Nutzung gewichteter Binärbäume zur Komprimierung

Huffman-Codierung: Verlustfreie Komprimierung unter Nutzung gewichteter Binärbäume

Daten möglichst redundanzfrei ablegen/ übertragen zu können (doppelte Zeichen möglichst nicht doppelt oder zumindest kürzer als andere kodieren):
→ die einzelnen Zeichen mit Codes unterschiedlicher Binärlängen kodieren (Ascii: jedes Zeichen = 8 Bit):

- je häufiger ein Zeichen im Text vorhanden ist, je kürzer sollte seine Codierung sein!
- um einen Code auch wieder eindeutig dekodieren zu können, darf kein Code(wort) der Beginn eines anderen sein

Die Grundidee ist, einen *gewichteten Baum* für die Darstellung des Codes zu verwenden.

Nutzung gewichteter Binärbäume zur Komprimierung

Huffman-Baum:

Blätter = zu kodierende Zeichen
Pfad von der Wurzel zum Blatt = Codesymbol

Baum wird von den Blättern zur Wurzel (*bottom-up*) erstellt.

Huffman-Codierung benötigt keine Trennzeichen, obwohl die Zeichencodes unterschiedlich lang sind.

Der bei der Huffman-Kodierung gewonnene Baum liefert garantiert eine optimale und präfixfreie Kodierung.

D. h.: es existiert (bezogen auf die Eingangsdaten) kein Kodierverfahren, das einen kürzeren Code generieren könnte.

Huffman Codierung am Beispiel

Bsp.: “Mississippi”, ascii codiert: 11x8=88 bit

Definitionen:

X ist der Zeichenvorrat, aus dem die Eingangsdatenmenge besteht: ‚M‘, ‚i‘, ‚s‘, ‚p‘

px ist die Wahrscheinlichkeit des Symbols x (die relative Häufigkeit): ‚M‘: 0.09, ‚i‘: 0.36, ‚s‘: 0.36, ‚p‘: 0.18

C ist das Codealphabet – der Zeichenvorrat, aus dem die Codewörter bestehen: {0, 1}

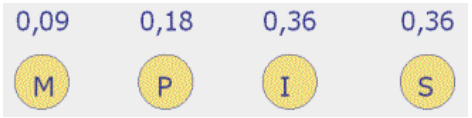
m ist die Mächtigkeit des Codealphabetes C – die Anzahl der verschiedenen Zeichen: 2

Huffman Codierung am Beispiel

Bsp.: “Mississippi”, ascii codiert: 11x8=88 bit

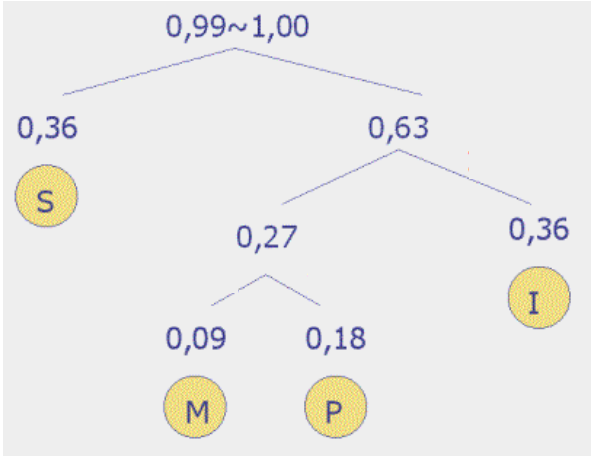
Aufbau des Baumes:

1. Ermittle für jedes Quellsymbol die relative Häufigkeit px, d. h. zähle, wie oft jedes Zeichen vorkommt, und teile durch die Anzahl aller Zeichen.



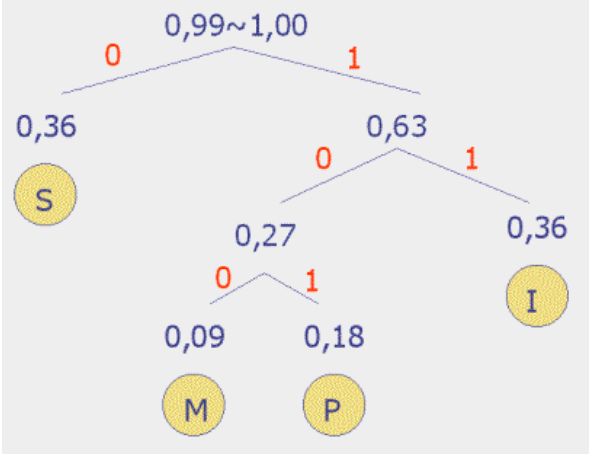
2. Erstelle für jedes Quellsymbol einen einzelnen Knoten (die einfachste Form eines Baumes/ Teilbaums) und notiere im/am Knoten die Häufigkeit.

