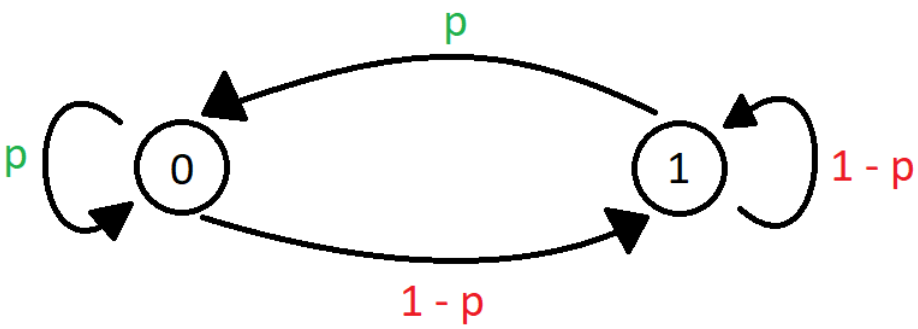
Pseudorandomness on Markov Chains

Bram van den Heuvel

15 mei 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 op basis van willekeur probeert een gegeven informatiebron te comprimeren. Het algoritme waarmee de Markovketen wordt gebouwd, is wellicht een vrij traag algoritme, maar het ontcijferen van de gecomprimeerde data is relatief gemakkelijk.

Deze lange rekenkracht hoopt het algoritme goed te maken met het feit dat het algoritme zeer dicht bij de ultieme ondergrens ligt, en dat het op basis van de willekeur een kans heeft om compacter te zijn dan verwacht, waardoor sommige databronnen efficiënter kunnen worden opgeslagen dan veel algoritmes op deterministische wijze kunnen bereiken.

Het algoritme bouwt een Markovketen, waarover het een pad loopt met een pseudowillekeurig algoritme. Het pseudowillekeurige algoritme wordt gebouwd met een gegeven seed. Het pseudowillekeurige algoritme wordt als statisch beschouwd, terwijl de seed en Markovketen variabel zijn. Beide waardes worden als onderdeel van het bericht meegegeven.

Titel: Pseudorandomness on Markov Chains
Auteur: Bram van den Heuvel, bachelorproject@bram.blmgroep.nl, 11273933
Begeleiding: dhr. dr. J.L. Dorsman, Mr. L.R. van Kreveld MSc
Einddatum: 15 mei 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 pseudowillekeurige website

Gedurende de eerste drie maanden van 2020 heb ik gewerkt aan een persoonlijk project waar ik veel plezier aan heb beleefd. Ik had op het internet een website gevonden die zichzelf een *Dungeons and Dragons Town Generator* noemde. Dit is een tool waarmee een dorp uit een fantasywereld kan worden gegenereerd.

De website had een hoop nuttige informatie waar men veel mee kon uitvoeren, en ik vond de website een fijne tool. Wat echter aan de website ontbrak, was een voldoende hoeveelheid geheugen; om te voorkomen dat de server volgepropt zou komen te zitten met steden die nooit meer bezocht zouden worden, had de server als procedure dat dorpen verwijderd werden als ze 24 uur lang niet meer bezocht werden.

Om deze reden besloot ik een soortgelijke website te maken. Ik ken veel mensen die zo'n website graag zouden willen gebruiken, en daarbij graag hun dorpen ook ná 24 uur weer zouden willen inzien. Ik heb echter geen harde schijf met onbeperkt veel ruimte, dus ik besloot om de website op een slimme manier aan te pakken.

In plaats van de gegenereerde dorpen op te slaan, gebruik ik een zogenaamd *pseudo-willekeurig algoritme* om de dorpen 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 dorp. Wiskundig gezien is een dergelijk algoritme 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 hetzelfde "willekeurige"dorp gegenereerd.

Deze website heb ik gebouwd. [Heuvel, 2020] De website heeft redelijk wat aandacht getrokken op Reddit, en werd vrij populair onder mensen die het spel Dungeons and Dragons spelen.

