Analyse van Huffman compressie-algoritmes

Alexander Van Dyck

7 december 2017

Inhoudsopgave

1 Inleiding

In dit verslag worden vijf verschillende Huffman algoritmes beschreven en vergeleken volgens compressieratio en snelheid. Elk algoritme heeft zijn voor- en nadelen, maar toch zullen we in staat zijn een definitieve winnaar te selecteren voor heel veel van de gevallen.
We nemen gedurende de loop van dit verslag aan dat een symbool 1 byte (8 bits) groot is. De berekeningen worden daardoor vereenvoudigd. Een andere symboolgrootte zal wellicht andere resultaten leveren, maar de berekeningen zijn volledig analoog.

2 Algoritmes

2.1 Standaard Huffman

Het standaard Huffman algoritme zal, in tegenstelling tot adaptieve algoritmes, geen wijzigingen maken aan de configuratie van de Huffman boom tijdens het en-/decoderen van een tekst. Op het eerste zicht is dit een nadeel. We zien echter dat, dankzij de statische natuur van de boom, we aan enorm veel preprocessing kunnen doen. We zullen later zien dat dit hierdoor sneller is dan een adaptief algoritme.

2.1.1 Opstellen boom

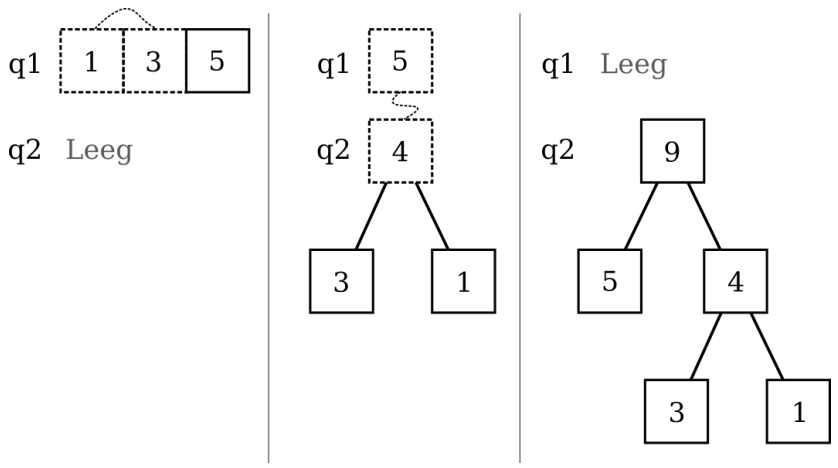
Een welbekende manier om een Huffmanboom op te stellen, is door middel van een prioriteitswachtlijn, waarbij we telkens de twee toppen met het laagste gewicht mergen en terug gesorteerd toevoegen. Hier wordt echter een alternatieve methode uitgewerkt die gebruik maakt van 2 wachtlijnen.

Algoritme We stellen twee wachtlijnen op: $q1$ en $q2$. $q1$ zal initieel een gesorteerde lijst van de bladeren voorstellen (oplopend gesorteerd volgens gewicht). $q2$ is initieel leeg.
In elke stap van het algoritme zullen we de twee toppen met het laagste gewicht van zowel $q1$ en $q2$ uit de wachtlijnen halen, mergen en achteraan $q2$ toevoegen. Het algoritme stopt als er maar 1 top in $\{q1 \cup q2\}$ zit. Deze top is dan de wortel.

Complexiteit We merken al snel op dat dit de complexiteit van dit algoritme $\theta(n)$ is, met n het aantal toppen in de Huffmanboom.

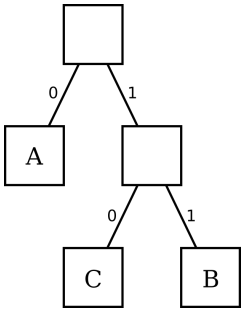
2.1.2 Encoder

Een naïeve implementatie bij het encoderen zou zijn om telkens wanneer een symbool wordt ingelezen, de Huffmanboom van de wortel tot het gewenste blad af te gaan om de code te achterhalen. We kunnen echter een bijzonder simpel algoritme gebruiken die een symbool rechtstreeks op zijn code zal afbeelden.



