Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Arbeitsgruppe Codes und Kryptographie
Fürstenallee 11
33102 Paderborn

Studienarbeit

Implementierung eines Verfahrens zur Berechnung optimaler längenbeschränkter Präfix-Codes

von
Sebastian Gesemann
Matrikelnummer: 6062848

Betreuer: Prof. Dr. Blömer

Paderborn, den 21. Juni 2004

Erklärung:

Ich versichere hiermit, daß ich diese Arbeit ohne Fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe.
Paderborn, den 21. Juni 2004

Sebastian Gesemann
Detmolderstraße 112a
33100 Paderborn

Inhaltsverzeichnis

Kapitel 1

Einleitung

Ein großer Teil der Kodierungstheorie beschäftigt sich mit der kompakten Darstellung von Daten wie zum Beispiel Text, Bild und Ton. Mit geschickten Kompressionstechniken kann man in der Regel solche Datenmengen ohne oder nur mit wenig Verlust reduzieren, um Ressourcen bei Übertragung oder Speicherung der Daten zu sparen. Dabei ist zwischen zwei Varianten der Kompression zu unterscheiden. Es gibt zum einen die verlustfreie Kompression, die es erlaubt, die Originaldaten wieder Bit für Bit korrekt zu rekonstruieren, und zum anderen verlustbehaftete Kompression.

Bei der verlustfreien Datenkompression versucht man, die Daten möglichst kompakt darzustellen, ohne Informationsgehalt zu verlieren. Zu den rein verlustfreien Verfahren zählen zum Beispiel die „General-Purpose“ Kompressionsprogramme ZIP, BZIP, RAR, sowie „Special-Purpose“ Kompressionsverfahren FLAC (Free Lossless Audio Codec), JPEG2000 (im „Lossless“-Modus), die speziell für die Kompression von Audiodaten und Bilddaten ausgelegt sind.

Verlustbehaftete Verfahren erlauben im Vergleich zu verlustfreien Verfahren eine viel höhere Kompressionsrate. Man verlangt nicht mehr, dass sich aus der komprimierten Version die ursprünglichen Daten wieder exakt rekonstruieren lassen. Solche Verfahren eignen sich in der Regel sehr gut für multimediale Inhalte wie Video und Audio. Kleine Fehler können dort in Kauf genommen werden, ohne, dass die subjektiv empfundene Qualität stark beeinträchtigt wird.

Beispiele der verlustbehafteten Datenkompression sind die Formate MPEG[1] Audio Layer 3 (MP3), Ogg Vorbis (OGG), Advanced Audio Coding (AAC), Dolby Digital (AC3) für Audiodaten, MPEG1/2/4 Video, H.264 für Videodaten und JPEG, JPEG2000 für die Kompression von Bildern. Diese Verfahren setzen neben den Techniken der Redundanzreduktion wie prediktive Kodierung und Huffman-Kodierung, die bei verlustfreien Verfahren auch eingesetzt werden, zusätzlich noch Methoden zur Irrelevanzreduktion ein, um die Datenmenge weiter zu reduzieren.

Fast alle erwähnten Anwendungsbeispiele machen Gebrauch von „Codes“, die häufig vorkommenden Symbolen kurze Bitfolgen und weniger häufig vorkommen-

[1] Moving Pictures Experts Group

den Symbolen längere Bitfolgen zuordnen, um die durchschnittliche Länge einer als Bitfolge kodierten Nachricht zu minimieren. Ein sehr bekanntes und einfaches Verfahren, welches solche Codes konstruieren kann, ist das von David A. Huffman ([1]). Dieser Algorithmus kann aber auch unter Umständen Codeworte konstruieren, die länger als gewünscht sind und damit effiziente Implementierungen von Kodierern und Dekodierern verhindern. Der große Vorteil von optimalen längenbeschränkten Codes ist unter anderem die Möglichkeit, durch geeignete Wahl der Längenbeschränkung dafür sorgen zu können, dass jedes Codewort in genau ein Maschinenwort abgelegt werden kann. Dadurch wird eine schnelle und speichersparende Dekodierung möglich.

Muss ein Dekodierer auf relativ langsamen 8Bit-Systemen implementiert werden, wird eine Codewort-Längenbeschränkung umso wichtiger. Die Gefahr, dass der Huffman-Algorithmus Codes generiert, dessen maximale Codewortlänge die 8-Bit-Grenze überschreiten, ist im Gegensatz zu der Längenbeschränkung 32, wie sie für 32Bit-Systeme eingesetzt werden könnte, viel höher. Eine Beschränkung auf Codeworte mit einer maximalen Länge von 8 Bit wird auf solchen Systemen die Benutzung von schnellen, optimierten Dekodierverfahren erlauben.

Seit der Veröffentlichung von Larmore und Hirschberg [4] von 1990 ist ein Algorithmus bekannt, der optimale *längenbeschränkte* Codes sehr schnell finden kann. Dazu wurde ein neues Problem, das Münzsammlerproblem (Coin Collector's Problem), eingeführt. Larmore und Hirschberg geben eine Reduktion des Kodierungsproblems auf das Münzsammlerproblem und einen Lösungsalgorithmus („Package-Merge“) an, der beliebige Instanzen des Münzsammlerproblems lösen kann.

Turpin und Moffat zeigen in ihrer Veröffentlichung [4] von 1994, wie das von Larmore und Hirschberg beschriebene Verfahren für das ursprüngliche Kodierungsproblem weiter optimiert werden kann. Ihr Algorithmus ist um einiges schneller und speichersparender.

Ziel dieser Studienarbeit ist es, die Arbeitsweise des Verfahrens von Larmore und Hirschberg und die wichtigsten der in den zwei Veröffentlichungen vorgestellten Optimierungsideen zu erläutern. Es soll außerdem eine Implementierung des optimierten Lösungsverfahrens erstellt werden.

Dazu werden in Kapitel 2 nochmal grundlegende Definitionen und Sätze aus der Kodierungstheorie vorgestellt, auf die später noch in dieser Arbeit Bezug genommen wird. Das Kapitel 3 beschreibt formal die beiden Probleme OLBPC (das Problem, optimale längenbeschränkte Präfixcodes zu finden) und CCP (das Münzsammlerproblem). Es wird die Reduktion von OLBPC auf CCP angegeben und ihre Korrektheit ähnlich wie in [3] gezeigt. Kapitel 4 beschäftigt sich mit dem Verfahren zur Lösung des Münzsammlerproblems, dem Package-Merge Algorithmus. Es wird die rekursive Version des Algorithmus aus [3] vorgestellt und seine Korrektheit bewiesen. In Kapitel 5 geht es um die Optimierung des gesamten Verfahrens. Die spezielle Struktur der Münzsammlerprobleme, die durch die Reduktion erzeugt wird, wird untersucht und für eine effiziente Implementierung des Verfahrens ausgenutzt. Der in diesem Kapitel vorgestellte Algorithmus basiert auf der iterativen

Version des Package-Merge Algorithmus von Larmore und Hirschberg und enthält Optimierungen, die Turpin und Moffat in [3] beschrieben haben. Eine Java-Version des optimierten Verfahrens liegt dieser Studienarbeit auf CD bei, auf die am Ende des 5. Kapitels und im Anhang A weiter eingegangen wird. Im 6. Kapitel werden Ergebnisse von Laufzeittests der Java-Implementierung bei verschiedenen Eingabeparametern präsentiert.

Kapitel 2

Grundlegene Sätze und Definitionen

In diesem Abschnitt werden wichtige Sätze aus der Kodierungstheorie und grundlegende Definitionen aufgeführt, die später in dieser Arbeit noch benötigt werden.

Ein zentrales Thema der Kodierungstheorie ist das kompakte Kodieren von Nachrichten (bestehend aus einzelnen Symbolen) in Folgen von Nullen und Einsen. (Es werden in dieser Arbeit nur *binäre* Codes betrachtet.) Ein wichtiges Modell dabei ist das Modell der *gedächtnislosen Quelle*. Die gedächtnislose Quelle produziert voneinander unabhängige Symbole, die es zu kodieren gilt. Im Folgenden gehen wir davon aus, dass die Symbole die Ausgabe einer gedächtnislosen Quelle sind.

Sei $A = (a_1, a_2, \ldots, a_n)$ ein Alphabet bestehend aus n Symbolen. Ein *Code C* für dieses Alphabet ist nun eine Abbildung, die jedem Element aus A eine endliche Bitfolge aus Einsen und Nullen zuordnet.

Damit die ursprüngliche Nachricht aus den Bitfolgen wieder rekonstruiert werden kann, müssen eindeutig entschlüsselbare Codes benutzt werden.

Ist die Abbildung der Menge von allen endlich langen Nachrichten, bestehend aus Folgen von Symbolen, auf endliche Bitfolgen durch Konkatenation der einzelnen Codeworte der Nachrichtensymbole injektiv, nennt man den benutzten Code *eindeutig entschlüsselbar*.

Die *Präfixcodes* gehören zu den eindeutig entschlüsselbaren Codes und haben zusätzlich die Eigenschaft, dass kein Codewort Präfix eines anderen Codewortes ist. Aufgrund dieser Eigenschaft lässt sich ein Präfixcode als binärer Baum darstellen, in dem die Blätter die einzelnen Symbole darstellen (siehe Abbildung 2.0.1). Das Dekodieren wird dadurch stark vereinfacht. Der Dekodierer fängt für ein neues Symbol in der Wurzel des Baumes an. Solange der aktuelle Knoten kein Blatt ist, liest der Dekodierer ein Bit ein und läuft in Abhängigkeit dieses Bits entweder den linken oder rechten Ast herunter. Dieser Schritt wird solange wiederholt, bis der Dekodierer in einem Blatt landet. In dem Fall gibt er das zum Blatt dazugehörige Symbol aus und fängt erneut mit der Wurzel des Baumes für das nächste Symbol an.

