Aufgabe 1: Schmucknachrichten

Teilnahme-ID: 77075

Bearbeiter/-in dieser Aufgabe:
Felix Du

25. Mai 2025

Inhaltsverzeichnis

1 Lösungsidee **2**
1.1 Problemmodellierung . . . 2
1.2 Kommentar zur Komplexitätsklasse . . . 2
1.3 Greedy Huffman-Algorithmus . . . 2
1.4 Karp's IP-Modell . . . 5
1.5 Greedy-Heuristik . . . 8

2 Implementierung **8**
2.1 Greedy Huffman-Algorithmus . . . 8
2.2 IP-Modell . . . 11
2.3 Greedy-Heuristik . . . 13

3 Beispiele **14**
3.1 Gegebene Beispiele . . . 14
3.1.1 schmuck0.txt . . . 15
3.1.2 schmuck00.txt . . . 15
3.1.3 schmuck01.txt . . . 16
3.1.4 schmuck1.txt . . . 17
3.1.5 schmuck2.txt . . . 18
3.1.6 schmuck3.txt . . . 18
3.1.7 schmuck4.txt . . . 18
3.1.8 schmuck5.txt . . . 19
3.1.9 schmuck6.txt . . . 20
3.1.10 schmuck7.txt . . . 21
3.1.11 schmuck8.txt . . . 22
3.1.12 schmuck9.txt . . . 23

4 Quellcode **25**
4.1 Greedy Huffman-Algorithmus . . . 25
4.2 IP-Modell . . . 26
4.3 Greedy-Heuristik . . . 28

Literatur **29**

1 Lösungsidee

1.1 Problemmodellierung

Gegeben ist stets die Anzahl der Zeichen n (Perle), die verwendet werden, um den gegebenen Text zu kodieren, die eine Menge $A = \{i_0, i_1, \ldots, i_{n-1}\}$ bilden, zusammen mit einer Kostenfunktion $c_i, i \in A$, die die Kosten darstellt, um ein Zeichen i zu verwenden, den Durchmesser einer Perle i. Aus dem Text lassen sich alle verwendeten Zeichen $S = \{s_0, s_1, \ldots, s_m\}$ extrahieren, sowie die dazugehörige Häufigkeit $p_s, s \in S$. Die korrespondierenden normierten Häufigkeiten werden durch die Dichtenfunktion $\rho_s, 0 < \rho \leq 1$ dargestellt. Es soll ein präfixfreier Code C generiert werden, der den gegebenen Text mit minimalen Kosten kodiert. Jedes Wort c gehört zu einem Zeichen und hat Kosten, die durch die Summe der verwendeten Zeichen berechnet werden. Die dazugehörige Funktion wird durch ℓ_s, der erwarteten Kosten des Codewortes für s beschrieben. Insgesamt soll also folgende Summe minimiert werden:

$$\sum_{s \in S} p_s \cdot \ell_s \tag{1}$$

Dies ist die allgemeine Formulierung der Huffman-Kodierung; die gewöhnliche Huffman-Kodierung nach Huffman[3] ist ein präfixfreier Binärcode mit variablen Wortlängen. Die einzelnen Aufgabenteile a) und b) sind Spezifizierungen der allgemeinen Version. Für Aufgabenteil a) gilt stets $c_i = 1, \forall i$ –jeder Buchstabe des Kodierungsalphabets hat dieselben Kosten, beschränkt sich jedoch nicht nur auf das Binäralphabet. Dieses Problem kann auch als *n-ary Huffman Coding Problem*. Nach diesem Muster wäre die gewöhnliche Huffman-Kodierung ein 2-ary Huffman Coding Problem. Aufgabenteil b) fügt die variable Kostenfunktion c_i für das Kodierungsalphabet hinzu.

1.2 Kommentar zur Komplexitätsklasse

Für die Huffman-Kodierung ohne variable Kosten existiert ein Algorithmus, der in polynomieller Zeit lösbar ist und immer einen optimalen, präfixfreien Code für einen gegebenen Text generiert; Optimalität in Hinsicht auf die Gesamtlänge des kodierten Textes. Somit liegt das Problem aus Teilaufgabe a) in P und ein weiterer Beweis ist dadurch nicht notwendig. Für Teilaufgabe b) hingegen ist diese Frage nicht einfach zu beantworten. Golin et al.[1] beschreibt das Problem als noch nicht in eine genaue Klasse eingeordnet. Es existiert ein polynomielles Approximationsschema von Golin[2], sehr gute Heuristiken, sowie eine Formulierung von Karp[4] als ganzzahliges lineares Programm (ILP), jedoch kein Algorithmus, der optimale Lösungen in polynomieller Zeit findet. Allerdings kann eine Aussage über die Zugehörigkeit in NP getroffen werden. Wenn das Entscheidungsproblem des Problems folgendermaßen formuliert wird: *Gegeben eine Huffman Kodierung C, erzeugt diese für einen gegebenen Text einen Code mit Kosten* $\leq k$*?*, dann kann eine gegebene Lösung durch die oben formulierte Summe trivial berechnet und ein Zertifikat ausgestellt werden. So liegt das Problem also in NP, es ist jedoch nicht bekannt, ob es NP-vollständig ist oder in P.

1.3 Greedy Huffman-Algorithmus

Wie bereits beschrieben, kann das Problem als n-ary Huffman-Coding-Problem formalisiert werden, wenn n die Anzahl der zu verwendenden Zeichen (Perlenfarben) ist; dabei wird die gewöhnliche Huffman-Kodierung auf mehr als das Binäralphabet erweitert. Der von Huffman formulierte Algorithmus[3] basiert auf einem Binärbaum, dessen Blätter jeweils ein kodiertes Wort darstellen. Das Wort wird dabei durch seinen Pfad generiert, bei dem klassischerweise alle linken Kanten für 0, alle rechten für 1 stehen und für jeden Pfad von der Wurzel aus konstruiert wird. Ein durch diese Weise generierter Code ist stets präfixfrei. Dies soll nun auch für einen Baum bewiesen werden, dessen Wurzel sowie innere Knoten strikt n Knoten besitzen.

Theorem 1.1. *Ein Code C, der über die Blätter eines Baums generiert wird, ist präfixfrei.*

Seien c_1, c_2 zwei Codewörter eines Codes C und $c_1 \neq c_2$. Es wird gezeigt, dass kein Codewort ein Präfix des anderen ist.

Lemma 1. *Kein Blatt besitzt Kinder.*

Beweis. Per Definition des Blattes in der Baumstruktur haben diese keine weiteren Kinder. □

Lemma 2. *Jeder Pfad von der Wurzel zum Blatt ist eindeutig.*

Beweis. Die Einzigartigkeit eines Pfads soll per vollständiger Induktion über die Tiefe $d(v)$ eines Knotens v bewiesen werden.
Induktionsanfang: Nur die Wurzel r selbst hat eine Tiefe $d(r) = 0$. Der Pfad von r nach r ist der leere Pfad und somit eindeutig.
Induktionsvoraussetzung: Für alle Knoten u mit $d(u) \leq k$ sei bereits gezeigt, dass der Pfad von r nach u eindeutig ist.
Induktionsschritt auf v mit $depth(v) = k+1$: Man nehme den Knoten v mit $d(v) = k+1$. Da der Baum ein zusammenhängender Graph ist, existiert mindestens ein einfacher Pfad von r nach v. Jeder solche Pfad endet auf einer letzten Kante (u, v) mit $d(u) = k$. Nach Induktionsvoraussetzung ist der Teilpfad von r nach u eindeutig. Da Bäume nicht zulassen, dass ein Knoten mehrere Eltern hat, gibt es genau einen Vorgänger u für v. Somit erhält man den (einzigen) Pfad zu v eindeutig als:

$$(r \longrightarrow u) + (u \longrightarrow v) \tag{2}$$

Daher ist auch der Pfad $r \to v$ eindeutig. Somit gilt auch für jeden anderen Knoten v des Baums, dass sein Pfad w von r bis v einzigartig ist. □

Lemma 3. *Ein Wort c ist einzigartig.*

Beweis. Ein Wort c lässt sich durch seine einzelnen Stellen $c = l_1 l_2 l_3 \dots l_{|c|}$ beschreiben. Diese werden wiederum durch die Zeichen der Kanten seines Pfads generiert. Da in Lemma 1.3 bewiesen wurde, dass der Pfad zu einem beliebigen Wort einzigartig ist, so ist auch der Pfad w_c von c einzigartig, somit auch die Stellenfolge und letztendlich dadurch auch das Wort c. □

Aus Lemma 1.2 folgt, dass an keinem Blatt Kanten angehängt werden können, deren Zeichen neue Wörter erzeugen könnten. Kombiniert man das mit Lemma 1.4, so sieht man sofort, dass kein Wort c_1 ein Präfix von c_2 sein kann. Damit c_1 ein Präfix von c_2 werden könnte, muss $|c_1| \leq |c_2|$ gelten. Weiterhin müsste das Blatt von c_1 im Baum ein Teil des Pfades von c_2 sein. Insgesamt gilt jedoch, dass c_1 abgeschlossen ist und nicht verlängert werden kann, und dass für die zwei unterschiedlichen Wörter in C, die tatsächlichen Wörter unterschiedlich sind $c_1 \neq c_2$. Somit ist ein durch einen Baum generierter Code C stets präfixfrei.

