Seminar: Datenkompression
Bzip2

Florian Brohm, 7443251 Toprak Saricerci, 7445073

Abgabe: 21. Februar 2023

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Was ist Bzip2?

- Verlustfreies Datenkompressionsverfahren
- Ähnlich zu ZIP
- Basiert auf Burrows-Wheeler transform
- Autor: Julian Seward
- Veröffentlichung: 18. Juli 1996
- Verwendete Algorithmen
  - Run Length Encoding
  - Huffman Encoding
  - Move-to-Front Transform
  - Burrows Wheeler Transform

2 Was ist eine Datenkompression?

Als Datenkompression wird ein Vorgang bezeichnet, bei dem Daten so verarbeitet werden, dass sie mit weniger Bits darstellbar sind. Dabei kann die Kompression entweder `verlustfrei` oder `verlustbehaftet` sein.

2.1 Verlustfreie und verlustbehaftete Kompressionen

Für eine Kompression $k : E \to A$, die eine Eingabe $E \in \{0,1\}^*$ in eine Ausgabe $A \in \{0,1\}^*$ komprimiert, gilt:

$$k \text{ ist } \texttt{verlustfrei} \Leftrightarrow k \text{ ist invertierbar.}$$

Bei `verlustfreien` Kompressionen wird also die Forderung gesetzt, dass die komprimierte Ausgabe wieder in die originale Eingabe dekomprimiert werden kann.

Dass eine Kompression verlustbehaftet ist, ist nicht immer schlimm. Wichtig ist hierbei nur, dass die verlorene Information für den Menschen nicht erkennbar oder zumindest vernachlässigbar ist. So kann zum Beispiel bei einem Bild das

Farbspektrum verkleinert werden, ohne dass das menschliche Auge einen Unterschied bemerkt.

Abbildung 1: Ein Bild, welches mit dem mit dem verlustbehafteten JPEG-Verfahren komprimiert wurde (rechts), sodass Kompressionsartefakte zu erkennen sind [1].

In Texten hingegen können schon kleine Informationsverluste dafür sorgen, dass diese nicht mehr lesbar sind.

2.2 Entropiekodierungen und Substitiutionskompressionen

Bei verlustfreien Kompressionen unterscheidet man grundsätzlich zwischen den `Entropiekodierungen` und `Substitiutionskompressionen`.

`Substitiutionskompressionen` versuchen, oft wiederholende Zeichenfolgen zusammenzufassen oder Muster und somit Redundanz in der Eingabe zu eliminieren. Bekannte Vertreter sind hier das LZ77 Verfahren und das im folgenden Abschnit analysierte Run-Length Verfahren.

`Entropiekodierungen` arbeiten im Gegensatz zu Substitiutionskompressionen nicht mit Zeichenfolgen oder Mustern in der Eingabe, sondern mit den relativen Häufigkeiten der einzelnen Zeichen. Dabei ist das Ziel, dass häufig auftretende Zeichen weniger Bits zur Folge haben und seltene Zeichen mehr Bits. Die Huffman -und arithmetische Kodierung sind Beispiele von Entropiekodierungen.

3 Run-Length encoding

Die Run-Length Kodierung ist eine Substitiutionskompression, bei der sich wiederholende Zeichenfolgen zusammengefasst werden.

3.1 Kodierung

Sei $aw \in \Sigma^*$ die Eingabe, wobei $a = x^n,\ x \in \Sigma$ eine sich wiederholende Zeichenfolge der Länge n ist. Das Teilwort a wird zu nx umgeformt, wonach der Algorithmus mit w fortführt. Folgendes ist ein Beispiel der Kodierung:

jjjjjjjkkijjj $\Rightarrow$ 7jkkijjj $\Rightarrow$ 7j2kijjj $\Rightarrow$ 7j2k1ijjj $\Rightarrow$ 7j2k1i3j

