Der Huffman Algorithmus

Arman Habibi
Larissa Manalil
Andrei Stoica

Inhaltsverzeichnis

Problemstellung

- Grundproblem: Verlustfreie Kompression von Daten
- Lösung: Huffman Algorithmus
  - Zeichen, die häufiger vorkommen, mit weniger Bits speichern
  - Statt 8-Bit für jedes Zeichen, wie bei ASCII
- Zeichenbasiert
  - Jedem Zeichen wird ein zugehöriges binäres Codewort zugeordnet
  - Häufigkeitsverteilung der Zeichen muss bekannt sein
- Generierung eines präfix-freien, optimalen Codes
  - Kein Codewort darf Präfix eines anderen sein => Keine Trennzeichen beim Speichern benötigt
  - Optimal bedeutet es gibt keine Kodierverfahren mit kürzerem Code

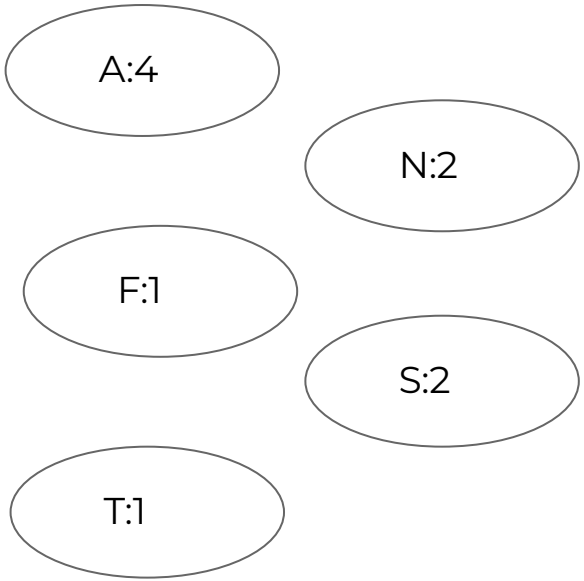
Lösungsansatz: Häufigkeitsanalyse

Häufigkeitsanalyse

Beispiel: ANANASSAFT

Zeichen	A	F	N	S	T
Häufigkeit	4	1	2	2	1

Einfügen in den Heap

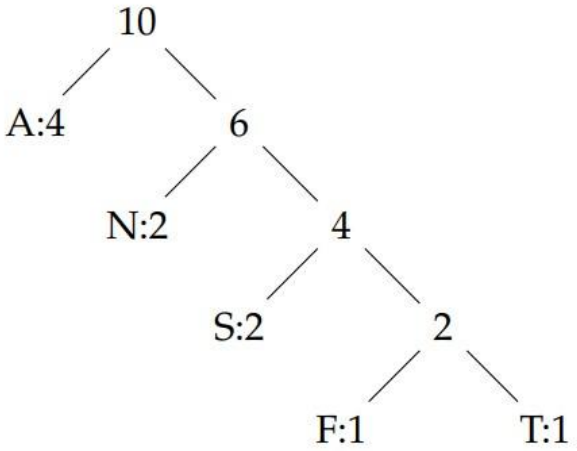


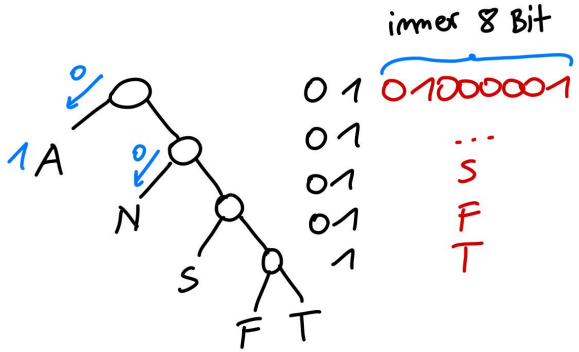
Step-by-step Heapify!