□

Somit kann Huffman's Algorithmus auf einen n-Baum erweitert werden. Folgend wird erläutert, wie der Greedy-Algorithmus funktioniert:
anfänglich werden $|S|$ Knoten, zusammen mit ihren Häufigkeiten $p(s)$ initialisiert. Anschließend werden in jedem Schritt n Knoten mit den geringsten Häufigkeiten in einen inneren Knoten zusammengefasst und dieser initialisiert. Seine „Häufigkeit" entspricht der Summe aller Häufigkeiten der Kinder. Dieses Prozedere wird so lange wiederholt, bis ein innerer Knoten übrig bleibt, der die Wurzel des Baums bildet. Es wird ein Min-Heap verwendet, um die Knoten nach kleinsten Häufigkeiten effizient zu sortieren. Da stets n Knoten ausgewählt werden und dafür ein neuer Knoten initialisiert wird, werden aus dem Min-Heap pro Schritt $n-1$ Knoten entfernt. Da schlussendlich genau 1 Knoten als Wurzel des Baums übrig bleiben muss, muss gelten:

$$|S| \bmod (n-1) = 1 \tag{3}$$

und somit auch:

$$(|S| - 1) \bmod (n - 1) = 0 \tag{4}$$

Da das jedoch nicht immer gilt, müssen, falls die Gleichung nicht ursprünglich erfüllt ist, „Dummy-Knoten“ mit einer Häufigkeit von 0 initialisiert werden. So wird die Gleichung erfüllt und sie verändern die Greedy-Entscheidung nicht. So werden Symbole mit einer niedrigeren Frequenz öfter in neue Knoten zusammengefasst und befinden sich „tiefer“ im Baum, besitzen dadurch also auch längere Codewörter. So kann der Baum konstruiert werden, dessen Blätter jeweils das Codewort für ein Symbol darstellen. Die tatsächlichen Codes können dann durch eine triviale Traversierung des Baums erlangt werden; dabei werden die kürzesten Codes jeweils den Symbolen mit der größten Häufigkeit zugeordnet. Zwei Codewörter c_1 und c_2 mit der selben Ziffernanzahl $|c_1|$ und $|c_2|$ haben auch dieselben Kosten $d(c_1)$ und $d(c_2)$, da einzelne Ziffern/Kodierungsbuchstaben keine variierenden Kosten haben. Somit ist die Zuweisung ebenfalls trivial. Die entstehenden Codes durch diesen Greedy-Algorithmus sind im Allgemeinen stets gut und approximieren eine optimale Lösung. Für das vorgestellte Problem liefert der Algorithmus tatsächlich stets die optimale Lösung. Dies soll im Anschluss bewiesen werden:

Theorem 1.2. *Der Greedy Huffman-Algorithmus liefert einen präfixfreien Code, dessen Länge für den zu kodierenden Text minimal, somit auch optimal ist.*

Lemma 4. *Ein vom Algorithmus erzeugter Code C ist stets präfixfrei.*

Beweis. Da der Algorithmus einen Baum erzeugt, ist durch Theorem 1.1 bewiesen, dass ein durch einen Baum erzeugter Code C stets präfixfrei ist. □

Lemma 5. *Seien $s_1, \ldots, s_n$ die n Symbole mit kleinster Wahrscheinlichkeit. Dann existiert immer ein optimaler Codebaum, in dem jene Knoten von $s_1, \ldots, s_n$ als Geschwister auf maximaler Tiefe existieren.*

Beweis. Sei T^* ein beliebiger optimaler Baum. Falls in T^* ein Blatt von s_i nicht zu den n tiefsten Blättern gehört, dann kann er mit einem Blatt höherer Wahrscheinlichkeit, das tiefer sitzt, ausgetauscht werden. Ein jeder solcher Austausch erhöht die Gesamtlänge nicht, da eine geringere Wahrscheinlichkeit an eine größere Tiefe und somit auch zu höheren Kosten verschoben wird. Nach endlich vielen Schritten liegen $s_1, \ldots, s_n$ gemeinsam in der tiefsten Ebene als Geschwister. □

Lemma 6 (Optimal Substructure)**.** *Wenn die Symbole $s_1, \ldots, s_n$ eines optimalen Baums T durch ein zusammenfassendes Zeichen s' mit $p_{s'} = \sum_{i=1}^{n} p_i$ ersetzt werden und auf der reduzierten Menge der Zeichen $S' = (S \setminus \{s_1, \ldots, s_n\}) \cup \{s'\}$ ein neuer Baum T' konstruiert wird, dann ist dieser auch optimal.*

Beweis. Wäre der reduzierte Baum nicht optimal, so ließe sich seine erwartete Länge durch einen besseren Baum mit denselben Zeichen S' senken. Durch anschließendes Aufspalten von s' in $s_1, \ldots, s_n$ an der tiefsten Stelle würde man dann einen Code für S mit geringerer Länge als durch T erhalten, das jedoch der Optimalität von T widerspräche. □

Das Theorem soll durch vollständige Induktion auf S bewiesen werden.
Induktionsanfang: Für $|S| \leq n$ erzeugt der Algorithmus in einem einzigen Schritt den Wurzelknoten, der alle Symbole als Blätter beherbergt. Ein beliebiger Präfixcode mit $|S| \leq n$ Symbolen weist dieselbe Struktur auf und ist optimal. Somit ist auch Greedy für diesen Fall trivialerweise optimal.
Induktionsvoraussetzung: Es wird angenommen, dass für $k < |S|$ Zeichen der Greedy-Algorithmus optimale Codes generiert.
Induktionsschritt: Sei nun $|S| > n$.

1. Wähle $s_1, \ldots, s_n$ mit den kleinsten Wahrscheinlichkeiten.

2. Nach Lemma 5 kann man jeden optimalen Baum so umformen, dass $s_1, \ldots, s_n$ als Geschwister in maximaler Tiefe vereint sind.

3. Fasse diese n Blätter zu einem Zeichen s' mit $p' = \sum_{i=1}^{n} p_i$ zusammen. Nun gilt $|S'| = |S| - (n-1)$.

4. Nach Lemma 6 ist der verbleibende Teilbaum optimal für die reduzierte Menge S'.

5. Per Induktionsvoraussetzung erzeugt der Greedy-Algorithmus auf dieser kleineren Menge $|S'| < |S|$ einen optimalen Code.

6. Durch Aufspaltung von s' in $s_1, \ldots, s_n$ an der tiefsten Ebene erhält man einen optimalen Code für die ursprüngliche Menge S.

Da nach Lemma 4 der erzeugte Code C präfixfrei ist, ist bewiesen, dass der Algorithmus über die Menge S einen Code C mit minimaler Länge über S erzeugt.

□

1.4 Karp's IP-Modell

Richard Karp präsentiert in seinem Paper „Minimum-Redundancy Coding for the Discrete Noisless Channel"[4] ein ganzzahliges Programm (IP) zum Lösen des in Aufgabenteil b) gestellten Problems, der Huffman-Kodierung mit variierenden Buchstabenkosten. Gegeben ist die Menge $A = \{i_0, i_1, \ldots, i_{n-1}\}$ aller zu verwendenden n Buchstaben und die zu kodierenden Symbole $S = \{s_0, s_1, \ldots, s_m\}$ mit ihren dazugehörenden normierten Wahrscheinlichkeiten $\rho(s)$. Außerdem die Menge D, die alle möglichen Gewichte $d, d \in D$ enthält. Eine Variable $x_{s,d}$ wird initialisiert, die 1 ist, genau dann, wenn das Kodierungswort von s ein Gewicht von d hat. Um zu garantieren, dass das IP einen präfixfreien Code erzeugt, wird die Struktur eines Baums implizit ausgenutzt, da dieser, wie in Kapitel 1.2 bewiesen wurde, stets einen präfixfreien Code bildet. Dafür wird eine Variable b_d eingeführt. Insgesamt wird das IP-Modell folgendermaßen formuliert:

$$x_{s,d} = \begin{cases} 1 & \text{if symbol } i \text{ is assigned cost } d, \\ 0 & \text{otherwise} \end{cases}$$

b_d: Anzahl an inneren Knoten auf dem Kostenlevel d

$$D = [c_{min}, d_{\max}], d \in D$$

$$\begin{aligned}
\text{minimiere} \quad & \sum_{s \in S} \sum_{d \in D} d \cdot p_s \cdot x_{s,d} && && (5) \\
\text{sodass} \quad & \sum_{d \in D} x_{s,d} = 1 && \forall\, s \in S && (6) \\
& b_d + \sum_{s \in S} x_{s,d} \;\leq\; \sum_{i \in A} b_{d-c_i} && \forall\, d \in D && (7) \\
& b_0 = 1 && && (8) \\
& b_d = 0 && \forall\, d < 0 && (9) \\
& x_{s,d} \in \{0, 1\} && && (10) \\
& b_d \in \mathbb{Z}_{\geq 0} && && (11)
\end{aligned}$$

