Aufgabe 1: Schmucknachrichten

Teilnahme-ID: 74749

Bearbeiter/-in dieser Aufgabe:
Christian Krause

28. April 2025

Inhaltsverzeichnis

Zusammenfassung

Für den ersten Aufgabenteil a) verwende ich eine Verallgemeinerung der klassischen Huffman-Codierung, um eine optimale Codierungstabelle für die Perlen zu erstellen. Im Aufgabenteil b) beziehe ich mich auf ein Paper, das einen Ansatz zur Erstellung von optimalen Codetabellen für Perlen mit unterschiedlichem Durchmesser vorstellt. Anhand dieses Paper habe ich meinen eigenen Ansatz zur Erstellung von optimalen Bäumen erarbeitet, der in der Praxis deutlich schneller ist.

1 Lösungsidee

1.1 Huffman Codierung

David Huffman hat 1952 eine Methode veröffentlicht, die heute als *Huffman Codierung* bekannt ist.[1] Die Huffman Codierung kann verwendet werden, um eine optimale präfixfreie Codierung für einen bestimmten Text zu finden. 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, also $p_1 \geq p_1 \geq \cdots \geq p_n$. 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, das 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, das 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 Perlen 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 sind definiert als das Produkt der Länge jedes Codeworts w_i mit der Frequenz p_i des codierten Buchstabens a_i:*

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

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.

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

Für die Kosten eines Baums 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 dem Buchstaben zuordnen, der am seltensten vorkommt, und das kürzeste Codewort dem Buchstaben zuordnen, der am häufigsten vorkommt. Dafür ordnen wir im Folgenden die Blätter Baums in aufsteigender Reihenfolge, also $|w_1| \leq \cdots \leq |w_n|$. Da das Alphabet nach absteigender Reihenfolge sortiert ist, also $p_1 \geq \cdots \geq p_n$, liefert die Zuordnung von w_i zu a_i für jeden Baum die geringsten Kosten.

[1] *Huffman-Kodierung.*

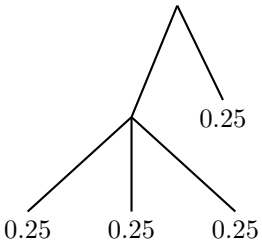


Abbildung 1: Suboptimaler Baum

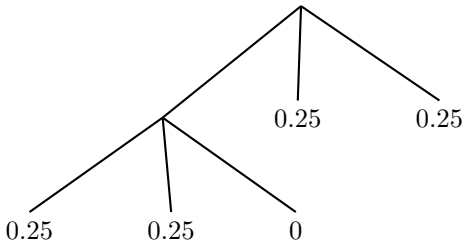


Abbildung 2: Optimaler Baum mit Platzhalterknoten

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.*

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 Anzahl an Blättern aber geringeren Kosten. □

Das gilt im allgemeinen Fall bei Bäumen mit $r > 2$ Kindern aber nicht. Für die allgemein bekannte Konstruktion 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 $n = 4$ und $r = 3$ 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 (Siehe Abbildung).*

Daher muss das Alphabet zuerst mit so vielen Platzhalterbuchstaben, deren Antreffwahrscheinlichkeit 0 ist, aufgefüllt werden, dass gilt: $n \mod (r-1) = 1$. Da bei jeder Iteration, bei der r Knoten zu einem zusammengefasst werden, die Gesamtanzahl der Knoten um $r-1$ reduziert wird, stellt diese Formel sicher, dass am Ende genau ein Knoten übrig bleibt. Die Konstruktion läuft nun mit einem Greedy-Algorithmus ab:

1. Erstelle für jeden Buchstaben a_i im Alphabet (inklusive Platzhalter) einen Teilbaum (der aus einem Wurzelknoten besteht) und beschriften diesen mit der Frequenz p_i des Buchstabens a_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 Teilbaum mit dem neuen Knoten als Wurzel und den gewählten Teilbäumen als Kinder.
4. Wiederhole Schritt 2 und 3, bis nur noch ein Baum übrig ist.

Aufgrund der Platzhalterknoten entsteht nun ein optimaler, vollständiger Baum (siehe Abbildung).

1.1.2 Beweis der Optimalität

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

Lemma 7. *Seien $l_1, l_2, \ldots l_r$ die r Buchstaben mit der geringsten Antreffwahrscheinlichkeit (In dieser Auswahl können auch die Platzhalterbuchstaben enthalten sein). Sie sind in einem Huffman Baum „Geschwister“, also Kinder des gleichen Knotens Y und liegen auf der untersten Ebene des Baums.*

Beweis. Die Buchstaben $l_1, \dots l_r$ sind sicher Geschwister, da sie als erstes ausgewählt werden.
Wenn $l_1, \dots l_r$ nicht auf der untersten Ebene des Baums liegen würden, dann müsste es einen internen Knoten X geben, dessen Antreffwahrscheinlichkeit kleiner ist als die des Knotens Y, dessen Kinder $l_1, \dots l_r$ sind. Da der Baum vollständig ist, müsste aber mindestens einer der Kinder von X eine geringere Antreffwahrscheinlichkeit haben als einer der Kinder $l_1, \dots l_r$ von Y. Das widerspricht aber unserer anfänglichen Annahme, dass $l_1, \dots l_r$ die r Buchstaben mit der geringsten Antreffwahrscheinlichkeit sind. Daher können wir sagen, dass $l_1, \dots l_r$ 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 beschriebene 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, dass ein Huffman Baum T mit n Blättern optimal ist. Seien $l_1, \dots l_r$ wieder die r Buchstaben mit der geringsten Antreffwahrscheinlichkeit und Y der Knoten in T, dessen Kinder $l_1, \dots l_r$ 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' (also wäre T' nicht optimal). Da das unserer Induktionshypothese widerspricht, können wir daraus schließen, 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 ← [ ]                                   ▷ Die Kinder des neuen Knotens
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 verläuft ähnlich wie bei binären Huffman Bäumen:[2]
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

