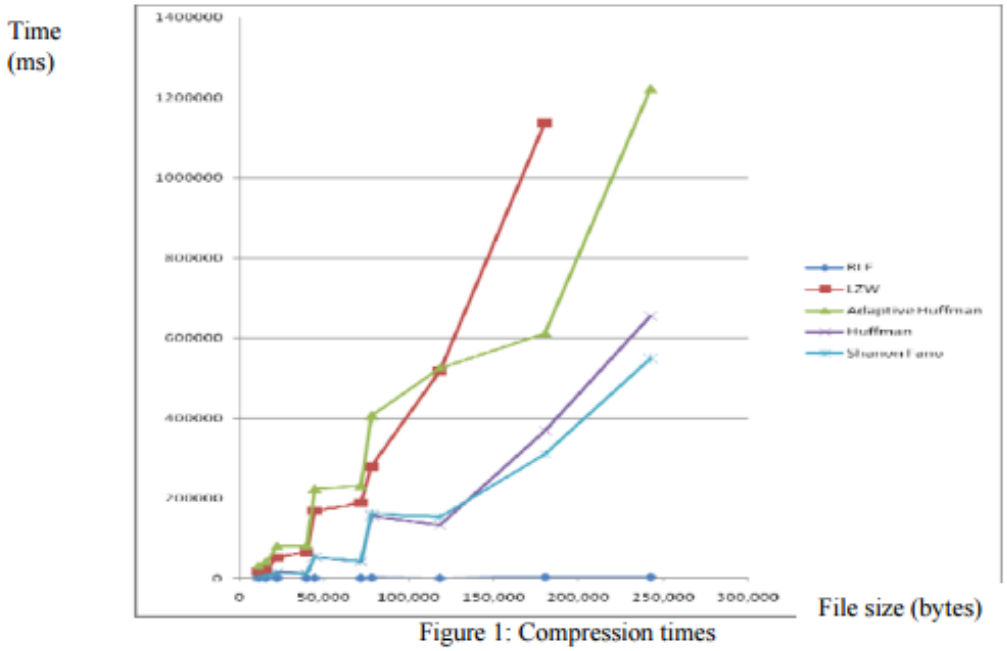


Abbildung 15: Vergleich der Performanz der Algorithmen

Eine Lösung für das Problem des LZW Algorithmus wäre die Parallelisierung. Allerdings gibt es hier kaum explizite Ansätze, sondern es wird meistens die Daten in Blöcke aufgeteilt oder mehrere Dateien gleichzeitig bearbeitet. Dies hat den Vorteil, dass die Algorithmen sich einfach implementieren lassen und hoch parallel arbeiten, da die Daten voneinander unabhängig sind. Jedoch kommt es bei sehr kurzen Eingaben bei der Aufteilung der Daten zu schlechten Kompressionsraten oder bei der Abarbeitung mehrerer Dokumente zu einer größeren Ressourcennutzung. Algorithmisch lässt sich das Parallelisierungsverfahren wie folgt darstellen:

```
1  def parallelCompression(data, compressionFunction):
2          for thread in threads:
3                  compressionFunction(partial data from data for thread)
```

oder

```
1  def parallelCompression(data, compressionFunction):
2          for document in data:
3                  compressionFunction(document)
```

Eine alternativ Lösung wäre die Nutzung der GPU, bei der aber ebenfalls kaum etablierte Lösungen existieren. Dennoch gibt es einige Versuche, LZSS zu parallelisieren. Hierbei wurde insbesondere der Vergleich der Eingabesequenzen mit bereits

gefundenen Sequenzen parallelisiert als auch das Suchen der Einträge im Wörterbuch. Zusätzlich wird noch das Huffman enconding parallelisiert. Trotzdem folgen hier weitere Probleme wie die begrenzte Bandbreite als auch die Limitierung der Programmierung der Hardware hinzu. Ein weiterer Ansatz ist die Komprimierung der Daten mit Hilfe der Hardware zu realisieren. Ein von Intel vorgestelltes Verfahren namens QuickAssist welches Deflate und LZS Kompression in der Hardware implementiert. Hierbei wird mit Hilfe der Intel QuickAssist API Funktionen bereit gestellt, welche an die Intel QuickAssist Treiber weitergegeben und letztendlich von der Hardware ausgeführt werden.
Einen letzten Denkanstoß gibt der Hutter Prize, welcher sich zum Ziel gesetzt hat die Komprimierung von Text zu verbessern. Hierbei gehen die Autoren davon aus, dass der ideale Komprimierungsalgorithmus eine ähnliche Problemstellung ist wie Machine Learning, da hier eine KI entscheidet, wie am besten komprimiert werden kann.