Jedem Wort darf nur ein Kostenlevel d zugeschrieben werden. Deshalb muss (5) gelten. (6) garantiert die Baumstruktur; wie genau, soll folgend erläutert werden. Für Bäume gilt allgemein, dass jeder Knoten, wenn er nicht die Wurzel darstellt, genau ein Elternteil hat. Die linke Seite der Gleichung $b_d + \sum_{s \in S} x_{s,d}$ zählt die Anzahl der Knoten auf Kostenlevel d, da b_d die Anzahl

an inneren Knoten, der andere Summand die Anzahl der Blätter zählt. Die rechte Seite zählt die Anzahl der Elternteile der Knoten auf Kostenlevel d. Für einen vollständigen Baum, für den alle Buchstaben/Kanten die Kosten/das Gewicht 1 haben, würde gelten:

$$b_d + \sum_{s \in S} x_{s,d} = n \cdot b_{d-1} \tag{12}$$

Durch variierende Buchstabenkosten hat das Elternteil eines Knoten mit den Kosten d nicht zwingend die Kosten $d-1$, stattdessen $d - c_i$, wenn c_i die Kosten sind, mit denen der Knoten von seinem Elternteil entstammt und der Knoten das i-te Kind ist. Wir müssen also über alle Buchstabenkosten c_i iterieren, um die Eltern der Knoten auf Kostenlevel $d - c_i$ alle mit einbeziehen zu können. Da im echten, konstruierten Baum, nicht jeder innere Knoten immer auch n Kinder hat, entsteht die Ungleichung. der Koeffizient n fällt weg, da wir durch das iterieren über die Kosten auch die Anzahl aller Buchstaben n berücksichtigen. So garantiert (6) also eine Baumstruktur. Da alle $x_{s,d}$ in der Gleichung als Blattknoten gewertet werden, wird hierbei also auch ein präfixfreier Code erzeugt. b_0 stellt die Wurzel des Baumes dar; da es nur eine Wurzel gibt, ist $b_0 = 1$. Für alle $b_d, d < 0$ existieren keine inneren Knoten, da alle Kosten positive ganze Zahlen sind. Noch nicht geklärt ist das Intervall D, dessen Grenzen relevant für die Gültigkeit des Modells ist. Die untere Grenze kann einfach ermittelt werden, denn kein Codewort kann geringere Kosten haben, als die eines Buchstabens mit geringsten Kosten. Die obere Grenze $d_{\max}$ ist hingegen nicht so trivial zu berechnen. Einerseits kann die Grenze durch simple Approximationen wie $|S| \cdot c_{max}$, denn ein beliebiges Codewort a sollte nie mehr Ziffern haben, als es zu kodierende Symbole gibt; diese können maximal die Kosten c_{max} haben. Allerdings wird schnell ersichtlich, dass ein zu lockeres Abschätzen der Grenzen schnell zu schlechten Laufzeiten führt. Somit besteht die Notwendigkeit einer engeren Approximation des Intervalls. Karp verwendet in seinem Paper die Kraft-Ungleichung, um Bedingungen für die Baumstruktur herzuleiten[4][5]; diese soll nun auch zur Approximation von $d_{\max}$ verwendet werden. Karp stellt folgende Formel zur Existenz eines präfixfreien Codes auf:

$$\sum_{s \in S} p_s r^{-\ell_s} \leq 1 \tag{13}$$

wobei p_s die Häufigkeit des Zeichens s und ℓ_s die Kosten des Codewortes von s darstellt. r lässt sich wiederum direkt durch eine Gleichung berechnen, die aus der Kraft-Ungleichung folgt:

$$\sum_{i \in A} r^{c_i} \geq 1 \tag{14}$$

Da wir r so gestalten wollen, dass die Grenze letztenendes so eng wie möglich gestaltet wird, kann daraus folgende Gleichung folgen:

$$\sum_{i \in A} r^{c_i} = 1 \tag{15}$$

Aus (13) folgt für ein bestimmtes ℓ_s:

$$p_s r^{-\ell_s} \leq 1 \implies r^{\ell_s} \geq \frac{1}{p_s} \implies \ell_s \geq -\log_r p_s \tag{16}$$

So gilt diese Ungleichung für alle möglichen Kosten ℓ der Codewörter von s. Weil ℓ_s eine ganzzahlige Variable ist, folgt:

$$\ell_s \geq \lceil -\log_r p_s \rceil \tag{17}$$

Da wir ein obere Grenze für ℓ_s finden wollen, sollte p_s maximal sein, um für alle ℓ_s gültig zu bleiben. Somit legen wir die Grenzen für ℓ_s folgendermaßen fest:

$$\min_i c_i \leq \ell_s \leq \max_s \lceil -\log_r p_s \rceil \tag{18}$$

und somit auch die Grenzen des Intervalls von $D = [d_{min}, d_{\max}]$. Es gilt folgend $d_{min} = \min_i c_i$ und $d_{\max} = \max_s \lceil -\log_r p_s \rceil$. Nach Lösen des Modells kann aus den Variablen $x_{s,d}$ entnommen werden, wie viele Variablen welche Kosten d haben. Mit diesen Informationen soll dann implizit der dazugehörige Baum konstruiert werden, um die Kodierung für die einzelnen Symbole zu erhalten. Dafür wird durch eine Breitensuche der Baum konstruiert; immer wenn die Kosten eines Knotens mit einem Wert des IP's übereinstimmt, wird das Codewort des Knotens gespeichert und der Knoten selbst zum Blatt gemacht. Wenn alle Werte zu einem Blatt zugewiesen wurden, terminiert der Algorithmus. Die Zuweisung von Symbol und Kodierung erfolgt wie beim Greedy-Algorithmus; das häufigste Symbol wird dem Codewort mit den geringsten Kosten zugewiesen. Dieses Lösungsverfahren produziert immer optimale Lösungen. Dies soll folgend bewiesen werden:

Theorem 1.3. *Mit den Mengen S und A erzeugt das formulierte IP-Modell von (5)-(11) eine optimale, ganzzahlige Lösung (x^*, b^*), dessen korrespondierender Code minimale Kosten $\sum_s p_s \ell_s$ hat.*

Lemma 7. *Jede ganzzahlige Lösung (x, b) des IP-Modells entspricht bijektiv genau einem präfixfreien Codebaum auf S.*

Beweis. Aus (x, b) kann stufenweise ein Baum erstellt werden: Auf Ebene $d, d \in D$ erzeugt man $\sum_s x_{s,d}$ Blätter und b_d innere Knoten und verbindet sie mit den Ebenen $d - c_i$. Die Ungleichung

$$b_d + \sum_s x_{s,d} \;\leq\; \sum_i b_{d-c_i}$$

bedingt genau genug Eltern und keine Zyklen, da, wie oben erklärt, diese genau eine Baumstruktur garantiert. Das Resultat ist ein präfixfreier Baum, dessen Blätter den Paaren (s, d) mit $x_{s,d} = 1$ entsprechen.
Sei nun ein beliebiger präfixfreier Baum T gegeben. Für jedes Blatt s, setze $x_{s,d_s} = 1$ wenn d_s die Tiefe von s darstellt, und für jede Ebene d, sei b_d die Anzahl innerer Knoten auf Ebene d. Da (7) ohnehin eine Baumstruktur forcieren soll, erfüllt die daraus entstehende Lösung (x, b) auch alle IP-Constraints. □

Lemma 8. *Durch die Bijektion aus Lemma 7 gilt:*

$$\sum_{s,d} d\, p_s\, x_{s,d} = \sum_s p_s\, \ell_s \tag{19}$$

Die Zielfunktion des IP-Modells stimmt mit der erwarteten Codelänge des Baums überein.

Beweis. Jedes Zeichen s hat eine Häufigkeit p_s und Tiefe des korrespondierenden Knotens im Baum ℓ_s. Jedes Zeichen trägt also $p_s \cdot \ell_s$ zur Gesamtlänge bei. Im IP wird es durch das eindeutige $d = \ell_s$ mit $x_{s,d} = 1$ erfasst, also summiert sich genau $\sum_s p_s\, \ell_s$. □

Nun soll die Aussage mit Hilfe von Lemma 7 und 8 bewiesen werden.

1. Sei C^* ein beliebiger optimaler präfixfreier Code mit Kosten $L^* = \sum_s p_s\, \ell_s^*$. Nach Lemma 7 entspricht C^* einer Lösung (x', b'), und nach Lemma 8 hat diese Lösung die Zielfunktion L^*. Daher gilt:
$$\min_{\text{IP}} \text{Obj} \leq L^* \tag{20}$$

2. Sei (x^*, b^*) eine optimale Lösung des IP. Nach Lemma 7 gibt es einen zugehörigen Code C, der wiederum nach Lemma 8 eine erwartete Länge von $\min_{\text{IP}}$ Obj hat.

3. Da L^* die minimale erwartete Länge aller präfixfreien Codes ist, gilt:

$$L^* \leq L(C(x^*, b^*)) = \min_{\text{IP}} \text{Obj} \tag{21}$$

Zusammen mit Punkt 1 folgt:

$$\min_{\text{IP}} \text{Obj} \leq L^* \leq \min_{\text{IP}} \text{Obj} \tag{22}$$

$$\min_{\text{IP}} \text{Obj} = L^* \tag{23}$$

So ist die Zielfunktion des IP gleich der minimalen erwarteten Länge L^* des Codes C^*. Somit sei bewiesen, dass eine Lösung des Modells (x^*, b^*) einen optimalen präfixfreien Code bedingt.