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 können 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 $|V|$. Die Tiefe des Wurzelknotens ist 0.*

Bis jetzt waren die Kinder v_1 bis v_r eines Knotens V immer genau eine Ebene tiefer als V, also $|v_i| = |V| - 1$. 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_1 bis v_r des Knotens V mit den Perlen o_1 bis o_r beschriften $v_i \to o_i$, soll gelten: $|V| = |v_i| + c_i$. Daraus resultieren sogenannte *schiefe* (engl. lopsided) Bäume (Siehe Abbildung 3).
Die oben beschriebene Huffman-Codierung deckt den Spezialfall $c_1 = c_2 \cdots = c_r$ ab, im allgemeinen Fall produziert sie jedoch keine optimalen Bäume. Das liegt dran, dass sich der Beweis für die Optimalität von Huffman-Bäumen darauf stützt, dass die r tiefsten Knoten Geschwister sind. Das ist in einem schiefen Baum im Allgemeinen aber nicht der Fall.

1.2.1 Vollständige Bäume

Golin und Rote stellen 1998 in ihrem Paper „A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs“[3] 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, dass dieser genau r Blätter besitzt.*

[2] *Huffman-Kodierung.*
[3] Golin und Rote, “A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs”.

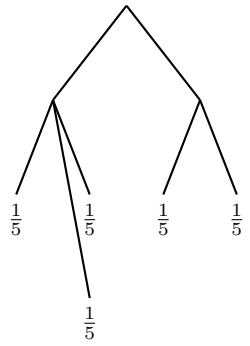


Abbildung 3: Ein nicht vollständiger Baum T

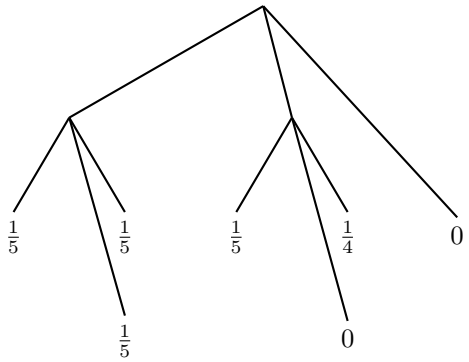


Abbildung 4: Der vervollständige Baum $Fill(T)$

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. In einem optimalen Baum hat jeder Knoten mindestens zwei Kinder (Sonst könnte man einen Knoten mit nur einem Kind zu einem Blatt machen und würde einen Baum mit geringeren Kosten erhalten). Die Anzahl der internen Knoten in einem vollständigen Binärbaum mit n Blättern beträgt $I = n-1$, also hat ein optimaler Baum höchstens I interne Knoten. Ein vollständiger Baum mit I internen Knoten hat genau $Ir-I+1 = (r-1)I+1$ Blätter, $Fill(T_{opt})$ hat also höchstens $1+(n-1)(r-1) \leq n(r-1)$ Blätter. □

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. *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:*

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

.

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

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

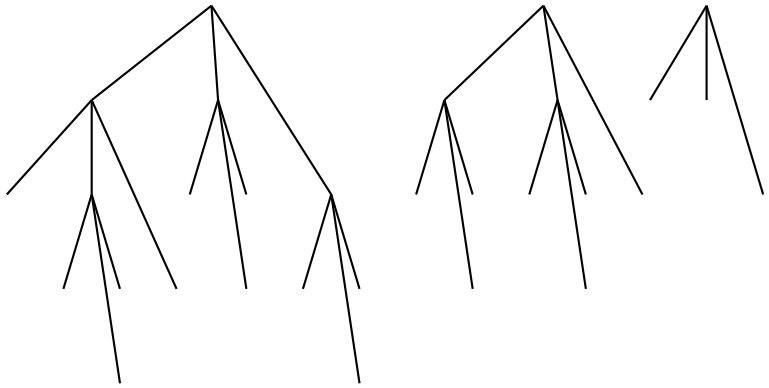
wobei $m = |\{v \in T \mid v \text{ ist ein Blatt und } |v| \leq i\}|$ der Anzahl der Blätter von T mit einer Tiefe von höchstens i enspricht und

$$l_k = |\{u \in T \mid |u| = i + k\}| \quad \text{für alle } k \in \{1, \ldots, C\}$$

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.

Beispiel 15. *Sei $T = Cut_2(T)$ ein Ebene-2-Baum mit $c_1 = c_2 = 1, c_3 = 2$ und der Signatur $sig_2(T) = (3, 6, 2)$ (links). In der Mitte sieht man den Baum $Cut_1(T)$ mit der Signatur $sig_1(Cut_1(T)) = (0, 5, 2)$. Ganz rechts sieht man den Baum $Cut_0(T)$ mit der Signatur $sig_0(Cut_0(T)) = (0, 1, 2)$.*



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

Definition 16. *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 $i > n$. 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 nun 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 Wurzel-Baums darzustellen, müssen wir zunächst den charakteristischen Vektor einer Perlensammlung definieren:

Definition 17. *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:*

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

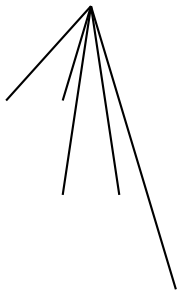
Beispiel 18. *Der charakteristische Vektor der Beispieldatei 7 mit den Durchmessern $C = (1, 1, 1, 1, 1, 1, 1, 2, 3, 4)$ beträgt $(7, 1, 1, 1)$ und der charakteristische Vektor der Beispieldatei 5 mit den Durchmessern $C = (1, 1, 2, 2, 3)$ beträgt $(2, 2, 1)$.*

