# Aufgabe 1: LaTeX-Dokument

Teilnahme-ID: ?????

Bearbeiter/-in dieser Aufgabe:
Vor- und Nachname

26. April 2025

## Inhaltsverzeichnis



**Anleitung:** Trage oben in den Zeilen 8 bis 10 die Aufgabennummer, die Teilnahme-ID und die/den Bearbeiterin/Bearbeiter dieser Aufgabe mit Vor- und Nachnamen ein. Vergiss nicht, auch den Aufgabennamen anzupassen (statt „LaTeX-Dokument“)!

Dann kannst du dieses Dokument mit deiner LaTeX-Umgebung übersetzen. Die Texte, die hier bereits stehen, geben ein paar Hinweise zur Einsendung. Du solltest sie aber in deiner Einsendung wieder entfernen!

# 1 Lösungsidee

TODO vlt kurzfassung/zusammenfassung

## 1.1 Huffman Codierung

David Huffman hat 1952 eine Methode veröffentlicht, die heute als *Huffman Codierung* bekannt ist.[1] Er stellte eine Methode vor, um eine optimale Präfixfreie Codierung für einen bestimmten Text zu finden.

[1] *Huffman-Kodierung.*

Dabei geht man von einem Alphabet $A$ mit $n$ verschiedenen Zeichen $a_1, \ldots, a_n \in A$ aus, gemeinsam mit der Häufigkeitsverteilung der Zeichen $p_i = \frac{\text{Häufigkeit von } a_i \text{ im zu codierenden Text}}{\text{Länge des Textes}}$. Man nennt diese Antreffwahrscheinlichkeit $p_i$ eines Zeichens auch *Frequenz* oder *Gewichtung*. Wir nehmen an, dass das Alphabet nach absteigender Häufigkeit sortiert ist ist, also $p_i \geq p_{i+1}$ für alle $0 \leq i < n-1$. Der Text soll mit einem Ausgabealphabet $O$ codiert werden, das aus $r$ verschiedenen Zeichen $o_1, \ldots, o_r \in O$ besteht. In der Bwinf-Aufgabenstellung besteht dieses Ausgabealphabet $O$ aus den $r$ verschiedenen Perlen (die aber alle den gleichen Durchmesser haben). Um den ursprünglichen Text zu codieren, müssen wir nun jedem Buchstaben $a_i$ ein *Codewort* $w_i$ zuordnen, dass aus einer Kette an Buchstaben aus dem Ausgabealphabet besteht $w_i = o_j o_k o_l \ldots$.
Diese Codierung soll *Präfixfrei* sein, also kein Codewort soll Teil eines anderen Codeworts sein. Dadurch kann der Text als Aneinanderreihung von Codewörtern (ohne „Komma“ dazwischen) übertragen und eindeutig decodiert werden.
Aufgrund von dieser Eigenschaft können wir den Code als einen Baum darstellen, der $n$ Blätter hat und bei dem jeder Knoten höchstens $r$ Kinder hat.

**Definition 1.** *Wir nennen einen solchen Baum mit n blättern und höchstens r Kindern pro Knoten* ***valide****, da er eine mögliche Codetabelle darstellt.*

Jedes Blatt eines solchen Baums repräsentiert ein Codewort $w$, dass eindeutig durch den Pfad von der Wurzel des Baums zu diesem Blatt definiert ist. Der Pfad wird durch die Kanten des Baums definiert, die mit den Perlen beschriftet sind. Da alle Perlen gleich groß sind, also die Länge aller Buchstaben des Ausgabealphabets gleich ist, ist die Beschriftung der Kanten beliebig. Es ist lediglich wichtig, dass alle Kanten eines Knotens mit unterschiedlichen Buchstaben beschriftet sind. Die Länge des Codeworts $|w|$ entspricht der Anzahl der Kanten auf dem Pfad von der Wurzel zu diesem Blatt.
Da wir eine optimale Codetabelle (also eine möglichst kurze Perlenkette) erstellen wollen, müssen wir die „Kosten“ eines Baums definieren. Diese Kosten eines Baums hängen natürlich auch davon ab, welches Codewort welchem Buchstaben zugeordnet wird. Für diese Definition gehen wir davon aus, dass wir eine solche Zuordnung $w_i \to a_i$ haben.

**Definition 2.** *Die Kosten eines Baums T ist definiert als das Produkt der Länge jedes Codeworts $w_i$ mit der Frequenz $p_i$ des codierten Buchstabens $a_i$:*

$$cost(T) = \sum_{i \leq n}^{i=0} |w_i| \cdot p_i$$

Da die Kosten eines Baumes proportional zu der Länge der resultierenden Perlenkette sind, kann die Aufgabenstellung darauf reduziert werden, den validen Baum $T$ mit den minimalen Kosten zu finden.
Dafür ist aber auch die Zuordnung, welcher Buchstabe von welchem Codewort codiert werden soll, entscheidend. Für diese Aufgabenstellung ist die Zuordnung aber für jeden Baum eindeutig, da wir die Kosten des Baums minimieren wollen. Um das zu erreichen, müssen wir das längste Codewort zu dem Buchstaben zugeordnet, der am seltensten vorkommt und das kürzeste Codewort zu dem Buchstaben zuordnen, der am häufigsten vorkommt. Dafür ordnen wir im Folgenden die Blätter Baumes in aufsteigender Reihenfolge, also $|w_0| \leq \cdots \leq |w_n|$. Da das Alphabet nach absteigender Reihenfolge sortiert ist, also $p_0 \geq \cdots \geq p_n$, liefert die Zuordnung von $w_i$ zu $a_i$ für jeden Baum die geringsten Kosten.

**Definition 3.** *Wir nennen einen solchen Baum, der die minimalen Kosten (kürzeste Perlenkette) für eine bestimmte Häufigkeitsverteilung hat* ***optimal***

### 1.1.1 Konstruktion des optimalen Baums

**Definition 4.** *Wir nennen einen r-när Baum, bei dem jeder Knoten maximal r Kinder hat* ***vollständig****, wenn jeder Knoten genau r Kinder hat.*

**Lemma 5.** *Ein optimaler binärer Huffman Baum ist vollständig. TODO braucht man das??*

*Beweis.* Ein optimaler binärer Huffman Baum ist im allgemeinen vollständig, da in einem Optimalen Baum kein Knoten mit nur einem Kind existieren kann. Würde man diesen Knoten zu einem Blatt machen hätte man einen Baum mit der gleichen 3Anzahl an Blättern aber geringeren Kosten. □

Das gilt im allgemeinen Fall bei Bäumen mit $r > 2$ Kindern aber nicht.
Für die allgemein bekannte Konstruktion (TODO Quelle) eines binären Huffman Baums werden wiederholt zwei Teilbäume zu einem zusammengefasst, bis schließlich nur noch einer übrig ist. Im allgemeinen Fall müssen immer $r$ Teilbäume zu einem zusammengefasst werden, was aber nicht für jede Anzahl an Buchstaben $n$ im Alphabet immer funktioniert.