1.5 Greedy-Heuristik

Für sehr große Eingaben mit vielen zu kodierenden Symbolen, dessen Bäume große Tiefen haben, kann das IP-Modell durch die große Laufzeit an seine Grenzen kommen. Für diese Eingaben kann eine Greedy-Heuristik verwendet werden, dessen Algorithmus sich nach Huffman's Algorithmus (Kapitel 1.3) richtet. Dabei werden für jede Iteration, bei der n Knoten in einen inneren Knoten zusammengefasst werden, die Knoten mit der größten Frequenz an die Kante mit den geringsten Kosten angefügt. Jeder Knoten, der kein Blatt ist, hat genau n Slots, die jeweils die Kosten der Kodierungsbuchstaben tragen. Dieses Verfahren bringt oft gute und gelegentlich auch optimale Ergebnisse, ist jedoch eine Heuristik; dies liegt daran, dass im Beweis der Optimalität des Huffman-Algorithmus' in Lemma 5 davon ausgegangen wird, dass die n Symbole mit geringsten Wahrscheinlichkeiten auch eine Tiefe d teilen. Falls dies nicht von Beginn an der Fall sein sollte, können Symbole schlichtweg getauscht werden, wobei dies nicht zu einer Verschlechterung des Ergebnisses führt. Diese Argumentation kann nicht mehr geführt werden, da die n Symbole mit geringster Wahrscheinlichkeit, wie auch aus der Formulierung des IP hervorgeht, nicht auf derselben Tiefe d und somit dieselben Kosten teilen müssen. Somit ist dieses Verfahren lediglich heuristisch, ist aber für große Eingaben dennoch gut bis sehr gut. Es wird für die Eingaben von BwInf nicht benötigt, allerdings für größere Eingaben.

2 Implementierung

Generell wurden alle beschriebenen Lösungen in Python 3.11 umgesetzt. Alle Programme, dessen Ergebnisse im nächsten Kapitel präsentiert werden, liefen auf einem MacOS-System mit einem 2.6 GHz 6-Core Intel i7 single Thread mit einem zugewiesenen Hauptspeicher von maximal 12 GB. Es werden folgend Pseudocodes für Algorithmen vorgestellt, ihre Zeitkomplexität analysiert und Einzelheiten erklärt.

2.1 Greedy Huffman-Algorithmus

Die Min-Heap, die verwendet wird, um die Knoten in der Queue stets nach kleinstem Gewicht/kleinster Häufigkeit zu sortieren, wird als Priority Queue umgesetzt. Dafür wird die Python-Bibliothek erwendet, die die Heapstruktur implementiert und somit für ein schnelles Abrufen sorgt. In Zeile 6-9 werden, aus den oben genannten Gründen, Dummy-Knoten zusätzlich der Queue hinzugefügt, falls die Gleichung $(|PQ| - 1) \bmod (n - 1) = 0$ nicht gilt. Anschließend beginnt das Iterieren über die Priority Queue. Es werden immer n Knoten zu einem neuen zusammengefügt. Zeile 16 berechnet die Summe aller Häufigkeiten der n Knoten, weil dies als neue Häufigkeit für den neuen inneren Knoten verwendet wird. Wenn $m = |S|$ gilt, dann wird insgesamt $\lceil (m-1) : (n-1) \rceil$ über die while-Schleife iteriert, da bei jeder Iteration $(n-1)$ Elemente aus der Priority Queue entfernt werden. Das Entfernen eines Elements aus der Priority Queue,

Algorithm 1 Aufstellen des Huffman-Baums für >2 Elemente

```
1:  function BuildHuffman(n, S)
2:      PQ ← leere Priority Queue                          ▷ Sortierung nach p_s
3:      for all Leaves s in frequencies do
4:          push (s, p_s) to PQ
5:      end for
6:      if (|PQ| − 1) mod (n − 1) ≠ 0 then                 ▷ Ermittlung von Dummy-Knoten
7:          for d ← 0 to (n − 1) − ((|PQ| − 1) mod (n − 1)) do
8:              push (d, 0) into PQ
9:          end for
10:     end if
11:     while |PQ| > 1 do
12:         children ← [ ]
13:         for i ← 0 to min(n, |PQ|) do                   ▷ Wenn n > |PQ|
14:             append pop(PQ) to children
15:         end for
16:         weight ← Σ_{c∈children} c.weight
17:         push (None, weight) to PQ
18:     end while
19:     return pop(Q)
20: end function
```

Zeitkomplexität: $O(m \log m)$ mit $m = |S|$
Platzkomplexität: $O(m)$ mit $m = |S|$

modelliert als Heap, dauert $O(|PQ|)$, also anfangs $O(m)$. Da für jede Iteration n Knoten aus der Queue entfernt werden müssen, ist die Zeitkomplexität dafür $O(n \log m)$. Daraus ergibt sich folgende Zeitkomplexität:

$$\lceil \frac{(m-1)}{(n-1)} \rceil \cdot O(n \log m) = O(\frac{m}{n-1} \cdot n \log m) = O(n \cdot \frac{m \log m}{n-1}) \implies O(m \log m) \tag{24}$$

Dies gilt, wenn n konstant ist. Die Queue besitzt höchstens $m+(n-1)$ Elemente, wobei $n-1$ die Maximalzahl der Dummies darstellt. Insgesamt ist die Summe der Knoten $\lceil \frac{(m-1)}{(n-1)} \rceil + m + (n-1)$. Für die Platzkomplexität gilt also:

$$\lceil \frac{(m-1)}{(n-1)} \rceil + m + (n-1) = (1 + \frac{1}{n-1}) \cdot m + (n-1) - \frac{1}{n-1} \implies O(1) \cdot m + \ldots \implies O(m) \tag{25}$$

Interessant ist nun die Weise, die Baumstruktur zu speichern, um die Codes später durch eine Traversierung wiedererlangen zu können. Dazu wird eine Knotenklasse `Node` erstellt, in der stets Häufigkeit, korrespondierendes Symbol, Code und Kinder gespeichert werden. Im oben erklärten Algorithmus wird lediglich die Baumstruktur generiert, nicht jedoch die Codes. Diese werden anschließend erst in der Traversierung zugewiesen. Sowohl die Zuweisung der Codes, als auch das Erlangen der Codes geschieht auf eine naive, rekursive Weise.

Algorithm 2 Zuweisung von Codewörtern

```
1: function AssignCodes(node: Node, code, n)
2:     node.code ← code
3:     for i = 0 to |node.children| do
4:         child ← node.children[i]
5:         if child.symbol not None or child.children ≠ ∅ then
6:             AssignCodes(child, code + i , n)
7:         end if
8:     end for
9: end function
```

Zeitkomplexität: $O(N)$ mit $N = \lceil \frac{(m-1)}{(n-1)} \rceil + m$

Nun haben alle Knoten, bis auf die Wurzel, einen zugewiesenen Code. Nun wird der Baum traversiert und die Codes der Blattknoten ausgelesen.

Algorithm 3 Codewörter speichern

```
1: function GetCodes(root)
2:     codes ← empty dictionary
3:     function Traverse(node)
4:         if node.children =∅ then                ▷ Wenn node ein Blatt ist
5:             codes[node.symbol] ← node.code
6:         end if
7:         for child in node.children do
8:             Traverse(child)
9:         end for
10:    end function
11:    Traverse(root)
12:    return codes
13: end function
```

Zeitkomplexität: $O(N)$ mit $N = \lceil \frac{(m-1)}{(n-1)} \rceil + m$

`codes` ist nun eine Hashmap, die für jedes Symbol $s \in S$ speichert, welches Codewort zugewiesen wurde. Beide Algorithmen besuchen jeden Knoten des Baums in einer Tiefensuche. Somit ist die Zeitkomplexität beider Algorithmen die Knotenanzahl des Baums N. Der gesamte Algorithmus läuft also wie folgt ab:

Algorithm 4 Huffman Code mit n Zeichen

```
1: function GetHuffmanCodes(n, S)
2:     root = BuildHuffman(n, S)
3:     AssignCodes(root, " ", n)
4:     codes = GetCodes(root)
5:     return codes
6: end function
```

Zeitkomplexität: $O(m \log m)$ mit $m = |S|$

Es gilt stets $n \geq 2$, da es trivial ist, dass nur dann ein präfixfreier Code existieren kann, wenn $|S| > 1$ gilt. Für N gilt nach wie vor $N = m + \frac{m-1}{n-1}$ und mit $n \geq 2$ gilt $0 < \frac{1}{n-1} \leq 1$ und somit auch $1 \leq 1 + \frac{1}{n-1} \leq 2$. Mit $N = m + \frac{m}{n-1} - \frac{1}{n-1}$ lässt sich einfach erkennen, dass folgendes gilt:

$$m \leq N \leq 2m \tag{26}$$

Da für Big-Θ (Average Case) folgendes gilt:

$$\begin{aligned} f(n) &= \Theta\big(g(n)\big) \\ k_1\, g(n) &\leq f(n) \leq k_2\, g(n) \end{aligned} \tag{27}$$

ist klar zu erkennen, dass dies auch auf N anzuwenden ist, und somit gilt:

$$N = \Theta(m) \implies N = O(m) \land m = O(N) \tag{28}$$

Damit lässt sich die Gesamtlaufzeit des Algorithmus akkurat ermitteln, und zwar mit:

$$O(m \log m) + O(N) + O(N) = O(m \log m) + 2 \cdot O(m) \implies O(m \log m) \tag{29}$$