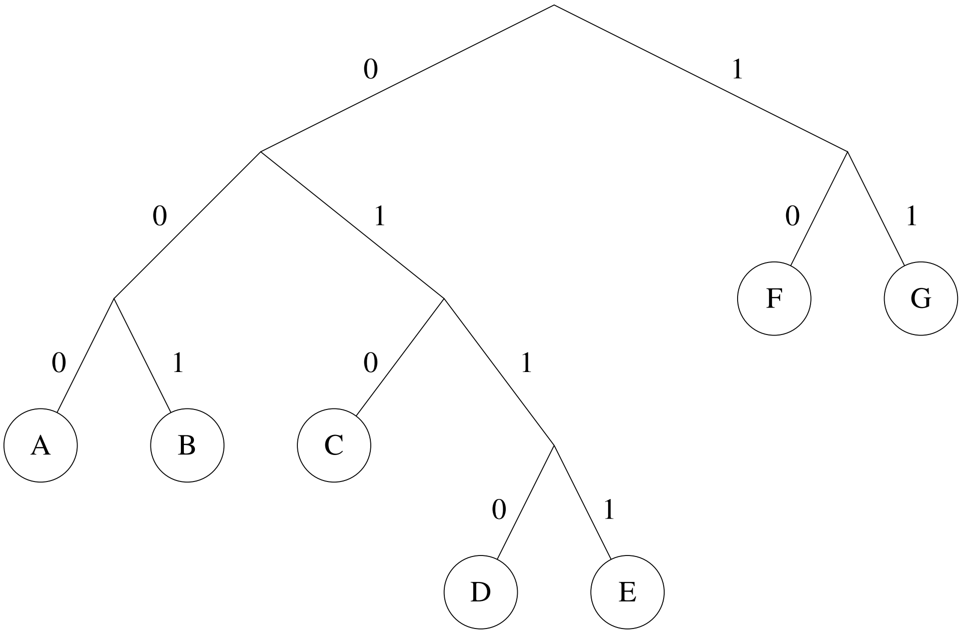


Abbildung 2.0.1: Beispiel der Binärbaumrepräsentation eines Präfixcodes, wie ihn der Huffman-Algorithmus ausgeben könnte

Ziel ist es, einen Code zu finden, der eine möglichst kurze Darstellung einer Nachricht liefert. Die Länge einer kodierten Nachricht wird davon abhängen, wie oft einzelne Symbole auftreten und wie lang die Codeworte (entspricht der Pfadlänge von der Wurzel zum Blatt im Baum) dieser Symbole sind.

Ist $p = (p_1, p_2, \ldots, p_n) \in \mathbb{R}^n$ mit $p_i > 0$ für alle i von 1 bis n die Wahrscheinlichkeitsverteilung für das Auftreten von n Symbolen $A = (a_1, \ldots, a_n)$ gegeben, kann für ein Code mit Codewortlängen $l = (l_1, l_2, \ldots, l_n) \in \mathbb{N}^n$ für das Alphabet A die *erwartete Codewortlänge* durch

$$\sum_{i=1}^{n} p_i l_i$$

berechnet werden.

Ein eindeutig entschlüsselbarer Code, der über allen möglichen eindeutig entschlüsselbaren Codes für eine gedächtnislose Quelle mit bestimmter Wahrscheinlichkeitsverteilung der Symbole eine minimale erwartete Codewortlänge besitzt, wird *kompakt* genannt.

Ein bekanntes Verfahren zur Bestimmung eines kompakten Präfixcodes ist der *Huffman*-Algorithmus (siehe [1] und [2]). Dieser kann einen solchen Code für n Symbole in Zeit $O(n \log n)$ (unsortierte Wahrscheinlichkeiten) beziehungsweise $O(n)$ (vorsortierte Wahrscheinlichkeiten) berechnen.

Satz 2.0.1. *(Kraft-McMillan) Sei ein Code mit Codewortlängen $l = (l_1, \dots, l_n) \in \mathbb{N}^n$ für ein Alphabet mit n Symbolen gegeben. Ist der Code eindeutig entschlüsselbar, dann gilt die folgende Ungleichung:*

$$\sum_i 2^{-l_i} \leq 1$$

Beweis. Kann in [2] nachgelesen werden. □

Satz 2.0.2. *Sei $n \in \mathbb{N}$ und $l = (l_1, l_2, \dots, l_n) \in \mathbb{N}^n$ gegeben. Erfüllt l die Ungleichung*

$$\sum_i 2^{-l_i} \leq 1 \, ,$$

kann ein Präfixcode mit Codewortlängen $l_1, \dots, l_n$ erzeugt werden.

Beweis. Kann in [2] nachgelesen werden. □

Definition 2.0.3. Eine Zahl $x \in \mathbb{R}$ ist genau dann eine *dyadische* Zahl, falls es ein $n \in \mathbb{N}$ mit $n \geq 0$ und $k_1, k_2, \dots, k_n \in \mathbb{Z}$ mit $k_1 < k_2 < \cdots < k_n$ gibt, so dass

$$|x| = \sum_{i=1}^{n} 2^{k_i}$$

gilt. Der kleinste Summand 2^{k_1} ist später von besonderem Interesse. Deshalb bezeichnen wir ihn mit *kleinstes dyadisches Glied* von x.

Beispiele: $12 = 2^3 + 2^2 \, , \; 1,125 = 2^0 + 2^{-3} \, , \; -0,75 = -(2^{-1} + 2^{-2})$

Kapitel 3

Die Probleme OLBPC und CCP

Im Folgenden wird gezeigt, wie das Problem, einen optimalen längenbeschränkten Präfixcode zu finden, auf das Münzsammlerproblem reduziert werden kann. Mit Hilfe eines Verfahrens, welches in der Lage ist, eine optimale Lösung des Münzsammlerproblems zu berechnen, kann dann auch unser ursprüngliches Problem gelöst werden.

Das Problem des Münzsammlers ist eine Variante des Rucksackproblems. Der Münzsammler ist im Besitz von einer Menge verschiedener Münzen, die neben ihrem eigentlichen Wert zusätzlich jeweils einen bestimmten Sammlerwert für den Münzsammler besitzen. Die Werte der Münzen in seinem Land sind immer Zweierpotenzen. Er möchte sich nun Lebensmittel im Wert von X kaufen, zum Bezahlen aber eine Teilmenge seiner Münzen auswählen, so dass er möglichst wenig an Sammlerwert verliert. Anstatt Sammlerwert wird nun der Begriff *Gewicht* benutzt, um Missverständnisse zu vermeiden.

3.1 Formale Beschreibung der Probleme

Definition 3.1.1. *OLBPC* (Optimal Length-Bounded Prefix Code) bezeichnet das Problem, einen optimalen längenbeschränkten Präfix-Code zu finden. Eine Instanz des OLBPC-Problems ist durch das Paar (p, L) gegeben, wobei $p = (p_1, \dots, p_n) \in \mathbb{R}^n$ mit $p_i > 0$ für alle $i = 1, \dots, n$ die Wahrscheinlichkeitsverteilung für das Auftreten von $n \geq 2$ Symbolen angeben und L die Längenbeschränkung für die zu bestimmenden Codewortlängen ist. Das Problem besteht darin, Codewortlängen $l_1, l_2, \dots, l_n \in \mathbb{N}$ zu bestimmen, so dass

1. $\sum_{i=1}^{n} 2^{-l_i} \leq 1$ (Es lässt sich nach Satz 2.0.2 ein Präfix-Code mit Codewortlängen l_i konstruieren)

2. $\forall i \; 1 \leq l_i \leq L$ (Die Codewortlängen sind durch L beschränkt)

3. Von all den möglichen Codewortlängen, die die ersten beiden Bedingungen erfüllen, sollen die Codewortlängen $l_1, l_2, \dots, l_n$ die erwartete Codewortlänge des Codes minimieren.

Definition 3.1.2. *CCP* bezeichnet das Münzsammlerproblem (Coin Collector's Problem). Eine CCP-Instanz ist durch das Tupel (M,w,X) gegeben, wobei $M \subset \mathbb{N} \times \mathbb{Z}$ eine endliche Menge ist, dessen Elemente die zur Verfügung stehenden Münzen des Münszammlers repräsentieren sollen. $n \in \mathbb{N}$ ist die kleinste Zahl, die

$$\forall (i,j) \in M : i \leq n$$

erfüllt. $w = (w_1, w_2, \ldots, w_n) \in \mathbb{R}^n$ sind Gewichte mit $w_i > 0$ für alle $i = 1, \ldots, n$ und X ist der zu zahlende dyadische Betrag mit $X > 0$. Jeder durch ein Paar $(i,j) \in M$ identifizierten Münze wird ein Gewicht durch $(i,j) \mapsto w_i$ und ein Wert durch $(i,j) \mapsto 2^j$ zugeordnet. Wir bezeichnen weiter die Summe der Werte aller Münzen einer Menge S mit

$$Wert(S) := \sum_{(i,j) \in S} 2^j$$

und die Summe der Gewichte aller Münzen einer Menge S mit

$$Gewicht(S) := \sum_{(i,j) \in S} w_i$$

Der Münzsammler möchte jetzt eine Teilmenge $S \subseteq M$ bestimmen, so dass:

1. $Wert(S) = X$ gilt
2. und S unter all den möglichen Teilmengen von M, die die erste Bedingung erfüllen, ein minimales Gewicht besitzt.

3.2 Reduktion von OLBPC auf CCP

Auf der Suche nach optimalen Präfixcodes für eine gegebene Wahrscheinlichkeitsverteilung und eine Rahmenbedingung wie eine Codewort-Längenbeschränkung können wir uns auf Codes mit Codewortlängen $l = (l_1, l_2, \ldots, l_n) \in \mathbb{N}^n$ beschränken, die die Gleichung

$$\sum_i 2^{-l_i} = 1$$

erfüllen, da andere Codes nach dem folgenden Lemma keine optimale Lösung sein können.

