Aufgabe 1: Schmucknachrichten

Teilnahme-ID: 74130

Bearbeiter/-in dieser Aufgabe:
Matthew Greiner

April 22, 2025

Contents

1 Teilaufgabe A

1.1 Lösungsidee

Verständnis der Aufgabe

Die Grundidee dieser Aufgabe ist es einen gegebenen Unicode-Text mit Hilfe von Perlen zu kodieren, die alle denselben Durchmesser haben. Die Anzahl der Farben der Perlen ist gegeben und beträgt mindestens zwei. Das Ziel ist es, den gegebenen Text so mit den Perlen zu kodieren, dass eine möglichst kurze Perlenkette daraus gemacht werden kann. Die verwendete Codetabelle für die präfixfreie Kodierung

und die Gesamtlänge der Botschaft muss am Ende ausgegeben werden.

Überlegungen zum Lösungsansatz

Da der Durchmesser aller Perlen gleich ist, muss die Anzahl der verwendeten Perlen in der Kodierung minimiert werden, um die Perlenkette möglichst kurz zu halten. Da jedes Zeichen aus dem Eingabetext einzeln durch einen Code aus Perlen dargestellt werden muss, macht es Sinn, dass häufig verwendete Zeichen kurze Codes und selten verwendete Zeichen längere Codes bekommen. Diese Anforderungen eine **optimale präfixfreie Kodierung basierend auf Zeichenhäufigkeiten zur Minimierung der Gesamtlänge** ist eine klassische Anwendung der Huffman-Kodierung, die auch als Basis für die Lösung dieser Teilaufgabe verwendet wird.

Die Huffman-Kodierung ist ein Algorithmus, der unter folgenden Bedingungen eine optimale binäre Kodierung liefert:

1. Jedes Zeichen bekommt einen eindeutigen Binärcode
2. Die Nachricht ist verlustfrei und eindeutig dekodierbar
3. Die durchschnittliche Länge (gewichtet nach Häufigkeit) wird minimiert

Da diese Bedingungen genau zu dieser Aufgabe passen, eignet sich diese Kodierung für diese Aufgabe gut. Standardmäßig funktioniert die Huffman-Kodierung aber binär. Da wir die gegebene Nachricht evtl. aber mit mehr als zwei Farben von Perlen kodieren müssen, muss der Algorithmus auf eine **n-ären** Huffman-Kodierung erweitert werden. Eine solche Verallgemeinerung des Algorithmus wurde bereits im Originalpaper von Huffman [1] beschrieben.
Diese Erweiterung des Algorithmus funktioniert analog zu der binären Huffman-Kodierung, nur die Erstellung des Huffman-Baums muss angepasst werden. Bei der normalen Huffman-Kodierung werden zunächst die zu kodierenden Zeichen in Knoten gespeichert. Dann werden die **zwei** Knoten mit den geringsten Häufigkeiten kombiniert und als Kinder von einem neuen Knoten gespeichert. Die "Häufigkeit" des neuen Knotens wird als Summe der beiden Häufigkeiten der Kinder festgelegt. Dieser Prozess wird wiederholt, bis nur noch ein Knoten übrig ist. Dieser ist dann die Wurzel des neuen Baums. Das bedeutet, bei jeder Kombinierung, verringert sich die Anzahl der betrachteten Knoten um eins. Der Baum wird also "bottom-up" erstellt, und garantiert somit eine optimale Kodierung (im Gegensatz zur Shannon-Fano-Kodierung, die einen Baum "top-down" erstellt und nicht immer die optimale Kodierung findet).

Der Unterschied zu der **n-nären** Huffman-Kodierung ist nur, dass die n Knoten mit den geringsten Häufigkeiten kombiniert werden und nicht nur zwei. Das hat zur Folge, dass mit jeder Kombination die Anzahl der betrachteten Knoten nun um $n-1$ sinkt. Daher muss gezielt darauf geachtet werden, dass nach allen Kombinationen nur noch ein Knoten übrigbleibt. Der Aufbau des n-nären Baums funktioniert, wenn folgende Gleichung erfüllt ist:

$$N \bmod (n-1) = 1 \quad (N = \text{Anzahl der Knoten}) \tag{1}$$

Wenn das nicht der Fall ist, können Platzhalterknoten hinzugefügt werden, die die Häufigkeit 0 haben, bis diese Bedingung erfüllt ist. Bei der Erstellung der Codetabelle wird der Graph (analog zur Standard Huffman-Kodierung) z.B. mit Tiefensuche traversiert. Dabei wird jeder Kante von einem Elternknoten zu einem Kindknoten eindeutig eine Zahl (z.B. von 0 bis k-1) zugeordnet. Eine Zahl repräsentiert in dieser Aufgabe eine Perlenfarbe. In der binären Version werden nur die Zahlen 0 und 1 benutzt.

