Technische Universität München
Lehrstuhl für Nachrichtentechnik
Prof. Dr. sc. techn. Gerhard Kramer

Bachelorarbeit

ICT Cubes Decode Webservice: Von Lamellensequenzen zu Klartext

Vorgelegt von:
Sebastian Baur

München, Juni 2013

Betreut von:
Dr. Georg Böcherer

Bachelorarbeit am
Lehrstuhl für Nachrichtentechnik (LNT)
der Technischen Universität München (TUM)
Titel : ICT Cubes Decode Webservice: Von Lamellensequenzen zu Klartext
Autor : Sebastian Baur

Sebastian Baur
Theresienstraße 100
80333 München
BaurSebastian@mytum.de

Ich versichere hiermit wahrheitsgemäß, die Arbeit, bis auf die dem Aufgabensteller bereits bekannte Hilfe, selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten Anderer unverändert oder mit Abänderung entnommen wurde.

München, 24. Juni 2013

Ort, Datum (Sebastian Baur)

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In Asien und insbesondere in Japan findet das sogenannte Mobile Tagging [8] Anwendung im mobilen Marketing. Beim Mobile Tagging wird ein Strichcode mit Hilfe der Kamera eines Mobiltelefons eingelesen. Meist handelt es sich dabei um zweidimensionale Strichcodes. Einer der in Deutschland am weitesten verbreitete Code ist der *Quick Response Code* (QR-Code). Der Strichcode enthält kontextspezifische Informationen. Beispielsweise kann es sich dabei um einen *Uniform Resource Identifier* (URI) handeln. So können Webanwendungen mit Gegenständen des täglichen Lebens verknüpft werden. Dieses Prinzip wird auch als Physical World Connection bezeichnet. Es findet Anwendung bei einem neuen Bauprojekt der RWTH Aachen. Seit März 2013 werden an der RWTH Aachen neue Gebäude für sechs Institute der Fakultät für Elektro- und Informationstechnik gebaut [12]. Die Außenwand dieser sogenannten *Information and Communication Technology Cubes* (ICT Cubes) ist mit nebeneinander angeordneten Lamellen verkleidet [7]. Abbildung 1.1 zeigt einen Entwurf der ICT Cubes. Die Lamellen sollen einerseits die Gebäude wie Würfel aussehen lassen und andererseits zur Beschattung dienen. Es gibt verschiedene Ausführungen der Lamellen. Sie werden in Kapitel 3.1 genauer beschrieben. Somit ist es möglich, durch geeignete Wahl der Lamellenreihenfolge einen Text in die Gebäudefassade einzukodieren. Wie bei den oben genannten QR-Codes sollen die Informationen mit Hilfe eines Mobiltelefons mit Kamera aus der Gebäudefassade wiedergewonnen werden können.

1.2 Beiträge

In dieser Arbeit wird ein Webservice entwickelt, der beliebige Ausschnitte der Lamellensequenz an der Fassade dekodiert. In Kapitel 3 werden Enkodierungsverfahren und Dekodierungsverfahren beschrieben und das Problem der Synchronisierung wird diskutiert. Mögliche Lösungsansätze werden erörtert. In Kapitel 4 wird eine wörterbuchbasierte Synchronisierung entwickelt. Die verwendeten Algorithmen und Datenstrukturen werden beschrieben. In Kapitel 5 wird für das entwickelte Dekodierungs- und Synchronisierungsverfahren die Wahrscheinlichkeit einer richtigen Dekodierung geschätzt. Auf Grundlage

Abbildung 1.1: Entwurf [3] der ICT Cubes an der RWTH Aachen.

der Ergebnisse wird der Algorithmus angepasst. Für Sequenzen mit 70 oder mehr Lamellen erreicht das angepasste Verfahren eine korrekte Dekodierung mit einer Wahrscheinlichkeit von mehr als 99%. In Kapitel 6 wird die Architektur des Webservices beschrieben. In Kapitel 7 werden die Google App Engine und die für die Implementierung verwendeten Elemente der Programmiersprache Go beschrieben.

2 Verwendung des Webservices

Die planenden Architekten von kadawittfeldarchitektur [3] in Kollaboration mit dem Institut für Theoretische Informationstechnik der RWTH Aachen haben aus ästhetischen und funktionellen Gründen entschieden, drei verschiedene Lamellentypen [7] für die Fassade zu verwenden. Durch die Anordnung der verschiedenen Lamellentypen soll ein kodierter Text dargestellt werden. Die Lamellen unterscheiden sich in ihrem Profil. Im Nachfolgenden werden diese Lamellen als rechte (r), linke (l) und mittlere (m) Lamellen bezeichnet. Die rechten und linken Lamellen haben als Querschnitt rechtwinklige Trapeze mit rechten Winkeln auf jeweils unterschiedlichen Seiten. Die mittleren Lamellen haben als Querschnitt eine symmetrische Trapezform. Die Querschnitte der Lamellen sind in Abbildung 2.1 dargestellt. Die Lamellen werden vertikal an der Fassade angeordnet. Die zu dekodierende Lamellensequenz muss von der Fassade eingelesen und dem dekodierenden Webservice übermittelt werden. In Abbildung 2.2 ist dargestellt, wie sich aus den Lamellen an der Fassade eine Sequenz ergibt, die aus den Buchstaben m, l und r, besteht.

Die Dekodierung einer Lamellensequenz wird durch einen HTTP GET request vom Webservice angefordert. Dies geschieht über den URI

`http://www.ict-cubes.appspot.com/?slats=rmrlrrrrmmrmllrrmrlrmrrrlrrrrl`

Im Query wird die aus m,l und r bestehende und zu dekodierende Lamellensequenz angegeben.
In der response auf den GET request steht im response body ein JSON string, der den dekodierten Klartext und dessen Länge enthält. Falls die angegebene Lamellensequenz

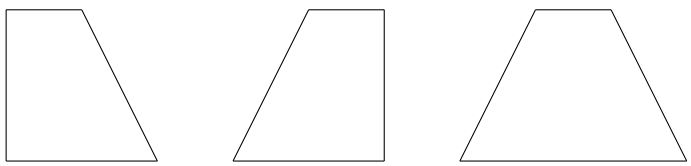

Abbildung 2.1: Querschnittsformen der verschiedenen Lamellentypen.

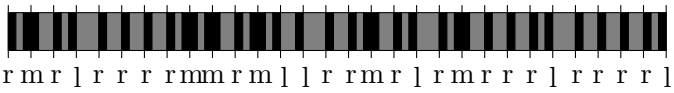

Abbildung 2.2: Ausschnitt aus der Fassade.

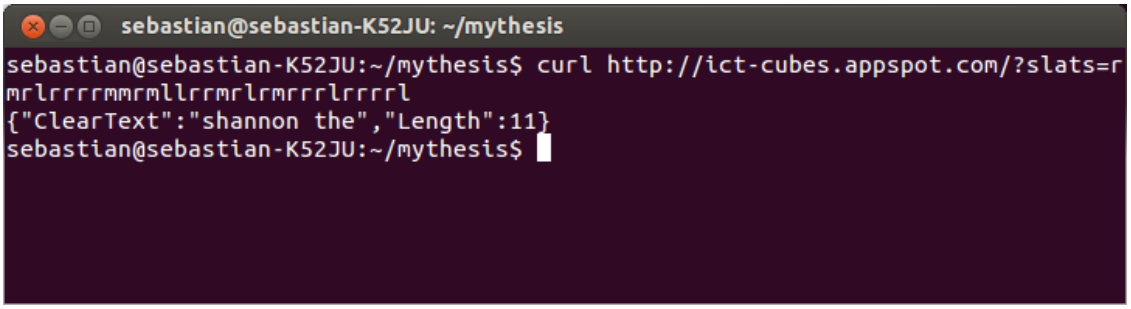

Abbildung 2.3: Beispielaufruf des Webservices mit Hilfe des Programms curl.

zu kurz ist und somit eine Dekodierung nicht möglich ist, antwortet der Webservice mit einem leeren String als dekodierten Klartext.
Abbildung 2.3 zeigt einen Beispielaufruf des Webservices mittels des Programms curl, einem Programm, das das HTTP Protokoll implementiert.
Abbildung 2.4 zeigt den Aufruf des Webservices mittels eines Browsers.
Der Webservice kann auch in Matlab mit dem Befehl urlread verwendet werden. Ein Beispielaufruf ist im Folgenden dargestellt.

```
resp = urlread ...
('http://www.ict-cubes.appspot.com/?slats=rmrlrrrrmmrmllrrmrlrmrrrlrrrrl')
```

In der Variablen resp wird dann der JSON string gespeichert, den der Webservice in seinem response body schickt. Der JSON string kann mit Hilfe von JSONlab [2] geparst werden. Dazu wird beispielsweise durch den folgenden Aufruf in der Variablen cleartext der dekodierte Klartext aus der Antwort des Webservices gespeichert.

Abbildung 2.4: Beispielaufruf des Webservices mittels eines Browsers.

```
cleartext = getfield(loadjson(resp),'ClearText')
```

Der Webservice erfüllt die Anforderungen an einen RESTful Webservice, wie er in Kapitel 6 beschrieben wird.

3 Der ICT Cubes Code

Die Lamellen sollen aus ästhetischen Gründen so aussehen, als seien sie in zufälliger Reihenfolge angebracht. Um ein Aufheizen der Räume zu vermeiden, ohne sie zu stark abzudunkeln, werden 40% linke, 40% rechte und 20% mittlere Lamellen angebracht. Auf Grundlage dieser Vorgaben wurde ein Enkodierungsverfahren [7] entwickelt, das diesen speziellen Anforderungen genügt. Die Codes für dieses Verfahren lassen sich den beiden Tabellen 3.1 und 3.2 entnehmen. In der ersten Kodierungstabelle werden alphanumerische Zeichen eindeutig Bitsequenzen variabler Länge zugeordnet und umgekehrt. Diese Bitsequenzen werden im Folgenden als $\mathcal{F}$-Bitstrings bezeichnet. In der zweiten Kodierungstabelle werden Bitsequenzen variabler Länge eindeutig Lamellentripeln zugeordnet und umgekehrt. Diese Bitsequenzen werden im Folgenden als $\mathcal{G}$-Bitstrings bezeichnet.

