Arithmetisches Kodieren

Vito Schopper - 7503386
Mario Navarro Krauss - 7463132

Pro-Seminar
Datenkompression WS 2022

Dozent: Dr.- Ing. The Anh Vuong
Graphische Daten Verarbeitung, Informatik Institut
Goethe Universität, Frankfurt am Main
vorgelegt am xx.01.2023

Inhaltsverzeichnis

1 Thema **2**

2 Theoretische Grundlagen **2**

3 Verfahren-Beschreibung **2**

3.1 Kodierung 2

3.2 Problematik - Finite-Precision Arithmetic Coding 3

3.3 Dekodierung 4

4 Anwendungsgebiet **5**

5 Qualitätsbewertung über das Verfahren **5**

6 Präsentation / Demonstration Kit **6**

6.1 naives Verfahren 6

6.1.1 Installation 6

6.1.2 Kodierung 6

6.1.3 Dekodierung 7

6.2 genaues Verfahren 7

6.2.1 Kodierung 7

6.2.2 Dekodierung 7

6.3 naive HTML-Kodierung 7

1 Thema

Arithmetisches Kodieren ist eine Form der Entropiekodierung. Im Gegensatz zu anderen Entropiekodierungen wie z.B. der Huffman-Kodierung, wo zunächst ein Baum erstellt wird, um anschließend für jedes Zeichen eine individuelle Kodierung zu berechnen, wird beim Arithmetischen Kodieren eine lange binäre Fließkommazahl ausgegeben, welche die komplette Eingabe kodiert. Es findet also keine Kodierung kleinerer Komponente statt, sondern es wird Zeichen für Zeichen kodiert. Hierdurch werden Kompressionsraten erreicht, welche sehr nahe am theoretischen Limit der Entropie liegen.[3]

2 Theoretische Grundlagen

Das Verfahren stützt sich wie auch viele andere Kompressionsverfahren auf die Stochastik. So wird zu Beginn die Häufigkeit und die entsprechende Wahrscheinlichkeit für jedes Zeichen einer Eingabe berechnet. Das Verständnis von Binary-Search wird vorausgesetzt und nicht genauer erklärt, da dies auch keine Charakteristik des Kompressionsverfahrens beschreibt, sondern nur zur Berechnung einer binären Zahl innerhalb eines Intervalles dient *(siehe 3.1)*.

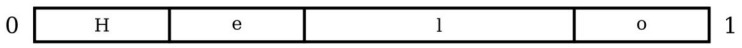
3 Verfahren-Beschreibung

3.1 Kodierung

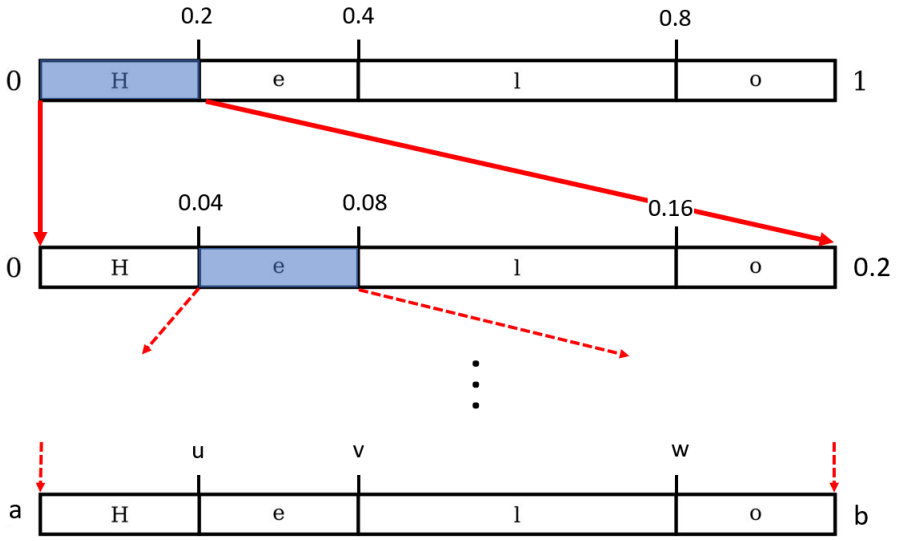
Zunächst wird für die Eingabe die Häufigkeit aller vorkommenden Zeichen berechnet. Anschließend wird das Interval [0,1] entsprechend dieser Häufigkeiten aufgeteilt. Dies wird gemacht, da am Ende beim Arithmetischen Kodieren die Eingabe als eine beliebig lange Fließkommazahl kodiert wird. Legen wir nun das Intervall [0,1] fest, so wird der später kodierte String irgendwo zwischen 0 und 1 liegen. So könnte z.B für die Eingabe **Hello** folgende Startbedingung erstellt werden.

