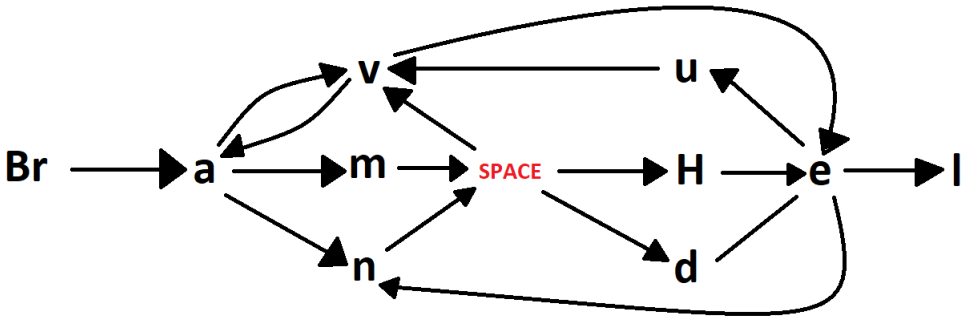
Pseudowillekeur op Markovketens

Bram van den Heuvel

27 juni 2020

Bachelorscriptie Wiskunde

Begeleiding: dhr. dr. J.L. Dorsman, Mr. L.R. van Kreveld MSc



Korteweg-de Vries Instituut voor Wiskunde
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Universiteit van Amsterdam

Samenvatting

Deze thesis is een korte beschrijving van een algoritme wat met behulp van een pseudowillekeurig algoritme een Markovketen construeert waarin een gegeven informatiebron kan worden gecomprimeerd. Het algoritme heeft een decompressietijd van $O(n)$ en een compressietijd van $O(2^n)$, wat betekent dat het algoritme voornamelijk toepasselijk is in scenario's waar rekenkracht op compressie niet relevant is.

Deze lange compressetijd hoopt het algoritme goed te maken met de efficiëntie bij onderling afhankelijke teksten welke hiermee makkelijk kunnen worden gecomprimeerd.

Het algoritme bouwt een Markovketen en zoekt vervolgens een passende seed waarmee een pseudowillekeurig pad kan worden bewandeld met behulp van een pseudowillekeurig algoritme. Het pseudowillekeurige algoritme wordt hierbij als een bekend gegeven beschouwd voor zowel de transmitter als de ontvanger, terwijl de seed en de Markovketen variabelen zijn die samen in één worden verzonden naar de ontvanger.

Titel: Pseudowillekeur op Markovketens
Auteur: Bram van den Heuvel, bachelorproject@bram.blmgroep.nl, 11273933
Begeleiding: dhr. dr. J.L. Dorsman, Mr. L.R. van Kreveld MSc
Einddatum: 27 juni 2020

Korteweg-de Vries Instituut voor Wiskunde
Universiteit van Amsterdam
Science Park 904, 1098 XH Amsterdam
`http://www.kdvi.uva.nl`

Inhoudsopgave

1 Inleiding

1.1 Een namengenerator

Een paar maanden voor ik begin met het onderzoeken voor mijn thesis, kwam ik een website tegen wat een programma beweerde te hebben wat mijn interesse wekte. De website zou een programma hebben wat nieuwe namen kon genereren! [don, 2020] Ik was nieuwsgierig, en besloot een kijkje te nemen.

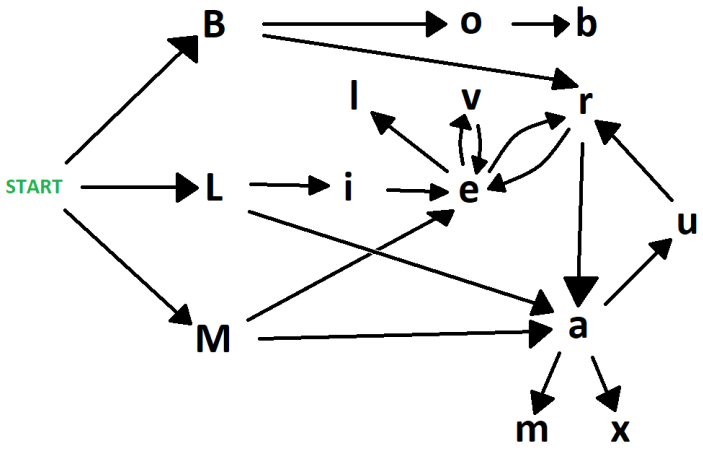
Wat ik vond, was een simpele webpagina met twee velden: in het eerste veld kon men zelf een aantal namen invullen, gescheiden door een spatie. Eronder zat een knop met "Generate", en als men op deze knop klikte, verschenen er in het tweede veld een aantal namen die op basis van de ingevoerde namen waren gemaakt. Zie hieronder een voorbeeld.

```
Bram Max Merel Lieve Bob Laura
===
GENERATE
===
Maur
Lax
Braur
Liera
Bram
```

Het programma heet een *Markov name generator*. Het programma maakt een Markovketen op basis van de gegeven namen, en loopt vervolgens willekeurige paden over de Markovketen. De Markovketen wordt gecontrueerd door voor elke letter te kijken in de namen wat de mogelijke volgende letters zijn. Een voorbeeld van deze keten is in figuur 1.1 te zien.

De namen worden vervolgens gegenereerd door een pad te lopen over de Markovketen. In het geval van de *Markov name generator* wordt er een extra toestand toegevoegd (genaamd "START") om aan te geven waar het algoritme begint, en vanuit waar het letters zal gaan kiezen. Vervolgens loopt het algoritme een willekeurig pad over de Markovketen, en het onthoudt elke letter die het tegenkomt. Als aan een eindconditie wordt voldaan, stopt het algoritme. Voorbeelden van eindcondities zijn dat de naam een gewenste lengte heeft, dat de Markovketen op een gegeven toestand moet eindigen, of dat de naam ¨mooi¨ of gewenst is.

Als aan de eindconditie is voldaan, dan kijkt men terug naar alle toestanden die tijdens het pad over de Markovketen zijn langsgekomen. De letters uit de toestanden



Figuur 1.1: Dit is de Markovketen die de *Markov name generator* zou gebruiken: het algoritme begint op ¨START¨ als startpunt, en kiest één van de letters B, L en M. Vervolgens loopt men de Markovketen af totdat men op een absorberend punt terecht komt, of totdat het algoritme uit andere redenen ervoor kiest om te stoppen, bijvoorbeeld als de gegenereerde naam te lang wordt geacht.

worden samengevoegd tot één tekst, en die tekst is de gegenereerde naam. Op deze manier worden dus de namen **Maura**, **Lax**, **Braur**, **Liera** en **Bram** gevormd met de bovenstaande namen.

1.2 Een pseudowillekeurige website

Het concept dat men een collectie namen kan gebruiken om nieuwe namen te genereren, wekte in eerste instantie mijn interesse. Met deze inspiratie heb ik een website gemaakt voor het spel *Dungeons and Dragons*: dit is een gezelschapsspel in een middeleeuws thema met elven, dwergen, orcs en andere gekke wezens. Dit soort wezens hebben namen nodig - en naar mijn ervaring bleken de namen gegenereerd uit dergelijke Markovketens vrij effectief! Ik heb daarom een website gebouwd die dorpen genereert. [Heuvel, 2020] Deze dorpen bevatten enorme hoeveelheden inwoners, die allemaal namen hebben die gegenereerd zijn met een Markovketen.

Ter voorbeeld hebben de vijf artificers uit het dorp **Thesis-town** van gemiddelde grootte de volgende namen:

```
Elias Serpenteye
Sylkas Halyhomin
Rhea Stonebluff
Edlyn Nightglory
```

```
Adison Flamevigor
```

Het nadeel van een dergelijke website, is dat het opslaan van alle namen van 4000 inwoners relatief veel opslagruimte kost. Om te voorkomen dat ik van talloze dorpen duizenden gegenereerde namen moest onthouden en opslaan, heb ik een truc toegepast.

In plaats van de gegenereerde namen op te slaan op mijn computer, gebruik ik een zogenaamd *pseudowillekeurig algoritme* om de namen te genereren. Dit is een algoritme wat op basis van een input een willekeurige output levert. Dit kan een getal zijn, een rij getallen, of in dit geval een lijst namen. Wiskundig gezien is een dergelijk algoritme echter niet werkelijk willekeurig aangezien dezelfde input altijd dezelfde output levert. Daarom noemen we het algoritme pseudowillekeurig.

Met deze pseudowillekeurige eigenschap is het echter mogelijk om een geheugenloze website op te zetten. De website vraagt de gebruiker om een dorpsnaam en -grootte. Als dezelfde naam en grootte geleverd worden, dan wordt dus altijd dezelfde "willekeurige" namen gegenereerd.

Dit gaf mij een interessante vondst! Op basis van een relatief kleine dorpsnaam kan een virtueel onbeperkte rij van namen worden gegenereerd - wat nou als we dit kunnen gebruiken om een gegeven namenlijst op te slaan? Het voorbeeld uit sectie 1.1 laat zien dat het mogelijk is om de originele namen te genereren, (in het voorbeeld wordt de naam **Bram**, één van de oorspronkelijke namen, gegenereerd) dus de Markovketen zou gebruikt kunnen worden om bestaande voor ons relevante namen op te slaan.