**Beispiel 6.** *Für $n = 4$ werden im ersten Schritt drei Knoten zu einem zusammengefasst. Im nächsten sind nur noch zwei Knoten übrig. Würde man diese Knoten nun zu einem fertigen Baum kombinieren wäre dieser für bestimmte Häufigkeitsverteilungen (z.B. $p_0 = p_1 = p_2 = p_3$) nicht optimal.*

Daher muss das Alphabet zuerst mit so vielen Platzhalterbuchstaben, deren Antreffwahrscheinlichkeit 0 ist, aufgefüllt werden, dass gilt: $n \mod (r-1) = 1$.
Jetzt können wir mit der Konstruktion beginnen:

1. Erstelle für jeden Buchstaben $o_i$ im Alphabet (inklusive Platzhalter) einen Teilbaum (der aus einem Wurzelknoten besteht) und beschriften diesen mit der Frequenz $p_i$ des Buchstabens $o_i$.
2. Wähle die $r$ Teilbäume mit den kleinsten Antreffwahrscheinlichkeiten. (Diese Wahl ist nicht eindeutig)
3. Beschrifte einen neuen Knoten mit der Summe der Antreffwahrscheinlichkeiten der $r$ gewählten Teilbäume und erstelle einen neuen Teilbeum mit dem neuen Knoten als Wurzel und den gewählten Teilbäumen als Kinder.
4. Wieder hole Schritt 2 und 3, bis nur noch ein Baum übrig ist.

TODO Bilder
TODO QUelle für allgemeine Konstruktion

### 1.1.2 Beweis der optimalität

Der Standartbeweis für die Optimalität von Binären Huffman Bäumen kann leicht verallgemeinert werden, daher Dazu müssen wir zuerst das „Sibling-Lemma“ auf $r-$när Bäume verallgemeinern:

**Lemma 7.** *Seien $l_0, l_1, \dots l_{r-1}$ die $r$ Buchstaben mit der geringsten Antreffwahrscheinlichkeit. Sie sind in einem Huffman Baum „Geschwister“, also Kinder des gleichen Knotens $Y$ und liegen auf der untersten Ebene des Baums.*

*Beweis.* Die Buchstaben $l_0, l_1, \dots l_{r-1}$ sind sicher Geschwister, da sie als erstes ausgewählt werden.
Wenn $l_0, l_1, \dots l_{r-1}$ nicht auf der unteresten Ebene des Baums liegen würden, dann müsste es einen Internen Knoten $X$ geben, dessen Antreffwahrscheinlichkeit kleiner ist als die des Knotens $Y$, dessen Kinder $l_0, l_1, \dots l_{r-1}$ sind. Dafür müsste aber mindestens einer der Kinder von $X$ eine geringere Antreffwahrscheinlichkeit haben als einer der Kinder $l_0, l_1, \dots l_{r-1}$ von $Y$. Das wiederspricht aber unserer Anfänglichen Annahme, das $l_0, l_1, \dots l_{r-1}$ die $r$ Buchstaben mit der geringsten Antreffwahrscheinlichkeit sind. Daher können wir sagen, dass $l_0, l_1, \dots l_{r-1}$ immer auf der untersten Ebene des Baums liegen. □

Damit können wir nun mithilfe von Induktion über die Größe $n$ des Alphabets den Beweis durchführen:

**Satz 8.** *Die oben beschrieben Konstruktion für $r-$näre Huffman Bäume liefert einen optimalen Baum und damit eine bestmögliche Codetabelle.*

*Beweis. Induktionsanfang*: Für $n \leq r$ ist ein Huffman Baum offensichtlich optimal.
*Induktionshypothese*: Alle Huffman Bäume mit $< n$ Blättern sind optimal.
*Induktionsschritt*: Nun müssen wir zeigen, ein Huffman Baum $T$ mit $n$ Blättern optimal ist. Seien $l_0, l_1, \dots l_{r-1}$ wieder die $r$ Buchstaben mit der geringsten Antreffwahrscheinlichkeit. Sei $Y$ der Knoten in $T$, dessen Kinder $l_0, l_1, \dots l_{r-1}$ sind. Sei $T'$ nun ein Huffman Baum identisch zu $T$, bei dem $Y$ zu einem Blattknoten konvertiert wurde (mit der gleichen Antreffwahrscheinlichkeit wie der Knoten $Y$ in $T$). Aufgrund der Induktionshypothese ist $T'$ optimal, da er $n - (r-1) < n$ Blätter hat.
Wenn $T$ nicht optimal wäre, gäbe es einen Baum $T_1$ mit geringeren Kosten als $T$. Aus $T_1$ könnte man wieder einen Baum $T_1'$ mit $n - (r-1)$ Blättern erstellen, in dem man $Y$ zu einem Knoten konvertiert. Wenn $T_1$ geringere Kosten als $T$ hätte, hätte auch $T_1'$ geringere Kosten als $T'$, woraus man schließen kann, dass $T$ optimal sein muss. □

### 1.1.3 Pseudocode

**Algorithm 1** ErstelleHuffmanBaum (r, A, p)

```
1:  function ErstelleHuffmanBaum(r, A, p)
2:      /* r: max. Kinder pro Knoten; A Alphabet; p[a]: Frequenz von Symbol a */
3:      Initialisiere MinHeap H
4:      for all a ∈ A do
5:          Erzeuge Blattknoten node mit node.symbol=a und node.freq=p[a]
6:          H.push(node)
7:      end for
8:      n ← |A|,    m ← 0
9:      while (n + m) mod (r − 1) ≠ 1 do
10:         Erzeuge Platzhalter-Blatt z mit z.freq = 0
11:         H.push(z)
12:         m ← m + 1
13:     end while
14:     while H.size() > 1 do
15:         /* Ziehe die r Bäume mit der kleinsten Frequenz aus dem Heap */
16:         C ← [ ]
17:         for i = 1 to r do
18:             C.append(H.pop())
19:         end for
20:         /* Erzeuge neuen inneren Knoten als Wurzel dieser r Kinder */
21:         new_node.freq ← Σ_{c∈C} c.freq
22:         new_node.children ← C
23:         H.push(new_node)
24:     end while
25:     return H.pop()                          ▷ Wurzel des fertigen Huffman-Baums
26: end function
```