1.2 Umsetzung

Um diesen Algorithmus umzusetzen und somit die Aufgabe zu lösen, wird wie folgt vorgegangen (die Nummerierung entspricht auch der Nummerierung in den Quellcode):

1. **Einlesen des Eingabetextes und der Anzahl der verschiedenen Perlenfarben**
   Da alle Eingaben in Dateien vorliegen, welche nur 3 Zeilen haben, werden sie zunächst eingelesen und an den Zeilenumbrüchen getrennt. Der Text sowie die Anzahl der verschiedenen Perlenfarben werden anschließend in entsprechenden Variablen gespeichert.

2. **Analyse der Häufigkeit der Zeichen im Text**
   Um zu zählen, wie häufig die Zeichen in dem Text vorkommen, wird über alle Zeichen des Texts iteriert. Die Werte werden in einer HashMap gespeichert, wobei das Zeichen den Schlüssel und die Anzahl seines Vorkommens im Text den zugehörigen Wert darstellt.

3. **Erstellung einen n-nären Huffman-Baums basierend auf den Häufigkeiten**
   Um den Baum zu erstellen, wird eine Klasse Node definiert, die einen Knoten im Huffman-Baum repräsentiert. Diese Klasse speichert das Zeichen, seine Häufigkeit, eine Liste seiner Kindknoten und einen Marker, ob der Knoten ein Blatt im Baum ist. Die einzelnen Knoten werden in einer PriorityQueue verwaltet, welche in Java als Min-Heap implementiert ist. Dadurch stehen die Knoten mit der geringsten Häufigkeit immer oben in der Warteschlange und diese Datenstruktur eignet sich daher gut, da für die Erstellung des Baums jeweils die n Knoten mit den **geringsten** Häufigkeiten kombiniert werden.
   Dies ist was den Huffman Algorithmus zu einem *greedy* Algorithmus macht. Er trifft in jedem Schritt eine lokal optimale Entscheidung (kombiniere die n unwahrscheinlichsten Knoten), die nachweislich zur globalen optimalen Lösung führt.

   Ablauf
   Zunächst werden die Zeichen des Text in Blattknoten gespeichert und in die PriorityQueue eingefügt. Um aus diesen Knoten den Huffman-Baum zu konstruieren, kann man Platzhalter-Knoten einführen (siehe 1.1), allerdings gibt es auch einen Weg diese zusätzlichen Knoten zu vermeiden:
   Statt Platzhalter mit der Häufigkeit 0 zu erzeugen, werden beim ersten Merging nicht zwingend n Knoten kombiniert, sondern eine bestimmte Anzahl r. Dieses r muss so gewählt werden, dass nach der ersten Kombination eine Gesamtanzahl an Knoten entsteht, mit der sich der Rest des Baums in gleichmäßigen n-ner Gruppen weiterbauen lässt.

   Es gilt die Gleichung 1.
   Wenn eine n-närer Baum gefordert ist, ist **N** die Anzahl an Blattknoten in der Queue. Die Gleichung muss auch nach dem ersten Merge mit r Knoten gelten:

$$(N - r + 1) \bmod (n - 1) = 1$$

   Nach r umstellen ergibt:

$$\begin{aligned} N - r &\equiv 0 \mod (n-1) \\ \Rightarrow \quad r &= N \bmod (n-1) \end{aligned}$$

   r muss mindestens 2 sein, damit ein Merge Sinn macht, daher wird:

$$r = (N - 2) \bmod (n - 1) + 2 \tag{2}$$

   in der Implementierung verwendet.
   In der Implementierung wird r also einmalig für den ersten Merge verwendet. Danach werden Gruppen zu je n Knoten kombiniert, bis nur noch ein Knoten übrig ist, das dann die Wurzel des Huffman-Baums ist.

4. **Erzeugen der Codetabelle anhand des Baums**
   Zur Speicherung der Codetabelle wird eine Hashmap verwendet, in der die Zeichen als Schlüssel hat und ihre zugehörige n-äre Huffman-Kodierung als Werte abgelegt sind. Die Werte werden dann schrittweise bei der Traversierung des Baums schrittweise aufgebaut.
   Um die Kodierung für jedes Zeichen zu finden, wird der Baum mit einer rekursiven Tiefensuche (DFS) durchlaufen. Dabei wird bei jedem rekursiven Aufruf der bisher zurückgelegte Pfad vom Wurzelknoten bis zum Aktuellen Knoten als String übergeben.

   Weil es sich um eine n-äre Huffman-Kodierung handelt, werden die Kanten des Baums durch die Ziffern 0 bis $n-1$ beschrieben. Die Ziffern entsprechen allgemein formuliert, den Kodierungsbuchstaben, die in diesem Fall die jeweilige Perlenart bzw. Perlenfarbe darstellt. Bei jedem Abstieg zu einem Kindknoten wird an die aktuelle Zeichenkette die Position des jeweiligen Kindes im Elternknoten angehängt. Wenn ein Blattknoten erreicht wird, also ein Knoten, der tatsächlich ein Zeichen repräsentiert und keine weiteren Nachfolger besitzt, wird der aktuell aufgebaute Pfad als Kodierung für dieses Zeichen in der Codetabelle gespeichert.

