Algoritmen en datastructuren III

Domien Van Steendam

3e bachelor of Science in de Informatica

1 Waarom Huffman?

LZ77 en LZW maken beide gebruik van een woordenboek. Er is echter geen (of alleszins zeer weinig) onderling verband tussen de cijfers. De komma's leveren ook geen informatie over het volgend karakter. Het kan echter wel nuttig zijn, voornamelijk voor lzw vanwege zijn globaal karakter, om bv. strings van 2,3 of 4 opeenvolgende cijfers in het woordenboek op te slaan. Maar voor een groter aantal opeenvolgende cijfers, waar men dan echt veel winst kan boeken, zal het woordenboek zijn nut verliezen aangezien een match zeldzamer wordt.

Burrows-Wheeler en Move to front bewijzen hun nut voornamelijk in lexicografische teksten. Dit staat letterlijk in de cursus en tevens in de paper van David Wheeler en Michael Burrows[1]. In lexicografische teksten zullen de karakters beter gegroepeerd worden wat dan voordelig is voor Move to Front. Echter bij ons formaat hebben we weinig garantie dat '12' gevolgd wordt door een '3'.

Huffman is optimaal voor een gegeven alfabet. We weten het alfabet, want het formaat is gegeven in de opgave. We kiezen als alfabet alle cijfers, de vierkantjes haakjes en de komma. We kunnen eventueel ook als optimalisatie een alfabet van 2 karakters of 3 kiezen per symbool, maar er is uiteraard een limiet. (Ik heb gekozen voor een alfabet van 2 karakters per symbool).

2 Algemeen algoritme

Het alfabet van dit formaat bestaat ten hoogste uit 13 tekens, namelijk de 10 cijfers, '[' , ']' en ','. We weten uit de cursus dat Huffman voor een vast alfabet steeds een optimale prefixcode zal genereren, t.t.z. die het dichtste aanleunt bij de entropie van dit alfabet.

Om mijn motivatie te bekrachtigen voer ik een experimentje uit op 'Resources\invoer\small.txt': Small.txt bevat 993870 karakters.
Om de entropie heb ik een scriptje geschreven dat te vinden is in 'Resources\scripts\bereken_entropie.py'. Dit script geeft tevens de telling van symbolen weer, om op dia manier een beeld te geven van de informatieverdeling in het meegegeven bestand. We voeren dit script uit op het invoerbestand en krijgen volgende output:

Kansen per karakter (totaal=993870):

, : 49999

0 : 89696

1 : 96169

2 : 95703

3 : 95173

4 : 95331

5 : 95358

6 : 94989

7 : 95183

8 : 95578

9 : 90689

[: 1

] : 1

Entropie = 3.4421945510415077

De entropie is ongeveer 3,44 bit.
Als we deze verdeling in een Huffman boom "gieten" en het gemiddelde aantal bits per symbool uitrekenen bekomen we 3,56 bits/symbool. Dit scheelt slechts 0.12 bit/symbool.

Na het comprimeren gebruikmakend van mijn implementatie (zie verder) bekomen we een bestand van 461382 bytes. Elk karakter heeft dus een gemiddelde grootte van 443435/993870= 0,446 bytes oftewel 3,569 bit.

Complexiteit:
Het Huffman algoritme bestaat uit volgende fasen:

1. De frequenties van ieder voorkomend symbool tellen.
2. Aan de hand van deze frequenties stellen we de Huffman-boom op.
   2.1. De boom wordt gecodeerd weggeschreven naar de output.
3. Elk karakter van het invoerbestand wordt gecodeerd aan de hand van deze boom en weggeschreven naar de output.