3.2 Dekodierung

Sei nxw mit $n \in \mathbb{N}$, $x \in \Sigma$ und $w \in (\mathbb{N} \cup \Sigma)^*$ die Eingabe. Das Teilwort nx wird zu x^n umgeformt, wonach der Algorithmus mit w fortführt. Folgendes ist ein Beispiel der Kodierung:

$$2j1k6i3k \Rightarrow jj1k6i3k \Rightarrow jjk6i3k \Rightarrow jjkiiiiii3k \Rightarrow jjkiiiiiikkk$$

3.3 Effizienz

Solange die Eingabe viele sich wiederholende Zeichenfolgen enthält, ist die Run-Length Kodierung sehr effektiv. Falls aber nicht, kann die Kompression sogar zu einem größeren Platzverbrauch führen. Die Zeichenfolge `abab`... würde zum Beispiel zu `1a1b1a1b`... komprimiert werden. Falls sowohl Counter als auch Zeichen genau ein Byte Platz bräuchten (kommt ganz auf die Implementierung an), würde die Kompression zu einer Verdopplung der Bits führen.

3.4 Implementierung

Eine einfache und effiziente Implementierung von Run-Length arbeitet mit 8-Bit Zeichen, meist `ASCII`, und hat für den Counter eine Obergrenze von 255, also auch 8-Bit. Das vereinfacht das Dekodieren deutlich, da Counter von Zeichen anhand der Bitposition des Textes unterschieden werden kann. Wenn das Byte auf einem ungeraden Index ist, handelt es sich bei diesem um einen Counterbyte und ansonsten um ein Zeichenbyte.

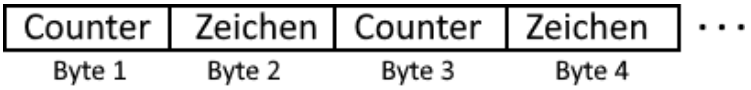

Abbildung 2: Counter -und Zeichenbytes beim Run-Length nach obiger Variante

Außerdem gibt es eine Variante von Run-Length, die zum Beispiel nur Zeichenwiederholungen der Länge 4 oder größer zusammenfasst, um die mögliche Verdopplung des Platzverbrauchs zu verhindern. `abab`... würde dann nicht verändert werden, da alle Zeichenwiederholungen die Länge 1 haben. So ist es aber nicht mehr einfach möglich, Counter von Zeichen zu unterscheiden.

3.5 Pseudocode

3.5.1 Kodierung

```
function run_length(input):
    out ← ϵ
    counter ← 1
    run_char ← input[1]
    for i in 2..n:
        if input[i−1] = input[i]:
            counter++
        else:
            out.append(counter, run_char)
            counter ← 1
            run_char ← input[i]
    out.append(counter)
    out.append(run_char)
    return out
```

3.5.2 Dekodierung

```
function run_length_decode(input):
    out ← ϵ
    for (count, char) in input:
        out.append(char * count)
    return out
```

4 Huffman encoding

Die Huffman-Kodierung errechnet für Zeichen $z \in \Sigma$ einer Eingabe einen Code

$$c : \Sigma \to \bigcup_{n \geq 1} \{0,1\}^n.$$

Dafür zählt die Kodierung die relative Häufigkeit jedes Zeichens und gibt Zeichen mit hoher relativer Häufigkeit kürzere Codes und Zeichen mit kleinerer relativer Häufigkeit entsprechend längere Kodes. Bei Eingaben mit ungleichen Häufigkeitsverteilungen machen einige wenige Zeichen den Großteil der Eingabe aus, sodass ein Großteil der Zeichen kurze Codes hat, was eine kleine durchschnittliche Codelänge und somit große Platzeinsparungen zur Folge hat.

4.1 Präfixfreiheit

Ein Code $c : \Sigma \to \Gamma$ gilt dann als `präfixfrei`, wenn

$$\forall x, y \in \mathsf{Bild}(c) : \nexists z \in \Gamma : xz = y.$$

Präfixfreie Codes wie bei der Huffman-Kodierung haben den Vorteil, man so aus einer Codefolge direkt die originale Eingabe linear errechnen kann. Eine Bitfolge ist genau dann ein Zeichen der originalen Eingabe, wenn die Bitfolge im Bild von c ist, welche bei der Huffman-Kodierung aus der gespeicherten Tabelle berechnet werden können.