5. **Berechnung der Gesamtlänge der kodierten Nachricht**
   Für die Berechnung der Gesamtlänge der kodierten Nachricht wird die Häufigkeit jedes Zeichens (bereits in Schritt 2 bestimmt) mit der Länge seines entsprechenden Huffman-Codes multipliziert. Die Gesamtlänge ergibt sich aus der Summe dieser Produkte multipliziert mit dem Durchmesser einer Perle.

6. **Ausgabe der Codetabelle und der Gesamtlänge**
   Die Tabelle wird nach Code-Länge und Frequenz sortiert (optional, allerdings ein "nice to have" um direkt ablesen zu können, welche Zeichen am meisten vorkommen, wenn dies auch ausgegeben wird, was in dieser Implementierung gemacht wird). Anschließend wird sie formatiert ausgegeben sowie die Gesamtlänge der kodierten Nachricht aus 5.

1.3 Komplexitätsanalyse

Nun wird die Komplexität der einzelnen Schritte des Algorithmus betrachtet. Dabei ist:

- L die Länge des Eingabetextes,
- k die Anzahl der verschiedenen Zeichen (Symbole),
- n die Basis des Huffman-Baums (Anzahl erlaubter Kindknoten je Knoten).

Schritt	**Laufzeit**	**Speicherbedarf**
Einlesen der Datei	$\mathcal{O}(L)$	$\mathcal{O}(L)$
Häufigkeitsanalyse	$\mathcal{O}(L)$	$\mathcal{O}(k)$
Aufbau des Huffman-Baums	$\mathcal{O}(k \log k)$	$\mathcal{O}(k)$
Generieren der Codetabelle	$\mathcal{O}(k)$	$\mathcal{O}(k)$
Berechnung der Gesamtlänge	$\mathcal{O}(k)$	$\mathcal{O}(1)$
Ausgabe Codetabelle (mit Sortierung)	$\mathcal{O}(k \log k)$	$\mathcal{O}(k)$
Gesamt	$\boldsymbol{\mathcal{O}(L + k \log k)}$	$\boldsymbol{\mathcal{O}(L + k)}$
Dominante Terme	*O(L) potentiell langer Text* *O(k log k) wegen Heap Operation/Sortierung*	

Table 1: Laufzeit- und Speicherkomplexität Teilaufgabe A

- **Einlesen und Häufigkeitsanalyse:** Beide Vorgänge durchlaufen den Text vollständig und brauchen daher eine Laufzeit von $\mathcal{O}(L)$.
- **Baumkonstruktion:** Wegen der Prioritätswarteschlange (Min-Heap) mit k Blattknoten entstehen $\frac{k-1}{n-1}$ Merges. Jede Insert- und Delete Operation im Heap braucht $\mathcal{O}(\log k)$, was insgesamt zu $\mathcal{O}(k \log k)$ führt.
- **Codetabelle:** Dieser Schritt ist durchläuft jeden Knoten (k Knoten) genau einmal mit DFS (Tiefensuche). Pro Knoten wird eine Schleife über alle n Kinder ausgeführt. Bei der n-nären Kodierung hat der Baum im schlimmsten Fall $\mathcal{O}(k)$ Knoten. Da n eine kleine Konstante ist, beträgt die Laufzeit für diesen Schritt $\mathcal{O}(k)$.
- **Gesamtlänge:** Dieser Schritt iteriert über alle verschiedenen Symbole, die einen Code bekommen und ist daher in $\mathcal{O}(k)$.
- **Ausgabe:** Die Codetabelle wird nach Frequenz und Code-Länge sortiert, was im Worst Case ebenfalls $\mathcal{O}(k \log k)$ erfordert. (Ohne die optionale Sortierung wäre das das in $\mathcal{O}(k)$)

Insgesamt ist die Laufzeit dominiert durch das Einlesen des Textes und die Sortiervorgänge beim Baumaufbau mit dem Min-Heap. Der Speicherbedarf ist moderat und wächst linear mit der Eingabegröße sowie der Anzahl unterschiedlicher Zeichen.
Daher erweist sich dieser Algorithmus auch bei größeren Texten als leistungsfähig und garantiert eine optimale Kodierung.

1.4 Beispiele

Hier die Ausgaben des Programmes zu den Beispielen auf der BwInf-Webseite:

1. **schmuck0.txt**

```
Codetabelle:
(Jede Ziffer steht für eine Perlenfarbe z.B. könnte '0' rot bedeuten und '1' blau)
{
  '␣': 111 (Freq: 5)
  'E': 110 (Freq: 5)
  'I': 100 (Freq: 4)
  'N': 101 (Freq: 4)
  'S': 000 (Freq: 3)
  'R': 0100 (Freq: 2)
  'D': 0010 (Freq: 2)
  'L': 0111 (Freq: 2)
  'M': 0110 (Freq: 2)
  'O': 0011 (Freq: 2)
  'C': 01010 (Freq: 1)
  'H': 01011 (Freq: 1)
}

Gesamtlänge der Botschaft 113 (Anzahl in Perlen) bzw. 11.3cm
```