3.1 Enkodierungsverfahren

Das Enkodierungsverfahren [7] besteht aus der Hintereinanderausführung zweier Abbildungen. Zuerst wird der Klartext T auf eine Bitsequenz B abgebildet. Diese Abbildung wird im Folgenden mit f bezeichnete. Dazu wird der Klartext geparst und dabei in die alphanumerischen Zeichen aus Tabelle 3.1 zergliedert. Sukzessive werden den alphanumerischen Zeichen die jeweils zugehörigen $\mathcal{F}$-Bitstrings zugeordnet. Sie ergeben konkateniert die Bitsequenz B. Bei dem in der Abbildung f verwendeten Code handelt es sich um eine Abbildung von alphanumerischen Zeichen mit fester Länge auf $\mathcal{F}$-Bitstrings mit varia-

_ : 000	a : 0111	b : 101110
c : 11110	d : 00110	e : 110
f : 10000	g : 010100	h : 11111
i : 0110	j : 00111000100	k : 00111001
l : 10110	m : 01011	n : 1001
o : 1010	p : 001111	q : 00111000101
r : 0010	s : 0100	t : 1110
u : 10001	v : 0011101	w : 101111
x : 001110000	y : 010101	z : 0011100011

Tabelle 3.1: Erste Kodierungstabelle. Der Code wurde entwickelt in [5].

0010 : lll	1101 : llr	00000 : llm
1100 : lrl	1111 : lrr	00011 : lrm
00010 : lml	01101 : lmr	0000111 : lmm
1110 : rll	1001 : rlr	01100 : rlm
1000 : rrl	1011 : rrr	01111 : rrm
01110 : rml	01001 : rmr	000010 : rmm
01000 : mll	01011 : mlr	001101 : mlm
01010 : mrl	1010 : mrr	001100 : mrm
001111 : mml	001110 : mmr	0000110 : mmm

Tabelle 3.2: Zweite Kodierungstabelle. Der Code wurde entwickelt in [7].

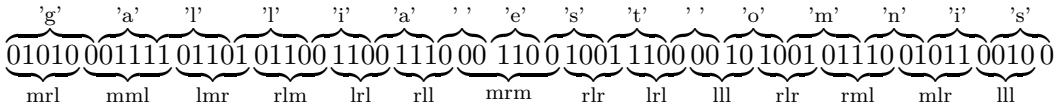

Abbildung 3.1: Beispiel für Enkodierung.

bler Länge und somit um einen *fixed-to-variable length* (f2v) Code. Anschließend wird die Bitsequenz B auf eine Lamellensequenz L abgebildet. Diese Abbildung wird im Folgenden mit g bezeichnet. Die Bitsequenez B wird geparst und dabei in die $\mathcal{G}$-Bitstrings zergliedert. Sukzessive werden den $\mathcal{G}$-Bitstrings die jeweils zugehörigen Lamellentripel aus Tabelle 3.2 zugeordnet. Diese ergeben konkateniert die Lamellensequenz L. Bei dem in der Abbildung g verwendeten Code handelt es sich um eine Abbildung von $\mathcal{G}$-Bitstrings mit variabler Länge auf Lamellentripel mit fester Länge und somit um einen *variable-to-fixed length* (f2v) Code.

3.1.1 Beispiel für die Enkodierung

Abbildung 3.1 zeigt die Enkodierung eines vorgegebenen Textes mittels des Enkodierungsverfahrens. Offensichtlich kann der letzte Buchstabe des Textes mit diesem zweistufigen Enkodierungsverfahren nicht komplett enkodiert werden. Dies hat zur Folge, dass er bei der späteren Dekodierung nicht mehr zurückgewonnen werden kann.

3.2 Dekodierungsverfahren

Die Lamellensequenz wird durch ein zweistufiges Verfahren dekodiert. Zuerst wird die Lamellensequenz L auf eine durch Dekodierung entstandene Bitsequenz $\hat{\mathrm{B}}$ abgebildet. Diese Abbildung wird mit g^{-1} bezeichnet. Dazu wird die Lamellensequenz L geparst und dabei in die Lamellentripeln aus Tabelle 3.2 zergliedert. Sukzessive werden den Lamellentripeln

$$\mathrm{T} \underbrace{\xrightarrow{f}}_{\text{f2v Kodierung}} \mathrm{B} \underbrace{\xrightarrow{g}}_{\text{v2f Kodierung}} \mathrm{L} \underbrace{\xrightarrow{g^{-1}}}_{\text{f2v Kodierung}} \hat{\mathrm{B}} \underbrace{\xrightarrow{f^{-1}}}_{\text{v2f Kodierung}} \hat{\mathrm{T}}$$

Abbildung 3.2: Veranschaulichung der Enkodierung und Dekodierung.

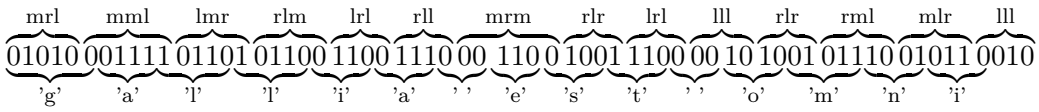

Abbildung 3.3: Beispiel für Dekodierung.

die jeweils zugehörigen $\mathcal{G}$-Bitstrings zugeordnet. Diese ergeben konkateniert die durch Dekodierung entstandene Bitsequenz $\hat{\mathrm{B}}$. Bei dem in Abbildung g^{-1} verwendeten Code handelt es sich um einen f2v Code. Dann wird die Bitsequenz $\hat{\mathrm{B}}$ auf den durch Dekodierung entstandenen Klartext $\hat{\mathrm{T}}$ abgebildet. Diese Abbildung wird mit f^{-1} bezeichnet. Dazu wird die Bitsequenz $\hat{\mathrm{B}}$ geparst und dabei in die $\mathcal{F}$-Bitstrings zergliedert. Sukzessive werden den $\mathcal{F}$-Bitstrings die jeweils zugehörigen alphanumerischen Zeichen aus Tabelle 3.1 zugeordnet. Die jeweils zugehörigen alphanumerischen Zeichen aus Tabelle 3.1 ergeben konkateniert den Klartext $\hat{\mathrm{T}}$. Bei dem in Abbildung f^{-1} verwendeten Code handelt es sich um einen v2f Code. Das Beispiel in Abschnitt 3.1.1 zeigt, dass nicht unbedingt $\hat{\mathrm{B}} = \mathrm{B}$ gilt.

Der gesamte Ablauf von der Enkodierung bis zur Dekodierung ist in Abbildung 3.2 veranschaulicht. Die Einzelbuchstaben des Klartextes T werden auf $\mathcal{F}$-Bitstrings abgebildet. Werden diese $\mathcal{F}$-Bitstrings konkateniert, ergibt sich die Bitsequenz B. Diese lässt sich in $\mathcal{G}$-Bitstrings unterteilen. Die $\mathcal{G}$-Bitstrings entsprechen Lamellentripeln, welche konkateniert die Lamellensequenz L ergeben. Bei der Dekodierung wird die Lamellensequenz L in Lamellentripel unterteilt, die auf $\mathcal{G}$-Bitstrings abgebildet werden. Die $\mathcal{G}$-Bitstrings ergeben konkateniert die aus der Dekodierung entstandene Bitsequenz $\hat{\mathrm{B}}$. Diese lässt sich in $\mathcal{F}$-Bitstrings aufteilen, welche auf alphanumerische Zeichen abgebildet werden. Diese ergeben konkateniert den dekodierten Klartext $\hat{\mathrm{T}}$.

3.2.1 Beispiel für die Dekodierung

Das vorher enkodierte Beispiel wird in Abbildung 3.3 entsprechend dem oben beschriebenen Dekodierungsverfahren dekodiert.

Es zeigt sich, dass der ursprünglich enkodierte Text wiedergewonnen werden kann. Wie erwartet lässt sich jedoch der letzte Buchstabe nicht rekonstruieren, da er bei der Enkodierung in Abbildung 3.1 nicht komplett enkodiert wurde.

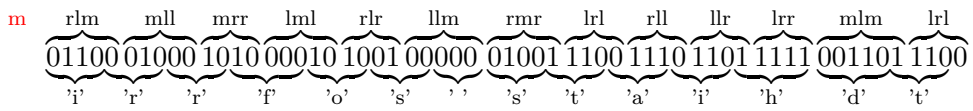

Abbildung 3.4: Beispiel für Dekodierung und Synchronisierungsproblem auf Lamellenebene. Im Unterschied zu Abbildung 3.3 wird mit der zweiten Lamelle mit der Dekodierung begonnen. Die weggelassene erste Lamelle ist rot dargestellt. Es ergibt sich eine andere Bitsequenz und dadurch auch ein anderer Klartext bei der Dekodierung.

3.3 Das Synchronisierungsproblem

Bisher wurde davon ausgegangen, dass die Dekodierung am Anfang der Lamellensequenz begonnen wird. Es ist aber offensichtlich, dass man beim Betrachten der Gebäudefassade den Anfang des enkodierten Textes nicht unbedingt erkennen kann. Unter der Voraussetzung, dass der enkodierte Klartext auf mehrere Seiten des Gebäudes verteilt ist, ist nicht ersichtlich, auf welcher Seite sich der Anfang des enkodierten Klartextes befindet. Das hat zur Folge, dass man unter Umständen an einer Stelle mit der Dekodierung beginnt, die nicht dem Anfang des enkodierten Klartextes entspricht.

3.3.1 Das Synchronisierungsproblem auf Lamellenebene

Wenn man nicht am Anfang des enkodierten Textes mit der Dekodierung beginnt, kann sich bei der Anwendung des oben beschriebenen Dekodierungsverfahrens ein Problem ergeben. In Abbildung 3.4 wird dieselbe Lamellensequenz wie in Abbildung 3.3 dekodiert, wobei die erste Lamelle weggelassen wird. Dabei ergibt sich eine falsche Zeichenfolge auf Klartextebene. Dieses Problem wird im Weiteren als Synchronisierungsproblem auf Lamellenebene bezeichnet.