### 1.1.4 Laufzeit

Die Laufzeitbetrachtung ist ähnlich wie bei binären Huffman Bäumen:
Sei $n$ die Länge des Alphabets und $t$ die Länge des Originaltextes. Um die Frequenzen der Buchstaben zu bestimmen, muss der Text einmal durchlaufen werden. Diese Funktion, die im Pseudocode oben nicht aufgeführt ist, hat also eine Asymptotische Laufzeit von $O(t)$. Die For-Schleife, die für jeden Blattknoten einen Buchstaben erzeugt hat eine Laufzeit von $O(n)$. Da $n < t$ erhöht diese For-Schleife die asymptotische Laufzeit aber nicht. Die While-Schleife, die Platzhalterknoten einfügt, iteriert höchstens $(r-1)$ mal. Da $r < n$ (sonst wäre die Codierung trivial) trägt diese While-Schleife auch nicht zur Asymptotischen Laufzeit bei.
In der Haupt-While-Schleife wird der Heap mit einer anfänglichen Größe von $O(n)$ in jeder iteration um $r-1$ verkleinert, sie hat also eine Laufzeit von $O(\frac{n}{r})$. In der Schleife werden $r$ Knoten aus dem Heap gezogen und einer wieder eingefügt, jeweils mit einer Laufzeit von $O(\log n)$. Der innere Teil der While-Schleife hat also eine Laufzeit von $O(r \cdot \log n)$ und damit gilt für die Laufzeit der While-Schleife: $O(n \cdot \log n)$. Das gesamte Programm hat also eine asymptotische Laufzeit von $O(t + n \cdot \log n)$

## 1.2 Ungleiche Perlengrößen

TODO entweder bei 1 anfangen und bis r oder bei 0 anfagnen und bis r-1 Im Aufgabenteil b) haben die Perlen unterschiedliche, ganzzahlige Durchmesser $c_1, \ldots, c_r$. Wir sortieren die Perlen ab jetzt so, dass gilt: $c_1 \leq c_2 \leq \cdots \leq c_r = C$ und wir bezeichnen den Durchmesser der größten Perle $c_r$ mit $C$.
Wir eine Codetabelle, die mit unterschiedlich großen Perlen erstellt wurde ähnlich wie in Aufgabenteil a) wieder als Baum darstellen.

**Definition 9.** *Wir bezeichnen die Tiefe eines Knotens $V$ mit $depth(V)$. Die Tiefe des Wurzelknotens ist 0.*

Bis jetzt waren die Kinder $v_0$ bis $v_{r-1}$ eines Knotens $V$ immer genau eine Ebene tiefer als $V$, also $depth(v_i) = depth(V) - 1$. Wir wollen die Eigenschaft, dass die Pfadlänge von der Wurzel eines Baumes

zu einem Blatt $|w|$ der Länge der entsprechenden Aneinanderreihung an Perlen entspricht aber natürlich weiterhin erhalten. Wenn wir also die Kinder $v_0$ bis $v_{r-1}$ des Knotens $V$ mit den Perlen $o_1$ bis $o_r$ beschriften $v_i \to o_i$, soll gelten: $depth(V) = depth(v_i) + c_i$.

### 1.2.1 Huffman

Da die oben beschriebene Huffman-Codierung den Spezialfall $c_1 = c_2 = \cdot = c_r$ darstellt, ist es zunächst sinnvoll, zu betrachten, weshalb der Huffman Algorithmus hier keinen optimalen Baum liefert. TODO (Die tiefsten Knoten sind nicht zwingend Nachbarn, weil c1 = c2 etc)

### 1.2.2 Vollständige Bäume

Golin und Rote stellen 1998 in ihrem Paper Golin und Rote, "A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs" einen neuen Ansatz zur Berechnung von optimalen Präfixfreien Codes für unterschiedliche Kosten der Buchstaben vor. Da das dem Problem in Aufgabenteil b) entspricht, orientiere ich mich in den folgenden Kapiteln an diesem Paper. Es ist aber anzumerken, dass meine Algorithmen nicht genau denen im Paper entsprechen, sondern durch einen etwas anderen Ansatz in der Praxis deutlich schneller sind. Am Ende diskutiere ich diese Unterschiede noch genauer.
Da es in diesem Aufgabenteil also nicht möglich ist, die Bäume von unten nach oben mit einem Greedy-Algorithmus zu erstellen, müssen wir alle mögliche Lösungen betrachten. In diesem Kapitel werden wir uns zuerst auf vollständige Bäume beschränken, indem wir ähnlich wie in Teil a) fehlende Buchstaben durch Platzhalter mit Frequenz 0 ersetzen:

**Definition 10.** *Für jeden Baum $T$ erhält man einen vollständigen Baum $Fill(T)$, indem man an jeden Knoten so viele zusätzliche Blätter mit Frequenz 0 hinzufügt, das dieser genau $r$ Blätter besitzt.*

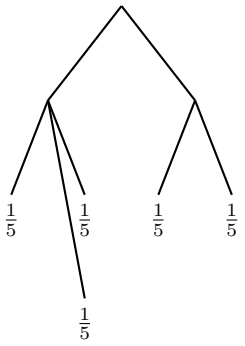



Abbildung 1: Ein nicht vollständiger Baum $T$

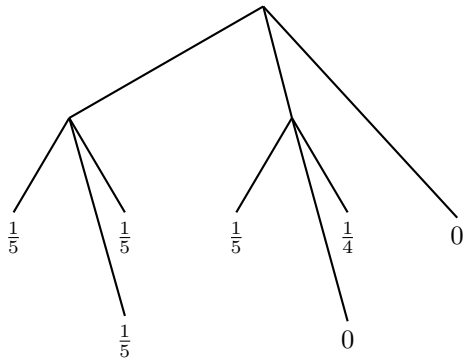



Abbildung 2: Der vervollständige Baum $Fill(T)$

**Lemma 11.** *Sei $T$ ein optimaler Baum mit $n$ Blättern. Die Anzahl der Blätter $m$ von $Fill(T)$ beträgt höchstens $m \leq n(r-1)$.*

*Beweis.* TODO □

Da $cost(Fill(T)) = cost(T)$ offensichtlich gilt, können wir nun das Problem umformulieren:
Finde für die Perlendurchmesser $c_1, \ldots, c_r$ und die Frequenzen $p_1 \geq \cdots \geq p_n$ (und $p_i = 0$ für alle $i > n$) den vollen Baum $T_{opt}$ mit $m$ Blättern ($n \leq m \leq n(r-1)$) mit minimalen Kosten:

$$cost(T_{opt}) = \min\{cost(T) : T \text{ ist vollständig und hat } m \text{ Blätter } (n \leq m \leq n(r-1))\}$$

Wenn wir $T_{opt}$ konstruiert haben, können wir die Blätter mit Frequenz 0 entfernen und erhalten eine optimale Codierungstabelle für $n$ Buchstaben.
Dafür müssen wir aber zuerst betrachten, wie wir vollständige Bäume allgemein darstellen und Ebene für Ebene konstruieren können.