Figuur 1: Een voorbeeld van het opstellen van een Huffmanboom met twee queues. De labels zijn de gewichten van de toppen.

Algoritme We berekenen voor elk symbool wat we moeten uitschrijven, en slaan dit op. Als we dan een symbool moeten encoderen, hoeven enkel de code voor dit symbool op te zoeken. Deze techniek haalt een constante tijd, en we moeten niet veel meer data opslaan tegenover een naïeve implementatie.



Symbool	Code
A	0
B	11
C	10

Figuur 2: In dit voorbeeld wordt de Huffmanboom in een dergelijke tabel omgezet. Bij het encoderen van de string “ABC” zal het algoritme dus 0 11 10 uitschrijven.

Complexiteit Bij de naïeve implementatie moeten we voor elk symbool een pad van de wortel tot een blad doorlopen. Een gemiddelde operatie is dus $O(d)$ met d de diepte van de Huffmanboom. Het ander algoritme zal per symbool slechts 1 opzoeking per symbool moeten doen, waardoor een gemiddelde operatie $O(1)$ is, bovendien met een zeer goede constante.

Geheugengebruik De maximale diepte van een Huffmanboom is 256. Aangezien de lengte van een code variabel is, moeten we deze ook bijhouden. Het totaal aantal geheugen dat maximaal bijgehouden moet worden (in bytes) is

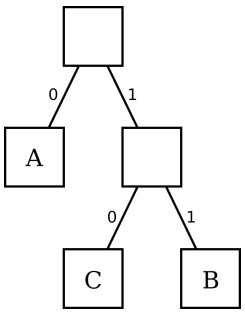
$$256\left(\frac{256}{8}+\frac{\log_2 256}{8}\right)=8448$$

Dit is een relatief kleine hoeveelheid.

2.1.3 Decoder

Een eenvoudig algoritme is om telkens 1 bit in te lezen en zo de Huffmanboom te traverseren. In theorie is dit al een efficiënte manier om te decoderen. We kunnen dit algoritme verbeteren door meerdere bits tegelijk in te lezen.

Een eerste algoritme Stel dat we telkens n bits inlezen. Een prefix van die n bits beschrijft dan een pad naar een blad van de Huffmanboom. We kunnen de waarde van dat blad (het symbool dat we moeten uitschrijven) en het aantal gebruikte bits (de padlengte) opslaan.
Nadat we dit voor alle mogelijke permutaties van n bits hebben opgeslagen, kunnen we de geëncodeerde tekst overlopen. We lezen dan telkens l nieuwe bits in (met l het aantal gebruikte bits van de vorige gelezen code), en concateneren vervolgens deze bits met de laatste $n-l$ (ongebruikte) bits van de vorige keer.



Code	Symbool	#Bits
00	A	1
01	A	1
10	C	2
11	B	2

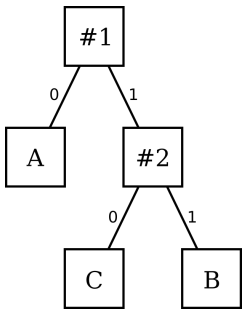
Figuur 3: Voorbeeld voor $n = 2$.
Bij "0010" zal het algoritme "A A C" uitschrijven en zal 2+1+1 bits inlezen.

Er is echter nog een addertje onder het gras. Indien er een pad naar een blad bestaat dat langer is dan n (het aantal bits dat we inlezen), dan kunnen we dit algoritme niet zomaar toepassen.
Wat we dan kunnen doen is (in de plaats van een symbool) een pointer naar een top in de Huffmanboom opslaan. Deze pointer zal wijzen naar waar het eenvoudige algoritme zou terecht gekomen zijn na het inlezen van n bits. We kunnen dan het eenvoudige algoritme gebruiken om tot een blad te geraken (en dus terug naar de

wortel te kunnen gaan).
Aangezien symbolen die weinig voorkomen in de tekst zich op een lage diepte bevinden, zal dit addertje niet veel invloed hebben op de uitvoeringstijd (bij voldoende grote n).