# Literatur


[1] *50'000 Prize for Compressing Human Knowledge*. Accessed: 29-04-2016. URL: http://prize.hutter1.net/.

[2] Dror Baron. URL: http://people.engr.ncsu.edu/dzbaron/research/parallel.html.

[3] Tim Kaldewey Eva Sitaridi Rene Mueller. *Parallel lossless compression using GPUs*. URL: http://on-demand.gputechconf.com/gtc/2014/presentations/S4459-parallel-lossless-compression-using-gpus.pdf.

[4] Edward Hartnet. *netCDF-4/HDF5 File Format*. URL: https://earthdata.nasa.gov/files/ESDS-RFC-022v1.pdf.

[5] *HDF5 Tutorial: Learning The Basics Creating a Compressed Dataset*. URL: https://support.hdfgroup.org/HDF5/Tutor/compress.html.

[6] Norman Hendrich. "64-040 Modul InfB-RS: Rechnerstrukturen, Kapitel 7". URL: https://tams.informatik.uni-hamburg.de/lectures/2015ws/vorlesung/rs/doc/rs-07.pdf.

[7] *Intel® QuickAssist Technology for Storage, Server, Networking and Cloud-Based Deployments*. Accessed: 29-04-2016. URL: http://www.intel.com/content/www/us/en/embedded/technology/quickassist/overview.html.

[8] Henning Fernau Maciej Li'skiewicz. "Datenkompression". URL: https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf.

[9] Anders L. V. Nicolaisen. *Algorithms for Compression on GPUs*. Aug. 2013. URL: http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/6642/pdf/imm6642.pdf.

[10] *Packet Processing*. Accessed: 29-04-2016. URL: https://01.org/packet-processing/intel%C2%AE-quickassist-technology-drivers-and-patches.

[11] Ravishankar. *HDF and NetCDF*. URL: http://www.cise.ufl.edu/~rms/HDF-NetCDF%20Report.pdf.

[12] U.S. AMARASINGHE S.R. KODITUWAKKU. *COMPARISON OF LOSSLESS DATA COMPRESSION ALGORITHMS FOR TEXT DATA*. Accessed: 29-04-2016. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.300.9031&rep=rep1&type=pdf.

[13] Steve Sullivan. *Comparison of Netcdf4 (HDF4) and Grib2 compression methods for meteorological data*. Accessed: 24.05.2016. Juni 2011. URL: https://wiki.ucar.edu/download/attachments/23364539/gridCompressionStudy-v1.pdf%3Fversion%3D1%26modificationDate%3D1317412688000.

[14] *Szip Compression in HDF Products*. Accessed: 29-04-2016. URL: https://www.hdfgroup.org/doc_resource/SZIP/.

[15] Wikipedia. *Data compression*. Accessed: 29-04-2016. Apr. 2016. URL: https://en.wikipedia.org/wiki/Data_compression.

[16] Wikipedia. *Data compression*. Accessed: 29-04-2016. Apr. 2016. URL: https://simple.wikipedia.org/wiki/Data_compression.

[17] Wikipedia. *Huffman coding*. Accessed: 29-06-2016. Mai 2016. URL: https://en.wikipedia.org/wiki/Huffman_coding.

[18] Wikipedia. *Hutter Prize*. Accessed: 29-04-2016. März 2016. URL: https://en.wikipedia.org/wiki/Hutter_Prize.

[19] Wikipedia. *Informationsgehalt*. Accessed: 29-06-2016. Apr. 2016. URL: de.wikipedia.org/wiki/Informationsgehalt.

[20] Wikipedia. *Lauflängenkodierung*. Accessed: 29-06-2016. Mai 2016. URL: https://de.wikipedia.org/wiki/LauflÃďngenkodierung.

[21] Wikipedia. *LZ77*. Accessed: 29-06-2016. Mai 2015. URL: https://de.wikipedia.org/wiki/LZ77.

[22] Wikipedia. *LZ77 and LZ78*. Accessed: 29-06-2016. Juni 2016. URL: https://en.wikipedia.org/wiki/LZ77_and_LZ78.

[23] Wikipedia. *Shannon-Fano-Kodierung*. Accessed: 29-06-2016. Feb. 2016. URL: https://de.wikipedia.org/wiki/Shannon-Fano-Kodierung.