**Definition 12.** *TODO braucht man des und wenn ja hier? Wir nennen einen Baum Ebene-i-Baum, wenn alle internen Knoten auf einer Ebene $\leq i$ liegen.*

**Definition 13.** *Wir kürzen einen Baum $T$ zu dem Ebene-$i$-Baum $Cut_i(T)$, indem wir alle Knoten entfernen, deren Eltern Tiefer als Ebene-$i$ liegen:*

$$Cut_i(T) = T - \{v \in T \mid |parent(v)| > i\}$$

TODO notation für |Tiefe| einführen TODO Beispiel

**Definition 14.** *Die Signatur einer Ebene $i$ des Baums $T$ ist das $C+1$-Tupel*

$$sig_i(T) = (m; l_1, l2, \ldots, l_C)$$

*wobei $m = |\{v \in T \mid v \text{ ist ein Blatt, } tiefe(v) \leq i\}|$ der Anzahl der Blätter von $T$ mit einer Tiefe von höchstens $i$ enspricht und*

$$l_k = |\{u \in T \mid tiefe(u) = i + k\}, k \in \{1, \ldots, C\}$$

*die Anzahl der Knoten auf Ebene $i + k$ ist.*

Diese Signatur wird es uns später erlauben, alle möglichen Bäume zu betrachten und dabei aber kleine Unterschiede, die nichts an den Kosten eines Baums ändern, zu vernachlässigen.

TODO Beispiele mit signatur und expand

Über die Signatur einer eines Ebene-$i$-Baums können wir nun die Kosten dieser Ebene definieren, was uns später dabei helfen wird, den „Billigsten“ Baum zu finden.

**Definition 15.** *Sei $T$ ein Ebene-$i$ Baum mit der Signatur $sig_i(T) = (m, l_1, l_2, \ldots, l_C)$. Für $m \leq n$ definieren wir die Kosten der Ebene-$i$ als:*

$$cost_i(T) = \sum_{j=1}^{m} |w_j| \cdot p_j + i \cdot \sum_{j=m+1}^{n} p_j$$

*wobei $|w_1| \leq \cdots \leq |w_m|$ die $m$ höchsten Blätter nach Tiefe geordnet sind.*

Der erste Term der Summe beschreibt Kosten der Blätter, für die bereits sicher ist, dass es sich um Blätter handelt, während der zweite Term die Kosten für die übrigen Blätter um die Ebene $i$ zu erreichen darstellt. Für $m \geq n$ erübrigt sich der zweite Term, da $p_i = 0$ für $i > m$. D.h. wenn $T$ ein Ebene-$i$-Baumm mit $sig_i(T) = (m, l1, \ldots l_C)$ ist und $m \geq n$, dann gilt $cost_i(T) = cost(T)$.
Um den „Billigsten“ Baum zu finden, konstruieren wir die Bäume Ebene für Ebene, beginnend bei dem *Wurzel-Baum*, der nur aus der Wurzel und deren Kindern besteht. Um die Signatur des Wurzelbaums darzustellen müssen wir zunächst den charakteristischen Vektor einer Perlensammlung definieren:

**Definition 16.** *Der charakteristische Vektor $(d_1, d_2, \ldots, d_C)$ einer Perlensammlung mit den Durchmessern $C = (c_1, c_2, \ldots, c_r)$ entspricht den Anzahlen, wie oft ein bestimmter Durchmesser in der Perlensammlung vertreten ist:*

$$d_i = \textit{Anzahl von } i \textit{ in } C$$

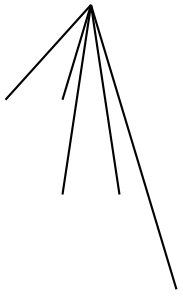
**Beispiel 17.** *Der charakteristische Vektor der Beispieldatei 7 mit den Durchmessern $C = (1, 1, 1, 1, 1, 1, 1, 2, 3, 4)$ beträgt $(7, 1, 1, 1)$ und der charakteristische Vektor der Beispieldatei 5 mit den Durchmessern $C = (1, 1, 2, 2, 3)$ beträgt $(2, 2, 1)$.*

**Definition 18.** *Der **Wurzel-Baum** $T_0$ ist ein Ebene-0-Baum, der aus dem Wurzelknoten und dessen $r$ Kindern, die alle Blätter sind, besteht. Die Signatur des Wurzel-Baums ist:*

$$sig_0(T_0) = (0, d_1, d_2, \ldots d_C)$$

*und $cost_0(T_0) = 0$.*

**Beispiel 19.** *Der Wurzel-Baum der Beispieldatei mit den Perlendurchmessern $C = (1, 1, 2, 2, 3)$ und*



*dem charakteristischen Vektor $D = (2, 2, 1)$ sieht folgendermaßen aus:*

Sei $T$ also ein Ebene-$i$-Baum und $T'$ ein Ebene-$i+1$-Baum, der bis auf Ebene-$i$ identisch zu $T$ ist, also: $Cut_i(T') = T$. $T$ und $T'$ haben also die selbe Anzahl an Knoten auf den Ebenen 0 bis $i$. Daher ist auch die Anzahl der Knoten auf Ebene $i+1$ identisch, der einzige Unterschied besteht darin, ob diese Knoten Blätter oder interne Knoten sind. In $T$ sind alle diese Knoten Blätter, da $T$ ein Ebene-$i$-Baum ist. In $T'$ könnten auch alle Knoten auf Ebene $i+1$ Blätter sein, es könnte aber auch eine bestimmte Anzahl $q$ an Knoten interne Knoten sein, maximal aber natürlich so viele, wie $T$ Blätter in Ebene $i+1$ hat. Sei $sig_i(T) = (m, l_1, \ldots, l_C)$ die Ebene-$i$ Signatur von $T$. Damit können wir sagen, dass $0 \leq q \leq l_1$, da $l_1$ die Anzahl der Blätter von $T$ auf Ebene $i+1$ ist.