Conclusie eerste algoritme Het grote nadeel aan dit algoritme is dat we telkens een dynamisch aantal bits moeten inlezen van de inputstroom (afhankelijk van ons laatst bezochte blad). Dit is niet enkel moeilijk om efficiënt te implementeren, maar zal bovendien maximaal 1 symbool als output genereren.
Om deze redenen wordt een ander algoritme voorgesteld, die deze problemen oplost en voortbouwt op dit idee.

Een tweede algoritme Deze keer lezen we telkens een statisch aantal bits in. We slaan dan voor elk niet-blad een structuur op die bijhoudt wat de simpele encoder zou doen na het inlezen van die n bits.
Dit houdt twee dingen in: enerzijds de symbolen die moeten worden uitgeschreven, en anderzijds de volgende top in de Huffmanboom.
Als we dan de tekst inlezen, moeten we voor de top waar we ons bevinden de corresponderende symbolen uitschrijven en ons voortbewegen naar de volgende top.



#1

Code	#Bytes	Next
00	AA	#1
01	A	#2
10	C	#1
11	B	#1

#2

Code	#Bytes	Next
00	CA	#1
01	C	#2
10	BA	#1
11	B	#2

Figuur 4: Voorbeeld voor $n = 2$. Bij het decoderen van 01 11 00 zullen we dus “A B CA” uitschrijven en de toppen 1-2-2-1 overlopen.

Geheugengebruik De lengte van het aantal bytes die we kunnen uitschrijven is variabel, daarom zullen we deze ook bijhouden. Het totaal aantal geheugen dat maximaal bijgehouden moet worden is dan (in bytes)

$$256(2^n(n + \log_2 n + 1))$$

Hierbij gaan we ervan uit dat een pointer naar een volgende Huffman top 1 byte in beslag neemt. Dit klopt aangezien we maximaal 255 pointers nodig hebben, waardoor

we een afbeelding kunnen definiëren van een byte tot een interne Huffman top.
Indien we per 1 byte inlezen wordt dit:

$$256(256(8 + 1 + 1)) = 655360$$

Dit is 640 KiB, wat voor moderne machines helemaal niet veel is.

Conclusie tweede algoritme Het tweede algoritme lost precies de problemen op van het eerste algoritme, en zal dus sneller zijn in het algemeen. Het nadeel is dat het 256 keer meer geheugen in beslag kan nemen.

2.2 Adaptive Huffman

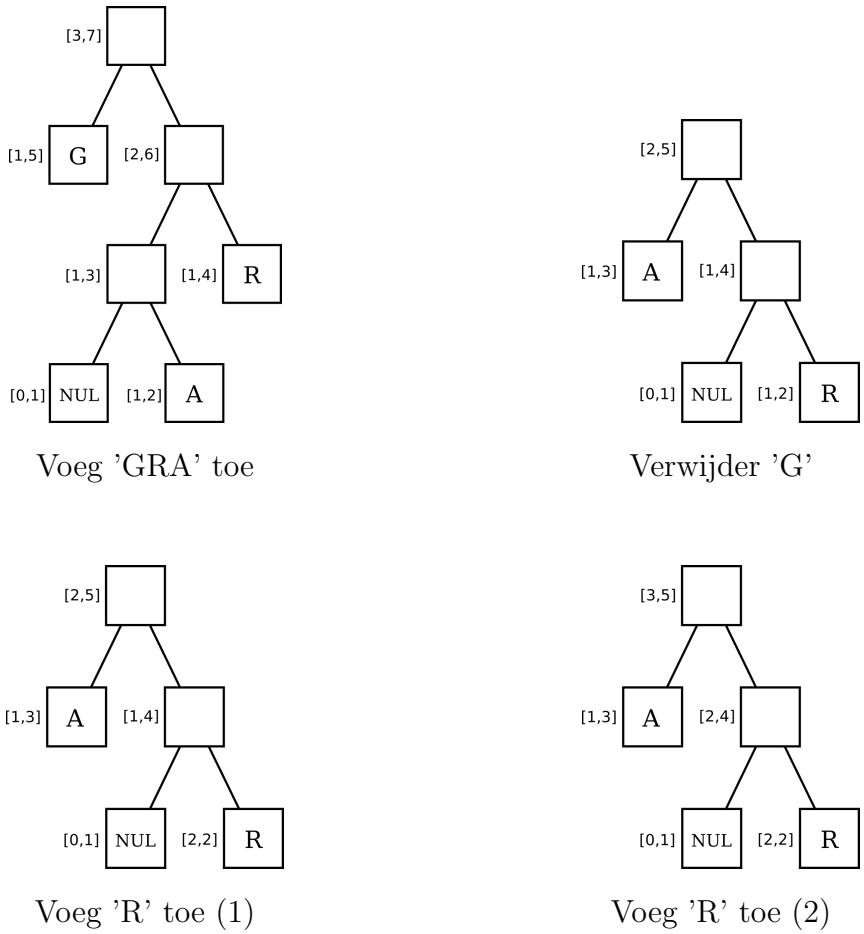
In de cursus Algoritmen en datastructuren 3[1] staat het algoritme dat we gebruiken voor adaptive Huffman.
Een alternatief op dit algoritme wordt omschreven door Donald E. Knuth[2]. Dit alternatief zal een betere theoretische complexiteit hebben. In volgende sectie wordt omschreven waarom we dit algoritme niet gebruiken.