2. **schmuck00.txt**

```
Codetabelle:
(Jede Ziffer steht für eine Perlenfarbe z.B. könnte '0' rot bedeuten und '1' blau)
{
  '␣': 21 (Freq: 24)
  'e': 12 (Freq: 18)
  'i': 11 (Freq: 16)
  't': 02 (Freq: 10)
  'n': 01 (Freq: 9)
  'h': 220 (Freq: 8)
  's': 222 (Freq: 8)
  'c': 202 (Freq: 7)
  'l': 201 (Freq: 6)
  'a': 100 (Freq: 4)
  'r': 102 (Freq: 4)
  'd': 001 (Freq: 3)
  'D': 2212 (Freq: 3)
  'u': 2211 (Freq: 3)
  'G': 1012 (Freq: 2)
  'g': 2210 (Freq: 2)
  'm': 2000 (Freq: 2)
  'w': 2002 (Freq: 2)
  'E': 0001 (Freq: 1)
  'P': 0020 (Freq: 1)
  'W': 0002 (Freq: 1)
  'Z': 1010 (Freq: 1)
  'b': 0022 (Freq: 1)
  'f': 1011 (Freq: 1)
  'k': 0021 (Freq: 1)
  'o': 0000 (Freq: 1)
  'A': 20010 (Freq: 1)
  '?': 20011 (Freq: 1)
}
```

```
Gesamtlänge der Botschaft 372 (Anzahl in Perlen) bzw. 37.2cm
```

3. **schmuck01.txt**

```
Codetabelle:
(Jede Ziffer steht für eine Perlenfarbe z.B. könnte '0' rot bedeuten und '1' blau)
{
  '␣': 2 (Freq: 85)
  'e': 0 (Freq: 66)
  'n': 43 (Freq: 53)
  'r': 41 (Freq: 34)
  'i': 40 (Freq: 33)
  's': 33 (Freq: 26)
  'a': 31 (Freq: 22)
  'l': 32 (Freq: 22)
  't': 30 (Freq: 22)
  'h': 13 (Freq: 17)
  'c': 12 (Freq: 15)
  'g': 11 (Freq: 14)
  'o': 10 (Freq: 14)
  'm': 443 (Freq: 13)
  'u': 442 (Freq: 12)
  '.': 441 (Freq: 10)
  'd': 424 (Freq: 9)
  'k': 440 (Freq: 9)
  'E': 423 (Freq: 8)
  ',': 422 (Freq: 8)
  'D': 344 (Freq: 7)
  'b': 421 (Freq: 7)
  'w': 343 (Freq: 6)
  'ü': 341 (Freq: 5)
  'I': 340 (Freq: 4)
  'S': 144 (Freq: 4)
  'f': 143 (Freq: 4)
  'p': 142 (Freq: 4)
  'v': 141 (Freq: 4)
  'V': 140 (Freq: 3)
  'F': 4442 (Freq: 3)
  'O': 4443 (Freq: 3)
  'ö': 4441 (Freq: 3)
  'z': 4444 (Freq: 3)
  'B': 4440 (Freq: 2)
  'G': 4203 (Freq: 2)
  'ä': 4204 (Freq: 2)
  'H': 3420 (Freq: 1)
  'K': 3421 (Freq: 1)
  'M': 3422 (Freq: 1)
  'N': 3424 (Freq: 1)
  'R': 4200 (Freq: 1)
  'W': 4202 (Freq: 1)
  'ß': 3423 (Freq: 1)
  '...': 4201 (Freq: 1)
}

Gesamtlänge der Botschaft 1150 (Anzahl in Perlen) bzw. 115.0cm
```

1.5 Quellcode

Wichtige Teile des Quellcodes:

1. Einlesung der Daten

```java
// 1. Read input
private static InputWrapper parseInput(String input) {
    // Save every line in array of Strings
    String[] lines = input.split("\n");
    List<Integer> list = new ArrayList<>();
    for (String s : lines[1].split("␣")) {
        list.add(Integer.valueOf(s));
    }
    return new InputWrapper(Integer.parseInt(lines[0].trim()), lines[2], list);
}
```

2. Häufigkeitsanalyse

```java
// 2. Calculate character frequency from input text
private static Map<Character, Long> buildFrequencyMap(String text) {
    Map<Character, Long> frequencyMap = new HashMap<>();
    for (char character : text.toCharArray()) {
        frequencyMap.put(character, frequencyMap.getOrDefault(character, 0L) + 1);
    }
    return frequencyMap;
}
```