**Definition 20.** *Sei $T$ ein Ebene-$i$-Baum mit der Signatur $sig_T = (m; l_1, l_2, \ldots, l_C)$. Die Expand Operation wandelt $0 \leq q \leq l_1$ Blätter auf Ebene $i+1$ in interne Knoten um, indem $r$ Kinder an jedes dieser Blätter angehängt werden:*

$$T' = Expand_i(T, q)$$

Die Expand-Operation legt nicht fest, welche der $l_1$ Blättern zu Knoten werden. Da wir uns auf vollständige Bäume beschränken, sind die Kosten dieser Bäume aber gleich, selbst wenn die Bäume selbst geringfügige Unterschiede haben. Um die Änderung der Signatur des Baums zu quantifizieren, können wir den charakteristischen Vektor einer Perlensammlung verwenden:

**Lemma 21.** *Sei $T$ ein Ebene-$i$-Baum mit der Signatur $sig_T = (m; l_1, l_2, \ldots, l_C)$ und $T' = Expand_i(T, q)$ die Erweiterung um $q$. Die Signatur von $T'$ ist dann:*

$$sig_{i+1}(T') = (m + l_1, l_2, \ldots, l_C, 0) + q \cdot (-1, d_1, d_2, \ldots, d_C)$$

*(Multiplikation und Addition werden Elementweise ausgeführt). Die Kosten der Erweiterung betragen:*

$$cost_{i+1}(Expand_i(T, q)) = cost_i(T) + \sum_{m < j \leq n} p_j$$

*Beweis.* TODO □

**Beispiel 22.** *TODO*

**Definition 23.** *Wir bezeichnen zwei Bäume $T_1$ und $T_2$ als equivalent, wenn sie die gleiche Anzahl an Blättern auf jeder Ebene haben:*

$$T_1 = T_2$$

**Korollar 24.** *Diese Äquivalenzrelation hat einige offensichtliche Eigenschaften:*

1. *Wenn $T_1 = T_2$, dann gilt $Expand_i(T_1, q) = Expand_i(T_2, q)$*
2. *Wenn $T_1 = T_2$, dann gilt $cost(T_1) = cost(T_2)$*
3. *Wenn $T_1 = T_2$, dann haben $T_1$ und $T_2$ auf jeder Ebene die gleiche Anzahl an internen Knoten*

**Lemma 25.** *Sei $T$ ein Ebene-$i$-Baum und $T'$ ein Ebene-$i+1$-Baum mit $Trunc_i(T') = T$. Es gibt ein $q$, sodass $Expand_i(T) = T'$.*

*Beweis.* Folgt aus der Definition von $Expand_i$ und daraus, dass Äquivalente Bäume auf jeder Ebene die gleiche Anzahl an internen Knoten haben. □

**Korollar 26.** *Jeder vollständige Ebene-$i$-Baum kann bis auf Äquivalenz durch $i$ Expand-Operationen vom Wurzel-Baum aus konstruiert werden.*

Da $Fill(T_{opt})$ höchstens $n(r-1)$ Blätter hat können wir $Fill(T_{opt})$ finden, indem wir vom Wurzel-Baum durch $Expand_i$ alle vollständigen Bäume durchsuchen, die $m \leq (r-1)$ Blätter haben.

**Algorithm 2** GetOptimalTree (Frequenzen, Farbengrößen)

```
1: function GetOptimalTree(p, c)                          ▷ Frequenzen und Perlengrößen
2:     n ← |p|                                            ▷ Anzahl verschiedener Codewörter
3:     C ← max(c)                                         ▷ Größte Perle
4:     r ← |c|                                            ▷ Anzahl Perlen
5:     // Initialisiere Stack mit Tupel (Signatur, Kosten, Expansionsliste)
6:     stack ← [([0, d_1, d_2, ..., d_C], 0, [])]
7:     best_cost ← +∞
8:     best_tree ← null
9:     while stack ≠ ∅ do
10:        (σ, cost, Qs) ← stack.pop()
11:        if Σσ > n (r − 1) then
12:            continue
13:        end if
14:        if σ[0] ≥ n then
15:            if cost < best_cost then
16:                best_cost ← cost
17:                best_tree ← Qs
18:            end if
19:            continue
20:        end if
21:        new_cost ← cost + Σ_{i=σ[0]+1}^{n} p_i
22:        if new_cost > best_cost then
23:            continue
24:        end if
25:        for q = 0 to σ[1] do
26:            σ' ← EXPAND(σ, q)
27:            stack.append(σ', new_cost, Qs + [q])
28:        end for
29:    end while
30:    return GENERATE_TREE(best_tree, color_sizes)
31: end function
```

Als erstes werden die Variablen $n$, $C$ und $r$ in Abhängigkeit von den Frequenzen der Buchstaben und der größen der Farben intitialisiert. Am Anfang enthält der Stack nur den Wurzel-Baum, dessen Signatur die Kosten 0 hat. Im Stack wird außerdem gespeichert, durch welche Erweiterungen (Werte von $q$) der jeweilige Baum erreicht wurde. Die *best_ cost* und *best_ tree* Variablen speichern die Kosten und die Werte der Erweiterungen des bisher besten Baums. Die *While*-Schleife läuft bis alle Bäume besucht oder für zu teuer befunden wurden. Für jedes Element in *stack* werden zuerst überprüft, ob es bereits zu viele Blätter hat, um die Vervollständigung eines optimalen Baums zu sein.
Falls das erste Element der Signatur (also $m$) bereits größer oder gleich $n$ ist, hat der Baum genug Blätter um alle $n$ Buchstaben zu codieren. Wenn der Baum „billiger“ als der beste Baum ist, der bisher gefunden wurde, werden die Werte von $q$ gespeichert, die verwendet wurden, um den Baum zu erreichen.
Falls der Baum noch nicht groß genug ist, muss er erweitert werden, das hat aber einen Preis, der nun berechnet und mit dem besten Preis verglichen wird.
Anschließend wird der Baum für jeden Wert von $q$ erweitert und die Ergebnisse werden wieder an den stack angehängt.
Nach der *While*-Schleife haben wir die Anzahl der internen Knoten des besten Baums. Dieser wird dann mit der GENERATE_TREE Funktion generiert. Die Codetabelle kann dann über die Pfade zu den $n$ höchsten Blättern generiert werden.
Da wir in Lemma 26 gezeigt haben, dass jeder vollständige Baum (und damit auch $Fill(T_{opt})$) durch mehrere *Expand* Operationen erzeugt werden kann, findet Algorithmus 2 sicher den optimalen Baum.