Lemma 3.2.1. *Sei n, p und L wie in Definition 3.1.1 gegeben. Sei weiter* $l = (l_1, l_2, \ldots, l_n) \in \mathbb{N}^n$ *eine optimale Lösung für die Probleminstanz* (p, L). *Es gilt dann:*

$$\sum_i 2^{-l_i} = 1$$

Beweis. Angenommen, die Lösung l erfülle *nicht* diese Gleichung, dann gilt

$$\sum_{i=1}^{n} 2^{-l_i} = 1 - x < 1 \text{ , mit } x \in \mathbb{R},\ 0 < x < 1,\ x \text{ dyadisch}$$

Sei $j \in \mathbb{N}$ mit $1 \leq j \leq n$ gegeben, so dass $l_j = \max(\{l_i \,|\, i = 1, \ldots, n\})$ gilt. Die Summe $\sum_{i=1}^{n} 2^{-l_i} = 1 - x$ ist damit ein Vielfaches von 2^{-l_j}. x ist damit auch ein Vielfaches von 2^{-l_j} und daher gilt $2^{-l_j} \leq x$. Es kann dann $l' = (l'_1, l'_2, \ldots, l'_n) \in \mathbb{N}^n$ durch

$$l'_i = \begin{cases} l_i & \text{für } i \neq j \\ l_i - 1 & \text{sonst} \end{cases} \quad \forall i = 1, \ldots, n$$

mit

$$\sum_{i=1}^{n} 2^{-l'_i} = \sum_{i=1}^{n} 2^{-l_i} + 2^{-l_j} \leq 1$$

konstruiert werden. Aus den Codewortlängen l' kann nach Satz 2.0.2 ein Präfixcode erzeugt werden. Da aber $l'_j < l_j$ und $p_j > 0$ gilt, besitzt dieser Präfixcode eine kürzere erwartete Codewortlänge als ein Code mit Codewortlängen l. Das ist ein Widerspruch zur Annahme, l sei eine optimale Lösung. □

Für die Reduktion muss zunächst gezeigt werden, dass bei bestimmten Ausprägungen des Münzsammlerproblems die optimale Lösung eine spezielle Gestalt besitzt. Anschließend wird bei der Reduktion aus der OLBPC-Instanz eine solche Ausprägung des Münzsammlerproblems erzeugt. Im Reduktionsbeweis wird dann die spezielle Gestalt der optimalen Lösung des Münzsammlerproblems ausgenutzt.

Lemma 3.2.2. *Sei $A \subset \mathbb{N} \times \mathbb{Z}$ eine endliche Menge mit $Wert(A) = X + k$ und $X \in \mathbb{N}$ mit $X > 0$, $k \in \mathbb{R}$ eine dyadische Zahl mit $0 < k < 1$ gegeben. Diese Menge besitzt nun eine Teilmenge $B \subset A$ mit $Wert(B) = k$.*

Beweis. Der Beweis erfolgt durch Induktion über die Anzahl der Elemente in M. Die Menge A muss mindestens 2 Elemente besitzen. Enthält die Menge genau 2 Elemente, muss M aus m_1 mit $Wert(\{m_1\}) = X$ und m_2 mit $Wert(\{m_2\}) = k$ bestehen. Die gesuchte Teilmenge B mit $Wert(B) = k$ ist dann $B = \{m_2\}$ (Induktionsanfang). Die Induktionsvoraussetzung ist die, dass die Aussage des Lemmas wahr für alle Mengen A' mit $|A'| < |A|$ ist. Enthält die Menge A mehr als 2 Elemente bezeichnen wir mit $m \in A$ ein Element, welches

$$\forall m' \in A : Wert(\{m\}) \leq Wert(\{m'\})$$

erfüllt. Das Element m ist nun eines der Elemente von A mit dem kleinsten Wert. $Wert(A)$ ist damit ein Vielfaches von $Wert(\{m\})$. Da $X < Wert(A) = X + k < X + 1$ gilt, muss der Wert von $\{m\}$ kleiner als 1 sein. Damit sind $1, X$ und $Wert(A) = X + k$ Vielfache von $Wert(\{m\})$. k muss dann auch ein Vielfaches von $Wert(\{m\})$ sein.

Gilt $Wert(\{m\}) = k$, ist eine Menge B mit $Wert(B) = k$ durch $B = \{m\}$ gefunden worden.

Gilt $Wert(\{m\}) < k$, kann nach Induktionsvoraussetzung eine Teilmenge B' von $A' = A \setminus \{m\}$ mit $Wert(B') = k - Wert(\{m\})$ berechnet werden, so dass $B \subset A$ mit $Wert(B) = k$ durch $B = B' \cup \{m\}$ bestimmt werden kann. □

Lemma 3.2.3. *Sei $n \in \mathbb{N}$, eine endliche Menge $M \subset \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le n,\ j < 0\}$, Gewichte $w = (w_1, w_2, \dots, w_n) \in \mathbb{R}^n$ mit $w_i > 0$ für alle $i = 1, \dots, n$ und ein $X \in \mathbb{N}$ gegeben. Existiert eine optimale Lösung S des Münzsammlerproblems (M, w, X), dann gilt*

$$\forall (i,j) \in S : (i, j+1) \in M \Rightarrow (i, j+1) \in S$$

Beweis. Sollte die Aussage nicht zutreffen, gäbe es mindestens ein $(i,j) \in S$ mit $(i, j+1) \in M$ und $(i, j+1) \notin S$. Das Lemma wird jetzt bewiesen, indem gezeigt wird, dass S in diesem Fall keine optimale Lösung sein kann.

Es kann aus der Menge S eine zweite Menge S' durch Ersetzen des Elementes (i,j) durch $(i, j+1)$ konstruiert werden:

$$S' \longleftarrow S \setminus \{(i,j)\} \cup \{(i,j+1)\}$$

Es folgt

$$Wert(S') = Wert(S) - 2^j + 2^{j+1} = Wert(S) + 2^j = X + 2^j$$

$$Gewicht(S') = Gewicht(S) - w_i + w_i = Gewicht(S)$$

Nach Lemma 3.2.2 kann eine Teilmenge A von S' mit $Wert(A) = 2^j$ bestimmt werden. Diese Menge A enthält mindestens ein Element und besitzt daher ein Gewicht größer als 0. Setzen wir $S'' := S' \setminus A$, so erhalten wir eine Teilmenge S'' von M mit

$$Wert(S'') = Wert(S') - 2^j = X + 2^j - 2^j = X$$

$$Gewicht(S'') = Gewicht(S') - Gewicht(A) = Gewicht(S) - Gewicht(A)$$

Damit ist aus S eine andere Menge konstruiert worden, die den gleichen Wert aber ein geringeres Gewicht besitzt. Das ist ein Widerspruch zur Annahme, die Menge S sei eine optimale Lösung für das Münzsammlerproblem. □

In Abbildung 3.2.1 ist ein Beispiel eines 8x5-Rasters zu sehen, dessen Felder die zur Verfügung stehenden Münzen (M) darstellen sollen. Die schattierten Felder sind Teil der optimalen Lösungsmenge S. Nach Lemma 3.2.3 gibt es innerhalb einer Spalte zwischen den Münzen, die in S enthalten sind, keine Lücken.

Satz 3.2.4. *OLBPC kann auf CCP reduziert werden.*

Beweis. Für den Beweis wird zunächst eine Reduktion angegeben und anschließend ihre Korrektheit bewiesen.
Reduktion: Sei n, p und L wie in 3.1.1 gegeben. Die Lösung des Problems kann mit Hilfe folgender Reduktion berechnet werden:

1. Setze
   $w \in \mathbb{R}^n$, $w_i := p_i$ für alle i,
   $M := \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le n,\ -L \le j \le -1\}$
   und $X = n - 1$

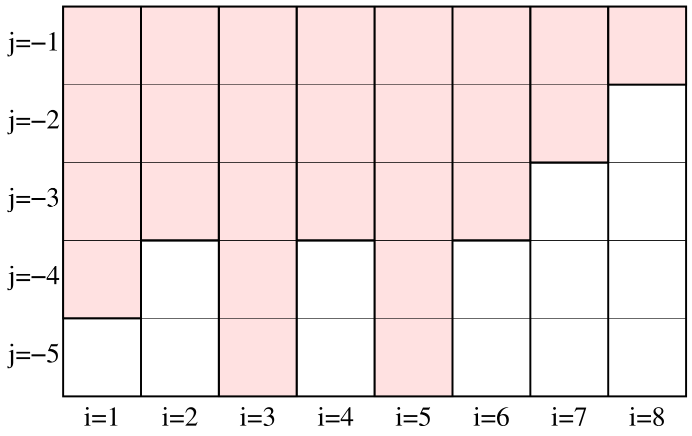


Abbildung 3.2.1: Beispiel für Lemma 3.2.3 der Lösungsmenge S (schattiert) bei $n = 8, X = 7, M = \{1, 2, \ldots, n\} \times \{-5, -4, \ldots, -1\}$ und willkürlich gewählten Gewichten w

2. Bestimme die optimale Lösung S des Münzsammlerproblems (M, w, X)
3. Bestimme die Liste $l \in \mathbb{N}^n$ als optimale Lösung durch

   $l_i := -\min(\{j \,|\, (i, j) \in S\})$

Um zu zeigen, dass die Codewortlängen $l_1, l_2, \ldots, l_n$ eine optimale Lösung für die OLBPC-Instanz (p, L) sind, muss überprüft werden, ob alle drei Bedingungen einer optimalen Lösung aus Definition 3.1.1 erfüllt sind.
Nach Lemma 3.2.3 folgt, dass sich die Menge S wegen ihrer speziellen Gestalt als

$$S = \{(i, j) \in \mathbb{Z}^2 \,|\, 1 \leq i \leq n, -l_i \leq j \leq -1\} \tag{3.2.1}$$

darstellen lässt. Mit Hilfe dieser Darstellung kann die Summe der Gewichte und Werte der Münzen, also das Gewicht der Menge S und der Wert der Menge S wie folgt berechnet werden:

$$\begin{aligned} Gewicht(S) &= \textstyle\sum_{i=1}^{n} w_i l_i \\ Wert(S) &= \textstyle\sum_{i=1}^{n} (\sum_{j=1}^{l_i} 2^{-j}) \\ &= \textstyle\sum_{i=1}^{n} (1 - 2^{-l_i}) \\ &= n - \textstyle\sum_{i=1}^{n} 2^{-l_i} \end{aligned}$$

Da S eine Lösung der CCP-Instanz $(M, w, X = n - 1)$ ist, gilt außerdem

$$Wert(S) = X = n - 1$$

Ein Vergleich der letzten 2 Gleichungen liefert, dass

$$\sum_{i=1}^{n} 2^{-l_i} = 1$$

gelten muss. Das bedeutet insbesondere: Es lässt sich nach Satz 2.0.2 ein Präfixcode mit den Codewortlängen l_i konstruieren. (Erste Bedingung von Definition 3.1.1 ist erfüllt.)

Weil die Werte der Münzen in M und daher auch in S größer oder gleich 2^{-L} sind, gilt $l_i \leq L$ für alle i von 1 bis n. (Zweite Bedingung von Definition 3.1.1 ist erfüllt.)

Weiter kann beobachtet werden, dass die erwartete Codewortlänge eines Codes mit Codewortlängen l_i, die es zu minimieren gilt, äquivalent zu *Gewicht*(S) ist, da $w_i = p_i$ für alle $i = 1, \ldots, n$ gilt. Es bleibt also zu zeigen, dass eine Lösung gefunden worden ist, die auch die dritte Bedingung der Definition 3.1.1 erfüllt.