3. Erstellung einen n-nären Huffman-Baums

```java
// 3. Build n-ary Huffman tree and returns root Node
private static Node buildHuffmanTree(Map<Character, Long> frequencyMap, int n) {
    // PriorityQueue for nodes (sorted by frequency, lower frequency first)
    PriorityQueue<Node> priorityQueue = new PriorityQueue<>();

    // Create leaf nodes (nodes with characters) and add to priority queue
    for (Map.Entry<Character, Long> entry : frequencyMap.entrySet()) {
        priorityQueue.add(new Node(entry.getKey(), entry.getValue()));
    }

    int numSymbols = priorityQueue.size(); // Number of unique symbols (leaf nodes)

    // Determine how many nodes should be merged in the first step
    int r;
    if (numSymbols <= 1) { // Handle edge case: if 0 or 1 symbol, no merging needed
        r = numSymbols;
    } else {
        // Formula: find smallest valid r so that:
        // After merging r nodes -> remaining nodes + 1 new node
        // -> total node count allows full n-ary merges
        int remainder = (numSymbols - 2) % (n - 1);
        r = remainder + 2; // Ensures 2 <= r <= n
    }

    // Build Huffman tree
    while (priorityQueue.size() > 1) {
        // Use r only for first merge if needed
        int nodesToMerge = (priorityQueue.size() == numSymbols) ? r : n;
        // Cannot merge more nodes than available
        nodesToMerge = Math.min(nodesToMerge, priorityQueue.size());

        if (nodesToMerge < 2) {
            // Should not happen if n > 1
            System.out.println("irgendwas␣ist␣broken");
            break;
        }

        List<Node> children = new ArrayList<>();
        long mergedFrequency = 0;
```

```
40
        // Extract nodes (num of nodesToMerge) with lowest frequencies
42      for (int i = 0; i < nodesToMerge; i++) {
            Node node = priorityQueue.poll();
44          children.add(node);
            mergedFrequency += node.frequency;
46      }

48      // Create new internal node with new values
        Node internalNode = new Node(children, mergedFrequency);
50
        // Add new internal node back to priority queue
52      priorityQueue.add(internalNode);
    }
54
    // Last node in queue is root of Huffman tree
56  return priorityQueue.poll();
}
```

4. Erzeugen der Codetabelle

```
1  // 4. Generate Huffman codes by traversing tree with recursion and DFS
   private static void generateCodes(Node node, String currentCode, Map<Character, String>
       codeTable) {
3      // Base case
       if (node == null) {
5          return;
       }
7
       // If it is a leaf, it represents a character, then assign a code
9      if (node.isLeaf()) {
           if (node.character != null) {
11             codeTable.put(node.character, currentCode);
           }
13         // Leaf is reached, stop recursion
           return;
15     }

17     // Assign codes 0 until k-1 to children
       for (int i = 0; i < node.children.size(); i++) {
19         generateCodes(node.children.get(i), currentCode + i, codeTable);
       }
21 }
```

5. Berechnung der Gesamtlänge

```
1  // 5. Calculate total length of encoded message
   private static long calculateTotalLength(Map<Character, String> codeTable,
       Map<Character, Long> frequencyMap,
3          int diameter) {
       long totalLength = 0;
5      // Iterate over each Character
       for (Map.Entry<Character, Long> entry : frequencyMap.entrySet()) {
7          totalLength += entry.getValue() * codeTable.get(entry.getKey()).length();
       }
9      return totalLength * diameter;
   }
```

6. Ausgabe

```
   // Print code table in a readable format
2  private static void printCodeTable(Map<Character, String> codeTable, Map<Character,
       Long> frequencyMap) {
       if (codeTable.isEmpty()) {
4          System.out.println("{}");
           return;
6      }
       List<Character> sortedKeys = new ArrayList<>(codeTable.keySet());
```

```
8       // Sorting is optional
        // Sort by length of code
10      sortedKeys.sort((a, b) -> Integer.compare(codeTable.get(a).length(),
        codeTable.get(b).length()));
        // Sort by frequency
12      sortedKeys.sort((a, b) -> Long.compare(frequencyMap.get(b), frequencyMap.get(a)));

14      System.out.println("{");
        for (Character character : sortedKeys) {
16          // Handle special characters for printing
            String c;
18          c = switch (character) {
                case '␣' -> "␣";
20              case '"' -> c = "\\\"";
                default -> String.valueOf(character);
22          };
            System.out.println("␣␣'" + c + "':␣" + codeTable.get(character) + "␣(Freq:␣"
24                  + frequencyMap.get(character) + ")");
        }
26      System.out.println("}");
    }
```

2 Teilaufgabe B

2.1 Lösungsidee

Verständnis der Aufgabe

Die Grundidee dieser Aufgabe bleibt gleich: Ein Unicode-Text soll mit Hilfe von Perlen kodiert werden. Allerdings haben die Perlen nun einen unterschiedlichen Durchmesser. Ziel ist es nun, die Gesamtlänge der resultierenden Perlenkette, also die kodierte Nachricht, zu minimieren.
Wenn dieses Problem weiterhin als Optimierungsproblem zu Erstellung einer optimalen, präfixfreien Kodierung (wie bereits im ersten Teil der Aufgabe) betrachtet wird, führt die Einführung variabler Perlendurchmesser zu einer Verallgemeinerung des ursprünglichen Problems. Denn in dieser Erweiterung entspricht der Durchmesser jeder Perle den Kosten des jeweiligen Symbols im Codealphabet. Im Folgenden wird einfachheitshalber, die Perle als Symbol bezeichnet, da diese verwendet werden, um den Text zu kodieren und der Durchmesser der Perle wird als Kosten des Symbols bezeichnet.