2.2 IP-Modell

Das IP-Modell wird mit Hilfe der Python-Bibliothek `pyscipopt`[1] implementiert und gelöst. Das Lösen des Modells gibt ein Array an Zahlen zurück, wobei jede Zahl die Kosten eines Codeworts für ein Symbol darstellt. Um den tatsächlichen Code zu generieren, wird über den impliziten Baum eine Standard-Breitensuche durchgeführt. Folgend wird skizziert, wie der Algorithmus funktionieren soll: Dabei erstellt MakeModel() das IP-Modell, SolveModel() löst das Modell

Algorithm 5 Huffman-Codes mit IP-Modell

```
1: function GetCodesIP(S, A)
2:     ip_model = MakeModel( )
3:     code_cost = SolveModel(ip_model)
4:     codes = RetrieveCodes(code_cost)
5:     return codes
6: end function
```

und gibt genanntes Array zurück, RetrieveCodes generiert den tatsächlichen Code zuletzt.

MakeModel() Das in Kapitel 1.3 formulierte Modell soll nun implementiert werden. Genauer gesagt wird die Klasse `Model` der Bibliothek `pyscipopt` verwendet, die zahlreiche Methoden und Funktionalitäten für das Implementieren von linearen Programmen bietet, unter anderem auch den IP-Solver. Das stumpfe Erstellen von Variablen und Constraints zu erläutern, stellt sich als trivial und auch öde heraus. Stattdessen soll die Anzahl an Variablen und Constraints betrachtet werden, da dies hauptsächlich die Laufzeit des Solvers beeinflusst. Das Erstellen von $x_{s,d}$, das Erstellen der Constraints $\sum_{d \in D} x_{s,d} = 1 \; \forall s$ sowie die Zielfunktion erfordern das Iterieren über die Mengen S und D. Daraus lässt sich schließen, dass jeder der soeben genannten Prozesse eine Zeitkomplexität von $O(|S| \cdot |D|)$ hat. Es werden insgesamt $|D| + 1$ b_d-Variablen erstellt, wobei b_0 berücksichtigt werden muss. (7) erfordert das Iterieren über alle alle Elemente von D sowie über alle Kosten c_i, somit über die Kosten der Elemente n. Somit also eine Zeitkomplexität von $O((|D| + 1) + (|D| \cdot n))$. Insgesamt ergibt das also eine Zeitkomplexität von:

$$3 \cdot O(|S| \cdot |D|) + O((|D| + 1) + (|D| \cdot n)) \tag{30}$$

Da D höchstens eine Zeitkomplexität von $O(d_{\max})$ hat, n offensichtlich konstant ist und für $m = |S|$ ist, gilt:

$$3 \cdot O(|D| \cdot m) + O(|D| + 1) + O(|D| \cdot n) \implies O(|D| \cdot m + |D| \cdot n) \tag{31}$$

Die Zeitkomplexität der Erstellung des Modells ist näherungsweise linear, zumindest für alle typischen Beispiele der Huffman-Kodierung, bei denen die Anzahl an Kodierungssymbolen n

[1] `https://pyscipopt.readthedocs.io/en/latest/whyscip.html`, zuletzt aufgerufen: 24. April 2025

hinreichend klein ist und im standardmäßigen ASCII höchstens 128 Zeichen vorkommen, wobei sich die häufigsten auf die 26 Buchstaben des lateinischen Alphabets beschränken werden. Man könnte für die meisten Beispiele und Anwendungsfälle die Zeitkomplexität stark zusammenfassen und insgesamt auf folgendes runterbrechen:

$$O(|D| \cdot m + |D| \cdot n) \implies O(m) \tag{32}$$

Zum Lösen des Faktors r, der benötigt wird, um $d_{\max}$ zu bestimmen, wird aus der Python-Bibliothek `scipy` die Funktion `newton` verwendet, um die Gleichung (15) zu lösen. Außerdem wird in der Anwendung eine kleine Zahl zu $d_{\max}$ dazuaddiert und somit das Intervall D vergrößert. Das liegt daran, dass in manchen Beispielen, diese Approximation für $d_{\max}$ zu eng anliegend ist und es dadurch keine valide Lösung für das IP-Modell gibt. Das Intervall muss also also vergrößert werden, um für manche Beispiele lösbar zu sein. Es ist trivial, dass dies vor allem für Beispiele geschieht, in denen n klein ist und stark schwankende Gewichte hat, und $|S|$ recht groß, da so viele Zeichen mit wenigen Buchstaben kodiert werden müssen.

SolveModel() Der Solver von `SCIP` löst das ihm gegebene Modell. Dabei verwendet er nicht einen gegebenen Algorithmus wie Simplex, sondern das `SCIP`-eigene MIP-Framework[2]. Genaueres soll an dieser Stelle zu dem Thema nicht erklärt werden, da keine Parameter zusätzlich hinzugefügt oder verändert wurden. In der Dokumentation von `SCIP`[3] kann mehr zum Lösungsverfahren gefunden werden. Nach Lösen des Modells muss aus den Variablen extrahiert werden, mit welchen Kosten die Zeichen von S kodiert werden. Dies geschieht auf naive Weise über das Iterieren aller Variablen $x_{s,d}$.

Algorithm 6 Alle d aus der IP-Lösung extrahieren

```
1: function Solve(model, x)
2:     model.Optimize( )
3:     lengths ← empty list
4:     for all (s, d) in x do
5:         if x_{s,d} = 1 then
6:             append d to lengths
7:         end if
8:     end for
9:     return lengths
10: end function
```

Zeitkomplexität: $O(|D| \cdot m)$ mit $m = |S|$

Mit diesem Array `lengths` können nun die einzelnen Codewörter und somit der gesamte Code C im nächsten Schritt erstellt werden. Insgesamt kann die Zeitkomplexität des IP-Solvers nicht klar bestimmt werden. Das Integer-Programming-Problem ist generel NP-schwer und so entwickeln sich viele Solver exponentiell zum Verhältnis der Anzahl der binären Variablen.

RetrieveCodes() In diesem Schritt wird durch eine Breitensuche der Baum implizit traversiert, um die Codewörter zu erlangen. Dazu wird das Array `lengths` aus dem letzten Schritt benötigt. Folgend ist zu sehen, wie der Algorithmus funktioniert.

[2] Mixed-Integer-Programing-Framework

[3] `https://pyscipopt.readthedocs.io/en/latest/whyscip.html`, zuletzt aufgerufen: 24. April 2025

Algorithm 7 BFS zur Codewortzuweisung

```
1: function ImplicitBFS(d_max, lengths, costs)
2:     n ← |costs|
3:     codes ← empty dictionary
4:     Q ← empty Queue
5:     Push Node(code = " ", cost = 0) to Q
6:     while not Q.empty() do
7:         node ← Q.pop()
8:         if node.cost ∈ lengths then
9:             remove node.cost from lengths
10:            codes[node.code] ← node.cost
11:            if |lengths| = 0 then
12:                return codes
13:            end if
14:        else if node.cost > d_max then
15:            continue                          ▷ Keine Kosten > d_max, skip
16:        else
17:            for i = 0 to n do
18:                child ← Node(node.code + str(i), node.cost + costs[i])
19:                Q.push(child)
20:            end for
21:        end if
22:    end while
23:    return costs
24: end function
```

Zeitkomplexität: $O(N)$ mit $N = \lceil \frac{(m-1)}{(n-1)} \rceil + m$ und $m = |S|$

Es wird über alle Knoten des Baums iteriert. Dadurch ist es recht irrelevant, ob eine Tiefen- oder Breitensuche angewandt wird, solange sich nur auf die Knoten des Baums beschränkt wird. So gilt für die Zeitkomplexität also:

$$O(\lceil \frac{(m-1)}{(n-1)} \rceil + m) \implies O(m) \tag{33}$$

2.3 Greedy-Heuristik

Der Algorithmus ist quasi der selbe wie beim Huffman-Algorithmus. Zumindest ist der Prozess und das Auslesen der Codes genau der selbe. Somit wird hier nur kurz erklärt wie genau die Heuristik umgesetzt wurde.