Sollten die Codewortlängen $l_1, l_2, \ldots, l_n$ keine optimale Lösung für das Problem sein, gäbe es andere Codewortlängen $l'_1, l'_2, \ldots, l'_n$, die eine optimale Lösung für das OLBPC-Problem wären. Ein Präfixcode mit den Codewortlängen l' hätte damit auch eine kleinere erwartete Codewortlänge als ein Präfixcode mit Codewortlängen l. Weil die Codewortlängen l' nach Voraussetzung eine optimale Lösung sind, muss nach Lemma 3.2.1

$$\sum_{i=1}^{n} 2^{-l'_i} = 1$$

gelten. Aus diesen Codewortlängen kann eine Menge S' durch

$$S' = \{(i, j) \in \mathbb{Z}^2 \,|\, 1 \leq i \leq n, -l'_i \leq j \leq -1\}$$

erzeugt werden. S' ist nun eine Teilmenge von M, da l' eine optimale Lösung mit $l'_i \leq L$ für alle i von 1 bis n ist. Die Menge S' besitzt folgende Eigenschaften:

$$\begin{aligned}
\mathit{Wert}(S') &= \textstyle\sum_{i=1}^{n}(\sum_{j=-l'_i}^{-1} 2^j) \\
&= \textstyle\sum_{i=1}^{n} 1 - 2^{-l'_i} = n - 1 = X \\
\mathit{Gewicht}(S') &= \textstyle\sum_{i=1}^{n} p_i l'_i < \sum_{i=1}^{n} p_i l_i = \mathit{Gewicht}(S)
\end{aligned}$$

Damit stellt S' auch eine bessere Lösung für das Münzsammlerproblem (M, w, X) dar. Das ist aber ein Widerspruch zur Annahme, S sei eine optimale Lösung für (M, w, X). $\square$

Kapitel 4

Lösung des Problems CCP

4.1 Der Algorithmus „Package-Merge“

In Definition 3.1.2 wurden Münzen durch Elemente aus $\mathbb{N} \times \mathbb{Z}$ identifiziert, deren Gewicht in Abhängigkeit der ersten Komponente eines der Gewichte $w = (w_1, \ldots, w_n) \in \mathbb{R}^n$ mit $w_i > 0$ für alle i von 1 bis n ist, damit sich die Reduktion leichter aufschreiben lässt.

Zur Vereinfachung wird für die Beschreibung des folgenden Algorithmus der Umweg über separate Gewichte weggelassen. Münzen seien als abstrakte Objekte gegeben, die eine Zweierpotenz als Wert und ein beliebiges positives Gewicht größer Null als Eigenschaften besitzen, welche sich durch $Wert(\cdot)$ und $Gewicht(\cdot)$ abfragen lassen. M sei eine endliche Menge dieser Münzen und X der zu zahlende Betrag. Die Probleminstanz wird mit (M, X) abgekürzt.

Ist X keine dyadische Zahl, kann sie nicht durch eine endliche Summe von Zweierpotenzen dargestellt werden. Weil die Münzmenge M endlich ist, ist auch jede Teilmenge von M endlich und kann damit keine Lösung für das Problem sein. Ist X dyadisch und größer als 0, so bezeichnen wir das kleinste dyadische Glied (siehe Definition 2.0.3) von X mit $KX(X)$. Den kleinsten Wert einer Münze der Menge M bezeichnen wir mit $KM(M)$.

Der folgende Algorithmus basiert auf dem rekursivem „Package-Merge“ Verfahren aus [3]. Es geht davon aus, dass X eine dyadische Zahl größer oder gleich Null ist. Die Funktion $Merge$ erzeugt dabei anhand zwei gegebener Münzen gleichen Wertes eine neue Münze, die genauso viel wiegt und wert ist, wie die zwei gegebenen Münzen zusammen.

Die Anweisung „abbruch“ im Pseudo-Code sorgt dafür, dass die Ausführung des Algorithmus ganz gestoppt und die angegebene Fehlermeldung zurückgegeben wird.

Algorithmus 1: RecursivePackageMerge(Menge von Münzen M, Betrag X)

```
Fall 1  wenn X = 0 dann
            rückgabe von ∅
        sonst
            // Es gilt hier X > 0
Fall 2      wenn M = ∅ dann
                abbruch mit „Es gibt keine Lösung“
Fall 3      wenn KX(X) < KM(M) dann
                abbruch mit „Es gibt keine Lösung“
            T ⟵ {t ∈ M | Wert(t) = KM(M)}
Fall 4      wenn KX(X) = KM(M) dann
                m ⟵ leichteste Münze aus T
                S ⟵ RecursivePackageMerge(M \ {m}, X − Wert(m))
                S ⟵ S ∪ {m}
                rückgabe von S
Fall 5      wenn KX(X) > KM(M) dann
                wenn |T| < 2 dann
                    S ⟵ RecursivePackageMerge(M \ T, X)
                    rückgabe von S
                sonst
                    m1 ⟵ leichteste Münze aus T
                    m2 ⟵ leichteste Münze aus T \ {m1}
                    m3 ⟵ Merge(m1, m2)
                    S ⟵ RecursivePackageMerge(M \ {m1, m2} ∪ {m3}, X)
                    wenn m3 ∈ S dann
                        S ⟵ S \ {m3} ∪ {m1, m2}
                    rückgabe von S
```

4.2 Korrektheitsbeweis für „Package-Merge“

Die Korrektheit des Algorithmus kann durch Induktion über die Probleminstanz (M,X) gezeigt werden. Zunächst werden alle möglichen Fälle, die auftreten können, einzeln betrachtet und erklärt.

Lemma 4.2.1. *Sei eine CCP-Instanz (M,X) mit $X = 0$ (Fall 1) gegeben. Die optimale Lösung dieser Instanz ist die leere Menge.*

Beweis. Es muss eine Teilmenge $S \subseteq M$ als Lösung ausgewählt werden, so dass der Wert der Lösungsmenge 0 ist. Da alle Münzen einen Wert größer 0 besitzen, erfüllt nur die Menge $S = \emptyset$ diese Bedingung. Sie ist automatisch eine optimale Lösung. □

Lemma 4.2.2. *Sei eine CCP-Instanz (M,X) mit $X > 0$ und $M = \emptyset$ (Fall 2) gegeben. Es gibt dann keine Lösung für dieses Problem.*

Beweis. Es muss eine Teilmenge $S \subseteq M$ als Lösung ausgewählt werden, so dass $Wert(S) = X > 0$ gilt. Die einzig mögliche Teilmenge von M ist aber nur die leere Menge selbst, welche den Wert 0 besitzt. Es gibt daher keine Lösung. Der Algorithmus bricht in diesem Fall mit einer Fehlermeldung ab. □

Lemma 4.2.3. *Sei eine CCP-Instanz (M,X) mit $X > 0$, $|M| > 0$ und $KX(X) < KM(M)$ (Fall 3) gegeben. Es gibt dann keine Lösung für dieses Problem.*

Beweis. Die Werte der Münzen aus M sind Zweierpotenzen und sind daher ein Vielfaches von $KM(M)$. Damit muss der Wert jeder beliebigen Teilmenge von M auch ein Vielfaches von $KM(M)$ sein. X ist aber in diesem Fall kein Vielfaches von $KM(M)$ da sonst $KX(X) \geq KM(M)$ gelten müsste. Es kann darum keine Teilmenge S von M existieren, die den Wert X besitzt. □

Lemma 4.2.4. *Sei eine CCP-Instanz (M,X) mit $X > 0$, $|M| > 0$ und $KX(X) = KM(M)$ gegeben (Fall 4). Es kann eine optimale Lösung S für diese Instanz mit Hilfe einer optimalen Lösung S' für eine Instanz (M',X') mit $|M'| < |M|$ und $X' < X$ konstruiert werden. Sollte es keine solche Lösung S' geben, gibt es für die Instanz (M,X) auch keine Lösung.*

Beweis. Der Beweis erfolgt in zwei Schritten. Erst wird gezeigt, dass mindestens eine Münze aus

$$T = \{m \in M \,|\, Wert(m) = KM(M)\}$$

Teil optimaler Lösungen sein muss. Dann wird gezeigt, dass eine bestimmte Münze aus T Teil einer optimalen Lösung sein muss.

T ist die Menge der Münzen aus M, die den Wert $KM(M)$ besitzen. Alle Werte der Münzen aus $M \setminus T$ sind daher ein Vielfaches von $2KM(M)$. Sei eine optimale Lösung S für die CCP-Instanz (M,X) mit $S \subseteq (M \setminus T)$ gegeben. Der Wert von S muss nun ein Vielfaches von $2KM(M)$ sein. X ist aber *kein* Vielfaches von

$2KM(M)$, da sonst $2KM(M) \leq KX(X)$ gelten müsste. Daraus folgt $Wert(S) \neq X$. Das ist ein Widerspruch zu S sei eine optimale Lösung. Daher muss jede optimale Lösung mindestens eine Münze aus T enthalten.

Angenommen, S sei eine optimale Lösung, die keine Münze m aus T enthält, die die Ungleichung

$$\forall m' \in T : Gewicht(m) \leq Gewicht(m') \tag{4.2.1}$$

erfüllt. Es kann nun eine solche Münze $m \in T$ bestimmt werden, die diese Bedingung erfüllt. Die Münze m' sei eine beliebige Münze aus $S \cap T$. Eine solche Münze gibt es, da $T \cap S$ nicht leer ist (wie vorher gezeigt). Es gilt nun $Gewicht(m) < Gewicht(m')$, da m' sonst eine Münze wäre, die die Ungleichung (4.2.1) doch erfüllen würde. Es kann nun eine neue Menge S' durch

$$S' \longleftarrow S \setminus \{m'\} \cup \{m\}$$

mit

$$\begin{aligned}
Gewicht(S') &= Gewicht(S) - Gewicht(m') + Gewicht(m) \\
Wert(S') &= Wert(S) - Wert(m') + Wert(m) \\
&\underbrace{=}_{m,m' \in T} Wert(S) - KM(M) + KM(M) = Wert(S) = X
\end{aligned}$$

konstruiert werden. Da die Münze m aber leichter als die Münze m' ist, ist das Gewicht von S' auch geringer als das Gewicht von S. S kann damit keine optimale Lösung gewesen sein.