Wat maakt deze website zo bijzonder? De pseudowillekeurige eigenschap heeft een heel groot voordeel voor de website: het kan voor duizenden internetgebruikers nuttige informatie herinneren en reproduceren, maar het heeft hiervoor geen schijfruimte nodig. De website werkt volledig geheugenloos, en de gebruiker krijgt hun data al meteen terug als ze de juiste dorpsnaam invullen.

Een website die geheugenloos gigabytes aan data kan bewaren - dat klonk als een interessant concept om te onderzoeken. Ik kreeg namelijk het volgende idee: het dorp genereert een namenlijst. Is het mogelijk om een dorpsnaam te vinden wat een namenlijst genereert die voor mij relevant is? Elk dorp genereert willekeurige namen, dus het kan

zijn dat één van de dorpen per toeval mijn familie opsomt in een lijstje. Wat ik dan zou kunnen doen, is dat ik niet de namenlijst van mijn gehele familie opsla, maar dat ik enkel de naam van het dorp opsla waarmee ik mijn gehele stamboom kan genereren. Dit is een compressiealgoritme!

In mijn bachelorthesis heb ik dit idee uitgewerkt naar een werkend algoritme. In plaats van een dorpsnaam in de website in te vullen, gaan we nu echter een pad over een Markovketen lopen en op die manier data proberen te genereren. In dit geval gaat ons pseudowillekeurige algoritme dus een willekeurig pad genereren, en dit willekeurige pad zal op de Markovketen dus een dataset vormen. Ons algoritme zal dan op zoek gaan naar een input *(we noemen zo'n input een seed)* waarmee de voor ons relevante data op de Markovketen zal worden gegenereerd.

1.2 Entropie

Bij dit onderzoek komt een hoop informatietheorie kijken. We gebruiken de informatietheorie om concreet vast te leggen hoe efficiënt het algoritme is, en tot in hoeverre het "beter¨ is dan een ander compressiealgoritme. Een term die in de informatietheorie veel gebruikt wordt, is de term *entropie*. Entropie is de *informatiedichtheid* van een databron. Kort gezegd zou je de informatiedichtheid kunnen beschouwen als de minimale hoeveelheid data die je moet versturen om de informatie over te brengen.

Stel dat een bepaalde databron aan symbolen een informatiedichtheid van 0.75 heeft, dan betekent dat dat je de dataset zodanig kunt comprimeren en compacter kunt maken, dat je per symbool gemiddeld 0.75 bits op een computer nodig hebt om alle informatie te kunnen overdragen.

De entropie van een databron hangt heel erg af van de inhoud. Beschouw namelijk de volgende twee datasets, welke allebei dezelfde symbolen uit hetzelfde alfabet gebruiken.

```
abababababababababababababababababababababababab
3435yofw5yct9esolurlit4ucstmieuloetiuxeitu4locmeul
```

We kunnen een computerprogramma bouwen waarmee we de datasets kunnen reproduceren. De eerste zou met een programma kunnen wat er ongeveer zo uitziet:

```
write "ab" 25 times
```

In plaats van 50 karakters heeft dit programma een opslagruimte van 19 karakters nodig. De andere code is echter lastiger om op te slaan op vergelijkbaar efficiënte wijze, aangezien er niet een merkbaar patroon in lijkt te zitten. Één van de meest efficiënte opties zou hier het volgende programma zijn:

```
write "3435yofw5yct9esolurlit4ucstmieuloetiuxeitu4locmeul"
```

Dit programma heeft echter 58 karakters nodig, dus het neemt meer ruimte in beslag dan de originele databron. Dergelijke bronnen noemen we *Kolmogorovwillekeurig*: als er geen programma bestaat wat minder karakters nodig heeft dan de originele databron om de databron te genereren, dan noemen we de bron Kolmogorovwillekeurig.

2 Wiskundige setting

Het is belangrijk om alle termen duidelijk gedefinieerd te hebben. Daarom zullen we in dit hoofdstuk alle relevante termen wiskundig definiëren als ze meerdere definities kennen, en we introduceren een paar nieuwe termen die in deze thesis voorkomen.