Algorithm 8 Greedy-Heuristik für ungleiche Buchstabenkosten

```
 1: function BuildHuffman(C, S)
 2:     n = |C|
 3:     PQ ← leere Priority Queue                          ▷ Sortierung nach p_s
 4:     for all Leaves s in frequencies do
 5:         push (s, p_s) to PQ
 6:     end for
 7:     if (|PQ| − 1) mod (n − 1) ≠ 0 then                 ▷ Ermittlung von Dummy-Knoten
 8:         for d ← 0 to (n − 1) − ((|PQ| − 1) mod (n − 1)) do
 9:             push (d, 0) into Q
10:         end for
11:     end if
12:     branch_positions ← [0, 1, ..., n − 1]
13:     sort branch_positions by ascending c_i             ▷ Sortiert Buchstaben nach Kosten
14:     while |PQ| > 1 do
15:         children ← [ ]
16:         for i ← 0 to n do
17:             append pop(PQ) to children
18:         end for
19:         arranged ← array of length n, all entries ← None
20:         for i = 1 to |children| do
21:             child ← children[i]
22:             slot ← branch_positions[i]
23:             arranged[slot] ← child                     ▷ Sortiert Kinder absteigend nach Frequenz
24:         end for
25:         weight ← ∑_{c∈arranged} c.weight
26:         node = (None, weight)
27:         node.children = arranged
28:         push node to PQ
29:     end while
30:     return pop(Q)
31: end function
```

Zeitkomplexität: $O(m \log m)$ mit $m = |S|$
Platzkomplexität: $O(m)$ mit $m = |S|$

Die Zeitkomplexität bleibt gleich, da die einzig hinzugefügten Zeilen 12-13 und 19-24 eine Sortierung über jeweils n Elemente durchführen, und diese jeweils eine Zeitkomplexität von $O(n \log n)$ haben. Da für große Instanzen gilt: $n \ll m$ sind die Laufzeiten dieser Sortierungen für die Zeitkomplexität unerheblich. Die hinzugefügten Listen beanspruchen aus selben Grund nicht signifikant mehr Platz. Die Liste `branch_positions` stellt alle Slots eines Knotens da, und zwar sortiert nach aufsteigenden Kosten, während `arranged` die letztendliche Reihenfolge der Knoten, wie sie an den neuen „angehängt“ werden sollen, darstellt. Da die Kinder zuvor nach absteigender Frequenz sortiert wurden, können die zwei sortierten Listen `children` und `branch_positions` „gezippt“ werden.

3 Beispiele

3.1 Gegebene Beispiele

Teilweise werden Tabellen eingekürzt, da sie zu groß sind. Es wird immer angegeben, was die Zeichen des Intervalls sind, in denen die Zeichen eingekürzt wurden. Alle vollständigen Ausgaben

sind im Ordner /output in der Einreichung zu finden. Die Beispiele 0, 00, 01 wurden gemäß des Aufgabenteil a) mit dem Greedy Huffman-Algorithmus gelöst. Allerdings lassen sich auch diese Beispiele mit dem IP-Modell lösen.

3.1.1 schmuck0.txt

Dies ist das Beispiel aus der Aufgabenstellung. Die gefundene Länge sowie der Code entsprechen dem aus der Aufgabenstellung.
Laufzeit zum Lösen des Modells: 0.015 s
Laufzeit für die Erstellung des Codes: 0.001 s
Länge des kodierten Textes: 11.3 cm

Zeichen	Häufigkeit	Kosten	Wort
‘E’	5	3	000
‘ ’	5	3	001
‘I’	4	3	010
‘N’	4	3	011
‘S’	3	3	100
‘D’	2	4	1010
‘O’	2	4	1011
‘L’	2	4	1100
‘R’	2	4	1101
‘M’	2	4	1110
‘C’	1	5	11110
‘H’	1	5	11111

3.1.2 schmuck00.txt

Laufzeit zum Lösen des Modells: 0.03 s
Laufzeit für die Erstellung des Codes: 0.001 s
Länge des kodierten Textes: 115.0 cm

Zeichen	Häufigkeit	Kosten	Wort
‘ ’	24	2	00
‘e’	18	2	01
‘i’	16	2	02
‘t’	10	2	10
‘n’	9	2	11
‘s’	8	3	120
‘h’	8	3	121
‘c’	7	3	122
‘l’	6	3	200
‘a’	4	3	201
‘r’	4	3	202
‘d’	3	3	210
‘D’	3	3	211
‘u’	3	4	2120
‘w’	2	4	2121
‘m’	2	4	2122
‘G’	2	4	2200

‘g’	2	4	2201
‘E’	1	4	2202
‘k’	1	4	2210
‘f’	1	4	2211
‘W’	1	4	2212
‘Z’	1	4	2220
‘P’	1	5	22210
‘o’	1	5	22211
‘?’	1	5	22212
‘A’	1	5	22220
‘b’	1	5	22221

3.1.3 schmuck01.txt

Laufzeit zum Lösen des Modells: 0.032 s
Laufzeit für die Erstellung des Codes: 0.001 s
Länge des kodierten Textes: 37.2 cm

Zeichen	Häufigkeit	Kosten	Wort
‘ ’	85	1	0
‘e’	66	1	1
‘n’	53	2	20
‘r’	34	2	21
‘i’	33	2	22
‘s’	26	2	23
‘l’	22	2	24
‘a’	22	2	30
‘t’	22	2	31
‘h’	17	2	32
‘c’	15	2	33
‘o’	14	2	34
‘g’	14	3	400
‘m’	13	3	401
‘u’	12	3	402
‘.’	10	3	403
‘d’	9	3	404
‘k’	9	3	410
‘E’	8	3	411
‘,’	8	3	412
‘D’	7	3	413
‘b’	7	3	414
‘w’	6	3	420
‘ü’	5	3	421
‘f’	4	3	422
‘S’	4	3	423
‘v’	4	3	424
‘I’	4	3	430
‘p’	4	3	431
‘O’	3	3	432

‘V’	3	3	433
‘ö’	3	3	434
‘z’	3	3	440
‘F’	3	3	441
‘B’	2	3	442
‘ä’	2	4	4430
‘G’	2	4	4431
‘K’	1	4	4432
‘R’	1	4	4433
‘M’	1	4	4434
‘ß’	1	4	4440
‘H’	1	4	4441
‘N’	1	4	4442
‘W’	1	4	4443
‘…’	1	4	4444

3.1.4 schmuck1.txt

Laufzeit zum Lösen des Modells: 0.02 s
Laufzeit für die Erstellung des Codes: 0.001 s
Länge des kodierten Textes: 19.1 cm

Zeichen	Häufigkeit	Kosten	Wort
‘e’	9	2	2
‘t’	6	3	02
‘ ’	5	3	12
‘r’	3	3	000
‘i’	3	3	001
‘n’	3	3	010
‘w’	3	3	011
‘B’	2	4	002
‘I’	2	4	012
‘s’	2	4	102
‘f’	2	4	112
‘„’	2	4	1000
‘b’	2	4	1001
‘W’	1	4	1010
‘N’	1	4	1011
‘F’	1	4	1100
‘h’	1	4	1101
‘ü’	1	5	1002
‘D’	1	5	1012
‘u’	1	5	1102
‘d’	1	5	1112
‘o’	1	5	11100
‘m’	1	5	11101
‘a’	1	5	11110
‘k’	1	5	11111

3.1.5 schmuck2.txt

An der Eingabe sieht man, dass in diesem Beispiel hauptsächlich ein Zeichen, „a“ präsent ist. Außerdem führt das Kodierungsalphabet mit den korrespondierenden Kosten dazu, dass die Kosten für die restlichen Codewörter schnell stark wachsen. Dass das Codewort für „a“ 0 ist, sollte trivial sein.

Laufzeit zum Lösen des Modells: 0.018 s
Laufzeit für die Erstellung des Codes: 0.001 s
Länge des kodierten Textes: 13.5 cm

Zeichen	Häufigkeit	Kosten	Wort
‘a’	33	1	0
‘ ’	1	11	101
‘b’	1	11	110
‘c’	1	11	1000000
‘d’	1	12	1001
‘e’	1	13	10001
‘f’	1	14	100001
‘g’	1	15	111
‘h’	1	15	1000001

3.1.6 schmuck3.txt

Laufzeit zum Lösen des Modells: 0.049 s
Laufzeit für die Erstellung des Codes: 0.001 s
Länge des kodierten Textes: 27.9 cm

Zeichen	Häufigkeit	Kosten	Wort
‘a’	34	2	1
‘b’	34	2	00
‘c’	34	3	2
‘ ’	3	4	02
‘d’	1	5	011
‘e’	1	5	0100
‘f’	1	6	012
‘g’	1	6	0101
‘h’	1	7	0102

3.1.7 schmuck4.txt

In diesem Beispiel haben alle Zeichen die selbe Wahrscheinlichkeit. Dies führt dazu, das die Kosten der Codewörter nicht stark voneinander abweichen und vor allem kein Codewort mit besonders geringen Kosten zugewiesen wird, da dies dafür sorgen würde, dass andere Wörter deutlich teurer werden. Da dies jedoch durch die Verteilung verhindert werden sollte, entsteht in diesem Beispiel ein Kostenintervall von $[7; 12]$.