Das bedeutet jetzt, dass eine Münze m, die die Ungleichung (4.2.1) erfüllt, Teil einer optimalen Lösung sein muss. Damit muss sich eine optimale Lösung S für die Instanz (M,X) als Vereinigung der Menge {m} und einer Menge $S' \subseteq (M \setminus \{m\})$ mit $Wert(S') = X - Wert(m)$ darstellen lassen. Dabei muss die Menge S' von all den möglichen Teilmengen von $(M \setminus \{m\})$ vom Wert $X - Wert(m)$ das Gewicht minimieren, damit auch S eine optimale Lösung für (M,X) ist. Die gesuchte Menge S' stellt damit eine optimale Lösung der CCP-Instanz (M',X') mit $M' = M \setminus \{m\}$ und $X' = X - Wert(m)$ dar. Falls es keine solche Lösungsmenge S' gibt, kann es nun auch keine optimale Lösung S für die Instanz (M,X) geben. □

Lemma 4.2.5. *Sei eine CCP-Instanz (M,X) mit $X > 0$, $|M| > 0$ und $KX(X) > KM(M)$ gegeben (Fall 5). Es kann eine optimale Lösung S für diese Instanz mit Hilfe einer optimalen Lösung S' für eine Instanz (M',X') mit $|M'| < |M|$ und $X' = X$ konstruiert werden. Sollte es keine solche Lösung S' geben, gibt es für die Instanz (M,X) auch keine Lösung.*

Beweis. Das Lemma wird in zwei Schritten bewiesen. Erst wird gezeigt, dass in diesem Fall eine gerade Anzahl von Münzen aus

$$T = \{m \in M \,|\, Wert(m) = KM(M)\}$$

in einer optimalen Lösung enthalten sein muss. Anschließend wird gezeigt, dass, wenn mindestens zwei Münzen aus T in einer optimalen Lösung S liegen, zwei Münzen aus $S \cap T$ bestimmt werden können, die eine bestimmte Bedingung erfüllen.

Sei eine optimale Lösung S für die CCP-Instanz (M,X) gegeben, die eine ungerade Anzahl von Münzen aus T enthält. Der Wert von S ist damit ein Vielfaches von $KM(M)$ aber kein Vielfaches von $2KM(M)$. X ist aber ein Vielfaches von $2KM(M)$, da $KX(X) \geq 2KM(M)$ gilt. Das ist ein Widerspruch zu S sei eine optimale Lösung. Eine optimale Lösung muss in diesem Fall also eine gerade Anzahl von Münzen aus T enthalten.

Es wird jetzt gezeigt, dass, falls eine optimale Lösung Münzen aus T enthält, zwei Münzen $m_1, m_2 \in (S \cap T)$ gefunden werden können, die die Ungleichung

$$\forall m' \in (T \setminus \{m_1, m_2\}) : \mathit{Gewicht}(m_1) \leq \mathit{Gewicht}(m') \geq \mathit{Gewicht}(m_2) \qquad (4.2.2)$$

erfüllen. Sei eine optimale Lösung S für die CCP-Instanz (M,X) gegeben. Wir wissen bereits, dass $S \cap T$ eine gerade Anzahl von Münzen enthalten muss. Es bleibt zu zeigen, dass wenn $|S \cap T| \geq 2$ gilt, es 2 Münzen m_1, m_2 aus $S \cap T$ geben muss, die die Ungleichung (4.2.2) erfüllen. Angenommen, es können *nicht* 2 Münzen m_1, m_2 aus $S \cap T$ ausgewählt werden, so dass die Ungleichung (4.2.2) erfüllt ist, dann muss es zwei Münzen $m \in (T \setminus S)$ und $m' \in (S \cap T)$ mit $\mathit{Gewicht}(m) < \mathit{Gewicht}(m')$ geben. Es kann nun die Menge S' durch

$$S' \longleftarrow S \setminus \{m'\} \cup \{m\}$$

mit

$$\mathit{Gewicht}(S') = \mathit{Gewicht}(S) - \mathit{Gewicht}(m') + \mathit{Gewicht}(m) < \mathit{Gewicht}(S)$$

$$\mathit{Wert}(S') = \mathit{Wert}(S) = X$$

konstruiert werden. Das ist aber ein Widerspruch zur Annahme, S sei eine optimale Lösung.

Ist $|T| = 1$, kann die Münze aus T nicht Teil einer optimalen Lösung sein. Das Problem (M,X) darf auf das Problem (M',X') mit $M' = M \setminus T$ und $X' = X$ reduziert werden. Die Lösung für (M',X') ist zugleich Lösung für (M,X). Gibt es für (M',X') keine optimale Lösung, kann es für die Instanz (M,X) auch keine Lösung geben.

Enthält die Menge T zwei oder mehr Münzen, können zwei Münzen m_1 und m_2 bestimmt werden, die die Ungleichung (4.2.2) erfüllen. Es wurde gezeigt, dass, falls es mindestens eine optimale Lösung für (M,X) gibt, es eine optimale Lösung S geben muss, die entweder beide Münzen m_1 und m_2 oder gar keine Münzen aus T enthält. Daher dürfen die zwei Münzen m_1, m_2 zu einer neuen Münze m_3 mit

$$\mathit{Wert}(m_3) = \mathit{Wert}(m_1) + \mathit{Wert}(m_2)$$

$$\mathit{Gewicht}(m_3) = \mathit{Gewicht}(m_1) + \mathit{Gewicht}(m_2)$$

zusammengefasst werden und eine neue CCP-Instanz (M',X') mit

$$\begin{aligned} M' &= M \setminus \{m_1,m_2\} \cup \{m_3\},\ |M'| < |M| \\ X' &= X \end{aligned}$$

erzeugt werden. Es gibt genau dann eine Lösung S für die Instanz (M,X), wenn es eine Lösung S' für die Instanz (M',X') gibt. S kann durch

$$S \longleftarrow \begin{cases} S' & \text{für } m_3 \notin S' \\ S' \setminus \{m_3\} \cup \{m_1,m_2\} & \text{sonst} \end{cases}$$

konstruiert werden. □

Satz 4.2.6. *Der Package-Merge Algorithmus kann für jede gegebene CCP-Instanz (M,X) eine optimale Lösung $S \subseteq M$ berechnen, die die Bedingungen aus Definition 3.1.2 erfüllt, falls es eine solche Lösung gibt. Er bricht, falls es keine Lösung gibt, mit einer Fehlermeldung ab.*

Beweis. Die Fälle 1 bis 3 stellen die Abbruchbedingung des Rekursionsalgorithmus dar. Nach Lemma 4.2.1, 4.2.2 und 4.2.3 ist der Satz für Instanzen (M,X) mit $X = 0$, $M = \emptyset$ oder $KM(X) < KM(M)$ wahr.

In den restlichen Fällen 4 und 5 reduziert der Algorithmus die CCP-Instanz (M,X) nach Lemma 4.2.4 und 4.2.5 korrekt auf eine Instanz (M',X') mit $|M'| < |M|$ und $X' \leq X$ und kann aus einer optimalen Lösung S' für (M',X'), falls es eine solche gibt, eine optimale Lösung S für (M,X) konstruieren. Es wurde gezeigt, dass, wenn es keine solche Lösung S' gibt, das Problem (M,X) auch nicht lösbar ist. Der Algorithmus kann damit alle Probleminstanzen sukzessiv auf die Fälle 1 bis 3 zurückführen. □

Kapitel 5

Optimierung und Implementierung

In diesem Kapitel geht es darum, den Algorithmus „Package-Merge“ für unser eigentliches Problem zu optimieren. Es wird einerseits die spezielle Struktur der CCP-Instanz beschrieben, die durch die in Kapitel 3 beschriebene Reduktion erzeugt wird und zum anderen gezeigt, wie sich die Struktur für Optimierungen des Verfahrens ausnutzen lässt.

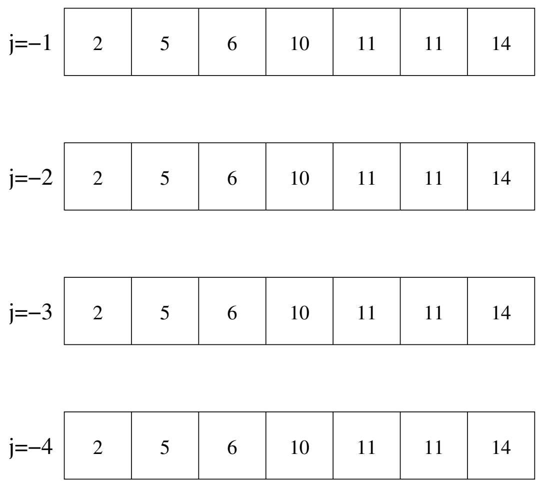
5.1 Die spezielle Struktur der erzeugten CCP Instanzen

Im Folgenden wird eine aufsteigend sortierte Wahrscheinlichkeitsverteilung der Symbole aus dem OLBPC-Problem vorausgesetzt. Die Allgemeinheit wird dadurch nicht eingeschränkt, weil jede Wahrscheinlichkeitsverteilung sortiert werden kann und die Codewortlängen, die der Algorithmus erzeugt, wieder so permutiert werden können, dass sie eine Lösung für die ursprüngliche Wahrscheinlichkeitsverteilung sind.

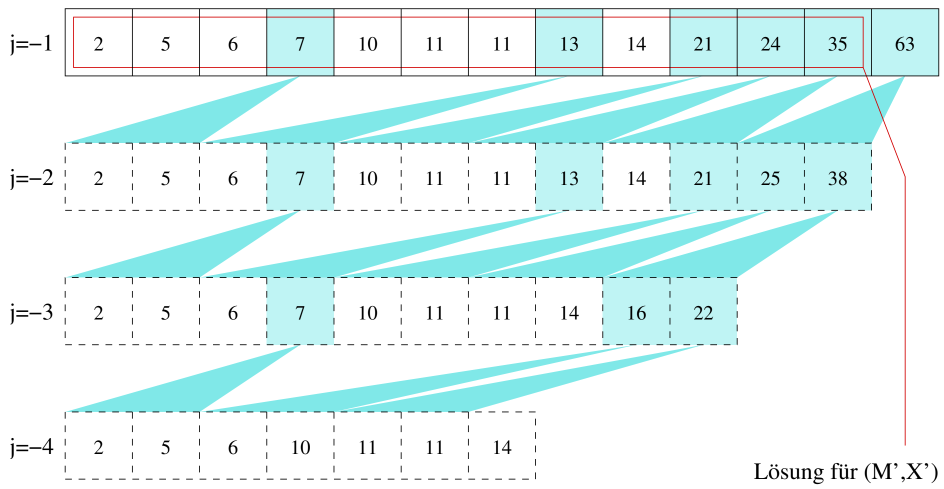
Die Reduktion von OLBPC auf CCP liefert eine Münzmenge mit nL Münzen. Jeweils n Münzen mit Wert 2^{-L}, n Münzen mit Wert 2^{-L+1} und so weiter bis zu n Münzen mit Wert 2^{-1}.