2.2.1 Knuth versus naïef

We definiëren een blok in een Huffmanboom als een opeenvolging van toppen met hetzelfde gewicht. De toppen binnenin een blok zijn gesorteerd per ordenummer.
In elke stap van het adaptieve algoritme moeten we meerdere keren wisselen met een andere top uit de boom. Om te weten met welke top we moeten wisselen, moeten we telkens het hoogste ordenummer van een blok berekenen.
Het naïeve algoritme zal hierbij gewoon het ordenummer van de initiële top verhogen totdat we het einde van een blok hebben bereikt. Het algoritme van Knuth zal in constante tijd het hoogste ordenummer van een blok kunnen bepalen.
Als dit het volledige verhaal is, kunnen we hieruit besluiten dat Knuth veel beter zal presteren.
Helaas is dit niet het volledige verhaal. In de praktijk zal het naïeve algoritme bijna nooit meer dan één vergelijking moeten maken. Dit wil zeggen dat het algoritme van Knuth overhead introduceert om iets op te lossen wat nooit een probleem was. Daarom gebruiken we beter het naïeve algoritme.

2.3 Adaptive Huffman met sliding window

Voor een adaptief algoritme met een sliding window moeten we een downdatefunctie definiëren. Deze methode zal, op het eerste zicht, exact het inverse zijn van de updatebewerking bij adaptive Huffman.
Er is echter een probleem: als we dit zomaar doen, dan kan een update soms de balans van de boom verstoren (zie figuur 5).



Figuur 5: Bij het encoderen van "GRAR" zal de uiteindelijke boom incorrect zijn. De gewichten zijn (volgens ordenummer) 0-2-1-2-3, wat niet kan in een Huffmanboom.

Probleem De ouder van de nng-top (nog niet gezien) moet altijd ordenummer 3 hebben. Als dit niet zo is, kan het zijn dat we een ordenummer verkeerdelijk overslaan. Voorbeeld: Indien we terugkijken naar figuur 5, zien we dat de top met label "R" op dezelfde plaats blijft omdat de top waarmee we moeten wisselen de ouder is van deze top. Hierdoor zal de top met label "A" overgeslagen worden.

Oplossing Telkens wanneer bij het downdaten een top gewicht 0 krijgt, moeten we deze top verwijderen. De nng-top wordt dan 1 naar boven geschoven.
Hierna wisselen we de nieuwe ouder van nng met de top met ordenummer 3. Op die manier kan hetzelfde probleem niet meer voorkomen.