Het zou goed kunnen dat voor een gegeven input, het pseudowillekeurige algoritme een pad over de Markovketen uit figuur 1.1 loopt op zodanige wijze, dat de namen **Bram**, **Max**, **Merel**, **Lieve**, **Bob**, en **Laura** gegenereerd worden. Wat we dan zouden kunnen doen, is dat we de input waarmee we de namen genereren in plaats van de namen zelf.

Dit is waar mijn onderzoek over gaat. Het is mogelijk om een gegeven namenset zodanig te verwerken in een Markovketen, dat de namen weer met de Markovketen en een zekere input op een pseudowillekeurig algoritme kunnen worden gegenereerd. Op de website leek dit een efficiënt alternatief te zijn wat significant minder opslagruimte kost. Mijn onderzoeksvraag is dus als volgt: hoe efficiënt is het algoritme dat pseudowillekeur en Markovketens gebruikt om informatie op te slaan in verhouding met andere, bestaande compressiealgoritmes?

2 Wiskundige achtergrond

Voordat we de onderzoeksvraag kunnen beantwoorden, is het belangrijk om de relevante termen uit de informatietheorie gedefinieerd te hebben. Ten eerste is het belangrijk om een maatstaf voor efficiëntie te definiëren. Hierna halen we inspiratie op uit de informatietheorie om nieuwe definities te introduceren die voor dit onderzoek relevant zijn.

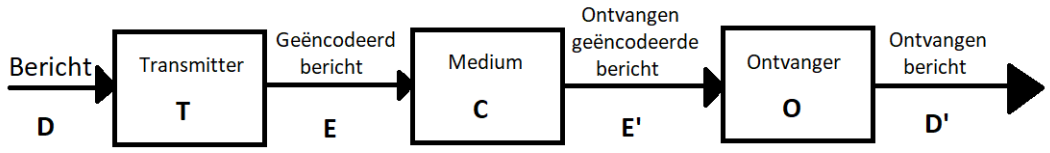
2.1 Broncodering

Een veelgebruikt schema in de informatietheorie ziet eruit zoals weergegeven in figuur 2.1: men begint met een bericht D wat verzonden moet worden (zoals een naam). De transmitter zet dit bericht om naar een zeker geëncodeerd bericht wat via medium (channel) C wordt verzonden naar de ontvanger O. Dit medium kan van alles zijn: denk bijvoorbeeld aan het internet, een telegraaflijn, een spreekbuis of een telefoongesprek. Via het medium kunnen echter kleine aanpassingen plaatsvinden. Er kan bijvoorbeeld ruis zitten in het medium, of het bericht kan onvolledig arriveren bij O. Hetgeen wat bij O arriveert, is bericht E'. In deze thesis gaan we er echter vanuit dat altijd geldt dat $E = E'$, dus dat het medium C elk bericht E compleet en foutloos verstuurt naar ontvanger O.

Ontvanger O ontvangt dus het bericht E van C, en zet dit weer om naar een bericht D'. We zeggen dat onze informatie succesvol is overgedragen indien $D' = D$. De onderstaande paragraaf laat hier een klein voorbeeld van zien.

2.2 Binaire codering

Vaak hebben media een gedefineerde verzameling symbolen die over het medium kunnen worden verstuurd. Het internet gebruikt bijvoorbeeld enkel de symbolen 0 en 1: elk bericht zal dus moeten worden geëncodeerd naar nullen en enen.



Figuur 2.1: Weergave van de methode

A	00
B	01
C	10
D	11

Tabel 2.1: Voorbeeld van een binairecoderingsalgoritme. De transmitter T zet elke letter om in de aangegeven binaire cijfers, terwijl de ontvanger O de aangegeven binaire cijfers gebruikt om op te zoeken welke letter ermee wordt bedoeld.

Stel dat we een bericht willen versturen dat de nieuwste CD van de muziekgroep ABBA beschikbaar is, en het bericht $D =$ "CDABBA" willen versturen. Dan kunnen we elk van de vier letters een getal van 0 tot 3 toekennen, en deze omzetten naar het binair. Dit levert ons de encodeertabel op als in tabel 2.1.

Als we het bericht op deze manier encoderen en versturen, dan kan de ontvanger O in dezelfde tabel kijken, en het bericht weer terughalen. Dit geeft ons de volgende gegevens:

```
D  = CDABBA
E  = 10 11 00 01 01 00
E' = 10 11 00 01 01 00
D' = CDABBA
```

Met andere woorden, er geldt dat $D' = D$, dus onze informatie is succesvol overgedragen.

2.3 Datacompressie

Vaak is er bij het verzenden van berichten een behoefte om het aantal tekens verzonden over het medium te minimaliseren. Er kan bijvoorbeeld sprake zijn van een beperkt aantal tekens wat per tijdseenheid kan worden verzonden, het medium kan mogelijk slechts een eindig aantal symbolen verzenden, of er komt geld kijken bij het verzenden van berichten. Bij datacompressie onderzoekt men de informatie van een bericht D verzonden kan worden met zo min overdracht via medium C.

Hierbij maakt men onderscheid tussen twee verschillende typen van datacompressie:

1. **Exact omkeerbare compressie.** (Lossless compression) Exact omkeerbare compressie gaat over het verzenden van berichten waarbij het algoritme van transmitter T omkeerbaar is, en dat altijd zal gelden dat $D' = D$, ondanks de inhoud van het bericht D.

2. **Niet-exact omkeerbare compressie.** (Lossy compression) Niet-exact omkeerbare compressie gaat over het verzenden van berichten waarbij het algoritme van transmitter T in sommige gevallen niet omkeerbaar is, dat wil zeggen, er zijn berichten waarvoor geldt dat $D' \neq D$, in ruil voor een kleinere hoeveelheid symbolen verzonden over het medium C.

C	1
D	01
A	001
B	0001

Tabel 2.2: Voorbeeld van een thermometercoderingsalgoritme. De transmitter T zet elke letter om in de aangegeven binaire cijfers, terwijl de ontvanger O de aangegeven binaire cijfers gebruikt om op te zoeken welke letter ermee wordt bedoeld.

In deze thesis wordt de onderzoeksvraag gesteld in de context van lossless compression: de voordelen van het algoritme onder lossy compression worden hier voorlopig buiten zicht gelaten.

Dit betekent dat we als vereiste stellen dat informatie door het algoritme altijd succesvol moet worden overgedragen, dat wil zeggen, er moet altijd gelden dat $D' = D$.

2.4 Thermometercodering

Een ander voorbeeld van codering voor een binair medium is thermometercodering. In plaats van dat een gegeven lengte wordt gekozen voor alle tekens uit het bericht, worden de tekens gesorteerd op hoe vaak ze voorkomen, en vervolgens wordt hen tekens van verschillende lengte toegekend.

Neem als voorbeeld dat we een bericht willen versturen dat de nieuwste CD van de muziekgroep ACDC beschikbaar is, en daarom het bericht $D =$ ‘CDACDC’ willen versturen. Dan kunnen we de tekens sorteren op hoe vaak ze voorkomen, en kunnen we de tekens toekennen als aangegeven in figuur 2.2.

Als we het bericht op deze manier encoderen en versturen, dan kan de ontvanger O nogmaals in dezelfde tabel kijken, en het bericht weer terughalen. Dit geeft ons de volgende gegevens:

```
D  = CDACDC
E  = 1 01 001 1 01 1
E' = 1 01 001 1 01 1
D' = CDACDC
```

Hier valt het op dat we over het medium C slechts 10 cijfer hoeven te versturen. De lezer kan narekenen dat er 12 cijfers nodig waren geweest indien de binaire codering werd toegepast. Dit betekent dat de thermometercodering in het geval van het bericht $D =$ ‘CDACDC’ efficiënter bleek.

2.5 Entropie

De conclusie van sectie 2.4 leidt tot de vraag hoe ver men kan gaan met compressie voor het medium. Hoe ver kan een bericht gecomprimeerd worden?

Wat belangrijk is om hiervoor te weten, is wat de *informatiedichtheid* van een bericht D is. Een andere term voor informatiedichtheid, welke veel wordt gebruikt in de informatietheorie, is entropie. Een bericht wat veel informatie bevat, heeft een hoge entropie. We gaan met behulp van de entropie bepalen hoe compact bepaalde berichten kunnen worden gecomprimeerd.

Zij A een stochast met een discrete verdeling over uitkomsten a_i van een uitkomstenruimte Ω met kans $\mathbb{P}(A = a_i)$. Dan is de entropie van A, geschreven als $H(A)$, als volgt gedefinieerd:

$$H(A) = \sum_i -\mathbb{P}(A = a_i) \log \left(\mathbb{P}(A = a_i)\right), \tag{2.1}$$

waarbij het logaritme basis 2 heeft. In deze thesis zullen alle logaritmes basis 2 hebben.

Beschouw het voorbeeld beschreven in sectie 2.4. Neem uitkomstenruimte $\Omega = \{"A", "B", "C", "D"\}$ waarbij A, B, C en D kansen $\frac{1}{6}$, 0, $\frac{1}{2}$ en $\frac{1}{3}$ respectievelijk hebben als uitkomst voor A. Dit geeft dan dat

$$H(A) = -\frac{1}{6} \log \left(\frac{1}{6}\right) - 0.5 \log 0.5 - \frac{1}{3} \log \left(\frac{1}{3}\right) = \frac{2}{3} + \frac{1}{2} \log (3) \approx 1,459.$$

De informatiedichtheid van ons bericht is dus $\frac{2}{3} + \frac{1}{2} \log (3)$. Wat betekent dit intuïtief? Zoals David J.C. Mackay in zijn boek beschreef, gaf de wetenschapper Shannon een bewijs voor het theoretische limiet van datacompressie. [MacKay, 2003] In informele taal bewees Shannon het volgende:

Voor n onafhankelijke en indentiek verdeelde stochasten A_i met entropie $H(A)$ in minstens $nH(A)$ symbolen moeten worden geëncodeerd als men exacte omkeerbaarheid wenst te bewaren.