4.2 Kodierung

Die Kodierung besteht aus folgenden Schritten:

1. Zähl in der Eingabe, wie oft jedes Zeichen vorkommt und berechne so die relative Häufigkeit.
2. Erstelle einen Graph, wobei jedes Zeichen einen Knoten hat, indem seine relative Häufigkeit und sein Zeichen gespeichert sind.
3. Solange es mehr als einen Knoten gibt, der kein Elternknoten besitzt:
   - Erstelle einen Knoten k, dessen beiden Kinder die Knoten im Graph mit der kleinsten relativen Häufigkeit sind.
   - Die linke Kante wird mit 0 und die rechte mit 1 markiert.
   - Setze die relative Häufigkeit von k auf die Summe der relativen Häufigkeiten seiner beiden Kinder.
4. Erstelle eine Tabelle, in der jedes Zeichen einem Code zugeordnet wird.
5. Traversiere den Baum von der Wurzel
   - Sobald ein Blatt erreicht wird, wird dessen Zeichen der zurückgelegte Weg als Kodierung zugewiesen.
6. Setze jedes Zeichen der Eingabe auf seinen Code in der Tabelle.

4.3 Dekodierung

Da bei der Kodierung der Eingabe auch die Tabelle mitgespeichert wird, kann man einfach den kodierten Text Bit für Bit durchgehen, bis die Bitfolge als Eintrag in der Tabelle zu finden ist. Da der Code präfixfrei ist, kann es keinen anderen Code in der Tabelle geben, der mit dieser Bitfolge erreicht werden könnte, sodass die Zuordnung eindeutig ist.

4.4 Effizienz

Da die Tabelle gespeichert werden muss und seine Größe linear von der Alphabetgröße abhängt, weil jedes Zeichen im Alphabet ein Eintrag besitzt, hat die Huffman-Kodierung einen riesigen Overhead. Bei einem Alphabet mit 256 Zeichen lohnt sich die Huffman-Kodierung für eine Eingabe mit weniger als 1000 Zeichen überhaupt nicht, da die Tabellengröße einen großen Anteil der Gesamtgröße ausmacht. Erst bei einer sehr großen Anzahl an Zeichen, z.B. 100000 ist die Tabellengröße vernachlässigbar.

4.5 Pseudocode

Da im 3. Schritt der Kodierung immer die Kinder mit der kleinsten relativen Häufigkeit gesucht werden, eignet sich ein Min-Heap als Datenstruktur für eine schnelle Suche.

4.5.1 Tabellengenerierung

```
function huffman_table(input):
    table: string ⇒ string
    heap ← MinHeap()
    for char in input:
        heap.insert(char, occurance(char))
    root ← {}
    while heap.hasItem():
        (char1, char1_occ) ← heap.pop()
        if heap.empty(): break
        (char2, char2_occ) ← heap.pop()
        root ← (char1_occ + char2_occ)
        heap.insert(root)
    for leaf in root:
        table.map(leaf.path → leaf.char)
    return table
```

4.5.2 Kodierung

```
function huffman_encode(input):
    table ← huffman_table(input)
    out ← ϵ
    for char in input:
        out.append(table(char))
    return table, out
```

4.5.3 Dekodierung

```
function huffman_decode(table, input):
    buffer ← ϵ
    out ← ϵ
    for char in input:
        buffer.append(char)
        if buffer in table:
            out.append(table(buffer))
            buffer ← ϵ
    return out
```

5 Transformation

5.1 Was ist eine Transformation?

Transformationen sind Funktionen auf einer Menge von Eingabewörtern Σ^*, welche diese auf eine, oft gleiche, Menge von Ausgabewörtern Σ'^* abbildet. Die Eingabe und das Ergebnis sind dabei jedoch gleich lang.