### 1.2.3 Laufzeit

Da der Algorithmus im Wort-Case alle Signaturen mit $\sum sig \leq n(r-1)$ durchläuft, läuft die *While*-Schleife Asymptotisch $O((n(r-1))^{C+1})$ mal durch. Die *For*-Schleife läuft $l_1 + 1$, also asymptotisch $O(n(r-1))$ mal durch und die *Expand*-Operation hat eine Laufzeit von $O(C)$. Damit hat dieser Algorithmus eine

asymptotische Laufzeit von $O((n(r-1))^{C+2} \cdot C)$.
In der Praxis werden aber sehr viel weniger Knoten besucht, da ein großer Teil bereits durch zu hohe Kosten ausgeschlossen werden kann.

## 1.3 Reduktion auf $n$ Blätter

Im vorherigen Kapitel haben wir uns der Einfachheit halber auf vollständige Bäume beschränkt. Dadurch mussten wir aber immer Bäume mit bis zu $n(r-1)$ Blättern betrachten, was vor allem für große $n$ und $r$ zu langer Rechenzeit führt. Da wir die optimalen Bäume trotzdem wie im vorherigen Kapitel Ebene für Ebene generieren wollen, definieren wir die *Reduce*-Funktion um „überschüssige“ Blätter zu entfernen.

**Definition 27.** *Wir reduzieren einen Baum $T$ zu $Reduce(T)$, indem wir nur die $n$ höchsten Blätter behalten und den Rest entfernen. Falls dabei interne Knoten zu Blättern werden, entfernen wir diese ebenfalls (und wiederholen diesen Prozess ggf.).*

**Korollar 28.** *Wir können wieder einige offensichtlichen Eigenschaften der Reduce-Funktion sammeln:*

1. *Falls $T$ weniger als $n$ Blätter hat, gilt $Reduce(T) = T$*
2. *$Reduce(T)$ ist einzigartig bis auf Äquivalenz*
3. *Für jeden Ebene-$i$-Baum $T$ gilt $cost_i(Reduce(T) = cost_i(T))$*

Um die Bäume wieder anhand der Signatur zu unterscheiden, müssen wir betrachten, wie sich die Signatur eines Baums $T$ mit $sig_i(T) = (m, l_1, \ldots, l_C)$ durch eine Reduktion verändert. Man kann die neue Signatur $sig_i(Reduce(T) = (m', l'_1, \ldots, l'_C))$ folgendermaßen berechnen:

**Algorithm 3** REDUCE

$m' \leftarrow \min\{m, n\}$
$l'_1 \leftarrow \min\{l_j, n - m'\}$
**for all** $j \in [2, \ldots, C]$ **do**
    $l'_j \leftarrow \min\{l_j, n - (m' + \sum_{k=1}^{j-1} l'_k)\}$
**end for**

**Lemma 29.** *Sei $T_{opt}$ ein optimaler Baum mit der Höhe $height(T) = h$. $T_{opt}$ kann bis auf Äquivalenz durch $h$-fache Iteration der Funktion $T' = Reduce(Expand(T', q))$ (für bestimmte Werte von $q$) erzeugt werden.*

*Beweis.* TODO □

Wir haben also gezeigt, dass wir den optimalen Baum durch mehrfaches Anwenden der *Expand* und *Reduce* Funktionen erzeugen können. Das können wir durch eine modifikation des vorherigen Algorithmus erreichen:

**Algorithm 4** GetOptimalTree (Frequenzen, Farbengrößen)

```
1:  function GetOptimalTree(p, c)                          ▷ Frequenzen und Perlengrößen
2:      n ← |p|                                            ▷ Anzahl verschiedener Codewörter
3:      C ← max(c)                                         ▷ Größte Perle
4:      r ← |c|                                            ▷ Anzahl Perlen
5:      // Initialisiere Stack mit Tupel (Signatur, Kosten, Expansionsliste)
6:      stack ← [([0, d_1, d_2, ..., d_C], 0, [])]
7:      best_cost ← +∞
8:      best_tree ← null
9:      while stack ≠ ∅ do
10:         (σ, cost, Qs) ← stack.pop()
11:         new_cost ← cost + Σ_{i=σ[0]}^{n-1} p[i]
12:         if new_cost > best_cost then
13:             continue
14:         end if
15:         if σ[0] ≥ n then
16:             if cost < best_cost then
17:                 best_cost ← cost
18:                 best_tree ← Qs
19:             end if
20:             continue
21:         end if
22:         for q = 0 to σ[1] do
23:             σ' ← reduce(expand(σ, q))
24:             stack.append(σ', new_cost, Qs + [q])
25:         end for
26:     end while
27:     return generate_tree(best_tree, color_sizes)
28: end function
```

Da wir den Reduktionsschritt hinzugefügt haben, müssen wir hier nicht mehr überprüfen, ob die Signatur mehr als $n(r-1)$ Blätter hat.
In dem Paper von Golin und Rote, "A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs" wird eine einfache Optimierung vorgeschlagen, mit der nicht immer alle Werte von $q$ durchlaufen werden müssen.

**Lemma 30.** *Es ist ausreichend, dass $q$ nur die Werte $[0, \ldots, \min\{l_1, n - \left(m + \sum_{j=1}^{c_2} l_j\right)\}]$ durchläuft.*

Leider wurde diese Optimierung jedoch nicht bewiesen oder näher erläutert, es wurde lediglich gesagt: „We leave it as an exercise for the reader to check that this is correct."
Als vorbildlicher Leser habe ich mir natürlich die Mühe gemacht und mir den Kopf über diesen Beweis zerbrochen:

*Beweis.* Der Beweis wird deutlich anschaulicher, wenn man zuerst den Fall $c_1 = c_2 = 1$ und $c_3 = 2$ betrachtet. Sei $sig_i = (m, l_1, l_2)$ also die Signatur eines Ebene-$i$-Baums $T$, den wir nun um $q$ erweitern wollen. Um zu zeigen, dass alle Erweiterungen mit $q > \min\{l_1, n-(m+\sum_{j=1}^{c_2} l_j)\}$ redundant sind, setzen wir $q \leftarrow n - (m + l_1) + 1$.
Sei $T' = Expand(T, q)$ mit der Signatur $sig_{i+1}(T') = (m', l'_1, l'_2)$. Durch die Regeln der Erweiterung wissen wir:

1. $m' = m + l_1 - q$
2. $l'_1 = l_2 + d_1 \cdot q = l_2 + 2 \cdot q$
3. $l'_2 = 0 + d_2 \cdot q = q$