Met andere woorden, in het optimale geval zou men gemiddeld $H(A)$ tekens door medium C moeten sturen per symbool in bericht D.

2.6 Kolmogorovcomplexiteit

Tot nu toe zijn er verschillende compressiealgoritmes besproken die een tekst vertalen naar een stel tekens die door het medium C heen gaan en bij de ontvanger O terugvertaald kunnen worden naar de oorspronkelijke tekst. De onderzoeksvraag ging echter over het idee om een algoritme wat de tekst genereert door te sturen in plaats van de vertaalde tekst zelf.

Stel dat ons geëncodeerde bericht E een representatie is van een algoritme α wat bij ontvanger O het bericht D kan genereren, (Zie figuur 2.1) en stel dat E een lengte heeft van n_E karakters, dat wil zeggen, er gaan n_E symbolen via medium C naar de ontvanger O. Dan zeggen we dat de Kolmogorovcomplexiteit van D gelijk is aan n_E: bericht D heeft n_E karakters nodig om met algoritme α te worden gegenereerd. Als alternatief schrijven we dat

$$K_\alpha(D) = n_E \text{ met } n_E = L(E).$$

Hierbij is L de lengtefunctie, welke aangeeft hoeveel symbolen via medium C zijn verzonden naar de ontvanger.

De Kolmogorovcomplexiteit van een bericht heeft een speciale relatie met de entropie van het bericht: in 1982 schreef A. A. Brudno een artikel waarin hij schrijft dat de genormaliseerde Kolmogorovcomplexiteit zou convergeren naar de entropie van de bron.

Met andere woorden, als ons bericht D bestaat uit een serie van n stochasten A_i, alle identiek en onafhankelijk verdeeld als A, dan geldt er dat

$$\lim_{n\to\infty} \frac{K_\alpha(D)}{n} = H(A). \tag{2.2}$$

Intuïtief betekent dit dat we voor een bericht van lengte n over het algemeen minimaal $nH(A)$ nodig zullen hebben, dus dat $nH(A)$ een ondergrens zal zijn voor ons algoritme.

2.7 Kolmogorovwillekeur

Laat een algoritme α wat een geëncodeerd bericht E vertaalt naar het oorspronkelijke bericht D, gegeven zijn. Dan geldt er dat een bericht D *Kolmogorovwillekeurig* als het kleinste programma dat beschreven kan worden met symbolen uit D, minstens even groot is als het oospronkelijke bericht.

Een voorbeeld hiervan zou een bericht zijn als de volgende:

```
01010101010101010101010101010101
98l3ynsir4tuse4utnlotu93w02tu4os
```

Het eerste bericht kan met een vrij klein programma worden gereproduceerd, terwijl de laatste tekst lastig te reproduceren is met een simpel programma in pseudocode. De simpelste optie zou namelijk 6 extra karakters vereisen:

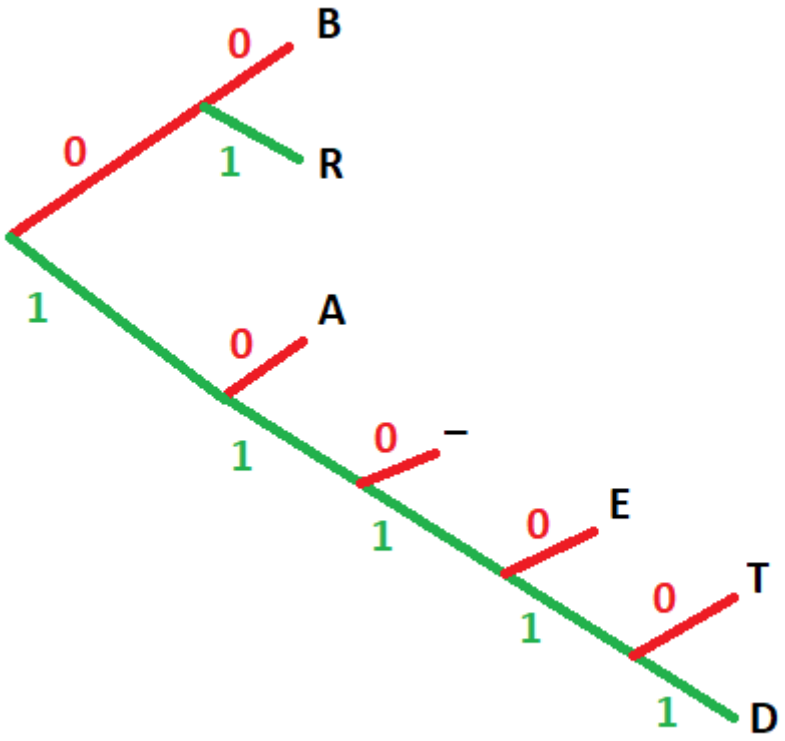
```
write 01 16 times
write 98l3ynsir4tuse4utnlotu93w02tu4os
```

Om deze reden noemen we het bericht D = "98l3ynsir4tuse4utnlotu93w02tu4os" Kolmogorovwillekeurig.

We kunnen met een telbewijs aantonen dat er voor elke lengte D een bericht van lengte n is wat Kolmogorovwillekeurig is. Stel dat onze verzameling symbolen twee elementen bevat: dan zijn er $2^{n-1} + 2^{n-2} + \cdots + 2^1 + 2^0 = 2^n - 1$ berichten van lengte $n-1$ of minder. Volgens het duiventilprincipe zal er dus minstens één van de 2^n berichten van lengte n een Kolmogorovcomplexiteit hebben van minstens n. [Wikimedia, 2020a] Dit bewijs kan veralgemeniseerd worden voor verzamelingen symbolen van arbitraire (doch eindige) lengte.

Wat betekent dit voor ons algoritme? Het betekent dat er altijd berichten zullen zijn die door het algoritme niet effectief gecomprimeerd kunnen worden - sterker nog, aangezien het algoritme sommige berichten beter zal comprimeren, zullen er volgens hetzelfde telbewijs ook berichten slechter worden gecomprimeerd, en juist meer symbolen kosten.

De truc zit hem erin om veelvoorkomende en belangrijke teksten zo veel mogelijk te comprimeren, en teksten die weinig voorkomen te vermijden. Als we bijvoorbeeld een



Figuur 2.2: De boomstructuur die voor de Huffmancodering wordt gebruikt in het voorbeeld, die tevens in tabel 2.3 kan worden gevonden.

namenlijst zouden willen comprimeren, dan zou een naam zoals **Anna** weinig symbolen nodig moeten hebben, terwijl een naam als **xYJk04e8** vrij onwaarschijnlijk is om aan een kind gegeven te worden, waardoor deze minder compact mag worden opgeslagen.

2.8 Huffmancodering

Huffmancodering is een alternatief op de thermometercodering die in een boomformaat bepaalt welke binaire getallen worden gebruikt om symbolen te encoderen. Beschouw de volgende voorbeeldtekst:

```
DE_BARBAAR_BARBARA_AT_RABARBER
```

Met een algoritme wat voor deze thesis irrelevant is, wordt vervolgens de boomstructuur in elkaar gezet en bepaald dat de waardes in tabel 2.3 optimaal zijn voor dit scenario. Een visuele representatie van deze structuur kan in figuur 2.2 gevonden worden.

A	10
B	00
D	11111
E	1110
R	01
T	11110
_	110

Tabel 2.3: Voorbeeld van een Huffmancoderingsalgoritme. De transmitter T zet elke letter om in de aangegeven binaire cijfers, terwijl de ontvanger O de aangegeven binaire cijfers gebruikt om op te zoeken welk symbool ermee wordt bedoeld.

Wat deze vorm van codering relevant maakt voor deze thesis, is dat er meerdere bewijzen bestaan dat dit algoritme optimaal zou zijn voor het comprimeren van individuele symbolen. [Nandakumar, 2020] Het blijkt echter minder efficïent te zijn als men probeert meerdere symbolen in één te verwerken - of, nog sterker, als de symbolen onderling afhankelijk zijn.

Tot nu toe hebben we aangenomen dat de symbolen binnen het bericht onderling onafhankelijk zijn - maar dat hoeft vaak niet waar te zijn. Een voorbeeld van een bericht dat mogelijk compact zou kunnen worden overgedragen, is het volgende:

```
{
    "name": "Julia",
    "age": "21",
    "university": "UvA",
    "country": "NL"
}
```

Berichten bevatten vaak duidelijke structuren die al snel door mensen worden gebruikt en opgepikt. Zo volgt er hier na een dubbele aanhalingsteken vaak (maar niet altijd) een dubbele punt of een komma. Na een komma komt altijd een nieuwe regel, en na een dubbele punt komt altijd een spatie en dubbele aanhalingstekens. Dergelijke structuren in een bericht zijn makkelijk om gebruik van te maken met behulp van een Markovketen. Als dergelijke patronen er structureel uit gepikt kunnen worden, dan kan het algoritme een hoop informatie vrij evident comprimeren.

Hier zal het algoritme het sterkst op moeten zijn, mocht het in zekere gevallen efficiënter dan Huffmancodering willen zijn.