Es stellt sich die Frage, ob es sich bei der beobachteten Falschdekodierung um ein Problem handelt, das nicht nur bei diesem speziellen Beispiel auftritt, sondern bei jeder Dekodierung. Durch den in Abbildung g^{-1} verwendeten Code werden, wie in Kapitel 3.1 beschrieben, ternäre Sequenzen der Länge drei auf $\mathcal{G}$-Bitstrings abgebildet. Somit steht jede Lamellenfolge der Länge drei für einen bestimmten $\mathcal{G}$-Bitstring. Beim Dekodierungsvorgang wird die zu dekodierende Lamellensequenz von ihrem Anfang ausgehend geparst und es werden jeweils Teilsequenzen der Länge drei gebildet. Wird der Anfang der Lamellensequenz L um eine Lamelle verschoben, so werden durch das Parsen andere Teilsequenzen der Länge drei gebildet. Diese stehen für andere $\mathcal{G}$-Bitstrings. Das führt zu dem beobachteten Fehler. Es handelt sich somit um ein generelles Synchronisierungsproblem.

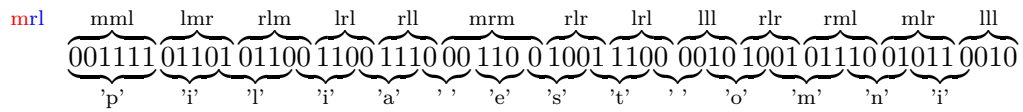

Abbildung 3.5: Beispiel für Dekodierung und Synchronisierungsproblem auf Bitebene. Die blauen Lamellen werden zur Synchronisierung weggelassen. Es ergibt sich nun die richtige Bitsequenz, bis auf fehlende Bits am Anfang. Trotzdem ergibt sich ein falscher Klartext.

3.3.2 Das Synchronisierungsproblem auf Bitebene

Um das Problem zu lösen, werden nun am Anfang der zu dekodierenden Lamellensequenz weitere Lamellen abgeschnitten. Ist die gesamte Anzahl der abgeschnittenen Lamellen ohne Rest durch drei teilbar, dann ergeben sich beim Parsen der so entstandenen Lamellensequenz L' dieselben Teilsequenzen der Länge drei, wie beim Parsen der Lamellensequenz L. In Abbildung 3.5 wird das oben angeführte Beispiel erneut dekodiert. Zwei zusätzliche Lamellen werden abgeschnitten. Da nun insgesamt drei Lamellen fehlen, ist die oben beschriebene Bedingung erfüllt.
Nach Durchführung der Dekodierung zeigt sich wieder eine falsche Zeichenfolge im Klartext. Synchronisierung auf Lamellenebene allein reicht nicht, um den enkodierten Klartext richtig zu dekodieren. Das in diesem Beispiel auftretende Problem wird als Synchronisierungsproblem auf Bitebene bezeichnet. Da vor der Dekodierung der Lamellensequenz L' die richtige Anzahl an Lamellen abgeschnitten wurde, wird der beobachtete Fehler nicht durch die Abbildung g^{-1} verursacht. Der Fehler entsteht bei Anwendung der Abbildung f^{-1} auf die Bitsequenz $\hat{B}$. Die Bitsequenz ist eine Schnittstelle zwischen der Lamellensequenz und dem Klartext. Sie wird einerseits durch die $\mathcal{F}$-Bitstrings und andererseits durch die $\mathcal{G}$-Bitstrings in jeweils unterschiedlich lange Teilbitsequenzen unterteilt. Wird ein Teil der Lamellensequenz L abgeschnitten, so fehlt der entsprechende $\mathcal{G}$-Bitstring in der Bitsequenz $\hat{B}$. Da dieser im Allgemeinen nicht mit einem $\mathcal{F}$-Bitstring übereinstimmt, fehlt nur ein Teil eines $\mathcal{F}$-Bitstrings in der Bitsequenz $\hat{B}$. Beim Parsen der Bitsequenz $\hat{B}$ zur Anwendung der Abbildung f^{-1} ergeben sich dann andere $\mathcal{F}$-Bitstrings als beim Parsen der Bitsequenz B. Diese werden dann auf andere alphanumerische Zeichen abgebildet und es ergibt sich ein falscher Klartext. Um dieses Problem zu lösen, muss der unvollständige $\mathcal{F}$-Bitstring am Anfang der Bitsequenz $\hat{B}$ entfernt werden. Die so entstandene Bitsequenz wird als $\hat{B}'$ bezeichnet. In Abbildung 3.6 wird nun dieses Verfahren angewendet. Die Lamellensequenz wird jetzt richtig dekodiert.
Das Dekodierungsverfahren mit Synchronisierung ist in Abbildung 3.7 veranschaulicht. Die Lamellensequenz L wird durch Abschneiden von Lamellen an ihrem Anfang auf die

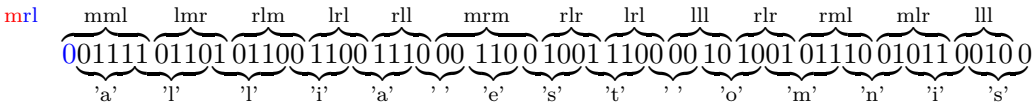

Abbildung 3.6: Beispiel für Dekodierung und Lösung beider Synchronisierungsprobleme. Neben den blauen Lamellen werden auch die blauen Bits zur Synchronisierung weggelassen. Es ergibt sich der richtige Klartext, bis auf fehlende Buchstaben am Anfang.

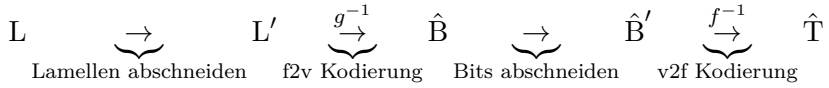

Abbildung 3.7: Veranschaulichung der Enkodierung und Dekodierung.

Lamellensequenz L' abgebildet. Durch Anwendung der Abbildung g^{-1} auf die Lamellensequenz L' wird diese auf die Bitsequenz $\hat{\mathrm{B}}$ abgebildet. Durch Abschneiden von Bits am Anfang der Bitsequenz $\hat{\mathrm{B}}$ wird diese auf die Bitsequenz $\hat{\mathrm{B}}'$ abgebildet. Diese Bitsequenz $\hat{\mathrm{B}}'$ wird durch Anwendung der Abbildung f^{-1} auf den dekodierten Klartext $\hat{\mathrm{T}}$ abgebildet.

3.3.3 Wie findet man den Synchronisierungspunkt?

In den vorausgegangenen Beispielen wird davon ausgegangen, dass der Beginn eines $\mathcal{F}$-Bitstrings innerhalb der Bitsequenz und die Aufteilung der Lamellensequenz in Lamellentripel dem Dekodierer bekannt sind. Ist das der Fall, so kann die richtige Anzahl an Bits bzw. Lamellen vor Anwendung des Dekodierungsverfahrens abgeschnitten werden. In der praktischen Anwendung soll aber ein beliebiger Ausschnitt der Lamellensequenz aus der Fassade dekodiert werden. In diesem Fall ist nicht gewährleistet, dass die oben angeführten Voraussetzungen erfüllt sind. Damit ist auch nicht bekannt, wie viele Lamellen bzw. Bits abgeschnitten werden müssen.

Pilotsymbole In praktischen Übertragungsverfahren der Nachrichtentechnik werden für die Synchronisierung unter Anderem Pilotsymbole verwendet. Dieses Verfahren kann auch auf das oben genannte Synchronisierungsproblem angewendet werden. Dabei wird in bestimmten Abständen vor dem Enkodierungsvorgang im Klartext eine dem Dekodierer bekannte Zeichensequenz eingefügt, die im Klartext nicht vorkommt. Diese Zeichensequenz fungiert dann als Pilotsymbol. Beim Dekodieren wird vom Anfang der Lamellen- und Bitsequenz so viel abgeschnitten, dass das Pilotsymbol im dekodierten Klartext erscheint. Die Sequenz ist dann synchronisiert und die Pilotsymbole müssen aus dem dekodierten Text wieder entfernt werden. Bei diesem Verfahren ergeben sich jedoch folgende

Probleme. Vor dem Enkodieren des Klartextes muss zusätzliche Redundanz in den Text eingefügt werden. Um auch kurze Ausschnitte aus dem enkodierten Klartext dekodieren zu können, müssten sehr viele Pilotsymbole eingefügt werden. Dadurch kann man weniger Text an die Fassade schreiben, da die Pilotsymbole auch Fläche beanspruchen. Daher wird nach einem anderen Verfahren gesucht.

Direkter Textvergleich Falls der gesamte Klartext dem Dekodierer bekannt ist, bietet sich als weitere Möglichkeit an, einen direkten Vergleich zwischen dem dekodierten Textabschnitt und dem Klartext durchzuführen. Ein Nachteil dieses Vorgehens ist, dass das Dekodierungsprogramm Zugriff auf den kompletten Klartext benötigt, und bei Veränderung des Klartextes oder Dekodierung eines komplett anderen Klartextes dieser erneut dem Dekodierer bekannt gegeben werden muss. Nachdem bei diesem Vorgehen der Text bei jeder Überprüfung auf Synchronisierung durchlaufen werden muss, ist es außerdem rechenintensiv.

Wörterbuch Eine weitere mögliche Lösung dieses Problems ergibt sich durch Verwendung eines Wörterbuchs, das alle im Text an der Fassade vorkommenden Wörter enthält. Bei der Dekodierung werden dann einige der dekodierten Wörter einer Lamellensequenz mit den im Wörterbuch vorkommenden Wörtern verglichen. Damit wird sichergestellt, dass Synchronität eingetreten ist. Der Dekodierer benötigt also nur die Codes und das Wörterbuch. Er kann somit universell eingesetzt werden, vorausgesetzt, das Wörterbuch enthält alle Wörter des Klartextes.

Automatische Synchronisierung Bei der Dekodierung kann man beobachten, dass bei längeren Lamellensequenzen eine automatische Synchronisierung auftritt. Das könnte man unter Zuhilfenahme eines Wörterbuchs zur Synchronisierung nutzen. Das Vorgehen ist dann ähnlich zum oben beschriebenen Verfahren. Abbildung 3.5 zeigt ein Beispiel für diese automatische Synchronisierung. Hier ergibt die Dekodierung der Lamellensequenz ab dem Wort „est“ den richtigen Klartext. Dieses Phänomen tritt nur auf, wenn die richtige Anzahl an Lamellen schon abgeschnitten wurde, um Synchronität auf Lamellenebene zu erreichen. Die Synchronität auf Bitebene ist aber noch nicht erreicht. In diesem Fall kann eine Synchronisierung im Verlauf der Dekodierung auftreten. Diese Synchronisierung tritt zwar relativ häufig auf, meist jedoch erst spät in der Lamellensequenz. Dadurch geht viel vom Klartext verloren. Abbildung 3.8 zeigt ein Beispiel, in dem durch diese Art der Synchronisierung ein komplettes Wort im Klartext verloren geht. Die automatische Synchronisierung liefert also nicht zuverlässig den bestmöglichen Synchronisierungspunkt. Deshalb wird dieser Ansatz nicht weiter verfolgt.