Definition 19. *Der **Wurzel-Baum** T_0 ist der 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 20. *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 21. *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, auch 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 22. *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. Die erste gleichung entsteht durch zwei Operationen: Das „Verschieben“ der Signatur um eine Ebene nach unten und das Erweitern des Baums.

$$sig'_{i+1} = (m + l_1, l_2, \ldots, l_C, 0)$$

Die Signatur sig'_{i+1} ist das Ergebnis des Verschiebens. Bei der Erweiterung werden q Blätter auf Ebene $i+1$ zu internen Knoten umgewandelt, weshalb q von der Anzahl dieser Blätter abgezogen wird. An diese neuen internen Knoten werden jeweils r Blätter angehängt. Für jeden neuen internen Knoten entstehen daher d_1 neue Blätter auf Ebene $i+2$, d_2 neue Blätter auf Ebene $i+3$ usw. Daraus folgt die erste Gleichung.
Die zweite Gleichung folgt aus einer ähnlichen Überlegung, die ich aufgrund der Kürze nicht weiter ausführen möchte. □

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 aus durch $Expand_i$ alle vollständigen Bäume durchsuchen, die $m \leq (r-1)$ Blätter haben.

Algorithm 2 GetOptimalTree1 (Frequenzen, Farbengrößen)

```
1:  function GetOptimalTree1(p, c)                          ▷ Frequenzen und Perlengrößen
2:      n ← |p|                                             ▷ Anzahl verschiedener Codewörter
3:      C ← max(c)                                          ▷ Größte Perle
4:      r ← |c|                                             ▷ Anzahl Perlen
5:      // Am Anfang enthält der Stack nur den Wurzel-Baum
6:      stack ← [([0, d_1, d_2, ..., d_C], 0, [])]          ▷ (Signatur, Kosten, Anzahl an internen Knoten)
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 initialisiert. Am Anfang enthält der Stack nur den Wurzel-Baum, dessen Signatur die Ebene-0-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* wird 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 dem Abschluss *While*-Schleife haben wir in *best_ cost* die Anzahl der internen Knoten des besten Baums. Dieser wird dann mit der GENERATE_TREE Funktion generiert und zurückgegeben. Die Codetabelle kann anschließend 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.
Ich verzichte hier auf eine Laufzeitbetrachtung, da im Folgenden ein effizienterer Algorithmus vorgestellt wird.

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' ← min{m, n}
l'_1 ← min{l_j, n − m'}
for all j ∈ [2, ..., C] do
    l'_j ← min{l_j, n − (m' + Σ_{k=1}^{j−1} l'_k)}
end for
```

Lemma 29. *Sei T_{opt} ein optimaler Baum der Höhe h. Sei $T_i = Cut_i(T)$, $T'_i = Reduce(Fill(T_i))$ und q_i die Anzahl an internen Knoten von T auf Ebene $i \in \{1, \ldots, h\}$. Dann gilt: $T'_i = Reduce(Expand_i(T'_{i-1}, q_i))$.*
In anderen Worten:
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. Wir wissen, dass

$$Fill(T_{i+1}) = Expand_i(Fill(T_i), q_{i+1}).$$

Das folgt daraus, dass beide Bäume vollständig sind und die gleiche Anzahl an internen Knoten auf jeder Ebene haben. Es gibt jetzt zwei Fälle, wir befassen uns zuerst mit dem Fall, dass $Fill(T_i)$ weniger als n Blätter hat:
In diesem Fall verändert die Reduktion den Baum nicht, es gilt also: $T'_i = Fill(T_i)$ und wir können schreiben:

$$Fill(T_{i+1}) = Expand_i(Fill(T_i), q_{i+1}) = Expand_i(T'_i, q_{i+1})$$

Wenn wir beide seiten reduzieren, haben wir das diesen Fall abgeschlossen:

$$Reduce(Fill(T_{i+1})) = T'_{i+1} = Expand_i(Fill(T_i), q_{i+1}) = Expand_i(T'_i, q_{i+1})$$

Der andere Fall folgt aus einer Betrachtung der Anzahl an Blättern auf den Ebenen i bis $h-1$ vor und nach der Erweiterung, woraus dann die gleichheit folgt. Genaueres findet sich in Lemma 10 des Papers.[4] □

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[5] 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 $\left[0, \ldots, \min\left\{l_1, n - \left(m + \sum_{j=1}^{c_2} l_j\right)\right\}\right]$ 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:

[4] Golin und Rote, "A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs".
[5] Golin und Rote, "A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs".

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 den Ebenen 0 bis $i+1$ 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

Im Worst-Case werden alle Bäume (bis auf Äquivalenz) mit $\leq n$ Blättern betrachtet. Ein solcher Baum hat maximal $I = n - 1$ interne Knoten und eine maximale Tiefe von $h = n - 1$. Sei q_i die Anzahl der internen Knoten auf Ebene i mit $1 \leq i \leq n-2$. Wir müssen folgende strikt steigende Sequenz betrachten:

$$(q_1, q_2 + 1, \ldots, q_1 + q_2 + \cdots + q_{n-2} + n - 3)$$

Diese Sequenz ist eine Teilmenge der n-er Teilmengen von $\{0, \ldots, 2n-4\} \subseteq \{0, \ldots, 2n\}$. Da die Expand und Reduce Operationen jeweils eine Laufzeit von $O(C)$ haben, können wir folgende Obergrenze für die Laufzeit des Algorithmus angeben:

$$O\left(C\binom{2n}{n}\right)$$

Falls gilt $p_0 = p_1 = \cdots = p_n$ können wir eine bessere Abschätzung machen. Da nun ein Baum gesucht ist, bei dem die Knoten alle auf einer ähnlichen Ebene liegen, kann die Tiefe durch $h = \lceil \log_r(n) \rceil \cdot C$ begrenzt werden. Mit dieser Abschätzung kann die Berechnung wie oben durchgeführt werden und man erhält eine Obergrenze von:

$$O\left(\binom{Cn+h}{h}\right)$$

1.4 Diskussion der Laufzeit

Die oben angegebene Laufzeit ist aber nur eine Obergrenze, die für die theoretische Korrektheit mehrere Extremfälle beachten muss: Die maximale Anzahl der internen Knoten entsteht bei einem Baum, bei dem jeder Knoten nur zwei Kinder hat und die maximale Tiefe entsteht bei einem Baum, bei dem jeder Knoten genau zwei Kinder und jede ebene genau einen internen Knoten hat, was extrem unrealistisch ist. Des Weiteren sind viele der möglichen n-er Teilmengen von $\{0, \ldots, 2n\}$ gar keine gültigen Bäume.
In dem Paper[6] wird ein Algorithmus mit der Laufzeit $O(C^{C+2})$ vorgestellt. Diese ist zwar asymptotisch

[6] Golin und Rote, "A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs".

deutlich besser als die Laufzeit von meinem Algorithmus, in diesem Algorithmus werden aber immer genau $\binom{n+C+1}{C+1}$ Knoten besucht, deren Kosten alle in einem Array gespeichert werden müssen. Die Stärke von meinem Algorithmus ist, dass zuerst große Bäume berechnet werden (da mit *it* immer der letzt Wert aus dem Stack entfernt wird) und dadurch schnell eine gute Obergrenze für die Kosten des optimalen Baums gefunden werden kann. Alle Bäume mit größeren Kosten können dann bereits vor der fertigstellung übersprungen werden. Dadurch, dass die fertigen Bäume nicht gespeichert werden müssen, benötigt mein Algorithmus nicht viel Speicherplatz. Um zu testen, ob mein Algorithmus trotz asymptotisch schlechterer Laufzeit tatsächlich weniger Knoten besucht, habe ich für die Beispieldateien berechnet, wie viele Bäume der Algorithmus aus dem Paper besucht hätte (und damit abspeichern müsste). Diesen Wert habe ich mit der Anzahl an besuchten Bäumen von meinem Algorithmus verglichen:

Beispieldatei	Anzahl Paper	Anzahl mein Algorithmus
0	455	159
1	23751	901
2	220	1096
3	715	104
4	680	2866
5	450978066	1629
6	73815	8454
7	60962903966874	1376
8	1621509963255	12321043

Man sieht, dass mein Algorithmus in der Praxis vor allem für Beispieldateien mit großem n deutlich schneller ist, als der theoretisch schnellere Algorithmus aus dem Paper. Beispieldatei 9 konnte mein Algorithmus nicht in sinnvoller Zeit lösen. Der optimale Algorithmus aus dem Paper hätte in dieser Beispieldatei mit $n = 674$ und $c = 4$ aber $1185106443885 \equiv 1.19\ldots10^{12}$ Knoten besuchen und auch abspeichern müssen, was bei 2 Byte pro Knoten über 2 Terrabyte benötigt hätte, was eine Ausführung sehr unrealistisch macht. Mein Algorithmus muss dagegen nur den besten Baum und den Stack speichern und könnte daher auch mit genug Zeit Probleme mit großem n lösen. (Ich habe meinen Algorithmus für einige Zeit für Aufgabe 9 laufen lassen, es wurden ca. 40 MB Arbeitsspeicher belegt).

1.5 Heuristik

Für große Eingabealphabete ($n \gg 500$) ist mein 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: $(p_1, \ldots, p_s) = \mathit{Chunks}[0]$.

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

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

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

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

Die höchsten $|P_h|$ Blätter von T sind für die häufigen Buchstaben „reserviert“. An die restlichen $|\mathit{Chunks}|$ 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.

1.5.1 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. Da die *GetOptimalTree* Funktion eine Laufzeit von $O\left(C\binom{2n}{n}\right)$ hat und zweimal mit verschiedenen n aufgerufen wird, ist die Laufzeit der Heuristik $O\left(C\binom{2x}{x}\right)$ mit

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

2 Umsetzung

Den ersten Aufgabenteil habe ich analog zum Pseudocode in Python implementiert. Die Algorithmen für Aufgabenteil b) habe ich ebenfalls in Python umgesetzt. Um die Ausführungszeit des Programms zu minimieren (und um Rust zu lernen), habe ich den optimalen Algorithmus auch in Rust implementiert. Dafür verwende ich das Tool *maturin*[7], das es erlaubt Rust Funktionen aus Python aufzurufen. Das Einlesen der Datei geschieht im Python Skript, das dann mit den Frequenzen und den Werten von c_i die Rust Funktion aufruft. Diese berechnet dann die Werte von q für den optimalen Baum, aus denen dann im Python-Programm ein Baum und daraus dann eine Codetabelle erstellt wird. Dieser Tree besteht aus Objekten der Node-Klasse:

```python
class Node:
    def __init__(self, parent, number):
        self.number = number
        self.parent = parent
        self.children = []
```