3. Wiederhole die folgenden Schritte so lange, bis nur noch ein Baum übrig ist:
   1. Wähle die m Teilbäume mit der geringsten Häufigkeit in der Wurzel, bei mehreren Möglichkeiten die Teilbäume mit der geringsten Tiefe.
   2. Fasse diese Bäume zu einem neuen binären (Teil-)Baum zusammen.
   3. Notiere die Summe der Häufigkeiten in der Wurzel.



Huffman Codierung

Bsp.: Mississippi



Konstruktion des Codes

1. Ordne jedem Kind eines Knotens eindeutig ein Zeichen aus dem Codealphabet zu (Gewicht).

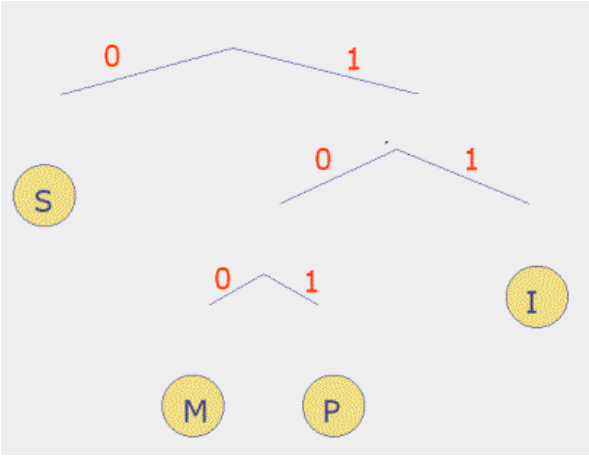
2. Lies für jedes Quellsymbol (Blatt im Baum) das Codewort aus:
   1. Beginne an der Wurzel des Baums.
   2. Die Codezeichen auf den Kanten des Pfades (in dieser Reihenfolge) ergeben das zugehörige Codewort.
      ‚S': 0 ‚I': 11 ‚M': 100 ‚P': 101 (seltenere Buchstaben: längere Codierung)

3. Codewort („Mississippi"):
   Setze das Codewort aus den Codezeichen zusammen: 100110011001110110111 21 Bit! (statt 88)
   M i ss i ss i p p i

Huffman Decodierung

Für die Dekodierung muss der Codierungsbaum mitgeschickt/ mitgespeichert werden, z.B. im Header der komprimierten Datei.
Das Verhältnis aus Datenmenge und Baumgrösse muss dafür adäquat sein – für sehr kleine Texte lohnt es sich nicht.

Der Huffman-Baum dient dann auch der Dekodierung:



10011001100111011011 entziffern:

1. Wir interpretieren auf Basis des Baumes die Ziffern 0 und 1 als Kanten, und folgen diesen, bis wir zu einem Blatt kommen.
   Das ist der Fall, wenn wir dem Pfad 1-0-0 gefolgt sind.
   Das Blatt das wir so erreicht haben trägt das Zeichen `M`.

2. Nun starten wir erneut bei der Wurzel des Baums und folgen den Kanten, den die Ziffernfolge unseres codierten Wortes vorgeben.
   Wir folgen also den Kanten 1-1 und kommen wieder an ein Blatt, das jetzt das Zeichen `I` trägt…

Zusammenfassung Graphen

Graphen sind eine extrem mächtige Datenstruktur zur Verwaltung von Daten mit Beziehungen!

Algorithmen:

- Minimum Spanning Tree,
- Binary search,
- Topological Sort,
- Bidirectional Search,
- Floyd- Warshall Algorithmus,
- Bellman-Ford-Algorithmus,
- Graph Coloring …

Die Effizienz kann jedoch auch noch in anderer Hinsicht als mit Blick auf die Laufzeit gemessen werden.

Es gibt manchmal größere Probleme als die Geschwindigkeit, und dann wir müssen uns mehr darum kümmern, wieviel Speicher eine Datenstruktur oder ein Algorithmus benötigt.

Im nächsten Abschnitt analysieren wir deshalb die Effizienz unseres Codes im Hinblick auf Speicherbedarf!