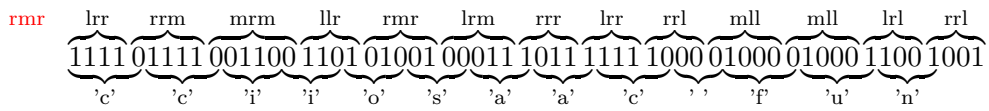

Abbildung 3.8: Beispiel für die automatische Synchronisierung. Würden die ersten drei Lamellen nicht abgeschnitten, so ergäbe sich der Klartext „shannon the fun“ . Neben dem unvollständigen Wort „shannon“ wird auch das Wort „the“ nicht richtig dekodiert.

3.3.4 Algorithmischer Ablauf der Dekodierung

Algorithmus 1 beschreibt den allgemeinen Ablauf der Dekodierung mit Synchronisierung.

Algorithmus 1 Ablauf der Dekodierung mit Synchronisierung

```
length ← Länge der Bitsequenz
for i = 0 to 2 do
  {Synchronisierung auf Lamellenebene}
  L' ← L_i...L_end
  B̂ ← g^-1(L')
  for j = 0 to length − 1 do
    {Synchronisierung auf Bitebene}
    B̂' ← B̂_j...B̂_end
    T̂ ← f^-1(B̂')
    if isSynchronised(T̂) then
      return T̂
    end if
  end for
end for
return leerer String
```

Die Funktion isSynchronised liefert den booleschen Wert true für den Fall der Synchronisierung. Andernfalls liefert sie den booleschen Wert false. Um zu ermitteln ob Synchronisierung eingetreten ist, wird eines der oben angeführten Verfahren angewandt.

4 Wörterbuchbasierte Dekodierung und Synchronisierung

Im Folgenden werden die für einen möglichen Dekodierer zu implementierenden Funktionen beschrieben. Hierzu werden entsprechende Schnittstellen definiert sowie die notwendigen Datenstrukturen und Algorithmen dargestellt. Die Funktionen können in vielen höheren Programmiersprachen, wie beispielsweise Java oder Python implementiert werden. In dieser Arbeit wurde die Programmiersprache Go gewählt, die in Kapitel 7 beschrieben wird. Für die Funktionen wird folgende Notation verwendet: **Funktionsname(Argument) Rückgabewert**.

4.1 Einlesen eines Codes

Die Funktion **getCodebook(nameOfCode) codebook** liest einen Code aus einer statischen Datei ein. Der Name der Datei wird der Funktion als Parameter übergeben. Ihr Rückgabewert wird als codebook bezeichnet. Die Datenstruktur des codebooks ist ein Wörterbuch [9]. Darunter versteht man ein array, dessen Elemente über beliebige Schlüssel anstelle von inkrementellen ganzen Zahlen indiziert werden. In dem codebook entsprechen die Schlüssel des Wörterbuchs den enkodierten Wörtern. Die Elemente des Wörterbuchs entsprechen den dekodierten Wörtern. Wird also der Code für die Abbildung g^{-1} eingelesen, so sind die Lamellentripel die Schlüssel und die $\mathcal{G}$-Bitstrings die Elemente des Wörterbuchs. Wird der Code für die Abbildung f^{-1} eingelesen, so sind die $\mathcal{F}$-Bitstrings die Schlüssel und die Einzelbuchstaben die Elemente des Wörterbuchs.

4.2 Fixed-to-variable length Dekodierung

Die Funktion **f2vDecode(slatSequence) bitSequence** führt eine f2v Dekodierung durch. Die zu dekodierende Lamellensequenz wird an diese Funktion als array slatSequence übergeben. f2vDecode ruft die Funktion getCodebook auf. Als Parameter wird dabei der Funktion getCodebook der Name der Datei, in der der Code für die Abbildung g^{-1} abgespeichert ist, übergeben. Das zurückgegebene codebook wird benötigt, um den

Dekodierungsalgorithmus durchzuführen. Das array der Lamellensequenz wird in Dreierschritten durchlaufen. Für jedes dabei auftretende Lamellentripel wird der entsprechende $\mathcal{G}$-Bitstring aus dem codebook in ein array mit dem Namen bitSequence gespeichert. Falls die Länge der Lamellensequenz nicht ohne Rest durch drei teilbar ist, so wird die entsprechende Anzahl an Lamellen am Ende der Lamellensequenz abgeschnitten. Ist die gesamte Lamellensequenz im array slatSequence durchlaufen, wird das neu entstandene array, das die nunmehr dekodierte Bitsequenz enthält, zurückgegeben. Algorithmus 2 illustriert diese f2v Dekodierung. Die Komplexität dieses Algorithmus ist linear in der Länge der zu dekodierenden Lamellensequenz.

Algorithmus 2 f2v Dekodierung

```
slatSequence ← zu kodierende Sequenz
codebook ← Code
for all wortAufLamellenebene ∈ slatSequence do
  dekodiertesWort ← codebook[wortAufLamellenebene ]
  bitSequence ← bitSequence + dekodiertesWort
end for
```

4.3 Variable-to-fixed length Dekodierung

Die Funktion **v2fDecode(bitSequence, rootPtr) clearText** führt eine v2f Dekodierung einer Bitsequenz durch. Da es sich bei dieser Dekodierung um eine Abbildung von $\mathcal{F}$-Bitstrings variabler Länge auf Einzelbuchstaben handelt, bietet es sich an, das codebook zur Dekodierung in einem Binärbaum zu speichern. Mit einem Binärbaum lässt sich die Bitsequenz effizient parsen. Die Funktion v2fDecode erhält ein array, in dem die Bitsequenz gespeichert ist und eine Referenz auf die Wurzel des Binärbaums als Parameter. Das array der Bitsequenz wird sukzessive durchlaufen. Parallel dazu wird der Baum durchlaufen. Wenn in der Bitsequenz eine binäre Eins vorkommt, rückt man im Binärbaum auf den linken Nachfolgerknoten vor. Stößt man auf eine binäre Null im array, rückt man im Binärbaum auf den rechten Nachfolgerknoten vor. Ist man im Baum an einem Blattknoten angelangt, wird der in diesem Blattknoten befindliche Einzelbuchstabe in das array mit dem Namen clearText gespeichert. Am Ende der Bitsequenz angelangt, wird das array clearText zurückgegeben. Ist der letzte $\mathcal{F}$-Bitstring in der Bitsequenz unvollständig, wird der Baum beim letzten Durchlauf nicht bis zu einem Blattknoten durchlaufen. Es wird in diesem Fall kein Wort mehr zum clearText array hinzugefügt. Dieses Verhalten ist erwünscht, um zu verhindern, dass ein unvollständiger $\mathcal{F}$-Bitstring zu einem falschen Wort dekodiert wird. Der Algorithmus 3 illustriert diese v2f Dekodierung. Die Komplexität dieses Algorithmus ist linear in der Länge der zu dekodierenden

Bitsequenz.

Algorithmus 3 v2f Dekodierung

```
bitSequence ← zu dekodierende Sequenz
rootPtr ← Zeiger auf codebook in Baumform
for all F − Bitstring ∈ bitSequence do
  durchlaufe den Baum gemäß F − Bitstring bis zu einem Blattknoten
  dekodiertesWort ← Inhalt des Blattknotens
  clearText ← clearText + dekodiertesWort
end for
```

4.4 Baumgenerierung

Die Funktion **generateTree() rootPtr** erstellt den Binärbaum, in dem der Code für die Abbildung f^{-1} für die v2f Dekodierung gespeichert ist. Sie gibt eine Referenz auf die Wurzel eines Binärbaums zurück. In der Funktion wird der Code für die Abbildung f^{-1} für die v2f Dekodierung aus einer statischen Datei eingelesen. Dazu wird die Funktion getCodebook mit dem Namen der Datei als Parameter aufgerufen, in dem der benötigte Code abgespeichert ist. Die Anzahl der Blattknoten im Binärbaum entspricht der Anzahl der Paare von $\mathcal{F}$-Bitstrings und Einzelbuchstaben aus dem Code. Um den Baum zu generieren, wird für jedes dieser Paare von der Wurzel aus beginnend der Baum durchlaufen. Parallel dazu wird der entsprechende $\mathcal{F}$-Bitstring durchlaufen. Wenn in dem $\mathcal{F}$-Bitstring eine binäre Eins vorkommt, rückt man im Binärbaum auf den linken Nachfolgerknoten vor. Stößt man auf eine binäre Null im $\mathcal{F}$-Bitstring, rückt man im Binärbaum auf den rechten Nachfolgerknoten vor. Existiert der Knoten auf den man vorrücken will noch nicht, so wird er als rechter bzw. linker Nachfolgerknoten angehängt. Am Ende eines Bitstrings wird im Baum in den entsprechenden Blattknoten der zugehörige Klartextbuchstabe, gespeichert. Sind alle Paare von $\mathcal{F}$-Bitstrings und Einzelbuchstaben im Baum gespeichert, so wird eine Referenz auf die Wurzel des Baumes zurückgegeben.

4.5 Einlesen des Wörterbuchs

Die Funktion **getDict(nameOfDict) dictionary** liest die Wörter des Wörterbuchs zur Synchronisierung aus einer statischen Datei in ein array ein. Der Name der Datei wird der Funktion als Parameter nameOfDict übergeben. Der Rückgabewert ist ein array von Wörtern mit dem Namen dictionary.