Laufzeit zum Lösen des Modells: 0.024 s
Laufzeit für die Erstellung des Codes: 0.001 s
Länge des kodierten Textes: 13.7 cm

Zeichen	Häufigkeit	Kosten	Wort

‘a’	1	7	001
‘b’	1	8	0001
‘c’	1	8	0100
‘d’	1	8	1000
‘e’	1	8	00000000
‘f’	1	9	00001
‘g’	1	10	11
‘h’	1	10	000001
‘i’	1	11	011
‘j’	1	11	101
‘k’	1	11	0000001
‘l’	1	12	0101
‘m’	1	12	1001
‘n’	1	12	00000001

3.1.8 schmuck5.txt

Laufzeit zum Lösen des Modells: 0.07 s
Laufzeit für die Erstellung des Codes: 0.002 s
Länge des kodierten Textes: 316.2 cm

Zeichen	Häufigkeit	Kosten	Wort
‘ ’	151	2	2
‘e’	110	2	00
‘t’	71	3	3
‘i’	67	3	02
‘o’	64	3	12
‘s’	61	3	010
‘a’	58	3	011
‘n’	56	3	100
‘r’	51	3	101
‘c’	44	3	110
‘d’	37	3	111
‘l’	33	4	4
‘m’	27	4	03
‘h’	26	4	13
‘u’	25	4	012
‘p’	20	4	102
‘f’	20	4	112
‘b’	16	5	5
‘y’	13	5	04
‘g’	11	5	14
‘.’	8	5	013
‘v’	5	5	103
‘x’	5	6	6
‘q’	5	6	05
‘,’	5	6	15
‘w’	4	6	014
‘T’	3	6	104

‘F’	3	6	114
‘S’	1	6	1130
‘–’	1	7	06
‘A’	1	7	16
‘H’	1	7	015
‘1’	1	7	105
‘9’	1	7	115
‘5’	1	7	1132
‘2’	1	7	11310
‘z’	1	7	11311
‘G’	1	8	016
‘”’	1	8	106
‘j’	1	8	116
‘;’	1	8	1133

3.1.9 schmuck6.txt

Dies ist das erste Beispiel mit asiatischen Zeichen. Obwohl dieses Beispiel noch moderat gehalten wird, stellen asiatische Sprachen diese Weise der Kodierung stark in Frage, da sie deutlich mehr Zeichen besitzen als das standard lateinische Alphabet, bzw. ASCII im allgemeinen.
Laufzeit zum Lösen des Modells: 0.036 s
Laufzeit für die Erstellung des Codes: 0.001 s
Länge des kodierten Textes: 23.4 cm

Zeichen	**Häufigkeit**	**Kosten**	**Wort**
‘，’	3	5	12
‘文’	2	5	21
‘馬’	2	5	002
‘。’	2	5	011
‘有’	2	5	020
‘古’	1	5	101
‘英’	1	5	110
‘雄’	1	5	200
‘未’	1	5	0001
‘遇’	1	5	0010
‘時’	1	5	0100
‘都’	1	6	22
‘無’	1	6	012
‘大’	1	6	021
‘志’	1	6	102
‘非’	1	6	111
‘止’	1	6	201
‘鄧’	1	6	0002
‘禹’	1	6	0011
‘希’	1	6	0101
‘學’	1	6	1001
‘、’	1	6	00001
‘武’	1	6	10000
‘望’	1	6	000000
‘督’	1	7	022

‘郵’	1	7	112
‘也’	1	7	202
‘晉’	1	7	0012
‘公’	1	7	0102
‘妻’	1	7	1002
‘不’	1	7	00002
‘肯’	1	7	10001
‘去’	1	7	000001
‘齊’	1	8	10002

3.1.10 schmuck7.txt

Die Tabelle ist von „,“ bis „P“ eingekürzt, da die Tabelle ansonsten zu groß ist.

Laufzeit zum Lösen des Modells: 0.125 s
Laufzeit für die Erstellung des Codes: 0.019 s
Länge des kodierten Textes: 13455.9 cm → 134.559 m

Zeichen	Häufigkeit	Kosten	Wort
‘ ’	13488	1	0
‘e’	11480	1	1
‘n’	7761	1	2
‘i’	5018	1	3
‘r’	4011	2	7
‘s’	3507	2	40
‘a’	3307	2	41
‘d’	3272	2	42
‘h’	3213	2	43
‘t’	3174	2	44
‘u’	2609	2	45
‘l’	2448	2	46
‘c’	1991	2	50
‘g’	1665	2	51
‘m’	1521	2	52
‘o’	1304	2	53
‘b’	1141	2	54
...	...	...	...
‘1’	16	4	6662
‘4’	13	4	6663
‘3’	9	4	6664
‘?’	8	4	6665
‘2’	7	5	49
‘!’	7	5	59
‘5’	7	5	69
‘6’	6	5	648
‘0’	5	5	658
‘q’	3	5	668
‘Ä’	3	5	6657
‘8’	3	5	6667

‘7’	2	5	66660
‘9’	2	5	66661
‘Q’	2	5	66662
‘ ̈’	2	5	66663
‘Ö’	1	5	66664
‘x’	1	5	66665
‘y’	1	5	66666
‘C’	1	6	649

3.1.11 schmuck8.txt

Die Tabelle ist von „ 都“ bis „ 婦“ eingekürzt. In diesem Beispiel offenbart sich das Problem mit asiatischen Bildsprachen. Es sind insgesamt 320 zu kodierende Zeichen. Die Problematik wird noch moderat durch ein ausreichendes Kodierungsalphabet mit $n = 5$ und ein kleines Kostenintervall $[1; 3]$ ausgeglichen.

Laufzeit zum Lösen des Modells: 0.49 s
Laufzeit für die Erstellung des Codes: 0.014 s
Länge des kodierten Textes: 328.7 cm

Zeichen	Häufigkeit	Kosten	Wort
‘,’	33	3	4
‘。’	27	3	02
‘「’	13	4	04
‘」’	13	4	14
‘》’	12	4	22
‘:’	11	4	23
‘《’	11	4	32
‘之’	10	4	33
‘格’	10	4	002
‘不’	9	4	003
‘有’	7	4	012
‘為’	7	4	013
‘云’	7	4	030
‘時’	6	5	24
‘其’	6	5	34
‘一’	6	5	004
‘武’	5	5	014
‘相’	5	5	032
‘人’	5	5	033
‘詩’	5	5	104
‘風’	5	5	114
‘也’	4	5	122
‘公’	4	5	123
‘是’	4	5	132
‘知’	4	5	133
‘在’	4	5	202
‘誰’	4	5	203
‘;’	4	5	212
‘未’	3	5	213
...	...	...	...

‘率’	1	7	100113
‘事’	1	7	101002
‘今’	1	7	101003
‘能’	1	7	101012
‘範’	1	7	101013
‘圍’	1	7	101102
‘否’	1	7	101103
‘況’	1	7	101112
‘皐’	1	7	101113
‘歌’	1	7	110002
‘國’	1	7	110003
‘雅’	1	7	110012
‘頌’	1	7	110013
‘豈’	1	7	110102
‘定’	1	7	110103
‘哉’	1	7	110112
‘許’	1	7	110113
‘渾’	1	7	111002
‘似’	1	7	111003
‘成’	1	7	111012
‘仙’	1	7	111013
‘里’	1	7	111102
‘莫’	1	7	111103
‘浪’	1	7	111112

3.1.12 schmuck9.txt

Die Tabelle ist von „ す“ bis „ 枠“ eingekürzt. Mit ca. 11 s ist es auch die aufwendigste Beispieleingabe von BwInf. Dies liegt vor allem an den 675 verschiedenen Zeichen, denen lediglich ein Kodierungsalphabet mit 4 Buchstaben gegeben wird. Die Problematik mit asiatischen Zeichensprachen wird nun vollkommen ersichtlich. Die teuersten Codewörter kosten 14 und das längste Codewort besteht aus 12 Stellen, mehr als in einer Kodierung mit $n = 4$ im lateinischen Alphabet je nötig gewesen wäre.

Laufzeit zum Lösen des Modells: 10.957 s
Laufzeit für die Erstellung des Codes: 0.030 s
Länge des kodierten Textes: 3659.7 cm → 36.597 m

Zeichen	Häufigkeit	Kosten	Wort
‘の’	173	5	03
‘た’	139	5	12
‘に’	138	5	21
‘い’	126	5	30
‘し’	123	6	13
‘と’	119	6	22
‘を’	112	6	31
‘て’	111	6	003
‘は’	108	6	012
‘、’	105	6	021
‘な’	89	6	102
‘。’	83	6	111

‘が’	81	6	201
‘る’	79	6	0002
‘で’	72	6	0011
‘こ’	72	6	0020
‘か’	68	6	0101
‘っ’	67	6	0110
‘ら’	61	7	23
‘う’	61	7	32
‘れ’	60	7	013
...	...	...	...
‘割’	1	14	0200000002
‘稲’	1	14	1000000003
‘妻’	1	14	1000000012
‘刻’	1	14	1000000102
‘印’	1	14	1000001002
‘諺’	1	14	1000010002
‘半’	1	14	1000100002
‘円’	1	14	1001000002
‘形’	1	14	1010000002
‘背’	1	14	1100000002
‘載’	1	14	2000000002
‘!’	1	14	00000000003
‘胆’	1	14	00000000012
‘連’	1	14	00000000102
‘応’	1	14	00000001002
‘ぜ’	1	14	00000010002
‘定’	1	14	00000100002
‘関’	1	14	00001000002
‘天’	1	14	00010000002
‘登’	1	14	00100000002
‘ギ’	1	14	01000000002
‘ぺ’	1	14	10000000002
‘拭’	1	14	000000000002

