Verlustfreie Datenkompression

Lempel-Ziv-Welch

Digitale AV Technik, MIB 5

Aus Sicht der Informationstheorie

Problem	**Kompression**	**Fehlerkorrektur**
Ziel	Effizienz	Verlässlichkeit
Anwendung	**Quellencodierung**	Kanalcodierung

Algorithmische Perspektive

Problem	**Kompression**
Algorithmen	Shannon–Fano coding
	Huffman-Code
	Arithmetische Codierung, CABAC
	Lempel-Ziv-Welch

Überblick

- **Verlustfreie Kompression** reduziert die Datenmenge ohne Informationsverlust.
- Wichtige Methoden:
  - Lauflängenkodierung (RLE)
  - LZ77, LZ78 und LZW

Lauflängenkodierung

Englisch: Run-Length Encoding, RLE

Idee

- **Erkennung und Komprimierung von aufeinanderfolgenden, gleichen Symbolen.**
- **Einfache, effiziente Methode**, vor allem bei sich wiederholenden Daten.

Beispiel

- Original: `AAABBBCCDAA`
- **RLE-kodiert**: `3A3B2C1D2A`

RLE - Erklärung

1. Zähle die Anzahl der aufeinanderfolgenden gleichen Symbole.
2. Speichere das Symbol und die Häufigkeit als Paar.

Vorteile:

- Sehr einfach zu implementieren.

Nachteile:

- Nicht effizient bei Daten mit wenig Wiederholungen.

RLE - Encoding in Python

```
def rle_encode(data):
    encoding = ''
    i = 0

    while i < len(data):
        count = 1

        while i + 1 < len(data) and data[i] == data[i + 1]:
            i += 1
            count += 1

        encoding += str(count) + data[i]
        i += 1

    return encoding
```

RLE - Decoding in Python

```
def rle_decode(data):
    decoding = ''
    i = 0

    while i < len(data):
        count = int(data[i])
        i += 1
        decoding += data[i] * count
        i += 1

    return decoding
```

RLE in der Anwendung

- Wann bringt RLE etwas?
- Welche Voraussetzung müssen die Daten erfüllen?
- Welche Daten sind in der Regel geeignet, welche nicht?

LZ77

Abraham Lempel and **Jacob Ziv** in 1977

LZ77 - Einführung

Idee

- **Ersetze wiederholte Daten durch Verweise auf vorherige Vorkommen** im Datenstrom.
- Die Verweise sind im Format `(distance, length, char)`:
  - `distance` oder `jump` zeigt an wie viele Schritte man zurück schaut.
  - `length` zeigt an, wie lang die gefundene Wiederholung ist.
  - `char` ist das neue Zeichen, das hinten angefügt wird.

LZ77 - Beispiel

Datenstrom: `ABABABCA`

1. Lese: `A` -> neues Zeichen -> Ausgabe: `(0,0,A)`
2. Lese: `B` -> neues Zeichen -> Ausgabe: `(0,0,B)`
3. Merke: `AB` -> neuer Substring
4. Lese: `ABAB` -> finde den Substring zweimal hintereinander
5. Lese: `C` -> Ausgabe: `(2,4,C)` (Verweis auf das erste `A` und dann Länge 4)
6. Lese: `A` +EOM — Ausgabe: `(0,0,A)`

Ergebnis: `(0,0,A) (0,0,B) (2,4,C) (0,0,A)`

LZ77 - Erklärung

1. **Suche nach dem längsten Präfix**, das in einem festen Fenster der bisherigen Zeichen vorkommt.
2. Speichere die **Distanz**, die **Länge** und das **nächste Symbol**.

Vorteile:

- Effizient für große, sich wiederholende Datenstrukturen.

Nachteile:

- Erfordert viel Speicher für das Fenster.

LZ77 - Compressing in Python

```
def compress(data):
        i = 0
        output_buffer = []

        while i < len(data):
            match = find_longest_match(data, i)

            if match:
                (bm_dist, bm_len) = match
                output_buffer.append((bm_dist, bm_len, data[i + bm_len]))
                i += bm_len + 1
            else:
                output_buffer.append((0, 0, data[i]))
                i += 1

        return output_buffer
```

- `find_longest_match` *finds the longest matching substring in the data*
- `bm` *prefix is used to abbreviate* `best_match`

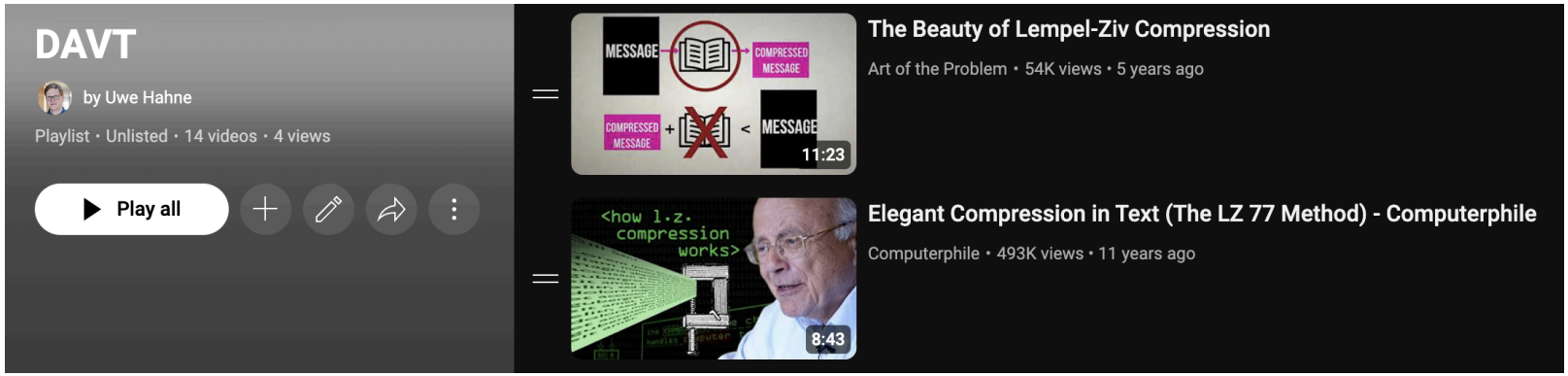
LZ77 - Decompressing in Python

```
def decompress(comp_data):
        decomp_data = []
        for item in comp_data:
            (dist, length, char) = item
            if dist == 0 and length == 0:
                decomp_data.append(char)
            else:
                start = len(decomp_data) - dist
                for i in range(length):
                    decomp_data.append(decomp_data[start + i])
                decomp_data.append(char)

        return ''.join(decomp_data)
```