Außerdem fällt auf, dass der zu zahlende Betrag X mit $X = n - 1$ eine ganze Zahl ist und dass die Werte aller Münzen durch 2^{-1} nach oben beschränkt sind. Der Package-Merge-Algorithmus wird sich also in der Anfangsphase immer in dem 5. Fall befinden. Hier fasst der Algorithmus die Münzen aus $T \subseteq M$, die alle Münzen vom Wert $KM(M)$ enthält, paarweise zu neuen Münzen zusammen und entfernt restliche Münzen, die nicht Teil einer optimalen Lösung sein müssen.

Im Folgenden wird eine Listen-Darstellung der Münzmenge verwendet. Die Münzen werden dazu in L sortierten Listen abgelegt. Jede Liste enthält nur Münzen eines bestimmten Wertes. Die Münzen sind innerhalb der Listen nach ihrem Gewicht sortiert.



(a) Instanz (M,X)



(b) reduzierte Instanz (M',X')

Abbildung 5.1.1: CCP Instanzen vor und während der Anwendung des Algorithmus

Die Abbildung 5.1.1 zeigt zum einen ein Beispiel einer CCP-Instanz, die durch die in Kapitel 3 beschriebene Reduktion aus der OLBPC-Instanz

$$(p = \left(\frac{2}{59}, \frac{5}{59}, \frac{6}{59}, \frac{10}{59}, \frac{11}{59}, \frac{11}{59}, \frac{14}{59}\right), L = 4), \;\; n = 7$$

erzeugt worden ist und zum anderen eine CCP-Instanz (M', X'), auf die der Algorithmus „Package-Merge“ die Instanz (M, X) nach einigen Rekursionsaufrufen reduziert hat. Die Gewichte der Münzen wurden der Übersicht wegen mit 59 multipliziert und als ganze Zahlen in die Felder geschrieben. Die Gewichte innerhalb der i-ten Spalte sind gleich und entsprechen der ursprünglichen Symbolwahrscheinlichkeit des i-ten Symbols ($i = 1, \dots, n$) aus dem OLBPC-Problem.

5.2 Optimierung des Verfahrens

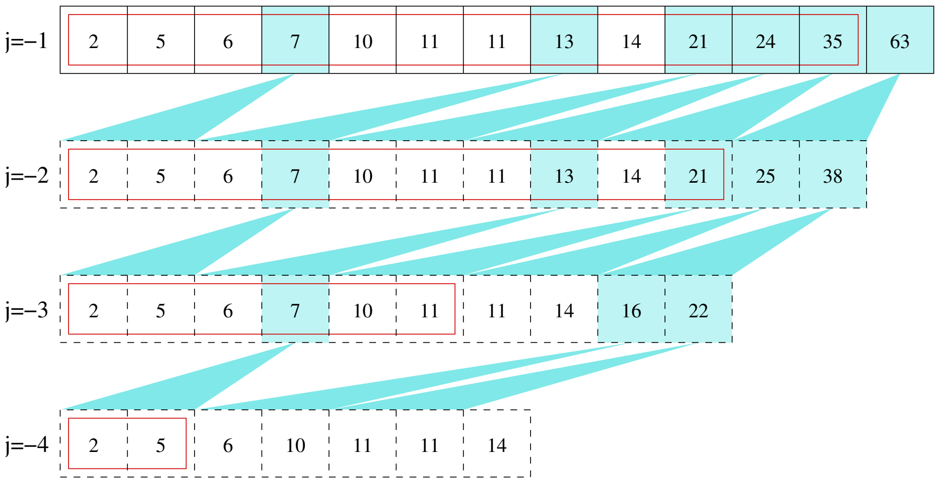
Wie in Abbildung 5.1.1b und 5.2.1a zu sehen ist, werden vom Algorithmus die Münzen vom Wert 2^{-L} der letzten Liste paarweise zusammengefasst und in der Liste darüber eingefügt. Dann werden die Münzen vom Wert 2^{-L+1} paarweise zusammengefasst und in der Liste mit Münzen vom Wert 2^{-L+2} eingefügt. Das ganze kann solange fortgesetzt werden, bis nur noch Münzen vom Wert 2^{-1} Teil der Menge M' sind, die in der ersten Liste enthalten sind. Der Package-Merge-Algorithmus wird genau das tun, weil bis hierhin $KX(X) \geq 1 > KM(M)$ (Fall 5) gelten wird.

Hat der Algorithmus eine solche CCP-Instanz (M', X') aus (M, X) erzeugt, in der nur noch Münzen vom Wert 2^{-1} in M' enthalten sind, können einfach die $2n-2$ leichtesten Münzen aus M' als Lösung ausgewählt werden. Das sind in der sortierten Liste dann die ersten $2n-2$ Elemente. Sie besitzen zusammen einen Wert von $(2n-2)2^{-1} = n-1 = X' = X$. Die Menge der $2n-2$ leichtesten Münzen aus M' ist somit eine optimale Lösung für die Instanz (M', X'). Es bleibt nun die Rücksubstitution der Münzen, die der Algorithmus aus zwei anderen Münzen erzeugt hatte, um hinterher die Codewortlängen l bestimmen zu können.

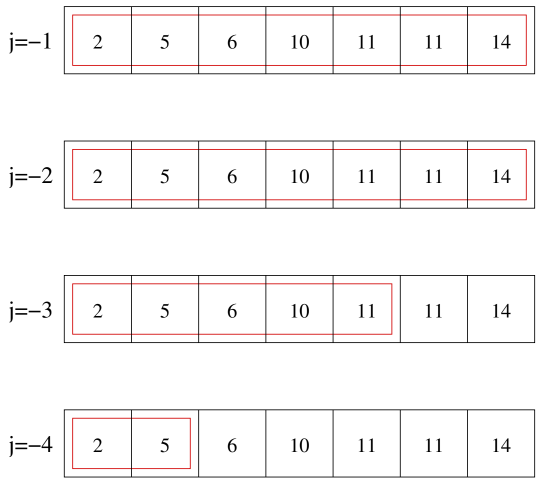
Es liegt nahe, zusammengesetzte Münzen als binäre Bäume zu modellieren, so dass die Münzen aus M, die zusammengefasst wurden, als Blätter in diesen Bäumen auftauchen. Damit kann dann die Lösungsmenge S aus S' bestimmt werden, indem die Bäume der zusammengesetzten Münzen aus S' nach den Blättern durchsucht werden. Diese Methode ist allerdings relativ zeit- und speicherintensiv.

Es ist gar nicht nötig, einen kompletten Baum für jede zusammengesetzte Münze zu speichern oder alle Münzlisten gleichzeitig im Speicher zu behalten (Beobachtung von Turpin und Moffat, siehe [4]). Die einzige Information, die ausreicht, um die Liste der Codewortlängen zu erzeugen ist die, welche Münzen zusammengesetzt worden sind und welche nicht.

Betrachten wir dazu die Abbildung 5.2.1a. Die Menge der ersten $2n-2 = 12$ Elemente der ersten Liste, die die Lösung S' darstellen, enthält 7 Münzen, die auch in M enthalten waren (hell) und 5 zusammengesetzte Münzen (dunkel). Diese 5 zusammengesetzten Münzen bestehen nun aus den ersten 10 Münzen der Liste



(a) reduzierte Instanz (M', X')



(b) Instanz (M, X) mit optimaler Lösung S

Abbildung 5.2.1: Bestimmung der Lösungsmenge S

darunter. Unter den ersten 10 Münzen der zweiten Liste befinden sich 3 zusammengesetzte Münzen, die aus den ersten 6 Münzen der nächsten Liste bestehen. Unter den ersten 6 Münzen aus der dritten Liste ist nur eine zusammengesetzte Münze, die aus den ersten zwei Münzen der vierten Liste bestehen. Es kann also nur mit Hilfe der Informationen, welche Münzen zusammengesetzt worden sind, die Lösung S ermittelt werden.

In der Abbildung 5.2.1b ist die optimale Lösung S (eingerahmt) des CCP-Problems (M,X) zu sehen. Aus ihr kann direkt die optimale Lösung des OLBPC-Problems l abgelesen werden. Die l_i geben – wie in Kapitel 3 gezeigt – die Anzahl der Münzen in der i-ten Spalte an, die in der Lösung S enthalten sind, wenn die Münzmenge wie in der Abbildung als Raster dargestellt wird. Es muss nur in jeder Spalte die Anzahl der Münzen, die in S enthalten sind, bestimmt werden. Die Codewortlängen l sind damit

$$l = (4,4,3,3,3,2,2).$$

Die Listendarstellung der Münzmenge kann direkt für eine iterative Version des Verfahrens ausgenutzt werden. Die Werte der Münzen müssen dadurch nicht mehr gesondert gespeichert werden, da es für jeden Münzwert eine entsprechende Liste gibt. Die Werte der Münzen liegen damit implizit fest. Die iterative Version des Lösungsverfahrens braucht dann nur noch die Gewichte in den Listen verwalten, Gewichte paarweise zusammen fassen, sortierte Listen zusammenfügen und aufzeichnen, wo zusammengesetzte Gewichte (von zusammengesetzten Münzen) in den Listen aufgetreten sind.