3 Constructie van het algoritme

In het afgelopen hoofdstuk hebben we context gegeven uit de informatietheorie en een paar concepten bestudeerd waar we rekening mee moeten houden. In dit hoofdstuk zal het algoritme geconstrueerd worden, wat we in het hoofdstuk 4 zullen bestuderen op efficiëntie. We zullen de wiskundige termen algemeen definiëren, maar de voorbeelden uit de inleiding erbij houden voor perspectief en intuïtie.

3.1 Databron

In de context van het algoritme waar deze thesis over gaat, zullen we vaak naar het bericht D verwijzen als een *databron*. We doen dit met opzet ter herinnering dat het bericht geen rij onafhankelijke of identiek verdeelde stochasten hoeft te zijn. Het is mogelijk dat (een deel van) de tekens afhankelijk of verschillend verdeeld zijn.

Zij Ω een aftelbare verzameling symbolen. Dan noemen we een eindige rij D een *databron* als

$$D = (s_1, \dots, s_n), \text{ waarbij } s_i \in \Omega \tag{3.1}$$

voor alle $i \in \{1, \dots, n\}$ met $n \in \mathbb{N}$. We definiëren de entropie van D als de entropie van de meest aannemelijke stochast die de databron zou kunnen vormen: definieer een stochast A als een stochast met de gegeven verzameling Ω als uitkomstenruimte, waarbij voor alle $x_i \in \Omega$ geldt dat

$$\mathbb{P}(A = x_i) = \frac{1}{n} \sum_{j=1}^{n} 1_{(s_j = x_i)}. \tag{3.2}$$

Met deze definitie zeggen we dan dat

$$H(D) \coloneqq H(A) = -\sum_{x_i \in \Omega} \mathbb{P}(A = x_i) \log(\mathbb{P}(A = x_i)). \tag{3.3}$$

Merk op dat we hier niet kiezen om de eerder gegeven entropie gebruiken op Ω omdat deze eerdere entropie aanneemt dat alle tekens onderling onafhankelijk zouden zijn.

3.2 Pseudowillekeurige generator

Zoals beschreven in de inleiding, gebruiken we een pseudowillekeurig algoritme om een pseudowillekeurig pad over de Markovketen te lopen. Zij P een functie die een natuurlijk getal afbeeldt naar een reeks pseudowillekeurige getallen, gegeven door

$$P : \mathbb{N} \mapsto (x_1, x_2, x_3 \dots), \text{ met } x_i \in (0,1), \text{ als uitkomsten van } X \sim \text{unif}(0,1)\ \forall i \in \mathbb{N}. \tag{3.4}$$

Dan heet P een *pseudowillekeurige generator*.

Met een reeks pseudowillekeurige getallen is het mogelijk om een pad te lopen over een Markovketen. In de volgende paragraaf definiëren we hoe de Markovketen eruit zal zien.

3.3 Markovpakket

Zij M een Markovketen waarvan de *toestandsruimte* I gedefinieerd is als deelverzameling van de volgende verzameling:

$$I \subseteq \{(i, (d_{i,1}, \ldots, d_{i,n})) \mid d_{i,j} \in D, i, j \in \{1, ..., n\}, n \in \mathbb{N}\}. \tag{3.5}$$

Dit is een formele manier om te verwoorden dat I een verzameling is van toestanden die bestaan uit letters uit Ω. De toestanden bevatten daarnaast ook nog een uniek cijfer om te waarborgen dat meerdere toestanden dezelfde combinatie letters mag bevatten.

De Markovketen op de voorpagina heeft de volgende toestandsruimte:

```
I = {
    (1, "Br"), (2, "a"), (3, "m"), (4, "v"), ( 5, "n"),
    (6, " " ), (7, "d"), (8, "e"), (9, "H"), (10, "l")
}
```

Aangezien het eerste getal enkel uniciteit waarborgt en toestaat dat meerdere toestanden dezelfde letters bevat, geven we enkel de letters weer die de toestanden bevatten.

Als de toestandsruimte I van dergelijk formaat is, dan noemen we M een *Markovpakket van* D indien er een pad over M bestaat zodanig dat

$$(d_1, \ldots, d_{k_1}), \ldots, (d_{k_1+1}, \ldots, d_{k_2}), \ldots, (d_{k_{i-1}+1}, \ldots, d_{k_i}) = (s_1, \ldots, s_{k_i}) \supseteq (s_1, \ldots, s_n) = D. \tag{3.6}$$

Als dit het geval is, dan zeggen we dat er een pad bestaat waarvan de databron D de kop vormt. In het voorbeeld is de Markovketen een Markovpakket van $D =$ "Bram van den Heuvel", omdat er een pad over de Markovketen bestaat zodanig dat men de naam **Bram van den Heuvel** maakt. Andere voorbeelden van berichten waarvoor die Markovketen een Markovpakket is, zijn de verichten **Br**, **Bram deuvel** en **Bram Hel**, omdat deze berichten de kop vormen van een mogelijk pad over de Markovketen.

3.4 Pad

Het ligt vrij voor de hand hoe men een pad kan lopen over een Markovketen. Wat we in dit geval echter willen toepassen, is dat we de pseudowillekeurige generator P combineren met onze Markovketen M om een programma te bouwen waarmee we een databron D kunnen genereren.

Met de manier waarop we de toestandsruimte I hebben gedefinieerd, kunnen we een volgorde definiëren over een toestandsruimte I van de Markovpakket op basis van de index die elke toestand uniek maakt. Met deze volgorde gaan we de getallen die onze

pseudowillekeurige generator voortbrengt, afbeelden op een pad over de Markovketen M. We definiëren dat het pad begint bij het eerste element van I. Vervolgens gaat het een pad lopen over M.

Voor de op de Markovketen volgende punten worden de kansen verdeeld worden over de uitkomsten van het i-de getal van de pseudowillekeurige generator $P(n)$. Met andere woorden,

$$p_{\alpha+1} = i_k \text{ indien } \sum_{j=1}^{k-1} \lambda_{ij} < P_\alpha(n) \leq \sum_{j=1}^{k} \lambda_{ij}, \tag{3.7}$$

waarbij $P_i(n)$ het i-de element van $P(n)$ is, i_k het k-de element van I en λ_{ij} de kans dat men van toestand i naar toestand j springt in de Markovketen.

Ik zal hier een kort voorbeeld geven. Neem de symbolenverzameling $\Omega = \{1, 2, 3, 4\}$ met de triviale toestandsruimte $I = \{(1), (2), (3), (4)\}$. Stel dat er geldt voor de Markovketen M dat

$$\Lambda = \begin{pmatrix} 0.25 & 0.5 & 0 & 0.25 \\ 0.5 & 0 & 0.4 & 0.1 \\ 0 & 0.05 & 0.34 & 0.61 \\ 0.01 & 0 & 0.98 & 0.01 \end{pmatrix},$$

als transitiematrix op M, en stel dat we een pad willen lopen met een pseudowillekeurige generator waarvoor geldt dat

$$P(1) = (0.7567252, 0.0216422, 0.2967996, 0.5974098, 0.8150187, \dots). \tag{3.8}$$

Dan zien we dat

$$\begin{aligned} p_1 &= (1) \text{ per definitie,} \\ p_2 &= (4), \text{ want } 0.75 < 0.7567252 \leq 1, \\ p_3 &= (3), \text{ want } 0.01 < 0.0216422 \leq 0.99, \\ p_4 &= (3), \text{ want } 0.05 < 0.2967996 \leq 0.39, \\ p_5 &= (4), \text{ want } 0.39 < 0.5974098 \leq 1, \\ p_6 &= (3), \text{ want } 0.01 < 0.8150187 \leq 0.99, \\ &\dots \end{aligned}$$

Op deze manier kunnen we de pseudowillekeurige generator P afbeelden op een pad over de Markovketen M. Het enige wat we nog nodig hebben, is een natuurlijk getal $n \in \mathbb{N}$. Gegeven de Markovketen M en pseudowillekeurige generator P kunnen we dus elk natuurlijk getal afbeelden op een pad over de M.

Beschouw nu een pad over de Markovketen M wat we met $P(n)$ genereren. Dan zeggen we voor een gegeven reeks R dat

$$P(n) \xrightarrow{M} R \text{ indien } R \text{ de kop van } P \text{ vormt.} \tag{3.9}$$

3.5 Seed

Om ons algoritme te laten werken, willen we een getal vinden waarvoor de pseudowillekeurige generator en de Markovketen gezamenlijk onze databron genereren. In de praktijk zal dit een kwestie van *trial and error* zijn: men gaat elk getal vanaf 0 langs om te controleren of de databron D wordt gegenereerd.

Zij P een pseudowillekeurige generator. Dan zeggen we dat S de *seed* van D op M is indien

$$S = \min\left\{n \in \mathbb{N} \mid P(n) \xrightarrow{M} D\right\} \tag{3.10}$$

We spreken er hier van als $P(S)$ databron D *genereert*.

We willen graag de waarde van S schatten. Aangezien P (pseudo)willekeurig is, is S een waarde die niet exact te berekenen valt. We kunnen echter wel de verwachtingswaarde berekenen. Stel dat de Markovketen een kans $u \in [0, 1]$ heeft om een pad te lopen over M waarvan D de kop vormt.