Diese Variante ist bekannt als das Problem der **präfixfreien Kodierung mit ungleichen Symbolkosten** (prefix-free coding with unequal letter cost). Im Gegensatz zur klassischen Huffman-Kodierung, die von gleichen Symbolkosten ausgeht, ist diese Verallgemeinerung deutlich komplexer. Es ist bis heute **nicht bekannt**, ob dieses Problem in **P** liegt oder **NP-schwer** ist.

Überlegungen zum Lösungsansatz

Die aus Kapitel 1 implementierte n-näre Huffman-Kodierung scheitert bei dieser Problemstellung, da die Minimierung der Anzahl der Kodierungsbuchstaben nicht notwendigerweise zur Minimierung der Gesamtkosten führt (die Kosten werden nämlich nicht berücksichtigt).
Im Folgenenden werden drei Strategien zur Lösung dieses Problems untersucht:

2.2 Strategie 1: Heuristische n-näre Huffman Modifikation mit greedy Kostenzuweisung

2.2.1 Kernidee

Dieser Ansatz ist vermutlich der Intuitivste und am wenigsten aufwendige und ist in zwei Schritte aufgeteilt:

1. **Erstellung eines n-nären Huffman Baums**
   Zunächst wird ein n-närer Huffman mit dem aus Kapitel 1 beschriebenen unveränderten Algorithmus erstellt, bei der der Baum nur basierend auf den Symbolhäufigkeiten gebaut wird. Dieser Schritt ignoriert also die Symbolkosten und bestimmt nur die Topologie des Baums.

2. **Greedy Zuweisung der Kodierungsbuchstaben anhand der Kosten**
   Bei der Traversierung des Baums zur Erstellung der Codewörter erfolgt die Beschriftung der Kanten (von inneren Knoten mit n Kindern) standardmäßig mit den Ziffern bzw. Kodierungszeichen 0 bis $n-1$.
   Während bei der klassischen Methode aus Kapitel 1 diese Zuweisung oft der Reihenfolge der Kinder folgt (z.B. von links nach rechts), ändert sich dies in dieser Heuristik:
   Denn nun erfolgt diese Zuweisung greedy nach folgendem Prinzip:
   Die Kante, die zum Teilbaum mit der höchsten Häufigkeit führt, erhält jetzt den Kodierungsbuchstaben mit den geringsten Kosten. Die zweithäufigste Kante erhält das zweigünstigste Symbol usw. Dadurch wird angestrebt, dass teure Kodierungszeichen, also Perlen mit großem Durchmesser möglichst selben verwendet werden (in Pfaden mit seltenen Symbolen). So werden günstige Zeichen möglichst früh im Baum platziert, sodass sie häufiger vorkommen und damit mehr zur Gesamtkodierung beitragen können.

Diese lokale Regel zur Zuordnung führt nicht zu einer global optimalen Lösung, aber sie verteilt die teuren Kosten möglichst kosteneffizient in dem bestehenden Baum. Diese Heuristik ist leicht implementierbar, da diese nur eine kleine Anpassung der n-nären Huffman-Kodierung benötigt. Aus diesen Gründen wurde diese Strategie als erster Ansatz für die Lösung dieses Teilproblems gewählt. Sie dient als Baseline, an der weiterführende Verfahren verglichen werden können, die diese Strategie idealerweise übertreffen.

2.2.2 Umsetzung

Die Umsetzung dieser Strategie erfordert nur eine Anpassung der Teilschritte 4 und 5 des beschriebenen Algorithmus aus Kapitel 1. Die Schritte 1 bis 3 bleiben identisch (bis auf die Einlesung der Daten in Schritt 1 die minimal angepasst wird). Folgendes wird angepasst:

4. **Traversierung mit greedy Kostenzuweisung**
   Die eigentliche Traversierung des Baums mit der rekursiven Tiefensuche bleibt identisch. Für die Zuweisung der Kosten, werden zunächst die Kinder nach ihrer Häufigkeit absteigend sortiert. Die verfügbaren Kodierungssymbole (in diesem Fall die Ziffern 0 bis $n-1$), werden auch gemäß ihren zugehörigen Kosten aufsteigend sortiert. Mit einer Schleife über alle Kinder, wird den Kanten zu den Teilbäumen nun jeweils das günstigste noch verfügbare Symbol zugewiesen, beginnend mit dem häufigsten Teilbaum. Für jedes Kind wird dieser Prozess rekursiv wiederholt, wobei der aktuelle Pfad (die bisher zugewiesenen Kodierungssymbole) jeweils an das entstehende Codewort angehängt wird.

5. **Berechnung der Gesamtlänge der kodierten Nachricht**
   In diesem Schritt wird die Gesamtlänge der kodierten Nachricht bestimmt. Im Unterschied zur klassischen Huffman-Kodierung, bei der die Länge eines Codeworts einfach durch die Anzahl der Symbole bestimmt wird, müssen hier die individuellen Kosten der verwendeten Kodierungssymbole mit einberechnet werden. Die Gesamtlänge ergibt sich also nicht nur aus der Anzahl der Symbole im Codewort, sondern aus der Summe der Kosten aller Symbole, die zur Kodierung eines Zeichens verwendet werden.