2.1 Bekende termen

De volgende termen zijn algemeen bekend, maar voor de volledigheid worden ze desalniettemin vermeld.

2.1.1 Entropie

Entropie is de *informatiedichtheid* van een informatiebron. Hoe hoger de informatiedichtheid van onze bron, des te hoger de entropie.

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 $H(A)$ van A als volgt:

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

Beschouw het volgende voorbeeld. We nemen een verzameling $X = \{a, b, c\}$ als uitkomstenruimte van de stochast X, waarbij a, b en c kans 0.5, 0.25 en 0.25 respectievelijk hebben als uitkomst voor A. Dit geeft dan dat

$$H(A) = -0.5 \log 0.5 - 0.25 \log 0.25 - 0.25 \log 0.25 = \frac{3}{2}.$$

In deze thesis zullen alle logaritmes altijd basis 2 hebben.

2.1.2 Kolmogorovcomplexiteit

Zij C een computerprogramma wat een gegeven string probeert te comprimeren. Dan noemen we de afbeelding K_C de Kolmogorovcomplexiteit als voor elke string D geldt dat $K_C(D)$ gelijk is aan de lengte van het kleinste programma wat D voortbrengt.

Neem hiervoor de twee strings uit de inleiding als voorbeeld:

```
abababababababababababababababababababababababababab
3435yofw5yct9esolurlit4ucstmieuloetiuxeitu4locmeul
```

In de inleiding hadden we voorbeelden gegeven van programma's die de twee bitstrings voortbrengen. De eerste bitstring kon met een programma van lengte 19 worden opgeslagen. Als dit het kleinste programma is waarmee deze bitstring kan worden gegenereerd, dan zeggen we dat de Kolmogorovcomplexiteit van de ab-string gelijk aan 19 is. Op dezelfde manier zien we dat de complexiteit van de andere string gelijk aan 58.

De Kolmogorovcomplexiteit van een informatiebron heeft nog een speciale relatie met de entropie van die informatiebron: in 1982 schreef A. A. Brudno een artikel waarin hij zegt dat de genormaliseerde Kolmogorovcomplexiteit zou convergeren naar de entropie van de bron. [Brudno, 1982]

Met andere woorden, voor een gegeven bitstring D van lengte n zou er gelden dat

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

2.1.3 Kolmogorovwillekeur

Zoals ook al in de inleiding is vermeld, noemen we een informatiebron *Kolmogorovwillekeurig* op het moment dat de Kolmogorovcomplexiteit groter dan of gelijk aan de grootte van de informatiebron zelf is.

We kunnen aantonen dat er voor elke lengte bitstrings geldt dat er ten minste één bitstring Kolmogorovwillekeurig is. Neem bijvoorbeeld een binaire bitstring van lengte n die we met enen en nullen willen comprimeren. Als we gaan tellen, zien we dat er $2^{n-1} + 2^{n-2} + \cdots + 2^1 + 2^0 = 2^n - 1$ mogelijke bitstrings van een lengte kleiner dan n zijn. [Wikimedia, 2020b]

2.2 Nieuwe termen

De volgende termen kennen meerdere definities, zijn door de auteur geïntroduceerd of zijn anders gedefinieerd voor gemakkelijk gebruik in de thesis.

2.2.1 Databron

Tot nu toe hebben we de termen *bitstring* en *informatiebron* een paar keer losjes voorbij laten komen. Voor de rest van de thesis zal dit allemaal onder één term vallen die hier staat gedefinieerd.