Dit algoritme **letterlijk** zoals hierboven implementeren kan efficiënter voor stap 3:
Enkel de bladeren bevatten informatie over de corresponderende karakters uit het alfabet.
Het opzoeken van een symbolen in de boom vereist dus telkens het overlopen van elk blad. Dit heeft een slechtste geval complexiteit van O(k) met k het aantal symbolen in het alfabet.
Inderdaad is k in dit geval constant, **maar** deze opzoekbewerking gebeurd per karakter in het invoerbestand. De complexiteit voor iedere karakter te encoderen is dus O(k*n). Deze constante k valt uiteraard weg voor het gegeven invoerformaat van 13 symbolen, maar dat is vanwege de verraderlijke O-notatie. In mijn implementatie heb ik voor een constante O(1) opzoeking gezorgd **onafhankelijk van k**. Dit heb ik gedaan door de boom te transformeren naar een array van 256 entries. Men gebruikt het karakter, dat als ASCII-integer kan geïnterpreteerd wordt, als index voor de entry. In de entry vindt men vervolgens het codewoord.
Stap 3 heeft natuurlijk nog steeds kost O(n), maar zal toch sneller uitgevoerd worden als voordien, want k speelt hier geen rol voor stap 3 en de huidige constante, die het opvragen van een arrayentry beslaat, is kleiner dan k.

De manier waarop ik het geïmplementeerd heb is analoog als hierboven, maar tussen stap 2 en 3 heb ik een omzetting gedaan van de Huffman-boom naar een map die een symbool afbeeldt op zijn codewoord. Dit codewoord kan vervolgens a.d.h.v. een symbool in constante tijd opgevraagd worden.
We itereren over de hele boom en als we een blad tegenkomen, plaatsen we het codewoord tezamen met het corresponderend karakter in de map. Dit itereren gebeurt éénmaal en kan in O(2k-1)=O(k).

Stap 1 vereist het overlopen van elk karakter in de invoer en in constante tijd een counter aanpassen. De complexiteit hiervan bedraagt dus O(n) met n het aantal karakters.

Stap 2 bevat de meeste code. Hier wordt de boom getransformeerd in een array met kost O(k). Vervolgens wordt elk karakter uit het invoerbestand gelezen en in constante tijd wordt zijn codewoord opgevraagd door middel van deze array. Dit gedrag is lineair.
Stap 3 is analoog aan stap 1 aangezien we opnieuw elk karakter in het invoerbestand inlezen en, dankzij de omzetting van hierboven, het codewoord van dit karakter in constante tijd opvragen. De complexiteit is hier dus opnieuw O(n).

De meeste ontwerp beslissingen van mijn Huffman implementatie zijn vrij intuïtief. De code is ook goed gedocumenteerd. Echter zijn er 2 aspecten waarop ik toch een woordje uitleg bij ga geven aangezien die vrij gedetailleerd zijn.

Het comprimeren van de boom: Het basisidee is om de boom depth-first te doorzoeken en telkens één bit wegschrijven per top. We schrijven 1 indien de top een blad is, 0 in het andere geval. Voor de bladeren schrijven we deze bit + het symbool horend bij dit blad.
Nadat we alle toppen hebben overlopen schrijven we als slotbit een 1. *Dit vereenvoudigt de code voor het decoden. Zonder het slotbit is het ook mogelijk, maar moeten we steeds helemaal terug naar de wortel om daar dan pas te ontdekken dat het proces beëindigd is.*
Om de gecomprimeerde boom terug uit te lezen, lezen we steeds 1 bit. Indien deze bit een 1 is, lezen we nog 8 bits extra die dan het symbool voorstellen. De pseudocode voor het uit te lezen ziet u hieronder:

0. Maak een node R aan die als wortel zal dienen.
1. De huidige node CUR is R.
2. Lees volgende bit b
3. Als b is 0 (interne node)
4. Maak nieuwe node N aan.
5. Als CUR-links vrij heeft hang N hier, (zoniet hang hem rechts) en ga terug naar 3 met CUR = N.
6. Als CUR zowel links als rechts als kinderen heeft, dan
    i. Als een CUR een ouder heeft CUR = ouder van CUR en probeer 6 opnieuw.
    ii. (CUR heeft altijd een ouder in het geval b==0, CUR is m.a.w. nooit wortel R)
7. Als b is 1 (blad)
8. Maak nieuwe node N aan.
9. Lees volgende 8 bits (karakterteken) en ken dit toe aan N.
10. Als CUR-links vrij heeft hang N hier, (zoniet hang hem rechts) en ga terug naar 3 met (CUR = CUR).
11. Als CUR zowel links als rechts als kinderen heeft, dan
    i. Als een CUR een ouder heeft CUR = ouder van CUR en probeer 11 opnieuw.
    ii. Als CUR geen ouder heeft (CUR is wortel R), is het algoritme beëindigd.