2.2.3 Komplexitätsanalyse

Die Komplexität des Algorithmus has sich wie folgt, geändert:

Schritt	**Laufzeit**	**Speicherbedarf**
Einlesen der Datei	$\mathcal{O}(L)$	$\mathcal{O}(L)$
Häufigkeitsanalyse	$\mathcal{O}(L)$	$\mathcal{O}(k)$
Aufbau des Huffman-Baums	$\mathcal{O}(k \log k)$	$\mathcal{O}(k)$
Generieren der Codetabelle	$\mathcal{O}(k \cdot n \log n)$	$\mathcal{O}(k \cdot n)$
Berechnung der Gesamtlänge	$\mathcal{O}(k)$	$\mathcal{O}(1)$
Ausgabe Codetabelle (mit Sortierung)	$\mathcal{O}(k \log k)$	$\mathcal{O}(k)$
Gesamt	$\boldsymbol{\mathcal{O}(L + k \log k)}$	$\boldsymbol{\mathcal{O}(L + k)}$

Table 2: Laufzeit- und Speicherkomplexität Teilaufgabe A

- **Codetabelle:** Dieser Schritt ist durchläuft jeden Knoten (insgesamnt k Knoten) genau einmal mit DFS (Tiefensuche). Pro Knoten werden die Kinder und Kodierungszeichen sortiert, was in $\mathcal{O}(n \log n)$ ist. Daher beträgt die Laufzeit für diesen Schritt $\mathcal{O}(k \cdot n \log n)$.
- **Gesamtlänge:** Dieser Schritt iteriert über alle verschiedenen Symbole, die einen Code bekommen. Es wird über jedes Zeichen in Code iteriert und somit ist dieser Schritt in $\mathcal{O}(k \cdot L_{max})$. L_{max} ist hier die maximale Code-Länge (konstant), daher $\mathcal{O}(k)$.

2.2.4 Beispiele

Hier die Ausgaben des Programmes zu den Beispielen auf der BwInf-Webseite. Mit Ausnahme der Datei *schmuck5.txt* (für die ein offizieller Vergleichswert vorliegt), wurde die Konsolenausgabe auf die Gesamtlänge der kodierten Nachricht reduziert. Dies wurde gemacht für eine klarere Vergleichbarkeit der Ergebnisse zwischen den noch folgenden Algorithmen.
Die gesamte Ausgabe ist hier **AusgabenBStrategie1.txt**

EDIT

zufinden. Die Ausgabe wird zu dem finalen (und besten) Programm ausführlicher gegeben.

1. **schmuck1.txt**

```
  ...
Gesamtlänge der Botschaft 197 (Anzahl in Perlen) bzw. 19.7cm
```

2. **schmuck2.txt**

```
  ...
Gesamtlänge der Botschaft 145 (Anzahl in Perlen) bzw. 14.5cm
```

3. **schmuck3.txt**

```
  ...
Gesamtlänge der Botschaft 279 (Anzahl in Perlen) bzw. 27.9cm
```

4. **schmuck4.txt**

```
  ...
Gesamtlänge der Botschaft 154 (Anzahl in Perlen) bzw. 15.4cm
```

5. **schmuck5.txt**

```
Codetabelle:
(Jede Ziffer steht für eine Perlenfarbe z.B. könnte '0' rot bedeuten und '1' blau)
{
  '␣': 2 (Freq: 151, Länge: 2)
  'e': 3 (Freq: 110, Länge: 3)
```

```
  't': 4 (Freq: 71, Länge: 4)
  'i': 5 (Freq: 67, Länge: 5)
  'o': 00 (Freq: 64, Länge: 2)
  's': 01 (Freq: 61, Länge: 2)
  'a': 02 (Freq: 58, Länge: 3)
  'n': 03 (Freq: 56, Länge: 4)
  'r': 04 (Freq: 51, Länge: 5)
  'c': 05 (Freq: 44, Länge: 6)
  'd': 06 (Freq: 37, Länge: 7)
  'l': 10 (Freq: 33, Länge: 2)
  'm': 11 (Freq: 27, Länge: 2)
  'h': 12 (Freq: 26, Länge: 3)
  'u': 14 (Freq: 25, Länge: 5)
  'f': 16 (Freq: 20, Länge: 7)
  'p': 15 (Freq: 20, Länge: 6)
  'b': 60 (Freq: 16, Länge: 7)
  'y': 61 (Freq: 13, Länge: 7)
  'g': 62 (Freq: 11, Länge: 8)
  '.': 63 (Freq: 8, Länge: 9)
  'q': 66 (Freq: 5, Länge: 12)
  ',': 130 (Freq: 5, Länge: 5)
  'v': 131 (Freq: 5, Länge: 5)
  'x': 132 (Freq: 5, Länge: 6)
  'w': 133 (Freq: 4, Länge: 7)
  'F': 134 (Freq: 3, Länge: 8)
  'T': 135 (Freq: 3, Länge: 9)
  'A': 650 (Freq: 1, Länge: 12)
  'G': 644 (Freq: 1, Länge: 14)
  'H': 651 (Freq: 1, Länge: 12)
  'S': 640 (Freq: 1, Länge: 11)
   ''': 645 (Freq: 1, Länge: 15)
  'j': 652 (Freq: 1, Länge: 13)
  '-': 643 (Freq: 1, Länge: 13)
  '1': 646 (Freq: 1, Länge: 16)
  '2': 136 (Freq: 1, Länge: 10)
  '5': 653 (Freq: 1, Länge: 14)
  '9': 654 (Freq: 1, Länge: 15)
  'z': 641 (Freq: 1, Länge: 11)
  ';': 642 (Freq: 1, Länge: 12)
}

Gesamtlänge der Botschaft 4010 (Anzahl in Perlen) bzw. 401.0cm
```