Zij X een verzameling symbolen met een gegeven volgorde. Dan noemen we een eindige rij D een *databron* als

$$D = (s_1, \ldots, s_n), \text{ waarbij } s_i \in X \tag{2.3}$$

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

$$\mathbb{P}(A = x_i) = \frac{1}{n} \sum_{j=1}^{n} 1_{(s_j = x_i)}. \tag{2.4}$$

Met deze definitie zeggen we dan dat

$$H(D) \coloneqq H(A) = -\sum_{x_i \in X} \mathbb{P}\left(A = x_i\right) \log\left(\mathbb{P}\left(A = x_i\right)\right). \tag{2.5}$$

2.2.2 Pseudowillekeurige generator

Zij P een functie die een natuurlijk getal afbeeldt naar een reeks pseudowillekeurige getallen, gegeven door

$$P : \mathbb{N} \mapsto (X_1, X_2, X_3 \dots), \text{ met } X_i \sim \text{unif}(0,1)\ \forall i \in \mathbb{N}. \tag{2.6}$$

Dan heet P een *pseudowillekeurige generator*.

Met een reeks pseudowillekeurige getallen is het mogelijk om een pad te lopen over een Markovketen. Dit definiëren we als volgt.

2.2.3 Markovpakket

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

$$I \subseteq \left\{(d_1, \dots, d_n) \mid d_i \in D, i \in \{1, ..., n\}, n \in \mathbb{N}\right\}. \tag{2.7}$$

Dan heet M een *Markovpakket van* D indien er een pad over M bestaat zodanig dat

$$(d_1, \dots, d_{k_1}) \cup (d_{k_1+1}, \dots, d_{k_2}) \cup \dots \cup (d_{k_{i-1}+1}, \dots, k_i) = (s_1, \dots, s_{k_i}) \supseteq (s_1, \dots, s_n) = D. \tag{2.8}$$

Met andere woorden, er bestaat een pad over M waarvan D de kop vormt.

2.2.4 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.

Ten eerste definiëren we een volgorde op de toestandsruimte I. De volgorde maakt niet veel uit, dus elke arbitraire volgorde is prima. Ik zal een klein voorbeeld geven om aan te tonen dat er een volgorde bestaat, en bovendien is dit een vrij intuïtieve volgorde om te kiezen. Ik heb het over de woordenboekvolgorde: aangezien X een volgorde heeft, kun je de elementen sorteren op basis van hun eerste element: als dit element overeenkomt, kijk je het naar het tweede element, daarna het derde element, etc. Hieronder is een voorbeeld gegeven van de toestandsruimte I van een bepaalde M gesorteerd op volgorde met verzameling $X = \{0, 1\}$.

```
(0)
(0, 0)
(0, 0, 0)
```

```
(0, 0, 0, 0)
(0, 0, 1)
(0, 0, 1, 0, 0, 1, 1)
(0, 1)
(0, 1, 1)
(1)
(1, 0, 0)
(1, 1)
(1, 1, 0, 1)
...
```

Op deze manier kunnen we een volgorde definiëren over een toestandsruimte I van een Markovpakket. 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.

We definiëren dat het eerste element van het pad $p_1 \in I$ het laagste element van I is. Dan zeggen we dat voor de volgende punten de kansen verdeeld worden over de uitkomsten van het i-de getal van de pseudowillekeurige generator $P(n)$. Met andere woorden,

$$p_{i+1} = i_k \text{ indien } \sum_{j=1}^{k-1} \lambda_{ij} < P_i(n) \leq \sum_{j=1}^{k} \lambda_{ij}, \tag{2.9}$$

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 $X = \{1, 2, 3, 4\}$ 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

$$P(1) = (0.7567252, 0.0216422, 0.2967996, 0.5974098, 0.8150187, \dots). \tag{2.10}$$

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

$$P(n) \xrightarrow{M} R \text{ indien } R \text{ de kop van } P \text{ vormt.} \tag{2.11}$$

2.2.5 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

$$S = \min\left\{n \in \mathbb{N} \mid P(n) \xrightarrow{M} D\right\} \tag{2.12}$$