Lemma 5.2.1. *Die Längen der einzelnen Listen, die die Münzgewichte speichern, sind durch* $2n-1$ *nach oben beschränkt.*

Beweis. Die letzte Liste, die die Gewichte der Münzen vom Wert 2^{-L} enthält, besitzt genau die Länge n. Die Liste darüber besitzt n Gewichte plus $\lfloor\frac{n}{2}\rfloor$ Gewichte von n aus der unteren Liste zusammengesetzten Münzen. Es kann daher die Folge von Listenlängen durch

$$\begin{array}{rcll} f_0 & = & n & \\ f_i & = & n + \lfloor\frac{f_{i-1}}{2}\rfloor & \text{für } i \in \mathbb{N},\ i > 0 \end{array}$$

angegeben werden. $f_i \le 2n-1$ kann per Induktion für alle $i \ge 0$ gezeigt werden. Für $i = 0$ ist es offensichtlich wahr. Für den Fall $i > 0$ gilt die Induktionsvoraussetzung:

$$\begin{array}{rrcl} & f_{i-1} & \le & 2n-1 \\ \Rightarrow & \lfloor\frac{f_{i-1}}{2}\rfloor & \le & n-1 \\ \Leftrightarrow & n + \lfloor\frac{f_{i-1}}{2}\rfloor & \le & 2n-1 \\ \Leftrightarrow & f_i & \le & 2n-1 \end{array}$$

□

Für die Implementierung dieser Listen bieten sich die in Bibliotheken vorhandene Datenstrukturen an. Sie können aber auch durch Arrays abgebildet werden. Arrays sind speichersparender, effizienter zu handhaben und stellen in diesem Fall keinen Nachteil gegenüber verketteten Listen dar. Die maximale Anzahl von Gewichten, die in einer Liste abgelegt werden müssen, ist bekannt. Es kann also ein Array der Größe $2n-1$ für die Speicherung der Gewichte-Liste erzeugt werden.

Der folgende Pseudo-Code (siehe Algorithmus 2) ist eine iterative Version des Verfahrens. Sie vereinigt die Reduktion von OLBCP auf CCP und eine speziell optimierte Version des Package-Merge-Algorithmus. Sie erwartet die Symbolwahrscheinlichkeiten aus dem OLBPC-Problem in einer aufsteigend sortierten Reihenfolge als Array mit der Codewort-Längenbeschränkung L und gibt die gewünschten Codewortlängen als Array zurück.

Der Algorithmus erzeugt nach und nach die sortierten Gewicht-Listen für Münzen vom Wert 2^{-L}, 2^{-L+1}, ..., 2^{-1} aus der vorherigen Liste (A) und den Symbolwahrscheinlichkeiten (W) und legt sie in B ab. Dabei wird das Bitfeld P aktualisiert, welches an der Position [j,x] genau dann eine Eins enthält, wenn in der Gewicht-Liste für die Münzen vom Wert 2^{-j} an der x-ten Stelle ein Gewicht einer zusammengesetzten Münze steht. Die alte Liste A wird für die nächste Wiederholung der Schleife (Zeile 1) nicht mehr benötigt, so dass die Array-Referenzen A und B vertauscht werden können.

Ab Zeile 2 werden die Codewortlängen berechnet. Wie in Abbildung 5.2.1a überprüft der Algorithmus, wie viele von den k leichtesten Münzen der aktuellen Liste zusammengesetzt waren und zählt diese in der Variablen z. Enthält das Bitfeld für eine Münze an der entsprechenden Stelle eine 0, so war diese Münze auch in der ursprünglichen Münzmenge M der CCP-Problem-Instanz (M,X) enthalten. Die Codewortlänge des zur Münze gehörenden Symbols wird inkrementiert. Die z gefundenen zusammengesetzten Münzen bestehen nun aus den $2z$ ersten Münzen der nächsten Liste. Diese Schleife wird solange fortgesetzt, bis all die ursprünglichen Münzen, die zusammengefasst wurden, gefunden und die Codewortlängen der dazugehörigen Symbole inkrementiert wurden.

5.3 Speicherverbrauch

Der Algorithmus arbeitet mit 4 eindimensionalen Arrays und einem zweidimensionalem Bitfeld. Angenommen, w_M sei die Anzahl der Bits eines Maschinenwortes und alle Elemente der Arrays (also Gewichte und Codewortlängen) lassen sich jeweils in einem Maschinenwort ablegen. Dann benötigt der Algorithmus zur Erzeugung der Codewortlängen für n Symbole und einer Längenbeschränkung L

$$2n+2(2n-1)+\left\lceil(2n-1)\frac{L}{w_M}\right\rceil+O(1) \text{ Maschinenworte,}$$

um die Lösung zu berechnen. Die Arrays W und R haben jeweils eine Länge von n, die Arrays A und B besitzen eine Länge von $2n-1$. Der dritte Summand wird für die Speicherung des Bitfeldes P benötigt.

Algorithmus 2: ItPackMerge(Gewichte-Array W, Längenbeschränkung L)

$n \longleftarrow$ Anzahl der Gewichte in W
wenn $2^L < n$ **dann**
  abbruch mit „Es gibt keine Lösung“
P sei ein $L \times (2n-1)$-Bitfeld gefüllt mit Nullen.
A und B seien Arrays für Gewichte der Größe $2n-1$
R sei ein Array für die Codewortlängen der Größe n gefüllt mit Nullen
für $i \leftarrow 1$ *bis* n **tue**
  $A[i] \longleftarrow w_i$
$l_A \longleftarrow n$
1 **für** $j \leftarrow L-1$ *bis* 1 **tue**
  $p_A \longleftarrow 1,\ p_B \longleftarrow 1,\ p_W \longleftarrow 1$
  solange $p_A + 1 \leq l_A$ **tue**
    $s \longleftarrow A[p_A] + A[p_A+1]$
    $p_A \longleftarrow p_A + 2$
    solange $p_W \leq n$ *und* $W[p_W] \leq s$ **tue**
      $B[p_B] \longleftarrow W[p_W]$
      $p_B \longleftarrow p_B + 1,\ p_W \longleftarrow p_W + 1$
    $B[p_B] \longleftarrow s$
    $P[j, p_B] \longleftarrow 1$
    $p_B \longleftarrow p_B + 1$
  solange $p_W \leq n$ **tue**
    $B[p_B] \longleftarrow W[p_W]$
    $p_B \longleftarrow p_B + 1,\ p_W \longleftarrow p_W + 1$
  vertausche Referenzen der Arrays A und B
  $l_A \longleftarrow p_B - 1$
2 $j \longleftarrow 0,\ k \longleftarrow 2n-2$
solange $k > 0$ **tue**
  $j \longleftarrow j+1,\ p_W \longleftarrow 1,\ z \longleftarrow 0$
  für $i \leftarrow 1$ *bis* k **tue**
    wenn $P[j,i] = 1$ **dann**
      $z \longleftarrow z+1$
    sonst
      $R[p_W] \longleftarrow R[p_W] + 1$
      $p_W \longleftarrow p_W + 1$
  $k \longleftarrow 2z$
rückgabe von R

Teil der Motivation für längenbeschränkte Codes ist die Möglichkeit, jedes Codewort in einem Maschinenwort ablegen zu können. Wenn wir auf einem 32Bit-System, wie es zum Beispiel Java ist, $L \leq w_M = 32$ wählen, kann das Bitfeld – bestehend aus $L(2n-1)$ Bits – mit $2n-1$ Maschinenworten (als `int[]`-Array der Länge $2n-1$) gespeichert werden. In diesem Szenario werden dann

$$8n - 3 + O(1) \text{ Maschinenworte}$$

für die Lösung des Problems benötigt.

Turpin und Moffat geben noch einen weiteren Optimierungstrick in ihrer Veröffentlichung [4] an. Dieser Trick macht die zwei Arrays A und B der Länge $2n-1$ überflüssig und erfordert dafür ein Array der Länge n, in dem „in-place“ die neuen Gewichte zusammengesetzter Münzen erzeugt werden. Allerdings muss man bei diesen Trick eine erhöhte Komplexität des Algorithmus und eine etwas längere Ausführungszeit in Kauf nehmen, da die Gewichte-Listen nicht mehr auf ein Array sondern auf 2 Arrays abgebildet werden. Bei der Handhabung dieser verteilt gespeicherten Listen werden noch zusätzliche Vergleiche benötigt, um die Ordnung zu rekonstruieren. Diese Optimierung ist nicht mehr Teil dieser Studienarbeit, da der Speicheraufwand mit etwa $8n$ Maschinenworten für einen Präfixcode mit n Codeworten in praktischen Anwendungsfällen ausreichend klein ist. Sollte der Speicheraufwand des in dieser Studienarbeit vorgestellten Verfahrens mit etwa $8n$ Maschinenworten noch zu hoch sein, kann er durch den erwähnten Trick um weitere 37% verringert werden.

5.4 Laufzeitklasse

Der Algorithmus berechnet L Listen von Gewichten, die eine maximale Länge von $2n-1$ besitzen können und durchläuft Teile des erzeugten $L \times (2n-1)$-Bitfeldes, um die Codewortlängen zu berechnen. Die Laufzeitklasse des Algorithmus ist damit:

$$O(nL)$$

Der optimierte Algorithmus „Package-Merge“ setzt aber aufsteigend sortierte Wahrscheinlichkeiten voraus. Eine Sortierung der einzelnen Wahrscheinlichkeiten ist in einer Zeit $O(n \log n)$ berechenbar. Im Falle einer lösbaren OLBPC-Instanz muss aber $\log n \leq L$ gelten, da ein Präfixcode, dessen Codeworte maximal L Bits lang sind, nur maximal 2^L verschiedene Codeworte besitzen kann. Das bedeutet, dass die Laufzeitklasse des Lösungsverfahren inklusive Sortierung der Wahrscheinlichkeiten die gleiche wie beim Verfahren ohne Sortierung bleibt.

5.5 Java-Implementierung

Die Virtual-Machine, auf der Java-Programme ausgeführt werden, arbeitet mit Maschinenworte einer Größe von 32 Bit. Es bietet sich daher an, die Gewichte als

32Bit-Ganzzahl (`int`) oder 32Bit-Fließkommazahl (`float`) zu speichern. In der Java-Implementierung wurden für die Gewichte `float[]`-Arrays verwendet, damit der Algorithmus direkt eine Wahrscheinlichkeitsverteilung als Eingabe annehmen kann. Die Summe der Gewichte muss aber nicht Eins ergeben, damit der Algorithmus korrekt arbeitet. Das Gewichte-Array könnte auch gemessene Symbolfrequenzen einer sehr langen typischen Nachricht enthalten.

Die maximale Längenbeschränkung L, die als Eingabe für den Algorithmus erlaubt wird, wurde durch 32 beschränkt. Dadurch lässt sich das Bitfeld P durch ein `int[]`-Array der Länge $2n-1$ darstellen. Falls Codewort-Längenbeschränkungen im Bereich $33 \ldots 64$ benötigt werden, müssen in der Java-Implementierung nicht viele Änderungen vorgenommen werden.