Sei T eine Transformation mit

$$T: \ \Sigma^* \to \Sigma'^*, \ w \mapsto T(w) \tag{1}$$

Die Anwendung einer Transformation wird im folgenden für ein Wort $w \in \Sigma^*$ gekürzt beschrieben.

$$w^T := T(w) \tag{2}$$

Aus der oben genannten Definition folgt

$$|w^T| = |w| \tag{3}$$

Beispiele simpler Transformationen:

$$abcd \overset{R}{\to} dcba \quad \text{(Umkehrung)}$$
$$dbca \overset{\text{sort}}{\to} abcd \quad \text{(Lexikographische Sortierung)}$$
$$abcd \overset{0}{\to} 0000 \quad \text{(Nulling)}$$

Man beobachte: die Permutation ist ein Spezialfall der Transformation.

5.2 Warum sind Transformationen nützlich?

Das Ziel der Datenkompression ist logischerweise das Einsparen von Speicherplatz. Wenn jedoch die Länge der Ausgabe einer Transformation der Länge der Eingabe gleicht, wozu benötigen wir sie dann?

Transformationen können benutzt werden, um Eingaben in eine für eine Kodierung (o.Ä.) günstige Form zu bringen. Dies steigert die Effizienz der Kodierung und spart somit mehr Speicherplatz.

Wir stellen die Forderung, dass eine Transformation invertierbar sein muss, um für (verlustfreie) Datenkompression nützlich zu sein. Die Umkehrfunktion einer Transformation T wird wie folgt definiert.

$$T^{-1}: \ \Sigma'^* \to \Sigma^*, \ w = \left(w^T\right)^{T^{-1}}$$

Jetzt gilt es nur noch, tatsächlich nützliche Transformationen zu finden und anzuwenden.

6 Move-to-front transform

6.1 Erläuterung

MTF ist eine Transformation, welche Zeichen der Eingabe auf deren Index in einer Liste "neulich genutzter" Zeichen abbildet.

Sei Σ das Eingabealphabet mit $|\Sigma| = n$. Wir initialisieren die Liste $A(1..n)$ folgendermaßen.

$$A_{1..n} = (\sigma_1, .., \sigma_n),\ \sigma_1, .., \sigma_n \in \Sigma \text{ paarweise Verschieden} \tag{4}$$

A enthält initial das gesamte Alphabet, per konvention lexikographisch sortiert.

Wir definieren die Hilfsfunktion t_{MTF}:

$$\begin{aligned} &t_{\text{MTF}} : \Sigma^n \times \Sigma \to \Sigma^n \times \Sigma, \\ &t_{\text{MTF}}(A, \sigma) := (A_i \circ (A \setminus A_i), i),\ i = \text{Index von } \sigma \text{ in } A \end{aligned} \tag{5}$$

t_{MTF} beschreibt einen Schritt des MTF-Algorithmus. Für ein Zeichen $\sigma \in \Sigma$ der Eingabe wird zunächst dessen Index in A in die Ausgabe geschrieben. Anschließend wird das Zeichen vorne an A gehängt.

Mithilfe von t_{MTF} lässt sich MTF nun definieren.

$$\begin{aligned} &\text{MTF}_A : \Sigma^* \to \Sigma^* \\ &\quad w^{\text{MTF}_A} := i \circ w' \\ &\qquad \text{mit } (A', i) = t_{\text{MTF}}(A, w_1) \\ &\qquad\quad w' = (w_2, .., w_k)^{\text{MTF}_{A'}} \end{aligned} \tag{6}$$

6.2 Inverse Transformation

Die Invertierung der MTF ist trivial, da gilt

$$\mathrm{MTF}_A^{-1} = \mathrm{MTF}_A \tag{7}$$

6.3 Implementierung

6.3.1 Transformation, Inverse Transformation

```
function mtf(input):
    A[|Σ|] ← Σ
    out ← ϵ
    for char in input:
        char_index ← A.indexOf(char)
        out.append(char_index)
        delete A[char_index]
        prepend(char, A)
    return out
```

6.3.2 Implementationsüberlegungen

Da A für Transformation und Inversion gleich sein muss, ist es effizienter, ein festes A in der Spezifikation des Algorithmus festzulegen.
Die Geschwindigkeit der Transformation wird dramatisch verbessert, wenn für A eine Linked List verwendet wird.