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

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. [Wikimedia, 2020a] 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:

$$L : \mathbb{N} \to \mathbb{N} : k \mapsto \lfloor \log k \rfloor, \tag{2.13}$$

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

2.2.6 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 heeft 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

$$K_P(D) = L(M) + L(S) = L(I) + L(S) + k^2 f. \tag{2.14}$$

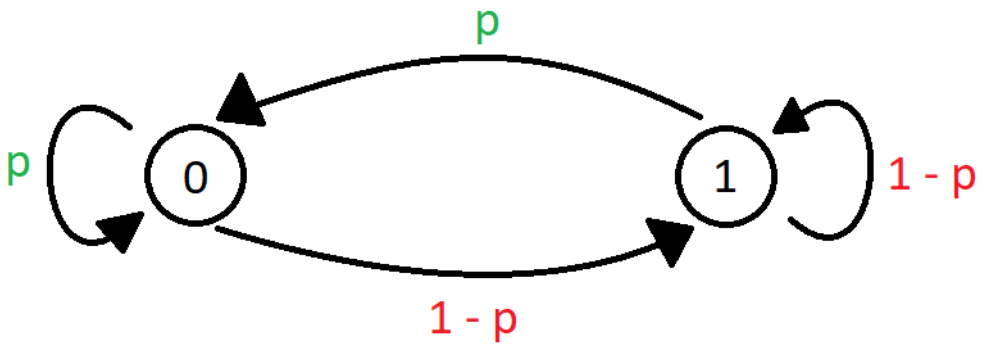
Intuïtief levert ons dit dus een aweging op. We willen graag zo min mogelijk in onze Markovketen bewaren en zo veel mogelijk laten genereren door P op, al levert dit de consequentie op dat S een groot getal wordt. We kunnen deze klein houden, maar dit betekent weer dat de Markoketen heel groot wordt.

3 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 scenarioś zijn doordat het goed vastleggen van afhankelijkheid buiten mijn kunde ligt.

3.1 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 $X = \{0, 1\}$, en neem een Markovketen M met toestandsruimte $I = \{(0), (1)\}$. 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 3.1: 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 waardes direct halen van de waardes $\frac{k}{n}$ en $\frac{n-k}{n}$, inspecteren we voor elke $n \in \mathbb{N}$ welke waardes het 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 $n \in \mathbb{N}$ geldt dat

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

-	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 3.1: 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.

Als $P(n)$ deze getallen produceert, dan kunnen we $p = 0.4$ kiezen, opdat $P(n) \xrightarrow{M} D$. In tabel 3.1 gaan we na dat met deze parameter de databron inderdaad gegenereerd wordt. 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

$$\mathbb{E}\left(\phi(B)\right) \geq \phi\left(\mathbb{E}(B)\right). \tag{3.1}$$

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

$$-\mathbb{E}\left(\psi(B)\right) = \mathbb{E}\left(\phi(B)\right) \geq \phi\left(\mathbb{E}(B)\right) = -\psi\left(\mathbb{E}(B)\right), \tag{3.2}$$

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

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

Volgens Jensens ongelijkheid volgt er dan dat $\mathbb{E}(L(S)) \leq nH(D) + O(\log(n))$. In de afleiding hierboven gebruiken we ook Stirlings benadering, welke zegt dat $\log n! = \log n - n + O(\log n)$. [Wikimedia, 2020d]

Voor een binaire Markovketen geldt er dus dat

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

3.2 Dubbele Markovketen

In het vorige stuk 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.

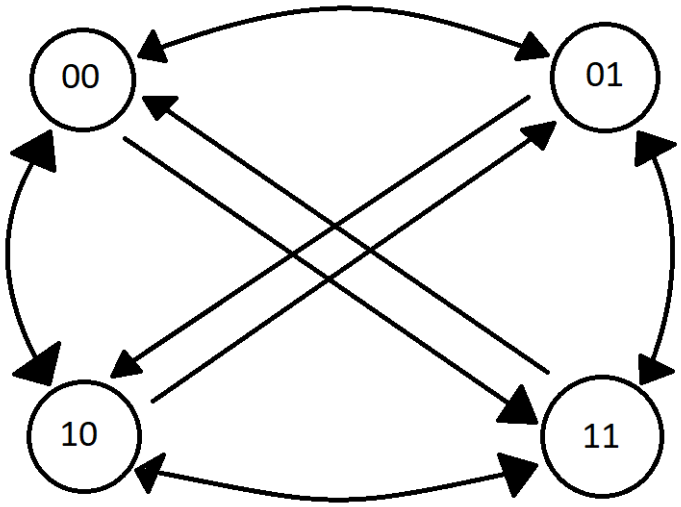
We gaan deze paragraaf ook gebruiken als opzet om deze Markovketen uit te breiden naar een algemeen voorbeeld van n symbolen op een Markovketen. Beschouw voor nu de Markovketen M met toestandenruimte $I = \{(0,0),(0,1),(1,0),(1,1)\}$. Zoals we in de vorige paragraaf hebben gezien, loont het om niet per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