4.6 Überprüfung der Synchronisierungsbedingung

In der Funktion **isSynchronized(clearText) synchronized** wird überprüft, ob das erste Wort des clearText arrays mit einem Wort aus dem Wörterbuch übereinstimmt. So wird überprüft, ob die richtige Anzahl an Lamellen bzw. Bits von den jeweiligen Sequenzen vor der jeweiligen Dekodierung abgeschnitten wurde. An diese Funktion wird das array clearText als Parameter übergeben. Es enthält den dekodierten Klartext. Die Funktion getDict liefert ein Wörterbuch als array. In dem Wörterbuch sind die im Klartext vorkommenden Wörter enthalten. Dieses Wörterbuch wird für die Synchronisierung benötigt. In der Funktion isSynchronized wird das erste Wort des clearText arrays mit allen Wörtern aus dem Wörterbuch verglichen. Falls es keine Übereinstimmung gibt, wurde nicht die richtige Anzahl an Lamellen bzw. Bits vor der jeweiligen Dekodierung abgeschnitten. Die Funktion liefert dann über ihren booleschen Rückgabewert synchronized zurück, dass es keine Übereinstimmung gibt. Falls synchronized den Wert true hat, besteht Synchronität, ansonsten hat synchronized den Wert false.

4.7 Ablaufsteuerung der Dekodierung

Die Funktion **decodeWithDict(slatSequence) clearText** dient zur vollständigen Dekodierung der Lamellensequenz. Sie steuert den Ablauf der Dekodierung und der Synchronisierung. Die Lamellensequenz wird an die Funktion als array slatSequence übergeben. Sie liefert das array clearText zurück. Dann wird die Funktion generateTree aufgerufen und so der Baum für die v2f Dekodierung erzeugt. Um sicherzustellen, dass am Anfang der Lamellensequenz die richtige Anzahl an Lamellen abgeschnitten wird, wird der gesamte Dekodierungsalgorithmus mit verschiedenen Teillamellensequenzen so lange konsekutiv durchgeführt, bis die Funktion isSynchronized den booleschen Wert true zurückliefert. Um die Teillamellensequenzen zu bestimmen, werden am Anfang der Lamellensequenz null bis maximal zwei Lamellen abgeschnitten. Die Teillamellensequenzen werden dann mit Hilfe der Funktion f2vDecode dekodiert. Um Synchronisierung auf Bitsequenzebene zu erreichen, werden anschließend Teilbitsequenzen erzeugt und diese konsekutiv mit Hilfe der Funktion v2fDecode dekodiert, bis die Funktion isSynchronized den booleschen Wert true zurückliefert. Die Teilbitsequenzen werden erzeugt, indem sukzessive Bits vom Anfang der zu dekodierenden Bitsequenz abgeschnitten werden. Algorithmus 1 beschreibt die Ablaufsteuerung.

5 Der Webservice im Test

Um zu schätzen, mit welcher Wahrscheinlichkeit der Webservice eine übermittelte Lamellensequenz richtig dekodiert, wurde ein Programm zum Testen des Webservices in Matlab implementiert.

5.1 Funktionsweise des Testprogramms

Das Testprogramm sendet Anfragen an den Webservice und überprüft die vom Webservice gegebenen Antworten auf Richtigkeit. Dazu müssen die Anfragen zuerst generiert werden. Es wird aus der gesamten Lamellensequenz ein zufälliger Startpunkt für eine Anfrage gewählt. Von diesem Startpunkt aus wird eine Anfrage einer vorgegebenen Länge generiert. Diese Anfrage wird dann an den Webservice geschickt. Die Antwort des Webservices wird auf Richtigkeit überprüft. Dazu wird die zu erwartende Antwort auf die Anfrage vom Testprogramm berechnet. Da die komplette Lamellensequenz dem Testprogramm bekannt ist, ist auch die Aufteilung der Lamellensequenz in Lamellentripel bekannt. Außerdem sind dem Testprogramm die komplette Bitsequenz und die Codes bekannt. Damit ist die Aufteilung der Bitsequenz in $\mathcal{F}$- und $\mathcal{G}$-Bitstrings bekannt. Es lässt sich aus dem Startindex der Anfragelamellensequenz der Index des nächstfolgenden Lamellentripels in der gesamten Lamellensequenz berechnen. Daraus wird der Startindex des zugehörigen $\mathcal{G}$-Bitstrings in der gesamten Bitsequenz berechnet. Dann wird der Startindex des nächstfolgenden $\mathcal{F}$-Bitstrings bestimmt. Dieser entspricht dem Startindex der korrekten Antwort. Das Ende der korrekten Antwort wird analog bestimmt. Eine Antwort wird als richtig gewertet, wenn sie mit der vom Testprogramm berechneten Antwort übereinstimmt. Es wird eine vorgegebene Anzahl an Anfragen mit fester Länge der Lamellensequenz an den Webservice gesendet. Dann wird die relative Häufigkeit einer richtigen Antwort als Schätzer für die Wahrscheinlichkeit einer richtigen Dekodierung bei dieser Länge der Lamellensequenz verwendet.

5.2 Auswertung der Ergebnisse

Um die Wahrscheinlichkeit einer richtigen Dekodierung zu schätzen, werden je 1000 Anfragen der Längen 10,20,...,100 an den Webservice geschickt und die jeweiligen Antworten ausgewertet. Zusätzlich werden die 95%-Konfidenzintervalle für jeden Punktschätzer bestimmt.

5.2.1 Auswertung des ersten Algorithmus

Bei Verwendung des in Kapitel 4 beschriebenen Algorithmus ergibt sich das in Abbildung 5.1 als Testlauf 1 bezeichnete Ergebnis. Bei diesem Algorithmus wird die in Abschnitt 4.6 beschriebene Funktion zur Überprüfung der Synchronisierung benutzt. Dabei wird das erste dekodierte Wort mit dem kompletten Wörterbuch verglichen. Für kurze Anfragen ist die Wahrscheinlichkeit einer richtigen Dekodierung sehr gering. Aber es ergibt sich auch für lange Anfragen eine Wahrscheinlichkeit für eine richtige Dekodierung von nur ungefähr 20%. Lange Anfragen sind relevant, da der Nutzer in den meisten Fällen an sinnvollen Sätzen oder längeren Textpassagen Interesse hat.

5.2.2 Verkleinerung des Wörterbuchs

Bei den meisten Fehlern, die bei der Dekodierung auftreten, wird während der Synchronisierung ein Wort gefunden, das im Wörterbuch vorkommt, obwohl noch keine Synchronisierung eingetreten ist. Das Dekodierungsergebnis entspricht dann nicht dem Klartext. Um die Wahrscheinlichkeit zu verringern, dass ein solches Wort gefunden wird, wird der Umfang des Wörterbuchs verkleinert. Wörter, die nur aus einem Buchstaben bestehen, sind in diesem neuen Wörterbuch nicht mehr enthalten. Dadurch werden aber auch Anfragen falsch dekodiert, die korrekt dekodiert mit einem Wort der Länge eins beginnen. Es zeigt sich, dass trotzdem die Wahrscheinlichkeit einer richtigen Dekodierung mit dem kleinerem Wörterbuch steigt. Dieser Sachverhalt ist in Abbildung 5.1 als Testlauf 2 dargestellt. Er ergibt sich mit der in Kapitel 4.6 beschriebenen Synchronisierungsbedingung und dem neuen Wörterbuch. Die Wahrscheinlichkeit für eine richtige Dekodierung ist gestiegen.

5.2.3 Neue Synchronisierungsbedingung

Ein noch besseres Ergebnis lässt sich durch Verwendung einer geänderten Synchronisierungsbedingung erzielen. Dabei wird nicht nur das erste Wort der dekodierten Sequenz mit dem Wörterbuch verglichen, falls dieses erste Wort kurz ist. Zusätzlich wird jetzt überprüft, ob auch das zweite Wort übereinstimmt, falls die Sequenz lang genug ist.