Die Node speichert ihren Elternknoten (*None* bei der Root-Node) und ihre Kinderknoten. Die *number* ist wichtig, um aus dem fertigen Baum dann eine Codetabelle zu generieren. Sie codiert nämlich, durch welche Kante dieser Knoten erreicht wurde. Jeder Knoten kann maximal r Kanten der längen $c_1, \ldots, c_r$ haben. Die *Node* mit der *number* i wurde durch die Kante der Länge c_i erreicht. Da jeder Knoten also jeweils nur eine Kante mit einer bestimmten Nummer hat, kann ein Pfad von der Wurzel des Baums zu den Blättern eindeutig anhand der Nummern identifiziert werden. Die *generate_ tree* Funktion ist dafür zuständig anhand der Werte von q den Baum zu generieren:

```python
def generate_tree(Qs: list[int], color_sizes: list[int]):
    color_sizes = sorted(color_sizes)
    root = Node(None, 0)
    Levels = defaultdict(list)
    Levels[0].append(root)
    for i, c in enumerate(color_sizes):
        root.children.append(Node(root, i))
        Levels[c].append(root.children[-1])

    for i, q in enumerate(Qs):
        if q == 0:
            continue
        for j, n in enumerate(Levels[i + 1]):
            for ii, c in enumerate(color_sizes):
                n.children.append(Node(n, ii))
                Levels[c + i + 1].append(n.children[-1])
```

[7] https://github.com/PyO3/maturin

```
            if j + 1 == q:
                break
    return root
```

In der ersten For-Schleife wird der Wurzel-Baum generiert. Anschließend wird in jeder Ebene die entsprechende Anzahl an Blättern zu internen konvertiert. Jeder der neuen internen Knoten wird vollständig aufgefüllt, d.h. die *generate_tree* Funktion erzeugt einen vollständigen Baum. Das ist aber keine Einschränkung, da überschüssige Blätter beim Erstellen der Codetabelle ignoriert werden. Um die Codetabelle zu generieren werden zunächst alle Blätter des Baums und deren Code, der aus den Nummern der Knoten auf dem Pfad von der Wurzel zu dem Blatt besteht, ermittelt:

```
def generate_code_from_tree(root, color_sizes):
    leaves = get_leaves_rec(root)
    codes = []
    for l in leaves:
        code = []
        while l.parent:
            code.append(l.number)
            l = l.parent
        cost = 0
        for c in code:
            cost += color_sizes[c]
        codes.append((code[::-1], cost))
    return codes
```

Diese Codetabelle kann nun nach der Länge der Codes sortiert werden. Die n kürzesten Codes werden dann verwendet um die Buchstaben zu codieren.

3 Beispiele

3.1 Aufgabenteil a)

Ich in der Tabelle sieht man die länge der optimalen Perlenkette für die gegebenen Beispieldateien und die Laufzeit des Huffman-Algorithmus. Da der optimale Algorithmus auch für den Fall $c_1 = c_2 = \cdots = c_r$ optimale Bäume erzeugt, habe ich auch diese Laufzeit mit eingezeichnet.

Beispieldatei	Optimale Perlenkette	Laufzeit Huffman	Laufzeit optimaler Algorithmus
0	113	0.06ms	0.04 ms
00	372	0.12ms	0.09 ms
01	1150	0.36ms	0.08 ms
9	17505	10.89 ms	6236.16 ms

Man sieht, dass beide Algorithmen für die vorgegebenen Beispieldateien sehr schnell sind. Der optimale Algorithmus ist teilweise sogar geringfügig schneller, was vermutlich daran liegt, dass er in Rust implementiert ist. Als letzte Beispieldatei habe ich Beispieldatei 9 so verändert, dass es 4 gleich große Perlen gibt. Hier sieht man, dass der Huffman Algorithmus für große n in seinem spezialfall deutlich schneller ist als der optmale Algorithmus für allgemeine Fälle.

3.2 Aufgabenteil b)

In der Folgenden Tabelle sieht man die Laufzeit die Länge der optimalen Perlenkette und die Laufzeit des optimalen Algorithmus. Daneben ist die Länge der Perlenkette aufgeführt, die durch die Heuristik ermittelt wurde und deren Laufzeit. Die Codetabellen sind in separaten .txt Dateien in der Abgabedatei enthalten.

Beispieldatei	Optimale Perlenkette	Laufzeit optimal	Perlenkette Heuristik	Laufzeit Heuristik
0	113	0.05ms	113	1.61 ms
1	191	0.1ms	192	4.34 ms
2	135	0.11ms	135	1.86 ms
3	279	0.03ms	279	1.37ms
4	137	0.24ms	137	9.59 ms
5	3162	0.19ms	3618	18.99 ms
6	234	0.62ms	238	8.09 ms
7	134559	0.19ms	134800	87.79 ms
8	3287	1198.61 ms	3362	28.21 ms
9	-	-	36668	93.74ms

Man sieht auf den ersten Blick, dass der optimale Algorithmus mit den Beispieldateien 1 bis 8 sehr gut zurecht kommt. Beispieldatei 9 konnte aber nicht sinnvoller Zeit gelöst werden. Die Werte der Heuristik der ersten Beispieldateien haben wenig bedeutung, da ein Großteil der Buchstaben als häufig angesehen wird und es generell wenig Buchstaben gibt, was zu einer sehr kleinen Chunkgröße führt. Die Ergebnisse bei Beispieldatei 8 und 9 sind aber interessant, da die Heuristik bei Beispieldatei 8 eine Codetabelle produziert hat, deren Gesamtlänge der Perlenkette nur 75mm länger als die optimale Perlenkette ist, was einer Abweichung von ca 2.3% entspricht. Auch das Ergebnis bei Beispieldatei 9 ist sehr gut, es weicht nur um 71mm (ca. 0.2%) von der Länge ab, die auf der BWINF Internetseite als Ergebniss eines brauchbaren Programms angegeben wurde.