2.4 Adaptive Twopass Huffman

2.4.1 Opstellen boom

De beste manier om een adaptieve boom deterministisch op te stellen is om eerst een standaard boom op te stellen. Hierbij moet bij het mergen telkens het kind met het kleinste gewicht links in de boom terechtkomen. Daarna kunnen we gewoon de boom op een breedte-eerst manier overlopen en zo de ordenummers toekennen.

2.4.2 Optimalisatie

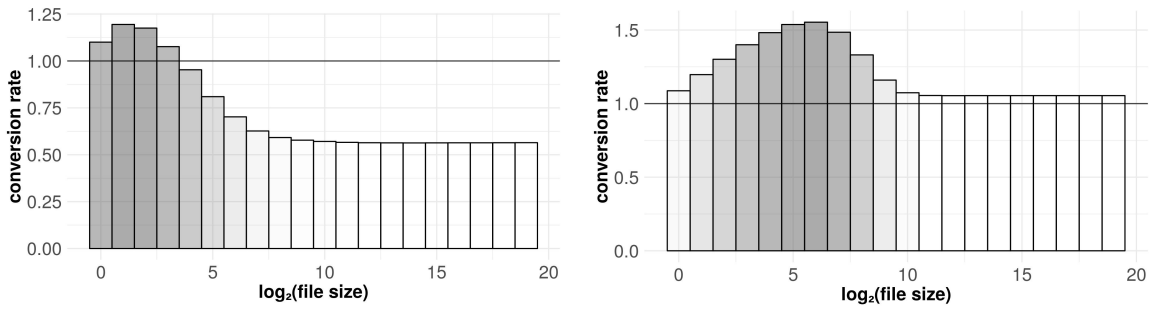
Aangezien we nooit de updatefunctie moeten oproepen, mogen we nng volledig weglaten. Op deze manier zal het encoderen van het minstgebruikte karakter telkens 1 bit minder in beslag nemen.

3 Optimale blokgrootte en windowlengte

Bij sliding window en blockwise adaptive Huffman kunnen we respectievelijk een windowlengte en blokgrootte kiezen. De volgende secties beschrijven optima voor deze parameters.

3.1 Optimale windowlengte

In het geval van sliding window willen we een zo klein mogelijke windowlengte om lokaliteit van symbolen uit te buiten. Als we echter een te klein sliding window gebruiken, is de kans groter dat we nodeloos een symbool uit de Huffmanboom verwijderen, en zo een byte extra moeten uitschrijven bij het reïntroduceren van dit symbool.
Als we kijken naar figuur 6, dan zien we dat zowel bij gewone tekst als binaire bestanden een minimale windowgrootte van rond de 2^{12} oftewel 4096 uitstekend werkt.

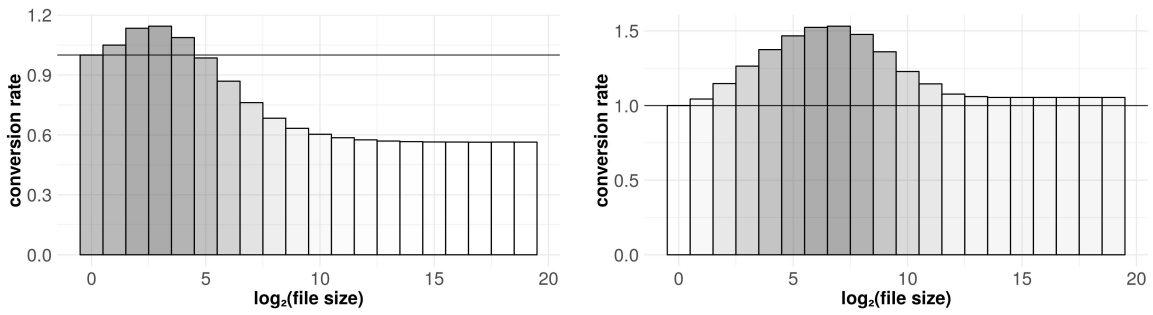


Figuur 6: Sliding window compressieratio voor tekstbestanden (links) en random bestanden (rechts).