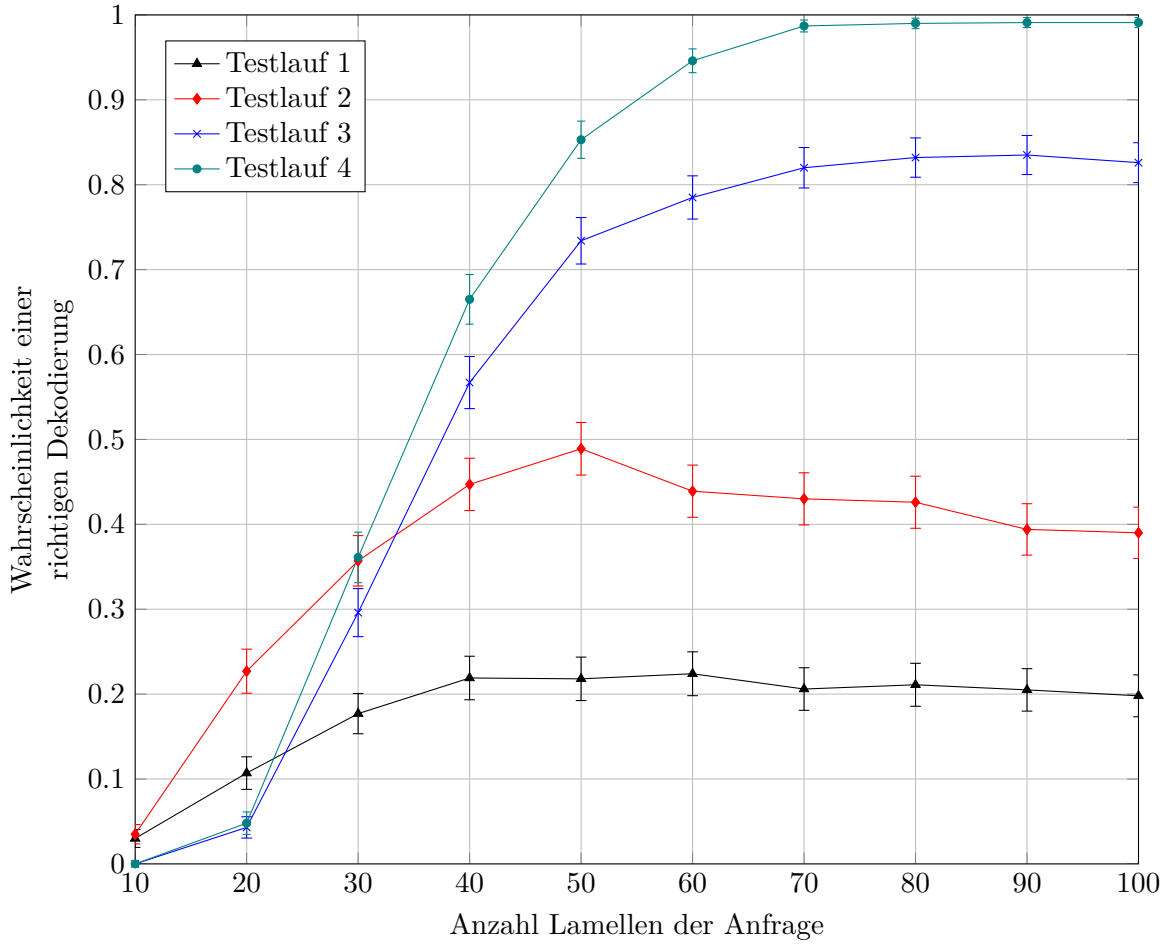

Abbildung 5.1: Ergebnisse der verschiedenen Testläufe. Jeder Punktschätzer entspricht der relativen Häufigkeit einer korrekten Dekodierung bei 1000 Anfragen an den Webservice. Bei Testlauf 1 dient als Synchronisierungsbedingung die Übereinstimmung des ersten Wortes der dekodierten Lamellensequenz mit einem Wort aus dem kompletten Wörterbuch. Testlauf 2 zeigt die Wahrscheinlichkeit einer richtigen Dekodierung, wenn Wörter der Länge eins aus dem Wörterbuch gestrichen sind. Bei Testlauf 3 dient als Synchronisierungsbedingung die Übereinstimmung des ersten und zweiten Wortes mit einem Wort aus dem reduzierten Wörterbuch. Hat das zweite Wort die Länge eins, so wird stattdessen das dritte Wort auf Übereinstimmung überprüft. Bei Testlauf 4 werden auch dekodierte Lamellensequenzen als richtig dekodiert gewertet, wenn ein Wort der Länge eins an ihrem Anfang fehlt oder ein zusätzliches Wort an ihrem Anfang steht.

Ist die Sequenz zu kurz, wird angenommen, dass keine Synchronisierung eingetreten ist. Außerdem wird, falls das zweite Wort nur die Länge eins hat, das dritte Wort anstelle vom zweiten Wort auf Übereinstimmung mit dem Wörterbuch überprüft. Dieser Algorithmus wird zusammen mit dem oben beschriebenen neuen Wörterbuch verwendet. Aus Abbildung 5.1 lässt sich die neue höhere Wahrscheinlichkeit einer richtigen Dekodierung ablesen. Sie wird als Testlauf 3 bezeichnet.

5.2.4 Anpassung der Fehlerbewertung

Bei der in Kapitel 5.2.3 beschriebenen Synchronisierungsstrategie lassen sich die meisten auftretenden Fehler zwei Fehlerarten zuordnen. Bei der ersten Fehlerart beginnt die richtig dekodierte Anfrage mit einem Wort der Länge eins. Diese Wörter kommen nicht im Wörterbuch vor und werden dementsprechend am Anfang eines Wortes nicht als richtig erkannt. Da bei langen Anfragesequenzen der dekodierte Klartext aber aus mehreren Wörtern besteht, fällt es nicht ins Gewicht, wenn ein Wort der Länge eins am Anfang des Textes fehlt. Deshalb werden im Folgenden diese Fehlerfälle vom Testprogramm als richtige Dekodierung gewertet. Bei der zweiten Fehlerart wird zwar die Synchronisierung richtig erkannt, aber trotzdem ein im Wörterbuch vorkommendes Wort in der eigentlich abzuschneidenden Sequenz gefunden. Bei so dekodierten Sequenzen steht also ein Wort vor dem eigentlichen Klartext, das dort nicht hingehört. Auch das stört bei langen Anfragesequenzen meist nicht, weil es in der Regel kurz ist. Deshalb werden vom Testprogramm nun auch diese Klartextsequenzen als richtig gewertet. Abbildung 5.1 zeigt die Ergebnisse bei dieser Auswertung der Webserviceantworten in Testlauf 4. Die Fehlerwahrscheinlichkeit ist kleiner als 1%.

5.3 Statistische Methoden

Ziel ist es, einen Schätzer für die Wahrscheinlichkeit einer richtigen Dekodierung zu finden [15]. Das Zufallsexperiment, eine Anfrage an den Webservice zu senden und die Antwort als falsch oder richtig zu werten, ist ein Bernoulli Experiment. Es gilt $P(\text{richtig dekodiert}) = \theta$ und $P(\text{falsch dekodiert}) = 1 - \theta$. Es werden N Anfragen an den Webservice gesendet und die Antworten auf Richtigkeit überprüft. Das entspricht N stochastisch unabhängigen Bernoulli Experimenten. Die Zufallsvariable X zählt die Anzahl der richtigen Dekodierungen und kann die Werte $\{0, 1, \ldots N\}$ annehmen. Da eine einzelne Anfrage, wie oben beschrieben, ein Bernoulli Experiment ist, ist die Zufallsvariable X binomialverteilt, also

$$Pr\{X = x\} = B_{N,\theta}(X) = \binom{N}{x}\theta^x(1-\theta)^{N-x}, x \in \{0, 1, \ldots, N\}. \tag{5.1}$$

Es ist der Parameter θ der Binomialverteilung zu schätzen. Um den Schätzer zu finden, wird im Folgenden das maximum likelihood Prinzip verwendet. Es maximiert die Wahrscheinlichkeit dafür, dass die Zufallsvariable den beobachteten Wert annimmt. Die likelihood Funktion $L(x;\hat{\theta})$ ist die Zähldichte der Zufallsvariablen, also

$$L(x;\hat{\theta}) = \binom{N}{x}\hat{\theta}^x(1-\theta)^{N-x}. \tag{5.2}$$

Die log-likelihood Funktion, mit der sich im Folgenden leichter rechnen lässt, ist

$$\log L(x;\hat{\theta}) = \log\binom{N}{x} + x\log\hat{\theta} + (N-x)\log(1-\hat{\theta}). \tag{5.3}$$

Den maximum likelihood Schätzer θ_{ML} erhält man als das $\hat{\theta}$, das die log likelihood Funktion maximiert, also

$$\theta_{\mathrm{ML}} = \underset{\hat{\theta}}{\operatorname{argmax}}\{\log L(x;\hat{\theta})\} \tag{5.4}$$

$$\Rightarrow \frac{\mathrm{d}}{\mathrm{d}\hat{\theta}}\log L(x;\hat{\theta})|_{\hat{\theta}=\theta_{\mathrm{ML}}} = 0 \tag{5.5}$$

$$\Rightarrow \frac{x}{\theta_{\mathrm{ML}}} - (N-x)\frac{1}{1-\theta_{\mathrm{ML}}} = 0 \tag{5.6}$$

$$\Rightarrow \theta_{\mathrm{ML}}(x,N) = \frac{x}{N}. \tag{5.7}$$

Es wird überprüft, ob θ_{ML} ein Maximum ist. Dazu wird die zweite Ableitung gebildet.

$$\frac{\mathrm{d}^2}{\mathrm{d}\hat{\theta}^2}\log L(x;\hat{\theta})|_{\hat{\theta}=\theta_{\mathrm{ML}}} = -\frac{N^2}{x} - \frac{N^2}{N-x} < 0, x \in \{0,1,\ldots,N\} \tag{5.8}$$

Es handelt sich somit um ein Maximum. Der maximum likelihood Schätzer für die Wahrscheinlichkeit, dass eine Anfrage richtig dekodiert wurde, ist somit bei N Anfragen mit $\theta_{\mathrm{ML}}(x,N) = \frac{x}{N}$ zu berechnen. Der Erwartungswert der binomialverteilten Zufallsvariablen X ist $\mathrm{E}(X) = N\theta$. Der Erwartungswert des maximum likelihood Schätzers ist damit $\mathrm{E}(\theta_{\mathrm{ML}}(X,N)) = \frac{\mathrm{E}(X)}{N} = \frac{N\theta}{N} = \theta$. Es handelt sich also um einen erwartungstreuen Schätzer. Die Varianz der Zufallsvariablen X ist $\mathrm{Var}(X) = N\theta(1-\theta)$. Somit ergibt sich für die Varianz des maximum likelihood Schätzers $\mathrm{Var}(\theta_{\mathrm{ML}}(X,N) = \mathrm{Var}(\frac{X}{N}) = \frac{1}{N^2}\mathrm{Var}(X) = \frac{\theta(1-\theta)}{N}$. Diese Varianz entspricht, da es sich um einen erwartungstreuen Schätzer handelt, der mittleren quadratischen Abweichung des Schätzers. Im Testprogramm wird der maximum likelihood Schätzer als Schätzer für die Wahrscheinlichkeit einer richtigen De-

kodierung verwendet. Das Konfidenzintervall [6] des binomialverteilten Schätzers zum Niveau $1 - \alpha$ lässt sich annähern durch

$$|\theta - \theta_{\mathrm{ML}}| \leq \tau_{1-\alpha/2} \sqrt{\frac{\theta_{\mathrm{ML}}(1 - \theta_{\mathrm{ML}})}{N}}. \tag{5.9}$$

Im Falle eines 95%-Konfidenzintervalls gilt $\tau_{1-\alpha/2} = 1.96$. Für eine Anfragelänge von 100 Lamellen und $N = 1000$ Anfragen erhalten wir $\theta_{\mathrm{ML}} = 0.99$. Dies entspricht einem 0.95%-Konfidenzintervall von

$$|\theta - \theta_{\mathrm{ML}}| \leq 1.96 \sqrt{\frac{0.99(1 - 0.99)}{1000}} = 0.0061669\ldots \tag{5.10}$$

Der wahre Parameter θ liegt also mit einer Wahrscheinlichkeit von 95% in dem Intervall $[0, 983833, 0.996167]$.

6 RESTful Webservices

6.1 Webservice-Architekturen

Webservices können auf verschiedene Art und Weise realisiert werden. Eine häufig verwendete Architektur ist *Representational State Transfer konforme HTTP-Nutzung* (RESTful HTTP) [14]. Für die Realisierung des Webservices zur Dekodierung soll die RESTful HTTP Architektur verwendet werden.