	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 3.2: 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.

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 3.4 is beschreven.

We kunnen vervolgens op dezelfde manier per toestand proberen te regelen dat de overstapsgrenzen goed liggen: bij 10 is dit bijv. heel makkelijk: 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 3.2: 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 $\mathbb{E}\left(L(S)\right)$ 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

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

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 3.4) 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.

3.3 Vergelijking tussen de enkele en dubbele keten

Ik zal nog het algoritme van dubbele binaire getallen vergelijken 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

$$\mathbb{E}\left(L(S)\right) \leq 2nH(D) + O\left(\log\left(n\right)\right) = 2n + O\left(\log\left(n\right)\right), \tag{3.3}$$

terwijl er voor de dubbele Markovketen geldt dat

$$\mathbb{E}\left(L(S)\right) \leq \sum_{i \in I} \left(\frac{1}{4} nH(D_i)\right) + O\left(\log\left(n\right)\right) = 2n + O\left(\log\left(n\right)\right). \tag{3.4}$$

Dit is weliswaar een *worst-case scenario*, maar het laat goed zien hoe de keus van het aantal nodes een verschil van slechts een constante oplevert. Het scheelt pas daadwerkelijk als het lukt om toestanden uit te sluiten, of om afhankelijkheden te vinden en te gebruiken.

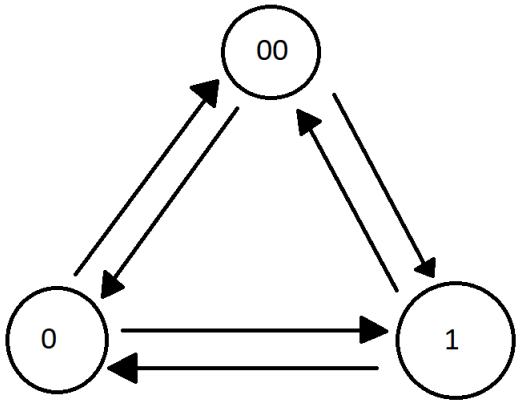
Wat volgt hieruit? Er geldt dus dat de lengte van het algoritme even lang zal zijn, ongeacht van de lengte van de tekens uit de Markovketen.

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

3.4 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 3.3: 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 2.14, dat

$$\mathbb{E}\left(K_P(D)\right) = L(I) + k^2 f + \mathbb{E}\left(L(S)\right) \leq L(I) + k^2 f + \sum_{i \in I} A_i H\left(D_i\right). \tag{3.5}$$

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

$$\lim_{n\to\infty} \frac{K_P(D)}{n} \leq \lim_{n\to\infty} \frac{L(I) + k^2 f + \sum_{i \in I} A_i H(D_i)}{n} = H(D), \tag{3.6}$$

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

3.5 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 3.3, dan zouden we 68 tekens nodig hebben

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

Tabel 3.3: 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 3.4: Tabel die de overgangen van onze Markovketen M uitschrijft. Op basis hiervan kunnen we makkelijk de lengtes berekenen.

We kunnen hier makkelijk zien dat $\mathbb{E}(S) = 1 * 2 * 1 * 6 * 42 * 4 * 2 * 1 = 4032$. Dit geeft ons dat $L(\mathbb{E}(S)) = L(4032) = 12$. 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.