Hello

char	#
H	1
e	1
l	2
o	1



Nun wird das erste Zeichen eingelesen. Dessen Intervall wird nun das komplette Intervall ersetzen. Die Verhältnisse der Intervalle untereinander ändert sich nicht. Es werden lediglich die Grenzen aktualisiert. Anschließend wird das nächste Zeichen eingelesen und die Intervallgrenzen entsprechend aktualisiert. Hat man dies nun für alle Zeichen der Eingabe durchgeführt, so liegt einem ein Intervall [a,b] vor.



Alle Zahlen innerhalb dieses Intervalls sind korrekte Kodierungen für unsere Eingabe. Es gibt somit theoretisch unendlich viele Kodierungen für eine Eingabe x, nämlich alle Zahlen im Intervall [a,b]. Da man bei einer Kodierung eine Eingabe aber natürlich mit möglichst wenig Bits kodieren möchte, wird die kürzeste Binärzahl genommen, welche im Intervall [a,b] liegt.
Dies kann z.B. durch einen Binary-Search Algorithmus erreicht werden. So wäre 0.11001 eine mögliche Kodierung für das Interval [0.78, 0.8].

3.2 Problematik - Finite-Precision Arithmetic Coding

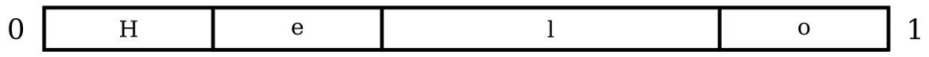
Normalerweise werden Zahlen mit 32 oder 64 Bit precision gespeichert. Das wird für uns ein Problem, da die Intervallgrenzen sich immer stärker annähern. Dies führt unvermeidlich zu einem Intervall, in welchem beide Grenzen den gleichen Wert annehmen [a,a] und fortan falsch Kodieren würde.
Es existieren mehrere Möglichkeiten, um diese Problematik mehr oder weniger effektiv zu umgehen. Oft wird direkt nach einem kodierten Zeichen geschaut, ob die Zahl im Intervall [0, 0.5] oder [0.5, 1] liegt und das jeweilige Intervall anschließend normalisiert, sodass Rechnungen mit hoher Präzision nie auftreten.

3.3 Dekodierung

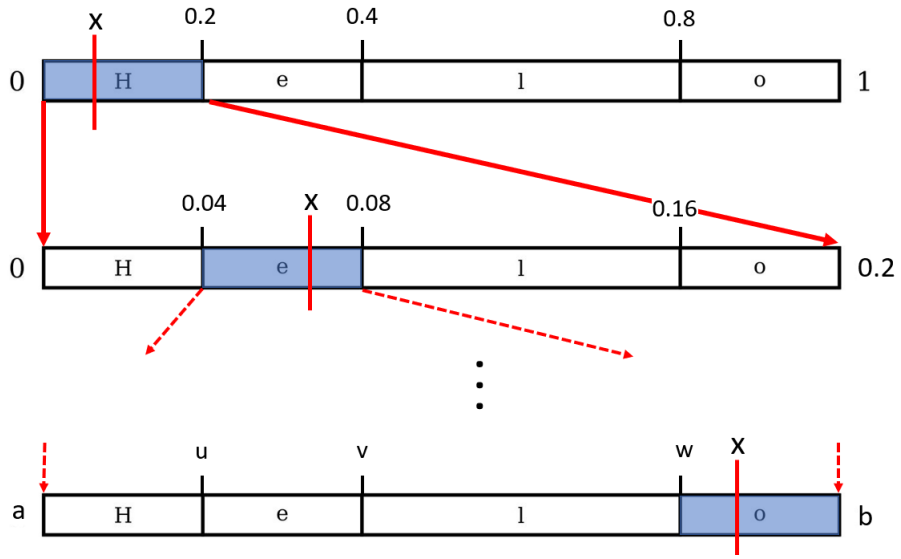
Hier wird als Eingabe ein kodierte Binärzahl x erwartet. Diese wird anschließend in eine Dezimalzahl umgerechnet. Ebenso wird eine Häufigkeitsverteilung aller Zeichen als Eingabe erwartet. Aus dieser wird analog zu 3.1 das Intervall [0,1] im Hinblick auf die Zeichen aufgeteilt. Eine Startbedingung könnte folgendermaßen aussehen:

$$x = 0.000100011_2 = 0.068359375_{10}$$

char	#
H	1
e	1
l	2
o	1



Durch die Häufigkeitsangaben wissen wir, dass die Dekodierung aus genau 5 Zeichen besteht.
Im ersten Schritt wird die Dezimalzahl x auf dem Intervall gesucht. Das kodierte Zeichen entspricht nun dem Intervall in welchem sich x befindet *(also H)*.
Anschließend werden die Intervallgrenzen analog zu 3.1 aktualisiert und x erneut auf dem Intervall [0, 0.2] gesucht.



Haben wir dies 5 mal durchgeführt erhalten wird unseren dekodierten String *(Hello)*.

4 Anwendungsgebiet

Implementierungen der Arithmetischen Kodierung können in der Text und Bildkompression genutzt werden. In Bildkompressionsoftware wird nur teilweise die Arithmetischen Kodierung als Entropie Kodierung genutzt. Aufgrund einer unklaren Patent Situation um die 1980er Jahre, wurde die Huffman-Kodierung oft bevorzugt[4]. JPEG zum Beispiel bietet jedoch beide Entropie-Kodierungen an, wobei eine Konvertierung von Huffman- zu Arithmetischer-Kodierung eine weitere Reduktion von bis zu 25% erreichen kann.

5 Qualitätsbewertung über das Verfahren

Die Arithmetische Kodierung kommt mit einer genauen Wahrscheinlichkeitstabelle des genutzten Alphabets nahe an das theoretische Limit der Datenkompression: der Entropie der Nachricht. Dabei kann es bessere Ergebnisse als die Huffman-Codierung erreichen, weist dafür jedoch auch einen höheren Rechenaufwand auf, da der Algorithmus komplizierter ist.
Die Asymmetric Numeral Systems (ANS)[6] kombinieren die nahe an der Entropie liegende Kompressionsrate dieses Verfahrens mit einer, der Huffman-Codierung vergleichbaren, Ausführungszeit.
Die Arithmetische Kodierung liefert also sehr gute Ergebnisse, aber benötigt dafür auch viele Rechenoperationen. Texte mit Millionen Zeichen können in Bruchteilen einer Sekunde komprimiert werden, aber wenn man große Mengen Bilder komprimieren will, kann man noch etwas mehr (zeitliche) Effizienz mit anderen Verfahren herausholen.

6 Präsentation / Demonstration Kit

Wir haben bei der Visualisierung/der Umsetzung des Demonstrations Kit zwischen 2 Fällen unterschieden. Da es bei den Berechnungen während der Kodierung/Dekodierung zu Schwierigkeiten mit den Intervallgrenzen kommt (siehe 3.2), haben wir ein **naives** und ein **genaues** Kit erstellt.

Das naive Kit erstellt eine gut verständliche, visuelle Animation. Jedoch funktioniert dieses Verfahren nur für relativ kleine Eingaben wie z.B "Hello" oder "Laterne". Bei größeren Eingaben benötigt die Erstellung der Animation erheblich länger. Da dieses Verfahren auch nicht das *Finite-Precision Arithmetic Coding* umsetzt, kommt es hier bei größeren Eingaben zu den in 3.2 erwähnten Fehlern. Auf kurzen Eingaben ist jedoch eine korrekte Kodierung/Dekodierung, sowie eine Erstellung einer visuell ansprechenden Animation kein Problem.

Hierfür haben wir auf das python-Modul **manim** zurückgegriffen. Dabei handelt es sich um ein Opensource-Projekt des Youtubers "3Blue1Brown", welcher es 2015 geschrieben hat, um visuell ansprechende Animationen für den Bereich der Mathematik zu erstellen.[5]
Mittlerweile wird dieses Model jedoch durch die Community erweitert und bietet viele Möglichkeiten Themen aus dem MINT-Bereich visuell darzustellen.

6.1 naives Verfahren

6.1.1 Installation

Es wird das python-modul **manim**, sowie viele kleine weitere module wie z.B. **ffmpeg** benötigt, welche jedoch automatisch mit **manim** heruntergeladen werden.
Der Installationsvorgang wird unter https://docs.manim.community/en/stable/installation.html genauer beschrieben.