6.2 Entwicklung des Architekturstils REST

Der Begriff REST wurde von Roy Fielding in seiner im Jahr 2000 veröffentlichten Dissertation [11] eingeführt. Roy Fielding war auch an der Entwicklung grundlegender Standards des WWW wie HTTP und URI beteiligt. Das frühe Web [14] bestand im Wesentlichen aus Dokumenten in einem standardisierten Format, die eindeutig identifiziert werden konnten. Die Dokumente enthielten gegenseitige Verweise und es stand mit HTTP ein einfaches Protokoll zur Verfügung, um Dokumente zu übertragen. Mit der Fortentwicklung wurde das Web mit Inhalten versehen, die auf Datenbanken oder Berechnungen beruhten und nicht mehr statisch waren. Es stellte sich die Frage nach einer grundlegenden Theorie, mit der man die unterschiedlichen Dienste des Web mit klaren Begriffen beschreiben kann und so eine formale Architektur erstellen kann. Fielding entwickelte ein Konzept, nach dem statische und dynamische Inhalte ein einheitliches Informationssystem [14] bilden. Durch Abstraktion der konkreten Architektur HTTP entstand so der Architekturstil REST. Die HTTP Architektur ist die einzige praxisrelevante Ausprägung dieses Architekturstils.

6.3 Beschreibung des Architekturstils REST und seine Ausprägung in RESTful HTTP

Fielding beschreibt in seiner Dissertation die Anforderungen an den REST Architekturstil. Sie werden im Einzelnen in den nachfolgenden Unterkapiteln beschrieben. Zudem wird beschrieben, wie HTTP die Forderungen des REST Architekturstils erfüllt [14].

6.3.1 Server-Client Architekturstil

Die erste Bedingung ist es, einen Server-Client Architekturstil zu verwenden. Dabei stellt ein Server einen Dienst zu Verfügung. Ein Client, der den Dienst nutzen will, sendet eine Anfrage an den Server und löst so beim Server entweder eine Ablehnung oder Bearbeitung der Anfrage aus. Durch diese Aufteilung der Aufgaben vereinfacht sich das Serverprogramm und wird skalierbarer, indem die Benutzerschnittstelle zur Aufgabe des Clientprogramms gemacht wird. Dadurch können sich Client und Server auch unabhängig voneinander weiterentwickeln.
Das HTTP Protokoll ist gemäß seiner Spezifikation [10] ein request/response Protokoll. Die Forderung nach einem Client-Server Architekturstil ist somit erfüllt.

6.3.2 Zustandslosigkeit

Die zweite Anforderung ist die der Zustandslosigkeit. Der Server darf keine Information über den Zustand der Interaktion zwischen Server und Client speichern. Dieser Zustand muss komplett vom Client gespeichert werden. Alle Informationen über den Zustand, die der Server für die Bearbeitung einer Anfrage benötigt, müssen in dieser Anfrage enthalten sein. Dadurch ist das System weniger fehleranfällig. Außerdem ist der Server skalierbarer und leichter realisierbar, da er keine Informationen über den Zustand zwischen zwei Anfragen speichern muss. Es ergeben sich aber auch Nachteile aus der Zustandslosigkeit. Da die Informationen über den Zustand bei jeder Anfrage übermittelt werden müssen, ergibt sich mehr Verkehr als bei einer Speicherung des Zustands beim Server.
In der Spezifikation des HTTP Protokolls wird dieses als zustandslos bezeichnet. Der REST Architekturstil fordert das Einhalten der Zustandslosigkeit. Es liegt aber am Entwickler, sich an diese Forderung zu halten.

6.3.3 Caching

Um die Netzwerkeffizienz zu erhöhen, soll der Client Anfrageergebnisse zwischenspeichern können, falls die Antwort vom Server dazu ausgezeichnet wurde (Caching). Die zwischengespeicherte Antwort wird verwendet, wenn sich Anfragen wiederholen.
Das HTTP Protokoll unterstützt auch Caching mit dem Ziel, in vielen Fällen Anfragen nicht senden zu müssen oder keine vollständigen Antworten senden zu müssen.

6.3.4 Einheitliche Schnittstelle

Die zentrale Anforderung Fieldings an den REST Architekturstil ist die einer einheitlichen Schnittstelle. Dadurch vereinfacht sich die Gesamtarchitektur des Systems. Außerdem er-

leichtert sie das Verständnis dafür, was bei Interaktionen abläuft. Dadurch kann das System problemlos weiterentwickelt werden, solange die Schnittstelle gleich bleibt. Die einheitliche Schnittstelle beeinflusst die Effizienz jedoch nachteilig, da der Verkehr zunimmt. Die einheitliche Schnittstelle muss die folgenden Kriterien erfüllen. Die Ressourcen müssen identifizierbar sein. Mit einer Resource ist jede Art von Information gemeint, die benannt werden kann. Dazu gehören beispielsweise Dokumente und Bilder aber auch nicht-statische Informationen, wie eine Wettervorhersage zum momentanen Zeitpunkt. In REST werden zur Identifizierung von Ressourcen resource identifiers benutzt. Jede Ressource hat mindestens einen solchen Bezeichner. Ein resource identifier kann aber nur auf eine Resource verweisen. Aktionen werden in REST auf Repräsentationen von Ressourcen durchgeführt. Diese Repräsentationen sind Darstellungen der Ressourcen in verschiedenen Formaten. Man kann Ressourcen also als Zusammenschluss von Repräsentationen und einem Bezeichner interpretieren. Auf welcher Repräsentation eine Aktion durchgeführt werden soll, wird durch Metadaten bestimmt. Außerdem ergibt sich der Zustand der Interaktion zwischen Server und Client aus den Verweisen, die der Server dem Client beim Zugriff auf eine Resource liefert. Fielding bezeichnet das als „hypermedia as the engine of application state“ [10].

In der HTTP Spezifikation ist auch eine einheitliche Schnittstelle zwischen Client und Server definiert. Sie besteht aus acht Methoden [14]. Die Kommunikation zwischen Server und Client läuft komplett über diese Methoden ab. Der Client stellt Anfragen und der Server beantwortet sie. In dem in dieser Arbeit erstellten Webservice wird von diesen Methoden nur die GET Methode verwendet. Durch diese Methode wird die durch einen Bezeichner spezifizierte Information vom Server geholt. Die Bezeichner, die im WWW verwendet werden, sind die URIs. Zu den verschiedenen Formaten, in denen die Repräsentationen der Ressourcen dargestellt werden können, gehört unter anderem das HTML Format. Dieses ist das übliche Format für Webseiten. Formate, die von Webservices verwendet werden, sind das XML Format und das JSON Format. Eine HTTP Nachricht besteht immer aus einem Header und einem optionalen Body. Im Header sind Metadaten enthalten, in denen unter anderem bestimmt wird, auf welcher Repräsentation die HTTP Methode durchgeführt werden soll. Die HTML Seiten, die einen Großteil des WWW ausmachen, funktionieren so, dass der Zustand der Interaktion zwischen Server und Client in den Verweisen gespeichert ist, die der Server dem Client schickt. Diese Verweise nennt man Links. Das sind URIs von Webseiten, die der Server dem Client bei seiner GET Anfrage geschickt hat. Der Server kann so steuern, auf welche Seiten der Nutzer von der momentan betrachteten aus wechseln kann. So wird der Zustand der Interaktion implizit gespeichert. Auch Webservices können so implementiert werden.

6.3.5 Schichten mit klaren Schnittstellen

Eine weitere Verbesserung des Architekturstils kann erreicht werden, indem das System in Schichten aufgebaut wird, wobei die Schichten durch klare Schnittstellen gekennzeichnet sind.
Das HTTP Protokoll reiht sich ein in die Protokolle des Internets, die sich den verschiedenen Schichten des Internetprotokollstapels zuordnen lassen.

6.3.6 Code-On-Demand Prinzip

Die letzte Eigenschaft ist das Code-On-Demand-Prinzip. Dies beinhaltet, dass die Funktionalität von Clients durch Downloads erweitert wird. Diese Bedingung ist optional.
Auch die Forderung nach dem Code-On-Demand-Prinzip ist mit den Webbrowsern erfüllt, da beispielsweise durch Plugins bzw. Downloads ihre Funktionalität je nach Bedarf erweitert werden kann.

7 Umsetzung in Go

7.1 Google App Engine

Um einen Webservice bereitzustellen, ist ein physikalischer Server notwendig. Für den ICT Decode Webservice wird die App Engine von Google verwendet [4]. Die Google App Engine bietet die Möglichkeit, eine Webanwendung zu hosten, ohne die dafür nötige Infrastruktur selbst zu besitzen und zu warten. Die Anwendung wird vom Benutzer hochgeladen und Google hostet und skaliert sie. Somit kann die zuverlässige und leistungsstarke Infrastruktur von Google für die eigene Webanwendung verwendet werden. Dieses Angebot ist kostenlos für Speicherplatz bis zu 500 MB und ungefähr 5 Mio. Seitenaufrufen pro Monat. Die Google App Engine eignet sich somit zum Hosten des in dieser Arbeit erstellten Webservices. Will man die Google App Engine zum Hosten seiner Webanwendung nutzen, hat man die Wahl zwischen verschiedenen Programmiersprachen. Die Programmiersprache, die zur Realisierung des ICT Decode Webservices verwendet wird, sollte vor allem die Möglichkeit bieten, die zur Dekodierung nötigen Algorithmen einfach zu implementieren. Die Programmiersprache Go bietet diese Möglichkeit, und sie kann zur Implementierung einer Webanwendung auf der Google App Engine verwendet werden.

7.2 Die Programmiersprache Go

Go [13] ist eine von Google als open source Projekt entwickelte Programmiersprache. Die Syntax von Go ist der Syntax von C sehr ähnlich. Die Programmiersprache unterstützt prozedurale, objektorientierte und parallele Programmierung. Im Folgenden werden die für den ICT Decode Webservice wichtigen Bestandteile der Programmiersprache Go beschrieben.

7.2.1 Go strings

Ein Go string ist eine beliebige, nicht veränderbare Abfolge von Bytes. Die Bytes repräsentieren in Go normalerweise einen in UTF-8 kodierten Unicode Text. Unicode ist ein internationaler Standard, um Schriftzeichen aus unterschiedlichen Sprachen zu kodieren.