Aangezien de verdeling van de kans of de kop van een willekeurig pad gelijk aan D is, Bernoulli verdeeld is, geldt er dat

$$\begin{aligned}
\mathbb{E}(S) &= \sum_{k=1}^{\infty} k\mathbb{P}(S = k) \\
&= \sum_{k=1}^{\infty} \left(k\mathbb{P}\left(P(k) \xrightarrow{M} D\right) \prod_{i=1}^{k-1} \mathbb{P}\left(\neg\left(P(i) \xrightarrow{M} D\right)\right)\right) \\
&= \sum_{k=1}^{\infty} ku(1-u)^{k-1} = \frac{1}{u}.
\end{aligned}$$

Men herkent op de laatste regel de geometrische verdeling, welke uitgerekend kan worden en dan gelijk aan $\frac{1}{u}$ blijkt. Uit deze afleiding volgt dus dat de grootte van de *seed* direct verbonden is met de waarschijnlijkheid dat men per toeval een juist pad loopt over de Markovketen.

Als we willen weten hoe goed dit algoritme werkt, is het echter niet per se relevant wat de waarde van S zal zijn; we willen enkel een schatting van S hebben om te bepalen hoe complex ons programma zal zijn. Hiervoor definiëren we een *lengtefunctie*. In essentie rekent deze functie uit hoeveel bits onze seed nodig zou hebben als deze zou worden opgeslagen in binaire code. Om deze netjes te definiëren, noteren we L als volgt:

$$L : \mathbb{N} \to \mathbb{N} : k \mapsto \lfloor \log k \rfloor, \tag{3.11}$$

met als bijzonder geval dat $L(0) = 1$. Met deze functie kunnen we de lengte van S bepalen.

3.6 Kolmogorovcomplexiteit

In principe is de Kolmogorovcomplexiteit een term die eerder genoemd is, maar met deze nieuwe definities kan de Kolmogorovcomplexiteit van het algoritme al worden vastegelegd. We bouwen de databron D immers met de pseudowillekeurige generator P,

de Markovketen M en de seed S. We kunnen aannemen dat P statisch zal zijn, dus deze hoeft geen onderdeel van de boodschap te zijn als we onze databrond D willen doorgeven. Wel moeten we dus de Markovketen M en onze seed S doorgeven. De lengte van de seed is gedefinieerd als $L(S)$. De hoeveelheid ruimte die de Markovketen kost, hangt af van onze Markovketen M: zo hebben we $L(I)$ bits nodig om alle toestand in bits over te dragen. (Bijv. als $I = \{01, 0, 1\}$, dan hebben we ongeveer 4 bits nodig.) Verder moeten we een gebroken getal van lengte $f \in \mathbb{N}$ overdragen voor elke verbinding, dus dat zijn $k^2 f$ bits, aannemende dat I in totaal k elementen bevat. Dit levert ons dus op dat

$$K_P(D) = L(M) + L(S) = L(I) + L(S) + k^2 f. \tag{3.12}$$

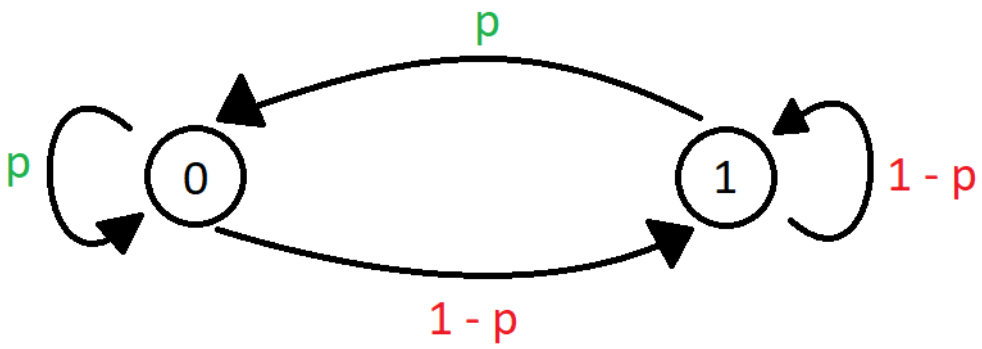
Intuïtief levert ons dit dus een afweging op. We willen graag zo min mogelijk in onze Markovketen opslaan, maar dat levert de consequentie op dat S een groot getal wordt. We kunnen deze klein houden, maar dit betekent weer dat de Markoketen heel groot wordt.

4 Beschouwing Markovketens

In het vorige hoofdstuk hebben we alle termen gedefinieerd. In dit hoofdstuk bekijken we enkele voorbeelden waarmee data zou kunnen worden gecomprimeerd. Sommige zullen algemene modellen zijn, enkele zullen specifieke scenario's zijn.

4.1 De binaire Markovketen

In dit simpele voorbeeld nemen we een zeer simpele Markovketen, en bekijken hiervan de lengte van de seed. Beschouw een databron D op de verzameling $\Omega = \{0, 1\}$, en neem een Markovketen M met toestandsruimte $I = \{(1, 0), (2, 1)\}$. Neem zonder verlies van algemeenheid aan dat D lengte $n+1$ heeft en begint met een 0. Stel dat van de volgende n karakters er k een 0 zijn en $n - k$ een 1.



Figuur 4.1: Een simpele Markovketen met 0 en 1 als toestanden. Vanuit beide toestanden is er een kans p dat men naar 0 springt, en een kans van $1 - p$ dat men naar 1 springt.

We kunnen nu het algoritme op een "slimme" manier bouwen. In plaats van dat we onze p direct halen van de waardes $\frac{k}{n}$ en $\frac{n-k}{n}$, inspecteren we voor elke $n \in \mathbb{N}$ welke getallen het algoritme P ons oplevert. Op basis daarvan kunnen we de grens verleggen zodat de Markovketen de juiste waardes genereert.

Neem als voorbeeld de databron $D = 0011010100$, en stel dat voor een zekere $j \in \mathbb{N}$ geldt dat

$$P(j) = (0.18342, 0.89678, 0.75301, 0.28454, 0.48854, 0.08603, 0.78775, 0.13326, 0.34409, \dots).$$

Als $P(j)$ deze getallen produceert, dan kunnen we $p = 0.4$ kiezen, opdat $P(j) \xrightarrow{M} D$. In tabel 4.1 gaan we na dat met deze parameter de databron inderdaad gegenereerd wordt.

-	0
0.18342,	0
0.89678,	1
0.75301,	1
0.28454,	0
0.48854,	1
0.08603,	0
0.78775,	1
0.13326,	0
0.34409,	0
...	x

Tabel 4.1: In deze tabel gaan we na dat we de databron $D = 0011010100$ correct genereren als we naar toestand 0 springen als het getal kleiner is dan 0.4, en dat we naar toestand 1 springen als het getal groter is dan 0.4. Het eerste getal heeft geen waarde naast zich staan, omdat we per definitie ervan uitgaan dat we bij toestand 0 beginnen en van daaruit verder bewegen door de Markovketen.

Op deze manier wordt het makkelijker om het goede pad te vinden: de getallen waar we naar toestand 0 springen, hoeven enkel lager te zijn dan de toestanden waar we naar 1 springen. Aangezien alle getallen uniform verdeeld zijn, heeft elke combinatie van k waardes een gelijke kans om de laagste k getallen te zijn. Hierdoor is de kans dat de juiste k getallen het laagste zijn, gelijk aan $\frac{1}{\binom{n}{k}}$.

De kans dat we het juiste pad over de Markovketen lopen is bekend, en hieruit kunnen we nu afleiden dat $\mathbb{E}(S) = \binom{n}{k}$. Echter willen we graag de verwachting van de lengte van S weten, dus we willen graag $\mathbb{E}(L(S))$ weten. Deze blijkt echter nog lastig te zijn om te berekenen, zeker omdat de lengtefunctie een vrij lastig te berekenen functie is. Om deze reden gaan we een afschatting maken.

Het blijkt dat $\mathbb{E}(L(S))$ lastig te berekenen is, maar $\mathbb{E}(S)$ is al bekend. Het zou dus fijn zijn als we de verwachte lengte kunnen schrijven in termen van $\mathbb{E}(S)$. Dit blijkt ook het geval te zijn, want we kunnen hiervoor Jensens ongelijkheid toepassen.

De wetenschapper J. Jensen toonde namelijk aan in een door hem geschreven artikel [Jensen, 1906] dat voor een willekeurige stochast B en convexe functie ϕ geldt dat

$$\mathbb{E}\left(\phi(B)\right) \geq \phi\left(\mathbb{E}(B)\right). \tag{4.1}$$

We kunnen deze stelling ook gebruiken om aan te tonen dat voor een concave functie ψ geldt dat $\mathbb{E}(\psi(B)) \leq \psi(\mathbb{E}(B))$. Definieer hiervoor $\phi \coloneqq -\psi$. In dit geval is ϕ dan convex, dus er geldt dat

$$-\mathbb{E}\left(\psi(B)\right) = \mathbb{E}\left(\phi(B)\right) \geq \phi\left(\mathbb{E}(B)\right) = -\psi\left(\mathbb{E}(B)\right), \tag{4.2}$$

waaruit volgt dat $\mathbb{E}\left(\psi(B)\right) \leq \psi\left(\mathbb{E}(B)\right)$. De lengtefunctie L is concaaf, dus we zien hiermee dat de verwachte lengte kan worden afgeschat op de lengte van de verwachting.