Für die Anzahl der Blätter auf den Ebenen $0, \ldots, i+1$ gilt jetzt:

$$\begin{aligned} m' + l_1' &= m + l_1 - q + l_2 + 2 \cdot q \\ &= m + l_1 + l_2 + q \\ &= m + l_1 + l_2 + n - (m + l_1) + 1 \\ &= l_2 + n + 1 \end{aligned} \tag{1}$$

Als nächsten Schritt kürzen wir den Baum mit der *Reduce* Funktion. Da wir bereits auf Ebene $i+1$ mehr als $n$ Blätter haben, werden die Blätter auf den nachfolgenden Ebenen auf jeden Fall gekürzt. Aber auch in der Ebene $i+1$ müssen $l_2 + 1$ Blätter gekürzt werden. Vor der Expansion waren auf der Ebene $i+1$ genau $l_2$ Blätter, zu denen wir $q \cdot 2$ Blätter hinzugefügt haben.
Um den Beweis weiterzuführen müssen wir die Struktur der Ebene $i+1$ betrachten, vor allem die der neu hinzugefügten Blätter. Jedes der $q$ Blätter aus Ebene $i$, das durch unsere Expand-Operation zu einem internen Knoten gemacht wurde, hatte vor der Reduktion 2 Kinder auf Ebene $i+1$ und eines auf Ebene $i+2$. Das Kind auf Ebene $i+2$ wird durch die Reduktion auf jeden Fall entfernt. Da wir nun $l_2 + 1$ Blätter aus Ebene $i+1$ kürzen müssen, wird mindestens ein Knoten, der vorher Erweitert wurde, nur noch ein Kind haben. (Da die Ebene $i+1$ vor der Erweiterung $l_2$ Blätter hatte und $l_2 + 1$ Blätter heraus gekürzt werden müssen.) Durch die Reduktion werden diese Knoten, die nur noch ein Kind haben, dann zu einem Blatt gemacht. Dadurch erreicht man einen Zustand, den man auch mit einer Erweiterung mit einem kleineren $q$ erreicht hätte. Dieser wurde aber schon in einer vorherigen Iteration der Schleife berücksichtigt. Wir haben also für das Beispiel $c_1 = c_2 = 1$ und $c_3 = 2$ gezeigt, dass es genügt, wenn $q$ die Werte $[0, \ldots, \min\{l_1, n - (m + \sum_{j=1}^{c_2} l_j)\}]$ durchläuft.
Die Verallgemeinerung dieses Beweises ist trivial und wird daher dem Leser überlassen. (Falls der Leser sich eine Lösung wünscht, ist diese im Anhang zu finden: Lemma 31.) □

### 1.3.1 Laufzeit

Der Algorithmus betrachtet im Worst-Case alle Signaturen mit $\sum sig \leq n$, also wird die *While*-Schleife $O(n^{C+1})$ mal durchlaufen. Die *For*-Schleife hat wieder eine asymptotische Laufzeit von $O(C \cdot n(r-1))$, da sie $O(n(r-1))$ mal ausgeführt wird und die *Expand* und *Reduce* Operationen jeweils eine Laufzeit von $O(C)$ haben. Insgesamt hat der Algorithmus also eine Asymptotische Laufzeit von $O(C \cdot n(r-1) \cdot n^{C+1})$.
TODO stimmt des überhautp??

In der Praxis werden aber deutlich weniger Bäume besucht, da viele bereits durch zu hohe Kosten und und die Obergrenze für $q$ aussortiert werden.

## 1.4 Heuristik

Für große Eingabealphabete ($n \gg 500$) ist dieser Algorithmus trotzdem deutlich zu langsam. Daher habe ich eine Heuristik entwickelt, mit der selbst große Codetabellen in kurzer Zeit approximiert werden können.
Dafür müssen die Buchstaben zuerst anhand einer Cutoff-Wahrscheinlichkeit $P_{cut}$ (z.B. $P_{cut} = 0.05$) in häufige und seltene Buchstaben aufgeteilt werden. Alle Buchstaben mit $p_i \geq P_{cut}$ werden ab jetzt als *häufige* Buchstaben $P_h$ bezeichnet, die restlichen Buchstaben mit $p_i < P_{cut}$ sind *seltene* Buchstaben $P_s$:

$$P_h = \{p_i \mid p_i \geq P_{cut}\} \qquad P_s = \{p_i \mid p_i < P_{cut}\} \quad \text{für alle } p_i$$

Für große $n \gg 500$ gibt es sehr viel mehr seltene Buchstaben, die aufgrund ihrer kleinen Wahrscheinlichkeit aber wenig zur gesamten Codelänge beitragen. Die Heuristik beruht auf der Annahme, dass die seltenen Wahrscheinlichkeiten $p_i < P_{cut}$ so geringe Unterschiede haben, dass sie als gleich angesehen werden können. Da es trotzdem noch deutlich zu viele seltenen Wahrscheinlichkeiten gibt, um einen passenden Baum mit der Funktion 4 zu generieren, werden die seltenen Wahrscheinlichkeiten anhand einer größe $S$ in „Chunks“ aufgeteilt. Für einen der Chunks wird dann mit der *GetOptimalTree* Funktion ein optimaler Baum $T_c$ generiert, mit der Eingabe: $(p_1, \ldots, p_s) = \mathit{Chunks}[0]$.

$$T_c = \mathit{GetOptimalTree}((p_0, p_1, \ldots, p_s), (c_1, c_2, \ldots c_r))$$

Anschließend wird für die häufigen Buchstaben und die Summen der Chunks der Haupt-Baum $T$ mit $|P_h| + |\mathit{Chunks}|$ Blättern generiert:

$$p_1, \ldots, p_{|P_h|} = P_h \qquad p_{|P_h|+i} = \sum \mathit{Chunks}[i]$$

$$T = \mathit{GetOptimalTree}((p_0, p_1, \ldots, p_{|P_h|+|Chunks|}), (c_1, c_2, \ldots c_r)$$

Die höchsten $|P_h|$ Blätter von $T$ sind für die häufigen Buchstaben „reserviert“. An die restlichen $|Chunks|$ Blätter wird jeweils eine Kopie von $T_c$ angehängt.
Wie „optimal“ der fertige Baum ist, hängt hauptsächlich von der Chunkgröße $S$ ab. Da der vorherige Algorithmus sehr schnell abläuft, kann auch ein Bereich an Chunkgrößen ausprobiert werden, um eine gute Approximation zu finden.