7 Burrows-Wheeler transform

7.1 Definitionen

Sei $w \in \Sigma^*$ mit $|w| = n$. Wir definieren die i-te Rotation von w für $1 \leq i \leq n$ folgendermaßen.

$$w^{\lambda_i} := (w_i, .., w_n, w_1, .., w_{i-1}) \tag{8}$$

Die Menge aller Rotationen von w sei

$$w^{\lambda} := \{w^{\lambda_i} : 1 \leq i \leq n\} \subset S_n \tag{9}$$

Ferner sei die Rotation von w des Ranges k

$$w^{\Lambda_k} := w^{\lambda_i}, \text{ sodass } |\{w^{\lambda_j} : w^{\lambda_j} \leq_l w^{\lambda_i}, \ 1 \leq i, j \leq n\}| = k \tag{10}$$

In einer lexikographisch sortierten Liste von Rotationen ist w^{Λ_k} der k-te Eintrag.

7.2 Erläuterung

BWT ist eine Transformation, welche die durchschnittliche Run-Length in der Eingabe erhöht. Dazu wird die lexikographische Rangfolge der Zeichen in betracht gezogen.

BWT sei für eine Eingabe $w \in \Sigma^*$ definiert als

$$\begin{gathered} \mathrm{BWT} : \Sigma^* \to (\Sigma \cup \{\$\})^*, \\ w^{\mathrm{BWT}} := (u_{n+1}^{\Lambda_1}, .., u_{n+1}^{\Lambda_{n+1}}) \\ \text{mit } u = w \circ \$ \end{gathered} \tag{11}$$

Aus einer sortierten Liste von Rotationen der Eingabe wird die Ausgabe jeweils aus dem letzten Zeichen jeder Rotationen gebildet. Davor wird jedoch ein "$" als Endzeichen an die Eingabe gehängt. Es gilt $\$ \notin \Sigma$.

7.3 Inverse Transformation

Für die Invertierung einer Ausgabe $w \in \Sigma^*$ und dessen lexikograpisch sortierte Permutation w' der Transformation wird zunächst eine Hilfspermutation definiert.

$$\gamma : \mathbb{N} \to \mathbb{N}, \ w_{\gamma(x)} = w'_x \tag{12}$$

γ bildet die Position eines Zeichens in w auf dessen Position in w' ab. Ein Zeichen wird also auf dessen neue Position abgebildet, nachdem w sortiert wurde.
Man beobachte:

$$\gamma \text{ besteht aus einem einzigen Zyklus der Länge } n+1. \tag{13}$$

Nun lässt sich die Inversion von BWT folgendermaßen definieren.

$$w^{\mathrm{BWT}^{-1}} := (w_{\gamma(\$)}, w_{\gamma(\gamma(\$)),..})^R \tag{14}$$

Merke: es gilt $|w| = |w^{\mathrm{BWT}^{-1}}|$

7.4 Implementierung

7.4.1 Transformation

```
function burrows_wheeler(input):
    rotations[n, n] ← input.rotations()
    rotations.sort()
    return rotations[1..n, n]
```

7.4.2 Invers

```
function inverse_burrows_wheeler(input):
    rotations[n, n]
    for k in n..1:
        rotations[1..n, k] ← input
        rotations.sort()
    return rotation ending with $
```

7.4.3 Implementationsüberlegungen

Suffix Arrays können in der Transformation in linearer Zeit konstruiert werden, um die Transformation letztendlich linear umzusetzen.
Die Platz-Effizienz der Pseudocode Beispiele ist relativ schlecht. Statt ganze Matrizen zu speichern, können Pointer in die jeweiligen Strings verwendet werden.

Literatur

[1] *JPEG Artefakte*. (besucht am 19.02.2023, 16:00). URL: https://de.wikipedia.org/wiki/Datenkompression#/media/Datei:Jpegartefakt_jpegartefact.jpg.