Lösungsansatz: Baum

Lösungsansatz: Baumencoding

Der Huffman Baum:





Interne Repräsentation der Knoten:

```
struct Node {
        char character;
        uint16_t frequency;
        Node *left;
        Node *right;
};
```

Lösungsansatz: Dictionary Übersetzung

Huffman Dictionary Erstellung:

```c
void tree_to_dic(Node *root, uint8_t *length_table, uint32_t *lookup_table, uint32_t path, uint8_t len)
{
    if (!root || !length_table || !lookup_table) {
        return;
    }

    tree_to_dic(root->left, length_table, lookup_table, path << 1, length + 1);

    if (root->character) {
        length_table[(uint8_t) root->character] = length;
        lookup_table[(uint8_t) root->character] = path;
    }

    tree_to_dic(root->right, length_table, lookup_table, path << 1 | 1, length + 1);
}
```

Dictionary:

Symbol	Bitsequenz
A	0
F	1110
N	10
S	110
T	1111

Lösungsansatz: Kodierung

Dictionary

Symbol	Bitsequenz
A	0
F	1110
N	10
S	110
T	1111

Die Codierte Nachricht:
ANANASSAFT = 0 10 0 10 0 110 110 0 1110 1111

Korrektheit: Kodierungstheorie

- *Quell-Alphabet* der Länge n (ASCII Zeichen)
- *Code-Alphabet* {0,1}
- *Gewichte* W =⟨$w_i > 0 \mid 0 \leq i < n$⟩zählen die absolute Häufigkeit der Zeichen
- *Code* T =⟨$l_i > 0 \mid 0 \leq i < n$⟩ordnet jedem Zeichen des Quell-Alphabets ein *Codewort* der Länge l_i zu.

Kraft-McMillian-Ungleichung:

$$\mathcal{K}(T) = \sum_{i=0}^{n-1} 2^{-l_i} \leq 1$$

Kosten eines Codes:

$$C(W,T) = C(\langle w_i \rangle, \langle l_i \rangle) = \sum_{i=0}^{n-1} w_i \cdot l_i$$

Korrektheit: Optimalität

Beweis der Optimalität des Huffman Codes per Induktion:

Induktionsbasis (n = 2): Es existiert nur der Code$\langle l_i \rangle = \langle 1, 1 \rangle$, nämlich die beiden Codewörter 0 und 1

Induktionsannahme: Der Huffman Code H hat die gleichen Kosten wie ein optimaler Code T bei (n-1)-Zeichen

Induktionsschritt:

Lemma: Es existiert ein optimaler Code, wo die zwei Knoten mit den geringsten Gewichten Geschwister sind

Bildung neuer Codes H’ und T’ der Länge (n−1), durch entfernen der zwei geringsten Gewichte und ersetzen des Elternknoten durch die Summe der Gewichte. Die Kosten der Codes sinken um diese Summe.

Beweis: $$C(H) = C(H') + w_{n-2} + w_{n-1} = C(T') + w_{n-2} + w_{n-1} = C(T)$$

Korrektheit: Kompressions-Effektivität

Entropie:

$$\mathcal{H}(W) = -\sum_{i=0}^{n-1} w_i \cdot log_2 \frac{w_i}{m}$$

Relativer Effektivitätsverlust:

$$\mathcal{E}(W,T) = \frac{C(W,T) - \mathcal{H}(W)}{\mathcal{H}(W)}$$

- Meist kommt der Huffman Code sehr nah an die entropischen Kosten und hat somit einen Effektivitätsverlust von wenigen Prozenten.
- Wenn w_0 im Vergleich zur Summe der anderen Zeichen sehr groß wird, ist der relative Effektivitätsverlust deutlich höher