Waarom 4096? Laten we veronderstellen dat de letters willekeurig verdeeld zijn over de tekst (voor de eenvoud). We willen een zo klein mogelijke waarde voor n vinden waarvoor we bijna zeker zijn dat alle 256 karakters in een tekst van n karakters zitten.
Men kan aantonen dat het verwachte aantal ontbrekende karakters (procentueel) gelijk is aan $\left(\frac{255}{256}\right)^n$. Inderdaad, vanaf $n = 4096$ komen we ongeveer 0% uit, precies wat we moeten hebben.
Uiteraard is dit geen bewijs dat dit geldt voor alle soorten teksten. De compressie van het sliding window algoritme hangt sterk af van tekst tot tekst.

3.2 Optimale blokgrootte

Als we kijken naar figuur 7 valt meteen op dat optimale blokgrootte een stuk hoger ligt dan de optimale window size. Dit komt omdat we hoe dan ook onze voortgang verliezen bij het overschakelen naar een nieuw blok, en telkens de boom helemaal opnieuw moeten opbouwen.
Een optimale blokgrootte betekent hier dus: de minimale grootte van een blok waarbij het verlies van onze boom bij de overgang naar een nieuw blok niet opvalt.
Op de figuur zien we dat deze blokgrootte rond de 2^{15} oftewel 32768 ligt.
Deze blokgrootte hangt, net zoals bij sliding window, enorm veel af van tekst tot tekst.



Figuur 7: Blockwise compressieratio voor tekstbestanden (links) en random bestanden (rechts).

4 Vergelijking compressieratio

4.1 Slechtste geval

Bij elk algoritme zal een random file het slechtste presteren. Met een random file wordt een file bedoeld waarbij elk symbool (ongeveer) evenveel voorkomt en de sym-

bolen even verspreid zijn.
Als de symbolen niet even verspreid zijn, zoals de string "aaaaaaabbbbbbb. . . zzzzzzz", dan zullen blockwise en sliding window enorm goed presteren (met de juiste window lengte en blokgrootte) in tegenstelling tot de anderen. Dit is zelfs het beste geval voor blockwise adaptive Huffman.

4.2 Beste geval

Het beste geval voor elk algoritme is uiteraard als we maar 1 verschillend karakter moeten encoderen. Bij standaard en twopass Huffman mogen dit zelfs 2 karakters zijn, aangezien de diepte van de boom niet verandert.

4.3 Gemiddeld geval

Het slechtste geval en beste geval zijn niet representatief voor een Huffman algoritme. In de praktijk zal men een randomfile niet proberen comprimeren (aangezien een randomfile reeds gecomprimeerd is per definitie). We zullen ook nooit een file met maar één karakter met Huffman encoderen (daarvoor gebruiken we beter bijvoorbeeld runlength encoding).
Om deze redenen gaan we naar de compressie van het "gemiddeld geval" kijken.

Algoritme	De Bijbel	Lord of the Rings	Kafka's Metamorphose
Origineel	4351186	1004478	139054
Standaard	2492114	567706	77408
Adaptive	2492159	567743	77425
Sliding window	2488564	568786	77259
Twopass	2492364	567962	77689
Blockwise	2493005	569031	77342

Figuur 8: Grootte van een bestand na compressie.