4 Quellcode

4.1 Aufgabenteil a)

Für den Aufgabenteil a) habe ich aufgrund der anderen Anforderungen eine andere Node Klasse als bei b) verwendet:

```
import heapq
import time


class Node:
    def __init__(self, p, a=None):
        """
        :param p: Frequenz
        :param a: Buchstabe (none wenn interner Knoten)
        """
        self.p = p
        self.a = a
        self.children = []

    def __lt__(self, other):
        return self.p < other.p


def generate_code_from_tree(root):
    codes = {}
    stack = [(root, [])]  # (node, turns to get there)
    while stack:
        node, code = stack.pop()
        for i, c in enumerate(node.children):
            if c.a is not None:
                codes[c.a] = code + [i]
            else:
                stack.append((c, code + [i]))
    return codes
```

```
def encode_huffman(r, frequencies):
    """
    :param r: Maximale Kinder pro Knoten
    :param frequencies: Dictionary mit Buchstabe : Frequenz
    """
    minHeap = []
    for a, p in frequencies.items():
        heapq.heappush(minHeap, Node(p, a))
    n = len(frequencies)
    m = 0
    if r > 2:
        while (n + m) % (r - 1) != 1: m += 1

    for _ in range(m):
        heapq.heappush(minHeap, Node(0))

    while len(minHeap) > 1:
        new_children = []
        for _ in range(r):
            new_children.append(heapq.heappop(minHeap))
        new_node = Node(sum(n.p for n in new_children))
        new_node.children = new_children
        heapq.heappush(minHeap, new_node)

    return generate_code_from_tree(minHeap[0])
```