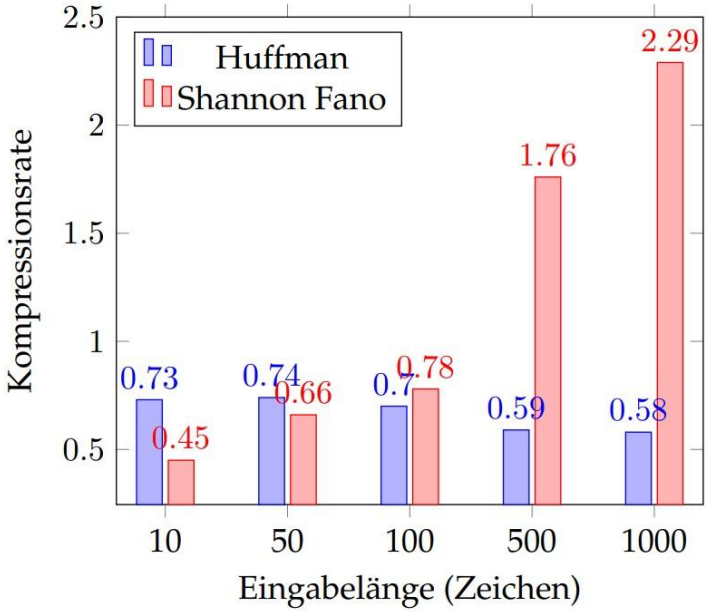
Beispiel:

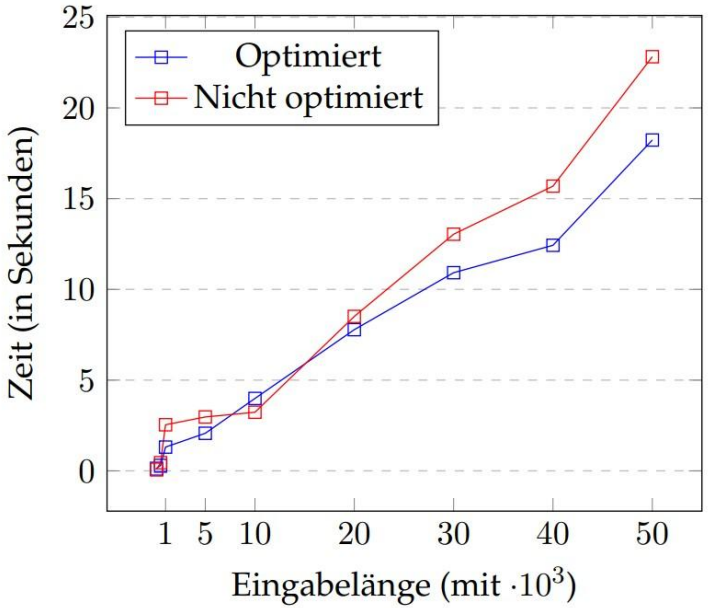
W =⟨96, 1, 1, 1, 1⟩mit H(W) = 32.2 Bits

T =⟨1, 3, 3, 3, 3⟩mit C(W, T) = 108 Bits

E(W, T) = 235%

Performanzanalyse





Performanzanalyse

Schneller Laufzeit durch:

- Einsparen von unnötigen strlen() Ausführungen → Direktes Zählen im Code
- Häufigkeitanalyse zuerst im Array zählen und anschließend in Knoten umformen
- Enkodierung der Buchstaben nicht entlang des Baumes sondern mithilfe eines Dictionaries

Bessere Speichereffizient durch:

- In-place Priority-Queue
- Bithacks (Shifts, Masken, etc.)
- Begrenzung des Speichers
- Nutzung von maximale 32 Bit pro Encoding
- Maximale Nutzung von Arrays anstatt Datenstrukturen

Weiter Optimierungen:

- Nur nicht/nur schwierig umsetzbar bei Baumerstellung durch Heap

Zusammenfassung und Ausblick

Huffman Codes

- Immer noch sehr relevant
- Meistens knapp an den entropischen Kosten
- Symbol-basierend und daher beschränkt
  - z.B Huffman Baum für Wörter statt Zeichen
- Anpassung an das Speicherformat
  - PNG für Bilder
  - FLAC für Audio

Implementierung

- Min-heap anschaulich, aber ineffizient
  - O(n) möglich, sogar in-place
- Optimierung des Speicherformats
  - Direkt binär speichern
- Parallelisierung

Zusammenfassung und Ausblick

Danke für die Aufmerksamkeit!