De nuance achter dit geweldige 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 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 aannemelijke 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.

4 Conclusie

Wat aan het begin van het project nog niet bekend was voor mij, was dat er ook al berekend is wat de maximale compressie is die een programma kan leveren. Deze is gelijk aan de entropie maal de lengte van de te comprimeren informatiebron. Met andere woorden, er zal altijd gelden dat

$$\lim_{n\to\infty} \frac{K_P(D)}{n} = H(D), \tag{4.1}$$

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.

Ten eerste blijkt ons algoritme tegen het gewenste maximum aan te zitten! Het algoritme heeft een bovengrens die aan het limiet van formule 4.1 voldoet, waardoor er volgens het Squeezelemma volgt dat ons algoritme convergeert naar de genoemde grens. [Wikimedia, 2020c]

Wat ten tweede ook blijkt, is dat het algoritme bijzonder sterk is voor databronnen waar de data onderling afhankelijk is. 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 $I = \{(0), (1)\}$ en $I = \{(0,0), (0,1), (1,), (1,1)\}$ leveren 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

[Brudno, 1982] Brudno, A. A. (1982). Entropy and the complexity of the trajectories of a dynamic system.

[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.

[Wikimedia, 2020a] Wikimedia (2020a). Geometric distribution. `https://en.wikipedia.org/wiki/Geometric_distribution`.

[Wikimedia, 2020b] Wikimedia (2020b). Kolmogorov complexity. `https://en.wikipedia.org/wiki/Kolmogorov_complexity`.

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

[Wikimedia, 2020d] Wikimedia (2020d). 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 4.1: 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.

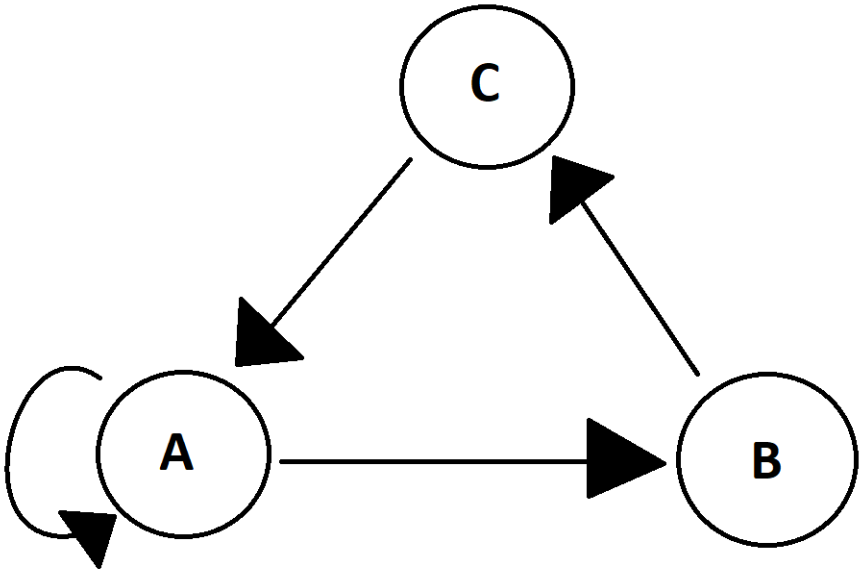
Als je op deze manier de symbolen gaat vertalen, dan krijg je de volgende tekst:

```
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.

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 4.1 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 4.1: 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 heb ik aangetoond dat je dan gemiddeld 256 keer moet proberen een pad te lopen voor je goed loopt. We kunnen het algoritme dan opslaan en het getal 256 meegeven aan de ontvanger van ons bericht: zij weten dan ook dat zij met hetzelfde algoritme 256 moeten proberen voor zij het goede pad lopen. We geven dan het getal 255 door, want we beginnen met tellen vanaf 0, dus 255 is het 256-e getal.

Als we dit zouden omzetten naar het binair, dan krijgen we dat we de volgende code mogen doorgeven:

```
1111 1111
```

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

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.