4.2 Aufgabenteil b)

Dieser Code ist nicht ausführbar, da hier einige Funktionen fehlen. Eine ausführbare Datei ist in der Abgabedatei enthalten.

```
from collections import defaultdict
import rust_encoder

class Node:
    def __init__(self, parent, number):
        """
        :param parent:
        :param number: Index in der sortierten color_sizes liste
        """
        self.number = number
        self.parent = parent
        self.children = []

    def is_leaf(self):
        return not self.children

def generate_tree(Qs: list[int], color_sizes: list[int]):
    """
    Generiert einen vollständigen Baum
    :param Qs: Anzahl an internen Knoten pro Ebene (Werte von q, mit denen Erweitert
    wurde)
    :param color_sizes: Größen der Perlen
    :return:
    """
    color_sizes = sorted(color_sizes)
    root = Node(None, 0)
```

```
    Levels = defaultdict(list)
    Levels[0].append(root)
    for i, c in enumerate(color_sizes):
        # Die neuen Kinder werden anhand der Position der Perlengröße in der
        # color_sizes Liste nummeriert
        root.children.append(Node(root, i))
        Levels[c].append(root.children[-1])

    for i, q in enumerate(Qs):
        if q == 0:
            continue
        for j, n in enumerate(Levels[i + 1]):
            for ii, c in enumerate(color_sizes):
                # Die neuen Kinder werden anhand der Position der Perlengröße
                # in der color_sizes Liste nummeriert
                n.children.append(Node(n, ii))
                Levels[c + i + 1].append(n.children[-1])
            if j + 1 == q:
                break
    return root



def generate_code_from_tree(root, color_sizes):
    """
    Gibt alle Codekombinationen zurück, die die Blätter dieses Baums ausdrücken
    :returns: [(code, cost), ...]
    """
    leaves = get_leaves_rec(root)
    # get codes
    codes = []
    for l in leaves:
        code = []
        # Der Code besteht aus den nummern der Knoten auf dem Pfad von der Wurzel
        # des Baums zu dem entsprechenden Blatt
        while l.parent:
            code.append(l.number)
            l = l.parent
        cost = 0
        for c in code:
            cost += color_sizes[c]
        # Der code wird hier umgedreht, da er sonst von der Wurzel zu dem Blatt gehen würde
        codes.append((code[::-1], cost))
    return codes


def attach_tree_at_leaves(main_root, sub_root, fixed_slots, color_sizes):
    """
    Hängt an jedes Blatt des Hauptbaums eine Kopie von sub_root an
    :param fixed_slots: Anzahl der obersten Blätter die nicht ersetzt werden sollen,
                        da sie zu häufigen Buchstaben gehören
    """
    leaves = get_leaves_rec(main_root)
    leaves = sorted(leaves, key=lambda x: x.get_depth(color_sizes))
    for i, leaf in enumerate(leaves[fixed_slots:]):
        s_r = sub_root.copy()
        leaf.parent.children.remove(leaf)
        leaf.parent.children.append(s_r)
```

```
        s_r.parent = leaf.parent
        s_r.number = leaf.number
    return main_root


def calc_cost(text, codes):
    """
    Berechnet die Kosten einer Liste an Codes, wenn sie optimal zu den Buchstaben
    zugewiesen werden
    """
    # Die Häufigkeiten werden absteigend sortiert
    occurrences = [text.count(i) for i in set(text)]
    occurrences = sorted(occurrences, reverse=True)

    # Die Kosten werden aufsteigend sortiert
    costs = sorted(i[1] for i in codes)

    total_cost = 0
    for i in range(len(occurrences)):
        # Da die Häufigkeiten absteigend und die Kosten aufsteigend sortiert sind,
        # ist diese Zuweisung optimal
        total_cost += occurrences[i] * costs[i]
    return total_cost


def reduce(sig, n):
    """
    Wendet die Reduktionsregel auf eine Signatur an.
    (Alle Blätter, die nicht zu den n höchsten Blättern gehören werden abgeschnitten.)
    """
    new_sig = [min(sig[0], n)]
    for i in range(1, len(sig)):
        new_sig.append(min(sig[i], n - sum(new_sig)))
    return new_sig


def extend(sig, q, D):
    """
    Erweitert die Signatur um q, indem q interne Knoten auf Ebene i+1 zu
    internen Knoten werden
    """
    new_sig = [sig[0] + sig[1]] + sig[2:] + [0]
    x = ([-1] + D)
    for i in range(len(new_sig)):
        new_sig[i] = new_sig[i] + q * x[i]
    return new_sig


class Encoder:
    """
    Enthält verschiedene Methoden, um eine Codetabelle für die gegebenen
    Frequenzen zu erstellen.
    """

    def __init__(self, frequencies, color_sizes):
        self.frequencies = frequencies
        self.num_colors = len(color_sizes)
        self.color_sizes = sorted(color_sizes)
```

```
    def encode_naive(self):
        """
        Alle vollständigen Binärbäume mit m  n * (r - 1) Blättern werden durchsucht
        """
        return generate_code_from_tree(
                    self.get_tree_naive(self.frequencies.values()), self.color_sizes)

    def get_tree_naive(self, frequencies):
        """
        Alle vollständigen Binärbäume mit m  n * (r - 1) Blättern werden durchsucht
        """
        # Frequenzen für jeden Buchstaben
        p = sorted(frequencies, reverse=True)
        # Die Anzahl der verschiedenen Buchstaben im Alphabet
        n = len(p)
        # Die größte Perle
        C = max(self.color_sizes)
        # Anzahl, wie oft jede Perlengröße vorkommt
        D = [self.color_sizes.count(i) for i in range(1, C + 1)]
        # Anzahl der verschiedenen Perlen
        r = len(self.color_sizes)

        print(f"N: {n}, C : {C}")
        # Speichert die Signatur, die Kosten und die Anzahl der Blätter,
        # die gerade bearbeitet werden
        stack = [([0] + D, 0, [])]  # [(sig, cost, Qs (Expansions)), ...]

        # Der bisher beste Baum
        best_cost = float("inf")
        best_tree = None

        while stack:
            sig, cost, Qs = stack.pop()

            if sum(sig) > n * (r - 1):
                # Der Baum hat zu viele Blätter, um Fill(T_opt) zu sein
                continue

            if sig[0] >= n:
                # Der Baum hat genug Blätter, um alle Buchstaben zu codieren
                if cost < best_cost:
                    best_cost = cost
                    best_tree = Qs
                continue

            # Die neuen Kosten des Baums
            new_cost = cost + sum([p[i] for i in range(sig[0], n)])
            if new_cost > best_cost:
                # Der Baum ist zu teuer
                continue

            for q in range(0, sig[1] + 1):
                # Der Baum wird für jedes q erweitert
                new_sig = extend(sig, q, D)
                stack.append((new_sig, new_cost, Qs + [q]))

        # Der beste Baum wird anhand der Anzahlen an inneren Knoten auf
```

```
        # jeder Ebene (Werte von q) generiert
        return generate_tree(best_tree, self.color_sizes)

    def encode_reduce(self):
        # Alle Bäume mit m  n Blättern werden durchsucht
        return generate_code_from_tree(
            self.get_tree_reduce(self.frequencies.values()), self.color_sizes)

    def get_tree_reduce(self, frequencies):
        # Probability for each character of the alphabet
        # Frequenzen für jeden Buchstaben
        p = sorted(frequencies, reverse=True)
        # Die Anzahl der verschiedenen Buchstaben im Alphabet
        n = len(p)
        # Die größte Perle
        C = max(self.color_sizes)
        # Anzahl, wie oft jede Perlengröße vorkommt
        D = [self.color_sizes.count(i) for i in range(1, C + 1)]

        print(f"Encode Reduce: N: {n}, C : {C}")

        print(f"N: {n}, C : {C}")
        # Speichert die Signatur, die Kosten und die Anzahl der Blätter,
        # die gerade bearbeitet werden
        stack = [([0] + D, 0, [])]  # [(sig, cost, Qs (Expansions)), ...]

        best_cost = float("inf")
        best_tree = None

        while stack:
            sig, cost, Qs = stack.pop()

            if sig[0] >= n:
                # Der Baum hat genug Blätter, um alle Buchstaben zu codieren
                if cost < best_cost:
                    best_cost = cost
                    best_tree = Qs
                continue

            new_cost = cost + sum([p[i] for i in range(sig[0], n)])
            if new_cost > best_cost:
                continue
            max_q = min(sig[1],
                    n - sum(sig[i] for i in range(self.color_sizes[1] + 1))) + 1
            for q in range(0, max_q):
                # Der Baum wird für jedes q erweitert (die erweiterung ist hier
                # angewandt, d.h. es werden keine "unnötigen" Werte von q durchlaufen

                # Der Baum wird erweitert und danach auf n Blätter reduziert
                new_sig = reduce(extend(sig, q, D), n)
                stack.append((new_sig, new_cost, Qs + [q]))

        # Der beste Baum wird anhand der Anzahlen an inneren Knoten auf jeder Ebene
        # (Werte von q) generiert
        return generate_tree(best_tree, self.color_sizes)

    def get_tree_rust(self, frequencies, silent=True):
        """
```

```
        Die get_tree_reduce Funktion in Rust implementiert
        """
        qs = rust_encoder.encode_optimal_py(
                frequencies, self.color_sizes, silent)
        return generate_tree(qs, color_sizes)

    def encode_rust(self, silent=True):
        """
        Die encode_reduce Funktion in Rust implementiert
        """
        return generate_code_from_tree(
                self.get_tree_rust(list(self.frequencies.values()), silent), self.color_sizes)

    def heuristic(self, chunk_size=50, color_sizes):
        cutoff = 0.05  # Alle Buchstaben, die häufiger vorkommen, sind häufige Buchstaben

        # Die häufigen Buchstaben
        common = [i for i in self.frequencies.values() if i > cutoff]

        # Die seltenen Buchstaben
        not_common = sorted(i for i in self.frequencies.values() if i <= cutoff)

        # Die seltenen Buchstaben werden in chunks aufgeteilt
        chunks = []
        for i in range((len(not_common) // chunk_size)):
            chunks.append(not_common[i * chunk_size: (i + 1) * chunk_size])
        chunks.append(not_common[(len(not_common) // chunk_size) * chunk_size:])

        # Für den ersten Chunk wird ein optimaler Baum generiert
        chunk_root = self.get_tree_rust(chunks[0])

        chunks = [i for i in chunks if len(i) > 0]
        # Für die häufigen Buchstaben und die Summen der Chunks wird ein Baum generiert
        main_root = self.get_tree_rust(common + [sum(c) for c in chunks])

        # Der chunk_root Baum wird an die passenden Blätter des main_root Baums angehängt
        main_root = attach_tree_at_leaves(main_root, chunk_root, len(common), color_sizes)

        # Der code wird anhand des fertigen Baums generiert
        return generate_code_from_tree(main_root, self.color_sizes)

    def optimize_heuristic(self, text):
        """
        Versucht einen möglichst guten Baum mit der heuristik zu finden,
        indem die chunk_size optimiert wird
        """
        n = len(self.frequencies)
        max_leaves = 40
        min_chunk_size = max(len(self.color_sizes), n // max_leaves + 1)
        best_cost = float("inf")
        best_code = None
        for i in range(min_chunk_size, max_leaves):
            codes = self.heuristic(i)
            cost = calc_cost(text, codes)
            if cost < best_cost:
                best_cost = cost
                best_code = codes
        return best_code
```