4 Quellcode

4.1 Greedy Huffman-Algorithmus

```python
# Für die Implementierung des Min-Heap (Priority Queue)
import heapq

# Klasse für die Knoten des Huffman Baums
class Node:
    def __init__(self, symbol, frequency, children=None):
        self.symbol = symbol
        self.frequency = frequency
        self.children = children if children is not None else []
        self.code = ""

    def set_freq(self):
        freq = 0
        for child in self.children:
            freq += child.frequency
        self.frequency = freq

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

def build_huffman(n, frequencies):
    pq = []
    for symbol, freq in frequencies.items():
        heapq.heappush(pq, Node(symbol, freq))

    remainder = (len(pq) - 1) % (n - 1)
    if remainder != 0:
        dummy_num = n - 1 - remainder
        for _ in range(dummy_num):
            heapq.heappush(pq, Node(None, 0))

    while len(pq) > 1:
        children = [heapq.heappop(pq) for _ in range(min(n, len(pq)))]
        internal_node = Node(None, None, children=children)
        internal_node.set_freq()
        heapq.heappush(pq, internal_node)

    return pq[0]

def code(n, root):
    def assign_codes(node, code, n):
        node.code = code
        for i, child in enumerate(node.children):
            if child.symbol is not None or child.children:
                assign_codes(child, code + str(i), n)

    def get_codes(root):
        codes = {}
        def traverse(node):
            if node.symbol is not None:
                codes[node.symbol] = node.code
            for child in node.children:
                traverse(child)
        traverse(root)
```

```
        return codes

    assign_codes(root, "", n)
    codes = get_codes(root)
    return codes
```

Listing 1: Huffman-Algorithmus mit n-ären Bäumen in Python

4.2 IP-Modell

```
# Bibliotheken für:
# Erstellung und Berechnung des IP-Modells
from pyscipopt import Model, quicksum
# Berechnen von r für d_{max}
from scipy.optimize import newton
import math
# Erstellen des Tabellenstrings für die Ausgabe
from tabulate import tabulate

# Klasse für die Knoten des Huffman Baums
class Node:
    def __init__(self, code, cost, children=None):
        self.code = code
        self.cost = cost
        self.children = children if children is not None else []

    def add_child(self, child):
        self.children.append(child)

# Implementiert die Queue Datenstruktur
class Queue:
    def __init__(self):
        self.queue = []

    def get(self):
        return self.queue

    def push(self, x):
        self.queue.append(x)

    def pop(self):
        return self.queue.pop(0)

    def is_empty(self):
        return len(self.queue) == 0

    def length(self):
        return len(self.queue)

def ip_model(costs_dict, probs_dict, text_len):
    model = Model("Word Cost IP")
    sources = list(range(len(probs_dict)))
    probs = [f/text_len for _, f in probs_dict.items()]
    costs = costs_dict
    x = {}

    def f(r):
```

```
        return sum(r ** (-c) for c in costs) - 1
    r = newton(f, 1.5)

    d_min = min(costs)
    d_max = max(math.ceil(-math.log(p, r)) for p in probs) + 5
    D_set = list(range(d_min, d_max))

    def add_x_vars():
        for s in sources:
            for d in D_set:
                x[(s, d)] = model.addVar(vtype="B", name=f"x_{s}_{d}")

    def add_objective():
        model.setObjective(quicksum(probs[s] * d * x[(s, d)] for s in sources for d in
    ↪  D_set), "minimize")

    def add_b_vars_cons():
        b = {}
        b[0] = model.addVar(vtype="I", lb=1, ub=1, name=f"b_{0}")
        for d in D_set:
            b[d] = model.addVar(vtype="I", name=f"b_{d}")
            lhs = quicksum(x[(s, d)] for s in sources) + b[d]
            rhs = 0
            for c in costs:
                if d - c >= 0:
                    rhs += b[d - c]
                else:
                    break
            model.addCons(lhs <= rhs)

    def add_cons():
        for s in sources:
            model.addCons(quicksum(x[(s, d)] for d in D_set) == 1)

    add_x_vars()
    add_objective()
    add_b_vars_cons()
    add_cons()

    return model, x, d_max

def solve(model, x):
    model.optimize()
    lengths = []
    for key, _ in x.items():
        if model.getVal(x[key]) == 1:
            lengths.append(key[-1])
    return lengths

def implicit_bfs(d_max, lengths, costs):
    n = len(costs)
    language_dict = {}
    q = Queue()
    q.push(Node("", 0, children=None))
    while not q.is_empty():
        node = q.pop()
        if node.cost in lengths:
            lengths.remove(node.cost)
```

```
            language_dict[node.code] = node.cost
            if len(lengths) == 0:
                break
        elif node.cost > d_max:
            pass
        else:
            for i in range(n):
                child = Node(node.code + str(i), node.cost + costs[i])
                q.push(child)
    return language_dict

def table(alphabet, freq, language, language_dict):
    header = ["Zeichen", "Häufigkeit", "Kosten", "Wort"]
    rows = []
    for i in range(len(alphabet)):
        row = []
        row.append(f"'{alphabet[i]}'")
        row.append(freq[alphabet[i]])
        row.append(language_dict[language[i]])
        row.append(language[i])
        rows.append(row)
    table = tabulate(rows, headers=header, tablefmt="github")
    length = 0
    for x in range(len(alphabet)):
        length += freq[alphabet[x]] * language_dict[language[x]]
    return table, length
```

Listing 2: Python Code für Huffman Coding IP-Modell

4.3 Greedy-Heuristik

```
import heapq

class Node:
    __slots__ = ("weight", "symbol", "children")
    def __init__(self, weight, symbol=None):
        self.weight = weight
        self.symbol = symbol
        self.children = []
    def __lt__(self, other):
        return self.weight < other.weight

def get_data(filename):
    with open(filename, "r") as f:
        n = int(f.readline())
        weight_function = list(map(int, f.readline().split()))

        frequency_dict = {}
        text = f.read()
        for char in text:
            if char != "\n":
                frequency_dict[char] = frequency_dict.get(char, 0) + 1
    return n, weight_function, frequency_dict, len(text.strip("\n"))

def greedy_heuristic(weights, letter_costs):
    n = len(letter_costs)
    # Priority QUeue mit allen Anfangsknoten
```

```
    pq = [Node(w, symbol=i) for i, w in enumerate(weights)]
    heapq.heapify(pq)

    # Dummy-Knoten auffüllen
    if n > 1:
        pad = (n-1 - (len(pq)-1) % (n-1)) % (n-1)
        for _ in range(pad):
            heapq.heappush(pq, Node(0))

    # Zweig-Slots nach Kosten sortieren
    branch_positions = list(range(n))
    branch_positions.sort(key=lambda i: letter_costs[i])

    # Baumaufbau
    while len(pq) > 1:
        # n leichteste Knoten entnehmen
        children = [heapq.heappop(pq) for _ in range(n)]
        # absteigend sortieren
        children.sort(key=lambda node: node.weight, reverse=True)

        # Kinder in Slots verteilen
        arranged = [None] * n
        for child, slot in zip(children, branch_positions):
            arranged[slot] = child

        # neuen Elternknoten erzeugen
        parent = Node(sum(ch.weight for ch in children))
        parent.children = arranged
        heapq.heappush(pq, parent)

    # Codezuweisung per Traversierung
    root = pq[0]
    codes = {}
    def _assign(node, prefix):
        if node.symbol is not None:
            codes[node.symbol] = prefix or "0"
        else:
            for digit, child in enumerate(node.children):
                _assign(child, prefix + str(digit))
    _assign(root, "")
    return codes
```

Listing 3: Greedy Heuristik für n-äre Huffman–Bäume

Literatur

[1] Mordecai J. Golin, Claire Kenyon und Neal E. Young. „Huffman Coding with Unequal Letter Costs“. In: *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*. 2002, S. 785–791. DOI: 10.1145/509907.510017. URL: https://citeseerx.ist.psu.edu/document?doi=4851061087e66651f971fdf9acbbbbc2c75aaeae.

[2] Mordecai J. Golin und Neal E. Young. „Huffman coding with unequal letter costs“. In: *Proceedings of the 20th Annual European Symposium on Algorithms (ESA)*. Bd. 7501. Lecture Notes in Computer Science. Springer, 2012, S. 469–481. DOI: 10.1007/978-3-642-33090-2_41. URL: https://www.cs.ucr.edu/~neal/publication/Golin12Huffman.pdf.

[3] David A. Huffman. „A Method for the Construction of Minimum-Redundancy Codes“. In: *Proceedings of the IRE* 40.9 (1952), S. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[4] R. Karp. „Minimum-redundancy coding for the discrete noiseless channel“. In: *IRE Transactions on Information Theory* 7.1 (1961), S. 27–38. DOI: 10.1109/TIT.1961.1057615.

[5] B. McMillan. „Two inequalities implied by unique decipherability“. In: *IRE Transactions on Information Theory* 2.4 (1956), S. 115–116. DOI: 10.1109/TIT.1956.1056818.