### 1.4.1 Laufzeit

Die *GetOptimalTree* Funktion wird zweimal aufgerufen, einmal $n = S$ und einmal mit

$$n = |P_h| + |Chunks| = |P_h| + \left\lfloor \frac{|P_s|}{S} \right\rfloor + 1$$

Da $|P_h| \leq 20$ (für $P_{cut} = 0.05$), muss $|P_h|$ in der Laufzeitbetrachtung nicht beachtet werden. Die Laufzeit der Heuristik beträgt also $O(m^{C+1})$ (TODO überprüfen) mit

$$m = \max\left\{S, \left\lfloor \frac{|P_s|}{S} \right\rfloor + 1\right\} \leq \max\left\{S, \left\lceil \frac{n}{S} \right\rceil\right\}$$

# 2 Umsetzung

Hier wird kurz erläutert, wie die Lösungsidee im Programm tatsächlich umgesetzt wurde. Hier können auch Implementierungsdetails erwähnt werden. TODO erweiterung: die Kosten der Perlen sind keine ganzen Zahlen mehr, sondern reelle Zahlen.

# 3 Beispiele

Genügend Beispiele einbinden! Die Beispiele von der BwInf-Webseite sollten hier diskutiert werden, aber auch eigene Beispiele sind sehr gut – besonders wenn sie Spezialfälle abdecken. Aber bitte nicht 30 Seiten Programmausgabe hier einfügen!

TOOD Latex datei der Dokumentation reinwerfen

# 4 Quellcode

Unwichtige Teile des Programms sollen hier nicht abgedruckt werden. Dieser Teil sollte nicht mehr als 2–3 Seiten umfassen, maximal 10.

# 5 Anhang

Tatsächlich:

**Lemma 31.** *Es ist ausreichend, dass $q$ nur die Werte $[0, \ldots, \min\{l_1, n - (m + \sum_{j=1}^{c_2} l_j)\}]$ durchläuft.*

*Beweis.* Sei $sig_i(T) = (m, l_1, \ldots, l_C)$ wieder die Signatur des Ebene-$i$-Baums $T$, den wir nun um $q$ erweitern wollen. Wir setzten $q \leftarrow n - (m + \sum_{j=1}^{c_2} l_j) + x$ mit $x \in \mathbb{N}$ und $x > 0$, um zu zeigen, dass alle Erweiterungen mit $q > \min\{l_1, n - (m + \sum_{j=1}^{c_2} l_j)\}$ redundant sind.
Sei $T' = \mathit{Expand}(T, q)$ mit der Signatur $sig_i(T') = (m', l'_1, \ldots, l'_C)$. Durch die Regeln der Erweiterung wissen wir:

1. $m' = m + l_1 - q$
2. $l'_j = l_{j+1} + q \cdot d_j$

Entweder ist $c_1 = c_2$ oder $c_1 < c_2$. Wir beschäftigen uns zuerst mit dem Fall, dass $c_1 < c_2$.

Für die Anzahl aller Blätter auf den Ebenen $0, \ldots, i + c_2$ gilt jetzt:

$$
\begin{aligned}
m' + \sum_{j=1}^{c_2} l'_j &= m + l_1 - q + \sum_{j=1}^{c_2} l_{j+1} + q \cdot d_j \\
&= m + l_1 - q + \left( \sum_{j=1}^{c_2-1} l_{j+1} + q \cdot d_j \right) + l_{c_2+1} + q \cdot d_{c_2} \\
&\quad \text{Da } c_1 < c_2 \text{ gilt } d_j = 0 \text{ für alle } j \in \{j \in \mathbb{N} \mid j < c_2 \wedge j \neq c_1\}. \text{ Es gilt aber } d_{c_1} = 1. \\
&\quad q \cdot d_j \text{ ist in der Summe also nur einmal relevant und kann daher „herausgezogen“ werden.} \\
&= m + l_1 - q + l_{c_2+1} + q \cdot d_{c_2} + \left( \sum_{j=1}^{c_2-1} l_{j+1} \right) + q \\
&= m + l_1 + l_{c_2+1} + q \cdot d_{c_2} + \sum_{j=1}^{c_2-1} l_{j+1} \\
&= m + l_1 + l_{c_2+1} + q + q \cdot (d_{c_2} - 1) + \sum_{j=1}^{c_2-1} l_{j+1} \\
&= m + l_1 + \left( n - (m + \sum_{j=1}^{c_2} l_j) + x \right) + q \cdot (d_{c_2} - 1) + \sum_{j=1}^{c_2-1} l_{j+1} \\
&= m + l_{c_2+1} + \left( n - (m + \sum_{j=1}^{c_2} l_j) + x \right) + q \cdot (d_{c_2} - 1) + \sum_{j=1}^{c_2} l_j \\
&= n + l_{c_2+1} + q \cdot (d_{c_2} - 1) + x
\end{aligned}
\tag{2}
$$

Wir können jetzt wieder wie im oberen Beweis vorgehen und den Reduktionsvorgang nachvollziehen. Auf allen Ebenen $> i + c_2$ müssen auf jeden Fall alle Blätter gelöscht werden, da schon bis Ebene $i + c_2$ mehr als $n$ Blätter erreicht sind. Da $c_1 \leq c_2$ hat jeder Knoten, den wir vorher erweitert haben genau $d_2 + 1$ Kinder. Wir müssen auf jeden Fall wieder die $l_{c_2+1}$ Blätter löschen, die bereits vor der Erweiterung da waren. Anschließend müssen wir weitere $q \cdot (d_{c_2} - 1)$ Blätter entfernen. Jeder der Knoten, die wir vorher erweitert haben, hat nun genau ein Kind auf Ebene $i + c_1$ und ein Kind auf Ebene $i + c_2$. Wir müssen aber noch $x > 0$ Blätter auf Ebene $i + c_2$ löschen, d.h. es gibt wieder mindestens einen Knoten, der nur ein Kind hat. Durch weitere Reduktion entsteht nun ein Zustand, der auch durch eine Erweiterung mit einem kleineren $q$ erreicht werden könnte (siehe Beweis von Lemma 30).
Der Fall $c_1 = c_2$ ist eine triviale Abwandlung dieses Beweises und wird dem Leser überlassen :) □

# 6 Literatur


Golin, M.J. und G. Rote. “A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs”. In: *IEEE Transactions on Information Theory* 44.5 (Sep. 1998). Conference Name: IEEE Transactions on Information Theory, S. 1770–1781. ISSN: 1557-9654. DOI: 10.1109/18.705558. URL: https://ieeexplore.ieee.org/document/705558 (besucht am 15. 01. 2025).

*Huffman-Kodierung*. In: *Wikipedia*. Page Version ID: 254369306. 20. März 2025. URL: https://de.wikipedia.org/w/index.php?title=Huffman-Kodierung&oldid=254369306 (besucht am 26. 04. 2025).