Hier ist der Rust Teil des Programs:

```rust
use pyo3::prelude::*;
use rayon::prelude::*;

/// Erweitert die Signatur, indem q Blätter auf Ebene i+1 zu internen Knoten gemacht
/// werden
pub fn extend(sig: &mut Vec<usize>, q: usize, D: &Vec<usize>) {
    sig[0] += sig[1] - q;
    let len = sig.len();
    for i in 1..len-1 {
        sig[i] = sig[i+1] + q * D[i-1];
    }
    sig[len - 1] = q * D[D.len() - 1];
}


/// Reduziert die Signatur, sodass es maximal n Blätter gibt
pub fn reduce(sig: &mut Vec<usize>, n: usize) {
    sig[0] = sig[0].min(n);
    for i in 1..sig.len() {
        sig[i] = sig[i].min(n - sig.iter().take(i).sum::<usize>());
    }
}

/// Gibt für eine gegebene Frequenzverteilung und eine Liste
/// von Perlengrößen die Anzahl der internen Knoten des Optimalen Baums zurück
fn get_tree_optimal(frequencies : &mut Vec<f64>, color_sizes : Vec<usize>,
                        silent: bool) -> Vec<usize> {
    frequencies.sort_by(|a, b| b.partial_cmp(a).unwrap());

    // Die Anzahl der verschiedenen Buchstaben im Alphabet
    let n = frequencies.len();

    // Die größte Perle
    let C = color_sizes.iter().max().unwrap();

    // Anzahl, wie oft jede Perlengröße vorkommt
    let mut D = vec![0; *C];
    for &size in &color_sizes {
        D[size - 1] += 1; // Increment the count for each size
    }

    if (!silent) { print!("C: {}, D: {:?}\n", C, D); }

    // Speichert die Signatur, die Kosten und die Anzahl
    // der Blätter, die gerade bearbeitet werden
    let mut stack: Vec<(Vec<usize>, f64, Vec<usize>)> =
                vec![([vec![0], (D.clone())].concat(), 0.0, vec![])]; // (sig, cost, Qs)

    // Der beste Baum, der bisher gefunden wurde
    let mut best_Qs : Vec<usize> = vec![];
    let mut best_cost =  f64::INFINITY;

    // Die Kosten für die Erweiterung der Signatur können im Voraus berechnet werden
    let mut new_cost : Vec<f64> = vec![];
    for i in 0..n {
        new_cost.push(0.0);
        for j in i..n {
```

```
            new_cost[i] += frequencies[j];
        }
    }

    // Anzahl an besuchten Knoten
    let mut visited: i64 = 0;

    while(stack.len() > 0) {
        visited += 1;

        let (sig, cost, Qs) = stack.pop().unwrap();

        if (sig[0] >= n) {
            // Der Baum hat genug Blätter, um alle Buchstaben zu kodieren
            if (cost < best_cost) {
                best_cost = cost;
                best_Qs = Qs.clone();
                if (!silent) {
                    print!("Found new best cost: {}, Qs: {:?}, stack size:
                            {}, visited: {}\n", best_cost, best_Qs, stack.len(), visited);
                }
            }
            continue;
        }

        let new_cost = cost + new_cost[sig[0]];
        if (new_cost > best_cost) {
            continue;
        }

        // Der Baum muss nur bis zu dem bewiesenen maximalen Wert von q erweitert werden
        let max_q = (sig[1]).min(n - sig.iter().take(color_sizes[1]+1).sum::<usize>())+1;
        for q in (0..max_q) {
            let mut new_sig = sig.clone();
            // Die Signatur wird erweitert
            extend(&mut new_sig, q, &D);
            // und auf maximal n Blätter reduziert
            reduce(&mut new_sig, n);
            stack.push((new_sig, new_cost, [Qs.clone(), vec![q]].concat()));
        }
    }
    if (!silent) {
        print!("Best cost: {}, Qs: {:?}\n", best_cost, best_Qs);
    }
    best_Qs
}
```

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 setzen $q \leftarrow n - (m + \sum_{j=1}^{c_2} l_j) + x$ 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 $T' = Expand(T, q)$ 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).