Naast Jensens ongelijkheid gebruiken we in de afleiding ook Stirlings benadering, welke zegt dat $\log n! = \log n - n + O(\log n)$. [Wikimedia, 2020c] Hiermee rekenen we dan uit dat

$$\begin{aligned}
L\left(\mathbb{E}(S)\right) &= L\left(\frac{1}{\binom{n}{k}}\right) = L\left(\frac{k!(n-k)!}{n!}\right) = \left\lfloor \log\left(\frac{k!(n-k)!}{n!}\right)\right\rfloor + 1 \\
&\leq \log\left(\frac{k!(n-k)!}{n!} + 1\right) + 1 = \log\left(k!\right) + \log\left(n-k\right)! - \log n! + 1 \\
&= e + 1 + \left(k + \frac{1}{2}\right)\log\left(k\right) + \left((n-k) + \frac{1}{2}\right)\log\left(n-k\right) - \left(n + \frac{1}{2}\right)\log\left(n\right) + O(\log\left(n\right)) \\
&= \left(k\log\left(k\right) + (n-k)\log\left(n-k\right) - n\log\left(n\right)\right) + O(\log\left(n\right)) \\
&= n\left(\frac{k}{n}\log\left(k\right) + \frac{n-k}{n}\log\left(n-k\right) - \log\left(n\right)\right) + O(\log\left(n\right)) \\
&= n\left(\frac{k}{n}\log\left(\frac{k}{n}\right) + \frac{n-k}{n}\log\left(\frac{n-k}{n}\right)\right) + O(\log\left(n\right)) \\
&= nH(D) + O(\log\left(n\right)).
\end{aligned}$$

Volgens Jensens ongelijkheid volgt er dan dat $\mathbb{E}\left(L(S)\right) \leq nH(D) + O\left(\log\left(n\right)\right)$.

Voor een binaire Markovketen geldt er dus dat

$$\begin{aligned}
\mathbb{E}\left(K_P(D)\right) &= L(M) + \mathbb{E}\left(L(S)\right) = L(I) + 4f + \mathbb{E}\left(L(S)\right) \\
&\leq 2 + 4f + nH(D) + O\left(\log\left(n\right)\right) \\
&= nH(D) + O(\log n).
\end{aligned}$$

Wat betekent dit? Dit betekent dat er voor een databron van lengte n geldt dat

$$\lim_{n\to\infty} \frac{\mathbb{E}(K_P(D))}{n} = \frac{nH(D) + O(\log\left(n\right))}{n} = H(D),$$

wat laat zien dat we voor grote berichten dicht komen bij de ondergrens die door formule 2.2 is gesteld.

4.2 Dubbele Markovketen

In de vorige paragraaf hebben we een Markovketen met de symbolen 0 en 1 bekeken. Het was extreem versimpeld, doordat er slechts twee toestanden waren en deze toestanden ook dezelfde overgangskansen hadden. In deze paragraaf maken we de Markovketen complexer door een Markovketen met de vier toestanden 00, 01, 10 en 11 te bekijken. Bovendien krijgt elke toestand eigen kansverdelingen over van welke toestand naar welke toestand kan worden gesprongen.

We gaan deze paragraaf ook gebruiken als opzet om deze Markovketen uit te breiden naar een algemeen voorbeeld van n symbolen op een Markovketen. Beschouw voor nu de Markovketen M met toestandenruimte $I = \{(0,0),(0,1),(1,0),(1,1)\}$. Zoals we in de

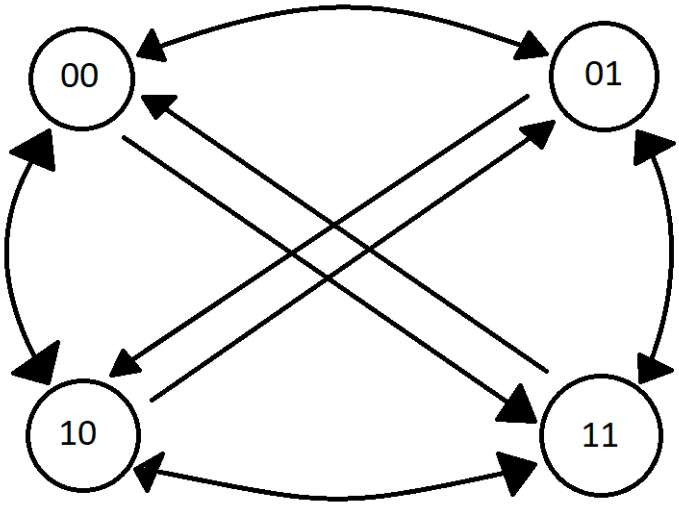
	00	01	10	11
00	0	2	0	1
01	0	1	2	1
10	1	0	0	0
11	1	1	0	0

Tabel 4.2: Tabel waarin het aantal overgangen van elke toestand naar elke volgende toestand staan uitgerekend. In $D = 00\,11\,00\,01\,01\,11\,01\,10\,00\,01\,10$ wordt er bijv. twee keer van 00 naar 01 gesprongen, maar van 00 wordt er geen enkele keer naar 10 gesprongen.

vorige paragraaf hebben gezien, loont het om niet per se de verhouding van de symbolen te gebruiken als parameters in onze Markovketen, maar is er een geldige reden om de getallen te bekijken die de pseudowillekeurige generator P genereert. Als we hiervoor willen bepalen hoe efficiënt deze aanpak is, loont het om de databron D te indexeren in een overgangsmatrix.

We gebruiken weer een voorbeeld om te laten zien hoe deze aanpak werkt. Neem nu de databron $D = 00110001011101100000110$. Als we de overgangen tellen in een matrix, dan krijgen we wat in tabel 4.4 is beschreven.

We kunnen vervolgens op dezelfde manier per toestand proberen te regelen dat de overstapsgrenzen goed liggen: bij 10 is dit bijvoorbeeld heel gemakkelijk: aangezien er vanaf 10 enkel één keer naar 00 wordt gesprongen, kunnen we de overgang van 10 naar 00 met kans 1 laten gebeuren. Bij toestand 00 moeten we dan weer hopen dat de waarde bij de overgang van 00 naar 11 het hoogst is, dan klopt de rest. Op deze manier kunnen we de Markovketen verder onderzoeken.



Figuur 4.2: Een Markovketen met de toestanden 00, 01, 10 en 11. Deze keten bevat meer punten, maar kan daardoor met grotere kans de databron genereren. Dit levert daarom zeer waarschijnlijk een kleinere seed op.

Als we de overgangaantallen veralgemeniseren naar de matrix $A = (a_{ij})$, waarbij a_{ij} het aantal overgangen van i naar j representeert, dan kunnen we stellen dat we vanaf punt i een juiste Markovketen hebben ontwikkeld indien de volgorde goed gekozen is. Dit kunnen we op dezelfde manier doen als bij de binaire Markovketen, wat betekent we we vanaf toestand i het aantal mogelijke configuraties gelijk is aan

$$\frac{(a_{i00} + a_{i01} + a_{i10} + a_{i11})!}{a_{i00}!a_{i01}!a_{i10}!a_{i11}!}.$$

Dit betekent dus dat $L(\mathbb{E}(S))$ het product is van het aantal mogelijkheden uit deze

staten. Als we $A_i \coloneqq \sum_{j \in I} a_{ij}$ definiëren, dan geeft dit ons dat

$$
\begin{aligned}
L\left(\mathbb{E}(S)\right) &= L\left(\prod_{i \in I}\left(\frac{\left(\sum_{j \in I} a_{ij}\right)!}{\prod_{j \in I} a_{ij}!}\right)\right) = \left\lfloor \log\left(\prod_{i \in I}\left(\frac{\left(\sum_{j \in I} a_{ij}\right)!}{\prod_{j \in I} a_{ij}!}\right)\right)\right\rfloor + 1 \\
&\leq \sum_{i \in I}\left(\log\left(\left(\sum_{j \in I} a_{ij}\right)!\right) - \sum_{j \in I} \log\left(a_{ij}!\right)\right) + 1 \\
&= \sum_{i \in I}\left(\log A_i!\right) - \sum_{i \in I}\sum_{j \in I}\left(\log\left(a_{ij}!\right)\right) + 1 \\
&= \left(\sum_{i \in I} A_i \log\left(A_i\right) - A_i + O\left(\log\left(A_i\right)\right)\right) - \sum_{i \in I}\sum_{j \in I}\left(a_{ij}\log\left(a_{ij}\right) - a_{ij} + O\left(\log\left(a_{ij}\right)\right)\right) \\
&= \left(\sum_{i \in I} A_i \log\left(A_i\right)\right) - \left(\sum_{i \in I} A_i\right) + O\left(\log\left(n\right)\right) + \left(\sum_{i \in I}\sum_{j \in I} a_{ij}\right) - \left(\sum_{i \in I}\sum_{j \in I} a_{ij}\log\left(a_{ij}\right)\right) \\
&= \sum_{i \in I}\left(A_i \log\left(A_i\right) - \sum_{j \in I}\left(a_{ij}\log\left(a_{ij}\right)\right)\right) - n + n + O\left(\log\left(n\right)\right) \\
&= \sum_{i \in I}\sum_{j \in I} -\left(\log a_{ij} - \log A_i\right) + O\left(\log\left(n\right)\right) = \sum_{i \in I}\sum_{j \in I}\left(-a_{ij}\log\left(\frac{a_{ij}}{A_i}\right)\right) + O\left(\log\left(n\right)\right) \\
&= \sum_{i \in I} A_i H(D_i) + O\left(\log\left(n\right)\right),
\end{aligned}
$$

waarbij $H(D_i)$ voor de entropie voor de data die op toestand i volgt, representeert. Als deze zeer eenduidig is, (zoals bij toestand 10 in tabel 4.4) dan zal deze erg laag zijn, terwijl deze heel hoog zal zijn voor punten waar afhankelijkheid aanwezig is.