Jedem Unicode Schriftzeichen wird eine eindeutige Hexadezimalzahl zugeordnet, die als code point bezeichnet wird. Es existieren unterschiedliche Unicode Transformations Formate, zum Beispiel UTF-8 und UTF-16. Die UTFs implementieren den Unicode Standard, indem sie die code points auf Folgen von Bytes abbilden. Go benutzt die UTF-8 Kodierung für strings. Auch für XML und JSON Dateien wird diese Standardkodierung verwendet. Go strings unterscheiden sich grundlegend von strings anderer Programmiersprachen wie zum Beispiel Java, C++ oder Python. Während strings in den letzteren Programmiersprachen Folgen von Zeichen mit fester Byteanzahl sind, bestehen Go strings aus Zeichen unterschiedlicher Byteanzahl. Auf Zeichen des Go strings kann somit nicht durch direkte Indizierung zugegriffen werden, sondern nur auf die einzelnen Bytes. Eine Ausnahme stellen strings dar, die nur Zeichen enthalten, die auch im ASCII Code vorkommen, da in UTF-8 diese Zeichen nur durch ein Byte kodiert sind. Dies erscheint zunächst als Nachteil. Go bietet aber die Möglichkeit, mit Hilfe der unten angegebenen for Schleife mit range zeichenweise über strings zu iterieren. Range liefert die beiden Rückgabewerte key und val. Im Rückgabewert key steht der Index des Zeichens des Strings beim momentanen Durchlauf, im Rückgabewert val steht das momentane Zeichen.

```
for key,val = range myString
```

Außerdem können durch slicing Teilstrings erzeugt werden. Das folgende Codebeispiel zeigt eine Implementierung dazu.

```
meinstring := "das ist ein test"
fmt.Printf(meinstring[1:5])
```

Mit der Printf Funktion erfolgt folgende Ausgabe.

```
as i
```

Dabei wird ein zusammenhängender Teilstring aus dem gesamten string selektiert. Die Bytes des gesamten strings sind mit einem Index von 0 bis n durchnummeriert. Will man einen Teilstring, der bei Index m beginnt und bei Index l endet, ausschneiden, so werden die Indizes in eckigen klammern wie folgt angegeben.

```
[m:l+1]
```

Enthält der zu slicende string Zeichen, die durch mehr als ein Byte kodiert sind, dürfen die Bytes dieses Zeichens nicht durch das slicing auf unterschiedliche Teilstrings verteilt werden. Der Buchstabe „ä“ ist beispielsweise in UTF-8 durch zwei Bytes kodiert. Durch das folgende slicing werden die beiden Bytes getrennt, was zu keiner sinnvollen Ausgabe führt.

```
meinstring := "ä"
fmt.Printf(meinstring[0:1])
```

Die Länge eines strings wird von der Funktion len() zurückgegeben. Dabei entspricht die Länge der Anzahl der Zeichen des Strings.

7.2.2 Collections

In Go gibt es drei verschiedene collections, nämlich arrays, slices und maps. Ein array in Go ist eine Folge von Elementen des gleichen Typs mit fester Länge. Arrays in Go werden durch den Indexoperator [ ] indiziert. Die Indizierung beginnt bei Null. Wenn das array nicht explizit initialisiert wird, wird es automatisch mit den Nullwerten des deklarierten Typs initialisiert. Arrays sind, im Gegensatz zu strings, veränderbar. Man kann einzelne Elemente austauschen.
Arrays kann man mit derselben Syntax, wie sie für strings verwendet wird, slicen. Das Resultat ist dann aber kein array mehr sondern ein slice. Ein slice in Go ist eine Referenz auf ein zugrunde liegendes array. Ein slice hat eine variable Länge und eine feste Kapazität. Die Kapazität entspricht der Länge des zugrunde liegenden arrays. Die Länge des slices ist kleiner gleich seiner Kapazität.
Eine map ist ein Wörterbuch in Go. Es ist ein ungeordneter collection Typ. Eine map enthält Paare von Schlüsseln und Werten. Die Schlüssel können alle Datentypen annehmen, die durch Vergleichsoperatoren vergleichbar sind. Die Werte dürfen beliebige Datentypen annehmen. Es müssen alle Schlüssel denselben Datentyp haben. Ebenso müssen auch alle Werte denselben Datentyp haben. Der Datentyp der Werte und Schlüssel kann sich aber unterscheiden. Eine map kann mit einer for...range Schleife durchlaufen werden. Dies geschieht in unbestimmter Reihenfolge.

7.2.3 JSON in Go

JSON (JavaScript Object Notation) ist ein verbreitetes Format zum Datenaustausch im Internet. Daten im JSON Format sind für Menschen einfach zu lesen und für Maschinen gut zu parsen. Als Zeichenkodierung wird standardmäßig UTF-8 verwendet. Außerdem erlaubt das JSON Format eine sehr kompakte Darstellung der Daten, vor allem kompakter als Alternativen wie XML. Ein Beispiel für Daten im JSON Format ist im Folgenden dargestellt.

```
{
    "Person": {
                "Name": "Mustermann"
```

```
                "Vorname": "Max"
            }
    "Haustier": "Hund"
}
```

Aufgrund der oben angeführten Vorteile wurde für den Webservice eine Darstellung der Antwort auf eine Anfrage im JSON Format implementiert. Dazu wurde das package encoding/json verwendet. Es bietet die Möglichkeit, die zu sendenden Daten im JSON Format darzustellen.

7.2.4 Beispiel für einen einfachen Webserver

Die Standard Bibliothek von Go enthält einige Pakete zur Netzwerkprogrammierung. Dazu gehört auch das net/http Package. Mit dessen Hilfe lässt sich ein HTTP Server implementieren. Um einen RESTful Webservice zu implementieren, wird im Webservice Programm ein solcher HTTP Server verwendet. Es folgt ein Beispiel für einen einfachen HTTP Server in Go. Dieses Beispiel wurde [1] entnommen. Der Server lauscht auf Port 4000. Host ist der localhost. Bei einem GET Aufruf schickt er eine response mit dem string „Hello!“ im body.

```
package main
import (
       "fmt"
     "net/http"
)

type Hello struct{}

func (h Hello) ServeHTTP(
     w http.ResponseWriter,
     r *http.Request) {
     fmt.Fprint(w, "Hello!")
}

func main() {
     var h Hello
     http.ListenAndServe("localhost:4000", h)
}
```

8 Zusammenfassung und Ausblick

Das Schreiben eines kodierten Textes auf eine Gebäudefassade ist ein interessantes architektonisches Gestaltungsmerkmal, das funktionelle und ästhetische Ansprüche erfüllen kann. Um den einkodierten Text für ein breites Publikum auf einfache Art und Weise zugänglich zu machen, ist eine Dekodierer notwendig. Durch die ausführliche Beschreibung der Dekodierung und der Analyse der dabei auftretenden Probleme und der algorithmischen Beschreibung eines möglichen Dekodierers, ist es möglich, schnell selbst einen Dekodierer zu implementieren. Zudem wurde in dieser Arbeit ein solcher Dekodierer als Webservice realisiert. Dieser Webservice kann für die in Kapitel 1.1 beschriebene Applikation, die mittels eines Mobiltelefons mit Kamera die Lamellensequenz an der Gebäudefassade dekodiert, verwendet werden. Er bietet eine einfach zu benutzende Schnittstelle zwischen dem Dekodierungsalgorithmus und dem noch zu implementierenden Smartphonefrontend. Dieses Frontend muss mit der Kamera erfasste optische Signale in eine Lamellensequenz umwandeln. Diese Lamellensequenz wird dann an den Webservice gesendet, der sie dekodiert. Der aus der Antwort des Webservices gewonnene Klartext soll dem Benutzer auf dem Smartphone grafisch dargestellt werden. Die für die ICT Cubes verwendete Art der Fassadenverkleidung mit einkodiertem Text wird vermutlich keine so weite Verbreitung finden wie QR-Codes, da die Realisierung verhältnismäßig aufwändig ist. Das öffentliche Interesse an derart gestalteten Gebäuden kann aber gesteigert werden und die Region in der das Gebäude steht kann dadurch an Attraktivität gewinnen.

Literaturverzeichnis

[1] "A Tour of Go." [Online]. Available: http://tour.golang.org

[2] "JSONlab." [Online]. Available: http://iso2mesh.sourceforge.net/cgi-bin/index.cgi?jsonlab

[3] "kadawittfeldarchitektur." [Online]. Available: http://www.kadawittfeldarchitektur.de

[4] "Warum App Engine," 2012. [Online]. Available: https://developers.google.com/appengine/whyappengine?hl=de

[5] F. Altenbach, G. Böcherer, and R. Mathar, "Short Huffman codes producing 1s half of the time," in *Proc. Int. Conf. Signal Process. Commun. Syst. (ICSPCS)*, 2011.

[6] T. Arens, F. Hettlich, C. Karpfinger, U. Kockelkorn, K. Lichtenegger, and H. Stachel, *Mathematik*. Spektrum, Akad. Verlag, 2008.

[7] G. Böcherer, F. Altenbach, M. Malsbender, and R. Mathar, "Writing on the facade of RWTH ICT Cubes: Cost constrained geometric Huffman coding," in *Proc. IEEE Int. Symp. Wireless Commun. Syst. (ISWCS)*, 2011.

[8] K. Bollhoefer, "Mobile Tagging mit 2D-Barcodes," 2007. [Online]. Available: http://www.pixelpark.com/de/pixelpark/_ressourcen/attachments/publikationen/070717_White_Paper_2D-Barcodes_Mobile_Tagging_final.pdf

[9] T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein, *Algorithmen—Eine Einführung*. Oldenbourg Verlag, 2010.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616-HTTP/1.1, the hypertext transfer protocol," 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2616.txt

[11] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, 2000.

[12] B. Klass, "Spatenstich zum Neubau der ICT-Cubes," 2013. [Online]. Available: http://www.rwth-aachen.de/go/id/dlzu/?#aaaaaaaaaaadlzv

[13] M. Summerfield, *Programming in Go: Creating Applications for the 21st Century*. Addison-Wesley Professional, 2012.

[14] S. Tilkov, "REST und HTTP," *Einsatz der Architektur des Web für Integrationsszenarien*, vol. 1, 2009.

[15] W. Utschick, *Lecture Notes Statistical Signal Processing*. Technische Universität München, Fachgebiet Methoden der Signalverarbeitung, 2013.