Dieses Ergebnis (4010) ist ca. 26.82% schlechter als der Vergleichswert (3162) von der BwInf Webseite, daher ist hier noch viel Platz für Verbesserung.

6. **schmuck6.txt**

```
  ...
Gesamtlänge der Botschaft 244 (Anzahl in Perlen) bzw. 24.4cm
```

7. **schmuck7.txt**

```
  ...
Gesamtlänge der Botschaft 153144 (Anzahl in Perlen) bzw. 15314.4cm
```

8. **schmuck8.txt**

```
  ...
Gesamtlänge der Botschaft 3615 (Anzahl in Perlen) bzw. 361.5cm
```

9. **schmuck9.txt**

```
...
Gesamtlänge der Botschaft 41056 (Anzahl in Perlen) bzw. 4105.6cm
```

Dieses Ergebnis (41056) ist ca. 12.18% schlechter als der Vergleichswert (36597) von der BwInf Webseite, daher ist hier noch etwas Platz für Verbesserung.

2.2.5 Quellcode

Wichtige Teile des Quellcodes:

4. Erzeugen der Codetabelle mit greedy Heuristik

```java
// 4. Generate Huffman codes with cost-aware digit assignment
private static void generateCodesCostAware(Node node, String currentCode, Map<Character,
    String> codeTable,
        List<Integer> costs) {
    // Base case
    if (node == null) {
        return;
    }

    // If it is a leaf, it represents a character, then assign a code
    if (node.isLeaf()) {
        if (node.character != null) {
            codeTable.put(node.character, currentCode.isEmpty() ? "0" : currentCode);
        }
        // Leaf is reached, stop recursion
        return;
    }

    List<Node> children = node.children;

    // 1. Sort children by frequency - Descending (made copy to not modify the main
    // list, not sure if needed)
    List<Node> sortedChildren = new ArrayList<>(children);
    sortedChildren.sort(Comparator.comparingLong(Node::getFrequency).reversed());

    // 2. Prepare digits sorted by cost - Ascending
    // DigitCost is a wrapper to map each digit with its corresponding cost
    List<DigitCost> digitCosts = new ArrayList<>();

    for (int i = 0; i < costs.size(); i++) {
        digitCosts.add(new DigitCost(i, costs.get(i)));
    }
    // Sort by cost ascending, for this input not needed as digits are already sorted
    Collections.sort(digitCosts);

    // 3. Assign cheapest digits to most frequent children
    for (int i = 0; i < sortedChildren.size(); i++) {
        Node child = sortedChildren.get(i);
        int assignedDigit = digitCosts.get(i).digit; // i-th cheapest digit
        generateCodesCostAware(child, currentCode + assignedDigit, codeTable, costs);
    }
}
```

5. Berechnung der Gesamtlänge

```java
// 5. Calculate total cost of encoded message
private static long calculateTotalCost(Map<Character, String> codeTable, Map<Character,
    Long> frequencyMap,
        List<Integer> costs) {
    long totalCost = 0;

    // Iterate over each character
    for (Map.Entry<Character, Long> entry : frequencyMap.entrySet()) {
        char character = entry.getKey();
        long frequency = entry.getValue();
```

```
        String code = codeTable.get(character);

        long costOfCode = 0;
        for (char digitChar : code.toCharArray()) {
            int digit = Character.getNumericValue(digitChar);
            costOfCode += costs.get(digit);
        }
        totalCost += frequency * costOfCode;
    }
    return totalCost;
}
```

2.3 Strategie 2

2.3.1 Umsetzung

2.3.2 Komplexitätsanalyse

2.3.3 Beispiele

2.3.4 Quellcode

2.4 Strategie 3

2.4.1 Umsetzung

2.4.2 Komplexitätsanalyse

2.4.3 Beispiele

2.4.4 Quellcode

2.5 Erweiterungen

2.6 Vergleich der Strategien

References

[1] David A. Huffman (1952) *A Method for the Construction of Minimum-Redundancy Codes*, *Proceedings of the IRE*, 40(9), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[2] Mordecai Golin and Li Jian (2007) *More Efficient Algorithms and Analyses for Unequal Letter Cost Prefix-Free Coding*, arXiv:0705.0253 [cs.IT]. https://doi.org/10.48550/arXiv.0705.0253

[3] M. J. Golin and G. Rote (1998) *A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs*, *IEEE Transactions on Information Theory*, 44(5), 1770–1781. https://doi.org/10.1109/18.705558