Dit laatste is waar het algoritme zo sterk in kan zijn. Zoals we zometeen zullen aantonen, maakt het qua efficiëntie niet veel uit of de toestanden uit één of twee staten bestaan. Waar het algoritme voornamelijk sterk in is, is in het comprimeren van onderlinge afhankelijkheden in een databron. Als na een toestand 01 vaak 00 volgt, dan kan deze vrij makkelijk worden verstopt in de Markovketen.

Dit kan in de praktijk heel makkelijk zijn. In HTML zit er bijvoorbeeld een duidelijke structuur die goed in een Markovketen kan worden verwerkt. Hier zitten veel afhankelijkheden in verwerkt die makkelijk ertussenuit kunnen worden gepikt.

4.3 Vergelijking tussen de enkele en dubbele keten

In deze paragraaf vergelijken we het algoritme van dubbele binaire getallen met de enkele binaire Markovketen. Neem een scenario waarin D van arbitrair hoge lengte $2 * n$ met $n \in \mathbb{N}$ is, waarbij elke toestand een gelijkwaardige kans om naar elke andere staat kan

springen. Dan geldt er voor de binaire Markovketen dat

$$\mathbb{E}\left(L(S)\right) \leq 2nH(D) + O\left(\log\left(n\right)\right) = 2n + O\left(\log\left(n\right)\right), \tag{4.3}$$

terwijl er voor de dubbele Markovketen geldt dat

$$\mathbb{E}\left(L(S)\right) \leq \sum_{i \in I} \left(\frac{1}{4} nH(D_i)\right) + O\left(\log\left(n\right)\right) = 2n + O\left(\log\left(n\right)\right). \tag{4.4}$$

Merk op dat de entropie groter kan zijn bij een hoger aantal mogelijke symbolen. In dit geval geldt er dat $H(D) = 1$ voor de binaire Markovketen, maar $H(D_i) = 2$ voor de dubbele Markovketen. Dit is weliswaar een *worst-case scenario*, maar het laat goed zien hoe de keus van het aantal nodes een logaritmisch verschil betekent. Het scheelt pas daadwerkelijk als het lukt om toestanden uit te sluiten, of om afhankelijkheden te vinden en te gebruiken.

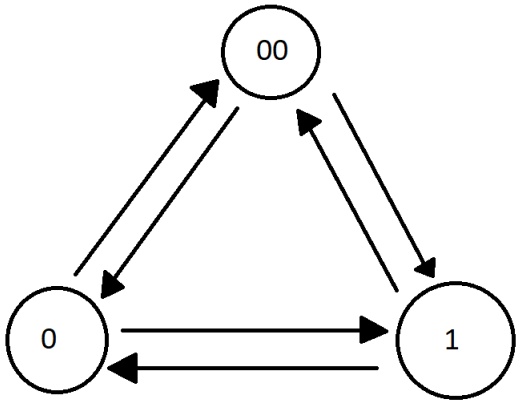
Wat volgt hieruit? Er geldt dus dat de lengte van het algoritme even lang zal zijn, ongeacht van de lengte van de tekens uit de Markovketen.

In de volgende paragraaf maken we gebruik van deze vondst om voor arbitraire symbolen de lengte van het algoritme te bepalen.

4.4 Markovketen met arbitraire lengte

In de vorige twee paragrafen hebben we Markovketens bekeken met toestandsruimtes van binaire waardes van lengte één, maar de berekening voor de Markovketen met lengte 2 was onafhankelijk van het daadwerkelijke aantal of de inhoud van de toestanden. We kunnen dus dezelfde berekening gebruiken om te concluderen voor bijna elke toestandsruimte I dat $\mathbb{E}\left(L(S)\right) \leq \sum_{i \in I} A_i H(D_i) + O(\log\left(n\right))$.

Er is één regel die hierbij belangrijk is om te onthouden, en dat is dat we er tot nu toe vanuit zijn gegaan dat er een uniek pad op de Markovketen bestaat. Dit is geldig wanneer geen $i \in I$ bevat zit in een andere toestand $i \in I$.



Figuur 4.3: Een Markovketen met de toestanden 0, 1 en 00. In deze Markovketen zijn er voor sommige databronnen meerdere paden mogelijk om het correcte resultaat te bereiken. Bijvoorbeeld, de databron $D = 1001$ kan van het pad $1 \to 00 \to 1$ worden gegenereerd, maar kan ook uit $1 \to 0 \to 0 \to 1$ ontstaan.

In zo'n scenario ontstaat namelijk de mogelijkheid dat er op meerdere manieren de databron kan worden gegenereerd. Dit verhoogt namelijk de kans dat men met een gegeven pseudowillekeurige generator de databron genereert. Zolang er echter één uniek pad blijft om de databron te kunnen genereren, blijft de uitspraak dat $\mathbb{E}\left(L(S)\right) \leq \sum_{i \in I} A_i H(D_i) + O(\log(n))$, waar.

Zolang dergelijke toestanden wel in I bevat zijn, zijn de kansen lastiger om te berekenen. Het is mij niet gelukt om deze kansen te berekenen. Mogelijk zal dit de kans kleiner maken, al is mijn vermoeden dat dit voordeel geen significant voordeel zal leveren.

Immers geldt op dit moment voor alle Markovketens, zoals uitgewerkt in vergelijking 3.12, dat

$$\mathbb{E}\left(K_P(D)\right) = L(I) + k^2 f + \mathbb{E}\left(L(S)\right) \leq L(I) + k^2 f + \sum_{i \in I} A_i H\left(D_i\right). \tag{4.5}$$

Dit is wellicht een afschatting, maar volgens de stelling van Brudno, zoals aangetoond in vergelijking 2.2, moet er gelden dat $\lim_{n\to\infty} \frac{K_P(D)}{n} = H(D)$. En aangezien er op dit moment al geldt dat

$$\lim_{n\to\infty} \frac{K_P(D)}{n} \leq \lim_{n\to\infty} \frac{L(I) + k^2 f + \sum_{i \in I} A_i H(D_i)}{n} = H(D), \tag{4.6}$$

zal het verschil miniem zijn. Immers geeft deze bovengreens aan volgens het Squeezelemma [Wikimedia, 2020b] dat ons algoritme al convergeert naar deze grens. Het is desalniettemin een onderwerp waar een vervolgonderzoek op zou kunnen indiepen.

4.5 Toegepast voorbeeld

Om de relevantie van een toepassing te tonen, beschouw de volgende databron die we in het binair willen verzenden:

```
DE_BARBAAR_BARBARARA_AT_RABARBER
```

Als we dit met het standaardcompressiealgoritme zouden werken als beschreven in tabel 4.3, dan zouden we 68 tekens nodig hebben

A	11
B	10
R	01
_	001
D	0001
E	00001
T	00000

Tabel 4.3: Tabel die tekens uit de databron afbeeldt op binaire tekens met hun betekenis.

We kunnen onze tabel echter efficiënt kiezen. We kunnen stellen dat:

x	D	E	_	B	A	R	RA	_AT_
D	0	1	0	0	0	0	0	0
E	0	0	1	0	0	1	0	0
_	0	0	0	2	0	0	0	2
B	0	1	0	0	5	0	0	0
A	0	0	0	0	1	5	1	0
R	0	0	1	3	0	0	0	0
RA	0	0	0	1	0	0	0	1
_AT_	0	0	0	0	0	0	1	0

Tabel 4.4: Tabel die de overgangen van onze Markovketen M uitschrijft. Op basis hiervan kunnen we makkelijk de lengtes berekenen.

We kunnen hier makkelijk zien dat $\mathbb{E}(S) = 1 * 2 * 1 * 6 * 42 * 4 * 2 * 1 = 4032$, want de verwachtingswaarde is de inverse van de kans om het goede pad te lopen, wat op zijn beurt weer de inverse is van het aantal mogelijke paden. Dit geeft ons dat $L\left(\mathbb{E}(S)\right) = L(4032) = 12$. Met andere woorden, waar een dergelijk standaardcompressiealgoritme 68 tekens nodig heeft om een dergelijke tekst te comprimeren, doet deze het met een string van lengte 12! Hier zit uiteraard nog de nuance achter dat de tabel die achter ons algoritme gaat, veel groter. Deze zal nog moeten worden overgedragen. Dit is echter een factor die bij een langer soortgelijk bericht zou wegvallen, en voor grotere berichten irrelevant wordt.

Een ander nuance bij het scenario is dat het scenario ook gekozen is op dat het algoritme hier bijzonder goed zou werken. Er zijn ook veel scenario's te noemen, zoals

het weerbericht, waar symbolen onderling onafhankelijk zouden kunnen zijn. In dit geval verliest het algoritme totaal het voordeel van deze eigenschap.

De reden dat het algoritme desalniettemin zo interessant is, is door de willekeur die achter het algoritme verschuilt. Aangezien de seed geometrisch verdeeld is, is er een redelijke kans dat de seed kleiner zal zijn dan verwacht. Op deze manier geldt er dus voor bepaalde berichten dat deze per toeval compacter kunnen worden opgeslagen dan verwacht. Dit is een scenario waar veel gebruik van kan worden gemaakt.