Weitere Details zur Implementierung sind in der kurzen Dokumentation der Java-Implementierung in Anhang A gegeben.

Kapitel 6

Laufzeitmessungen

In diesem Kapitel werden die Ergebnisse des der Java-Implementierung beiliegenden Test-Programms vorgestellt. Diese Ergebnisse wurden mit Hilfe eines Rechners mit Intel Pentium 4 Prozessor (2,4 GHz Taktfrequenz), einem Gigabyte Arbeitsspeicher und dem Standard Java Runtime Environment 1.4.2 von Sun für das Linux-Betriebssystem erstellt.

Das Test-Programm ermittelt bei mehreren Durchläufen die durchschnittliche Zeit in Millisekunden, die für die Erzeugung der Codewortlängen benötigt werden. Der Algorithmus muss für verschiedene Eingaben ($n = 2^8, 2^{12}, 2^{16}, 2^{20}$ und $L = 12, 16, 20, 24, 28, 32$ mit Gewichten, die durch

$$w_i = e^{\sigma^2(-1+\frac{i-0.5}{n})^2} \text{ für alle } i = 1, \ldots, n,\ \sigma = 3.$$

berechnet werden) eine Lösung ermitteln. Die Ergebnisse sind in der Tabelle 6.1 zusammengefasst. Die erste Zahl gibt dabei die benötigte Zeit für den Fall von bereits sortierten Gewichten an. Die zweite Zahl gibt die Laufzeit inklusive Sortierung der Symbolgewichte (QuickSort) und Permutation der Codewortlängen an. Dazu wurden mit Hilfe eines Zufalls-Generators die Symbolgewichte w_i vorher permutiert.

Die Wahl der Gewichte wurde nur der Vollständigkeit wegen angegeben. Experimente mit verschiedenen Verteilungen (verschiedene Werte für σ) haben ergeben, dass die Laufzeit des Verfahrens – zumindest bei diesem Modell – weitestgehend unabhängig von der Verteilung ist.

Laufzeit [ms] mit sortierten / unsortierten Gewichten	$n = 2^8$ (256)	$n = 2^{12}$ (4096)	$n = 2^{16}$ (65536)	$n = 2^{20}$ (1048576)
$L = 12$	0,14 / 0,17			
$L = 16$	0,17 / 0,22	3,14 / 4,11		
$L = 20$	0,21 / 0,26	3,78 / 4,55	119 / 155	
$L = 24$	0,25 / 0,29	4,41 / 5,41	132 / 171	1994 / 3310
$L = 28$	0,28 / 0,32	5,06 / 5,86	146 / 186	2246 / 3684
$L = 32$	0,32 / 0,37	5,67 / 6,45	161 / 203	2546 / 3979

Tabelle 6.1: Laufzeitmessungen (Angaben in Millisekunden)

Kapitel 7

Fazit

Es wurde gezeigt, wie optimale längenbeschränkte Codewortlängen für Präfixcodes effizient berechnet werden können. Der im Kapitel 5 vorgestellte Algorithmus lässt sich leicht in viele Programmiersprachen übersetzten, da die benötigten Konstrukte (Arrays, Schleifen) in fast jeder Sprache vorhanden sind.

Für praktische Anwendungsfälle (in der Regel kleine Alphabete, $n < 1000$) kann die beiliegende Java-Implementierung in nur wenigen Millisekunden eine optimale Lösung finden (siehe Kapitel 6) und für die berechneten Codewortlängen die entsprechenden Codeworte generieren (siehe Anhang A). Der Speicherverbrauch ist mit etwa $8n$ Maschinenworten (die Eingabe mitgerechnet) sehr gering. Dieser kann sogar noch weiter reduziert werden, wenn man die letzte Optimierungsidee von Turpin und Moffat aus ihrer Veröffentlichung [4] umsetzt.

Man könnte diesen Algorithmus aufgrund seiner Performanz generell Huffman-Varianten vorziehen, auch wenn gegebene Wahrscheinlichkeitsverteilungen beim Huffman-Verfahren keine Gefahr bezüglich der Codewort-Längenbeschränkung darstellen. Nach Turpin und Moffat (siehe [4]) benötigt das optimierte Package-Merge Verfahren für eine Eingabe mit etwa 2^{20} Gewichten (sortiert) und einer Längenbeschränkung von 32 auf einer Sun SPARC 10 Model 402 etwa genauso viel Zeit wie die optimierte Huffman-Implementierung aus [5].

Anhang A

Dokumentation der Java-Implementierung

A.1 Gliederung des Quellcodes

Auf der beiliegenden CD befindet sich die Java-Implementierung des optimierten Verfahrens als Quellcode (im Verzeichnis `source`) und als eine kompilierte JAR-Datei, die bei Ausführung ein Kommandozeilen-Interface startet.

Zwei Java-Klassen besitzen eine `main`-Methode. Das ist zum einen die Klasse `CLI`, die das Kommandozeilen-Interface für die Erzeugung von den Codes bereitstellt und zum anderen die Klasse `Test`, mit der die Performanz der Implementierung gemessen werden kann.

Beide Programme machen von der Klasse `OLBPC` Gebrauch, in der der optimierte Algorithmus enthalten ist. Die Klasse ist unabhängig von den anderen Klassen und kann daher ohne Änderungen allein wiederverwendet werden. Mit den Methoden `createCodeLengthsSortedFreqs` und `createCodeLengths` können die Codewortlängen für optimale längenbeschränkte Präfixcodes berechnet werden. Die erste von den beiden setzt aufsteigend sortierte Symbolgewichte voraus. Die zweite Methode sortiert vorher die Symbolgewichte, ruft dann die erste Methode auf und permutiert die Codewortlängen so, dass sie wieder zur ursprünglichen Eingabe passen. Die beiden Methoden werfen eine `IllegalArgumentException` falls das übergebene Gewichte-Array weniger als 2 Elemente besitzt, die Längenbeschränkung zu klein für die Symbolanzahl oder größer als 32 ist, da das Bitfeld P (siehe Algorithmus 2) durch das `int`-Array `packageFlags` abgebildet wird. Die Ordnung der Symbolfrequenzen genauso wie ihre Werte werden nicht zusätzlich überprüft.

Die Klasse `Tools` stellt die Methode `genCodes` zur Verfügung, mit der die Codeworte zu gegebenen Codewortlängen (deterministisch) erzeugt werden können. Das zurückgegebene `int[]`-Array wird die Codeworte eines Präfixcodes in den niederwertigen Bits der einzelnen `int`s enthalten (MSB[1] first).

[1] most significant bit

Mit Hilfe der Interfaces `ImportFilter` und `ExportFilter` können beliebige Eingabe- und Ausgabefilter zum Einlesen der Wahrscheinlichkeitsverteilung und Exportieren der Codes zu der Kommandozeilen-Anwendung hinzugefügt werden.

A.2 Anwendung des Kommandozeilen-Interfaces

Das Kommandozeilen-Interface ist in der Lage von der Standard-Eingabe oder aus einer Datei die Parameter Symbolanzahl, Längenbeschränkung und Symbolgewichte einzulesen. Es erzeugt mit Hilfe der Methoden aus den Klassen `OLBPC` und `Tools` einen optimalen längenbeschränkten Präfixcode und kann ihn bis jetzt in zwei Formaten auf die Standard-Ausgabe oder in eine Datei schreiben.

Der Eingabe-Filter „simple“ erwartet eine einfache Textdatei, die die Symbolanzahl, die Längenbeschränkung und die Gewichte enthält. Die Gewichte müssen dabei nicht sortiert vorliegen (siehe Abbildung A.2.1). Die Voreinstellung für die Ausgabe ist der Filter „bintext“, der die einzelnen Codeworte Zeile für Zeile als ASCII-Zeichenkette (binär) ausgibt (siehe Abbildung A.2.2). Der zweite Ausgabe-Filter mit Namen „javacode“ erzeugt Java Quellcode für den Präfixcode, den man sehr leicht in Java-Programme einbetten kann. Der Präfixcode wird als Definition eines `int[][]`-Arrays der Dimension $2 \times n$ ausgegeben. Es enthält die Codeworte als Ganzzahlen in den niederwertigen Bits (MSB first) und die Codewortlängen (siehe Abbildung A.2.3).

Das Programm kann bei korrekt installiertem Java Runtime Environment durch die Eingabe folgenden Befehls aus dem Verzeichnis, welches die Datei `cli.jar` enthält, gestartet werden:

`java -jar cli.jar` *weitere Parameter ...*

Ein Hilfe-Text wird automatisch vom Programm als Ausgabe generiert, wenn keine weiteren Parameter angegeben werden. Dieser erklärt die genaue Syntax der Aufruf-Parameter.

```
12      # Anzahl der Symbole
8       # Laengenbeschraenkung

1024.65 # Erstes Gewicht
512     # Zweites Gewicht
256     # und so weiter ...
128.42
64.24   # In einer Zeile wird alles hinter
32.42   # dem ersten Vorkommen von '#' ignoriert.
16.24
8.0815
4.39
3.14159265
1.3
1.101
```

Abbildung A.2.1: Beispiel-Eingabe für das Kommandozeilen-Interface CLI

```
0
10
110
1110
11110
1111100
11111010
11111011
11111100
11111101
11111110
11111111
```

Abbildung A.2.2: Die entsprechende Ausgabe vom `bintext`-Filter

```
static final int[][] CODE = {{
        0x00,0x02,0x06,0x0e,0x1e,0x7c,0xfa,0xfb,
        0xfc,0xfd,0xfe,0xff
},{
        1,2,3,4,5,7,8,8,
        8,8,8,8
}};
```

Abbildung A.2.3: Die entsprechende Ausgabe vom `javacode`-Filter

Literaturverzeichnis

[1] David A. Huffman. *A Method for the Constduction of Minimum-Redundancy Codes*. Procedings of the I.R.E. 1952

[2] Khalid Sayood. *Introduction to Data Compression*. 2nd Edition. ISBN 1-55860-558-4. 2000

[3] Lawrence L. Larmore, Daniel S. Hirschberg. *A Fast Algorithm for Optimal Length-Limited Huffman Codes*. 1990

[4] Andrew Turpin, Alistair Moffat. *Practical Length-Limited Coding for Large Alphabets*. 1994

[5] I.H. Witten, Alistair Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images*. 1994