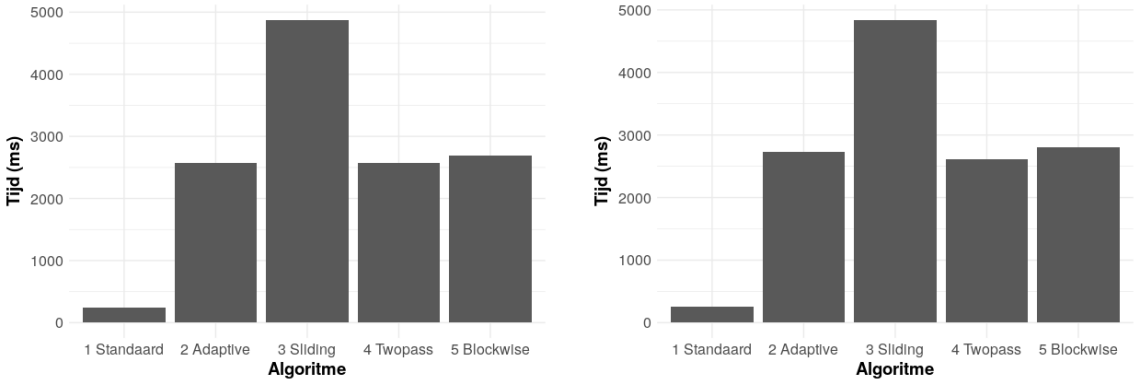
We zien op figuur 8 dat elk algoritme ongeveer even goed presteert op relatief grote teksten. Dit wil zeggen dat, tenzei we expliciet de lokaliteit van karakters willen uitbuiten, we altijd het snelste algoritme zullen willen gebruiken.

4.4 Twopass adaptive Huffman

Het twopass adaptive algoritme zorgt in theorie voor een fantastische compressie. We zien echter op figuur 8 dat het algoritme nooit het beste presteert. Dit komt omdat we ook de boom met de frequenties van elk karakter moeten uitschrijven. Omdat deze frequenties veel bytes kunnen innemen, wordt alle winst al snel tenietgedaan. Hierdoor zal er geen enkele situatie zijn waarvoor het twopass algoritme de beste compressie zal geven van alle algoritmes.

5 Snelheid

Een zeker niet verwaarloosbare metriek van een compressiealgoritme is de snelheid. Als een algoritme immers een fantastische compressie heeft, maar een waardeloze snelheid, zal die in de praktijk nooit gebruikt worden.



Figuur 9: Respectievelijk en- en decodeertijd van 10 keer de bijbel achter elkaar (grootte 43MB). De tijden zijn gelijkaardig omdat dezelfde stappen bij het encoderen en het decoderen worden gebruikt.

Vergelijking Zoals we zien op figuur 9 is het standaard algoritme veruit het snelste. Dit hadden we eerder voorspeld omdat we bij dit algoritme kunnen preprocessen.
Behalve sliding window halen de adaptieve algoritmes ongeveer dezelfde snelheid. Dit komt omdat we daarbij sterk gelijkaardige stappen doen.
Sliding window is uiteraard het traagste algoritme, aangezien we in (bijna) elke stap zowel een up- als downdateoperatie moeten doen.

Conclusie Het standaardalgoritme is veel sneller dan de andere algoritmes. Het enige algoritme dat in theorie sneller kan zijn dan standaard Huffman is blockwise adaptive Huffman, aangezien we die kunnen paralelliseren.

6 Blockwise standard Huffman

Met de kennis van vorige onderdelen kunnen we een algoritme ontwerpen die als vervanging voor blockwise adaptive Huffman zal dienen: blockwise standard Huffman. De reden is eenvoudig: we willen genieten van de voordelen van per blok te kunnen werken, en dit met aan de snelheid van het standaard Huffman algoritme.
Een volledige analyse van dit algoritme valt buiten de scope van dit verslag, maar van wat we nu al weten, kunnen we bijna zeker zijn dat dit heel goed zal presteren.

7 Besluit

Standaard Huffman zal in de praktijk (bijna) altijd gebruikt worden. Dit komt door de enorme snelheidswinst, in ruil voor een miniem verschil in compressie.
Zelfs als de data perfect is voor blockwise adaptive Huffman, kunnen we beter voor blockwise standard Huffman kiezen.
Enkel als snelheid helemaal niet van belang is en als de data globaal sterk verschillend is, maar lokaal niet, dan kunnen we overwegen om sliding window te gebruiken.

Het belangrijkste dat we uit dit verslag kunnen leren is dat complexe algoritmes, hoe goed ze ook lijken op papier, vaak verborgen consequenties met zich meedragen.

Referenties

[1] Gunnar Brinkmann, *Algoritmen en datastructuren 3*, 2017.

[2] Donald E. Knuth, *Dynamic Huffman Coding*, 1983.