Een scenario waar dit veelbetekenend kan zijn, is bijvoorbeeld de hoofdpagina van de website van Google. Deze simpele zoekmachinepagina is één die weinig ruimte in beslag behoort te nemen, omdat de pagina duizenden malen per seconde wordt opgevraagd. Één karakter kan daarom al megabytes per seconde schelen om te versturen. Als het pseudowillekeurige algoritme dus een compressie kan leveren van slechts één of twee bytes minder, dan kan dit al een enorme hoeveelheid data schelen en enorme winst opleveren.

5 Conclusie

Zoals Shannon's source coding theorem heeft gesteld, is er een gegeven limiet waar elk exact omkeerbaare algoritme aan zal moeten geloven. Met andere woorden, er zal altijd gelden dat

$$\lim_{n\to\infty} \frac{K_P(D)}{n} = H(D), \tag{5.1}$$

waarbij $K_P(D)$ staat voor de complexiteit van het programma wat onze databron D wil comprimeren, en $H(D)$ staat voor de entropie van databron D. Dit project heeft echter twee interessante bevindingen opgeleverd.

Het algoritme blijkt echter bijzonder sterk is voor databronnen waar de data onderling afhankelijk is. Denk hierbij aan namen, tekst, programmeercode of patronen. Op het moment dat data gecomprimeerd moet worden en er zitten bepaalde patronen in, dan kunnen deze netjes in onze Markovketen worden verwerkt. Het is extra makkelijk om makkelijke patronen in het pad te verwerken, en op die manier verder te komen.

Het blijkt ook dat de lengte van de onderlinge punten op de Markovketen niet per se relevant zijn: een toestandsruimte van $I = \{(1,0),(2,1)\}$ en $I = \{(1,\text{"00"}),(2,\text{"01"}),(3,\text{"10"}),(4,\text{"11"})\}$ leveren nagenoeg dezelfde resultaten op qua compressie, wat veralgemeniseerd kan worden naar elke lengte van de toestanden. Het algoritme wordt specifiek beter op het moment dat tussen de toestanden onderling een lage entropie te vinden is.

Het enige scenario wat niet in deze thesis is onderzocht, is het scenario dat er disjuncte paden op de Markovketen zijn die ieder de databron kunnen vormen. In dit scenario is het bijzonder lastig om de kans op het bewandelen van één van de paden te berekenen. Dit bijzondere geval is daarom een interessant onderwerp voor vervolgonderzoek.

Bibliografie

[don, 2020] (2020). donjon: Markov name generator. https://donjon.bin.sh/name/markov.html.

[Heuvel, 2020] Heuvel, B. v. d. (2020). D&d town generator. http://town.noordstar.me.

[Jensen, 1906] Jensen, J. L. W. V. (1906). Sur les fonctions convexes et les inégualités entre les valeurs Moyennes.

[MacKay, 2003] MacKay, D. J. C. (2003). *Information theory, inference, and learning algorithms*. Cambridge University Press.

[Nandakumar, 2020] Nandakumar, S. (2020). Huffman coding - optimality and issues. https://www.cse.iitk.ac.in/users/satyadev/au17/huffman.pdf.

[Wikimedia, 2020a] Wikimedia (2020a). Kolmogorov randomness. https://en.wikipedia.org/w/index.php?title=Kolmogorov_complexity#Kolmogorov_randomness.

[Wikimedia, 2020b] Wikimedia (2020b). Squeeze theorem. https://en.wikipedia.org/wiki/Squeeze_theorem.

[Wikimedia, 2020c] Wikimedia (2020c). Stirling's approximation. https://en.wikipedia.org/wiki/Stirling%27s_approximation.

Populaire samenvatting

Iedereen die weleens een collectie foto's naar een vriend of familielid heeft willen versturen, heeft het weleens gehad: de bestanden die je wil versturen, zijn te groot en kunnen niet via Whatsapp verstuurd worden. Of misschien zijn het er te veel om één voor één te versturen, en zou je liever één bestand versturen. Het liefst zou je de bestanden kleiner willen maken om ze te kunnen versturen naar je kennis.

De meeste computers kennen het zogenaamde zip-bestand. In een zip-bestand is een klein bestandje waarmee je andere bestanden kunt maken. Zo kun je een verzameling foto's heel compact opslaan in één klein bestandje.

Stel je bijvoorbeeld voor dat je de volgende tekst zo compact mogelijk wil opslaan:

```
A B C A A A B C A A A B C A A
```

Dan kunnen we deze tekst encoderen op de volgende manier. We gaan de letters omzetten, want dat is waar een computer mee werkt.

A	1
B	01
C	001

Tabel 5.1: In deze tabel kun je zien hoe we de tekens gaan vertalen. A komt heel vaak voor, dus deze slaan met maar 1 teken op, terwijl C, die heel weinig voorkomt, met 3 tekens opslaan.

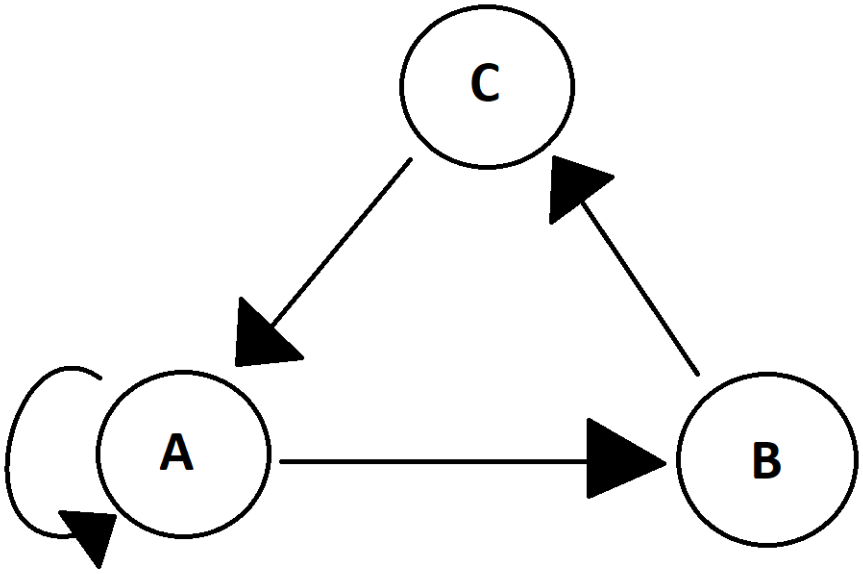
Als je op deze manier de symbolen gaat vertalen, dan krijg je de volgende tekst:

```
101001111010011110100111
```

Ga maar na! We beginnen met een A die een 1 wordt, daarna 01 van de B, dan 001 van de C, daarna weer 1 van de A, enzovoort. Als je dit uitrekent, komt het erop neer dat we de tekst met 24 bits (enen en nullen) kunt opslaan op een computer. Deze methode noemen wij *thermometercodering*, en deze vorm van coderen is één van de simpelste manieren om code te comprimeren.

In dit verslag vroeg ik mij af of er een slimmere methode is om dit soort teksten op te slaan. Als je goed naar de tekst kijkt, zie je namelijk dat er bepaalde patronen in zitten: na een B komt altijd een C, en na een C komt altijd een A! Misschien is er een makkelijkere methode om de tekst op te slaan, mogelijk met een methode waarmee we gebruik maken van deze extra informatie.

Deze lijkt te bestaan! Figuur 5.1 laat een voorbeeld zien van een Markovketen. Dit is een soort graaf waar je telkens stapjes maakt door van het elk teken naar een volgend teken te lopen.



Figuur 5.1: Dit is een Markovketen! Je kunt telkens stapjes maken door de pijltjes te volgen. Zoals je ziet, maken we hier gebruik van de informatie dat na een A een A of B maar nooit een C komt.

We kunnen vervolgens alle mogelijke paden over de ze Markovketen in willekeurige volgorde classificeren, en dan kunnen we op zoek gaan naar het goede pad. Als je gaat uitrekenen hoe groot de kans is dat we het goede pad over deze Markovketen lopen, dan zie je dat je 8 keer een kans van 50% hebt om goed te lopen. Hiermee kom je op een kans van 1 op 256 dat je goedloopt.

In deze thesis gebruiken we een zogenaamd *pseudowillekeurig algoritme*. Je zou kunnen zeggen dat dit algoritme probeert op willekeurige wijze paden op de Markovketen te classificeren, en aan elk pad een getalletje toekent. Ik heb in het verslag vervolgens aangetoond dat het getalletje wat het algoritme aan ons pad zal toekennen, gemiddeld rond de 265 zal liggen. Stel je bijvoorbeeld voor dat het algoritme ons pad als pad nummer 241 classificeert.

In plaats van dat wij dan onze boodschap opslaan, kunnen wij ook ervoor kiezen om de Markovketen op te slaan, en te zeggen dat wij van de Markovketen padnummer 241 willen gebruiken. Als wij dan willen weten wat onze boodschap is, dan hoeven we enkel ons algoritme padnummer 241 af te laten lopen, en dan ontvangen we ons bericht!

Waarom zouden we zo graag een getalletje willen opslaan? Als we het getal 241 naar het binair zouden omzetten, dan krijgen we dat we de volgende code mogen doorgeven:

```
1111 0001
```

Zoals je kunt zien, hebben we dan maar 8 bits nodig om het bericht door te geven. Dit is een stuk compacter dan de thermometercode, welke een totaal van 24 bits nodig heeft!

In deze thesis veralgemeniseer ik het scenario wat ik hier geschetst heb, en probeer ik te laten zien hoe efficiënt dit algoritme in andere scenario's is.