Videoempfehlung 01

Elegant Compression in Text (The LZ 77 Method) by *Computerphile*



LZ78 - Herkunft

Abraham Lempel and Jacob Ziv in 1978

LZ78 - Einführung

Idee

- **Erstelle ein Wörterbuch** während des Kompressionsprozesses.
- Neue Zeichenfolgen werden als Kombination eines existierenden Eintrags plus einem neuen Symbol gespeichert.

LZ78 - Beispiel

Datenstrom: `ABABABA`

Zeichen	Code	Wörterbuch
`A`	`(0,A)`	`{A:1}`
`B`	`(0,B)`	`{A:1, B:2}`
`AB`	`(1,B)`	`{A:1, B:2, AB:3}`
`ABA`	`(3,A)`	`{A:1, B:2, AB:3, ABA:4}`

Kodiert: `(0,A) (0,B) (1,B) (3,A)`

LZ78 - Erklärung

1. **Füge neue Einträge zum Wörterbuch hinzu**, wenn eine bisher unbekannte Zeichenfolge auftritt.
2. Kodiert als **Index des Präfixes** im Wörterbuch + **neues Symbol**.

Vorteile:

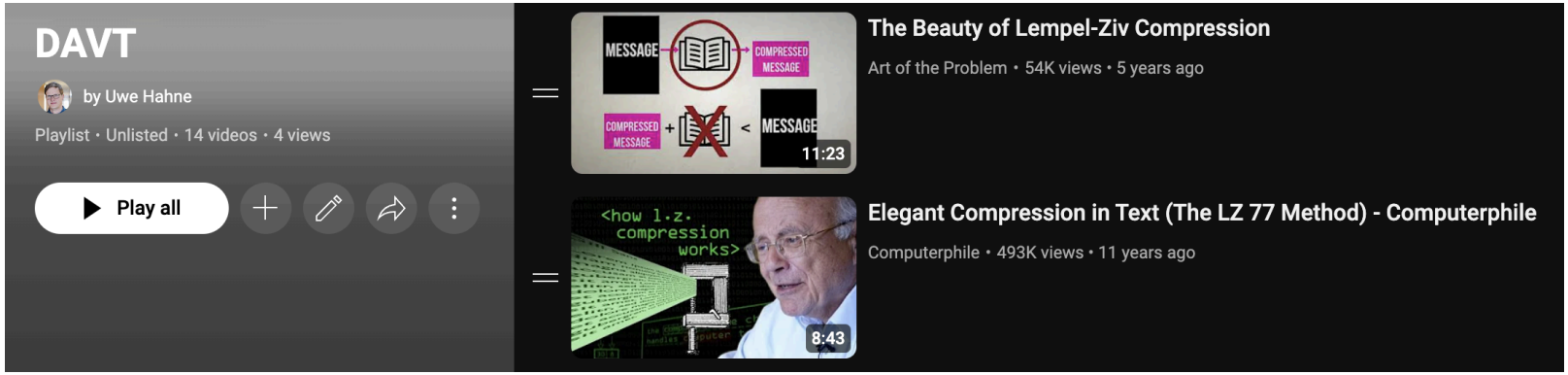
- Weniger Speicherbedarf als LZ77.

Nachteile:

- Wörterbuch muss übertragen oder erneut erstellt werden.

Videoempfehlung 02

The Beauty of Lempel-Ziv Compression by *Art of the Problem*



LZW - Herkunft

Abraham Lempel and Jacob Ziv and Terry Welch in 1984

LZW - Einführung

Idee

- Variante von LZ78, bei der das **Wörterbuch** für beide Parteien im Voraus bekannt ist.
- **Erstellt Einträge ohne den neuen Buchstaben**, d.h. es gibt nur Verweise.

LZW - Beispiel

an der Tafel nach Vorbild von **LZW-Kodierung: Informatik** by *bleeptrack*

LZW - Erklärung

1. **Nutze das initialisierte Wörterbuch**, das sukzessive erweitert wird.
2. Kodiert als **Indexverweise auf Wörterbucheinträge**.

Vorteile:

- Häufig verwendet in Formaten wie GIF und TIFF.
- Effizient und ohne zusätzliches Wörterbuch speicherbar.

Übersicht

- Es gibt zahlreiche Varianten der Kompressionsalgorithmen der LZ Familie.
- **Why the Lempel-Ziv algorithms are so dominant** by *Google for Developers* gibt einen guten Überblick.
- Auf Seiten der University of Stanford findet sich eine Tabelle.

Ausblick

- Nächste Woche: Verlustbehaftete Kompression