6.1.2 Kodierung

Für die Kodierung wird die Datei **encoding.py** mithilfe des Befehls

```
manim -pql -v critical encoding.py
```

im terminal ausgeführt.
Anschließend wird als Eingabe das zu kodierende Wort erfragt. Hier wie oben erwähnt, kein zu langes Wort eingeben. Daraufhin kann es je nach länge des Wortes 1-2 min dauern, bis die Animation erstellt wurde, welche sich direkt in einem neuen Fenster öffnet und angeschaut werden kann. Alternativ wurde die Animation unter

```
.\SeminarDatenkompression-2022\pythno\media\videos%encodin\480p15\Encoding.mp4
```

gespeichert.

6.1.3 Dekodierung

Für die Dekodierung wird die Datei **decoding.py** mithilfe des Befehls

```
manim -pql -v critical decoding.py
```

im terminal ausgeführt. Anschließend wird als Eingabe zuerst die zu dekodierende Binärzahl erfragt. Daraufhin wird eine Häufigkeitsverteilung der vorkommenden Buchstaben erfragt. Diese muss im Format eines python-Dictionaries eingegeben werden.
Beispiele:

```
{"H": 1, "e": 1, "l": 2, "o": 1}
{"A": 1, "e": 1, "f": 2}
```

Daraufhin kann es je nach Länge des Wortes mehrere Minuten dauern, bis die Animation erstellt wurde, welche sich direkt in einem neuen Fenster öffnet und angeschaut werden kann. Alternativ wurde die Animation unter

```
.\Seminar-Datenkompression-2022\python%\media\videos\decodin\480p15\Decoding.mp4
```

gespeichert.

6.2 genaues Verfahren

Hierfür bitte die html Datei **Datenkompression.html** unter *./Seminar-Datenkompression-2022/HTML* öffnen. Die erscheinende Seite, implementiert unsere Kodierungen/Dekodierungen gebündelt in einer html. Hier werden jedoch keine Animationen angezeigt. (Verweis auf 6.1)

6.2.1 Kodierung

Für die Kodierung wird eine Eingabe im Eingabefeld erwartet. Falls gewünscht kann man sich mit dem Button **Generiere Wahrscheinlichkeiten**, die Wahrscheinlichkeitsverteilung der Eingabe anzeigen lassen.
Um nun eine genaue Kodierung berechnen zu lassen, nach Eingabe ins obere Feld, den Button **Starte Genaues-Kompressionsverfahren** drücken.
Anschließend wird der kodierte Binärstring im unteren Feld angezeigt.

6.2.2 Dekodierung

Um den kodierten Binärstring zu dekodieren, sollte man am besten kurz das obere Eingabefeld manuell leeren. Fügt man nun den kodierten Binärstring ins untere Feld ein und drückt auf **Starte Genaues-Dekompressionsverfahren**, so wird einem die Dekodierung im oberen Feld angezeigt.

6.3 naive HTML-Kodierung

Auf der HTML Seite kann man zusätzlich noch einmal das naive Verfahren ausführen lassen. Das Vorgehen ist analog zu 6.2.
Anmerkung: Die Kodierung weicht leicht von der Kodierung in den Animationen ab, da der Algorithmus anders implementiert wurde.

Literatur

[1] Mark Nelson. (2014, October 19). Data compression with arithmetic coding. Mark Nelson. Retrieved January 12, 2023, from https://marknelson.us/posts/2014/10/19/data-compression-with-arithmetic-coding.html

[2] Arithmetic coding. The Hitchhiker's Guide to Compression. (n.d.). Retrieved January 15, 2023, from https://go-compression.github.io/algorithms/arithmetic/

[3] Wikimedia Foundation. (2022, November 6). Arithmetisches Kodieren. Wikipedia. Retrieved January 12, 2023, from https://de.wikipedia.org/wiki/Arithmetisches_Kodieren

[4] Wikimedia Foundation. (2023, January 21). Arithmetic coding. Wikipedia. Retrieved January 23, 2023, from https://en.wikipedia.org/wiki/Arithmetic_coding

[5] Wikimedia Foundation. (2022, December 30). 3Blue1Brown. Wikipedia. Retrieved January 15, 2023, from https://en.wikipedia.org/wiki/3Blue1Brown

[6] Wikimedia Foundation. (2022, August 9). Asymmetric Numeral Systems. Wikipedia. Retrieved January 23, 2023, from https://de.wikipedia.org/wiki/Asymmetric_Numeral_Systems