Het uitlezen van de prefixcodes uit het gecomprimeerd bestand:

We werken met 2 buffers zodat de tweede buffer de eerste kan verwittigen indien het einde van het bestand is bereikt. Zo vermijden we dat we de garbage bits op het einde zullen gebruiken. Dit is wel degelijk noodzakelijk ook al lijkt het onnodig op het eerste zicht. De pseudocode met gebruik van 2 buffers ziet u hieronder:

1. Cur node is wortel R
2. Voer volgende uit tot einde van de file.
3. Creëer een buffer van 16.
4. Lees volgende bit b uit buffer indien EOF nog niet gelezen is.
5. Indien EOF al in buffer, lees dan enkel de zinvolle bits uit de buffer a.d.h.v. het meegegeven aantal garbage bits.
6. Indien b==1
7. Cur is rechterkind
8. Indien b==0
9. Cur is linkerkind
10. Is deze curnode een blad?
11. Néé ga 4
12. Já schrijf corresponderend symbool en CUR = wortel R

3 Specifiek algoritme

Ik heb verschillende ideeën, maar uiteindelijk ben ik terug bij mijn eerste idee terechtgekomen aangezien die het beste presteerde. **De andere ideeën zal ik bij de extra's hieronder bespreken.**

Het basisidee is als volgt:

1. Berekenen de verschillen tussen 2 opeenvolgende getallen. Voor het allereerste getal berekenen we het verschil met 0.
2. Voer op deze verschillen het algemene Huffman algoritme toe

Stap 1 en 2 hebben beide duidelijk een lineaire kost.
Ik beschrijf hier hoe ik het geïmplementeerd heb. Voor sommige zaken bespreek ik kort hoe ze beter kunnen en waarom ze dan beter zijn. Voor de duidelijkheid staat deze *cursief en in groene kleur*. Voor de leesbaarheid van de code heb ik deze optimalisaties niet allemaal toegepast in de implementatie.

De verschillen zijn steeds strikt positief aangezien we van een strikt stijgende rij van getallen mogen uitgaan.
Stap 1 heb ik geïmplementeerd door middel van een tussenbestand.
In plaats van een tussenbestand zou het uiteraard een stuk sneller zijn, maar nog steeds O(n), als men de frequenties al begint te tellen tijdens het berekenen van de verschillen. Dit zou de code echter complexer maken. De compressiefactor zal ongewijzigd blijven en uiteindelijk speelt die de grootste rol indien beide methoden dezelfde tijdsorde hebben.

Vervolgens geef ik dit tussenbestand door aan stap 2 die gewoonweg het algemeen Huffman-algoritme uit deel 2.2 uitvoert op dit bestand.
Het wegschrijven van de verschillen kan op meerdere manieren:

- Men schrijft de verschillen als hun stringrepresentatie elk gescheiden door bv. een komma.
- Men schrijft de verschillen als een long long (8 bytes) weg.

Ik had oorspronkelijk (zie verder) voor de tweede optie gekozen om meerdere redenen:

- Voor getallen met meer dan 8 cijfers gebruikt de raw long long optie minder bytes als de string representatie. Aangezien de getallen kunnen oplopen tot 19 cijfers, zijn er dus meer getallen met meer dan 8 cijfers dan getallen met minder dan 8 cijfers.
  Het lijkt volgens mij echter een relatief zeldzaam geval gebeuren dat een verschil de volle

8bytes nodig zal hebben. Indien we een maximum weten voor de verschillen, kunnen we eventueel minder bytes gebruiken.

- Aangezien ik een vaste lengte gebruik voor de verschillen naar het tijdelijk bestand weg te schrijven, hoef ik geen komma's meeschrijven!
  Bij optie 1 moeten die wel meegeschreven worden, aangezien er geen vaste lengte is voor een stringrepresentatie van een getal.

Echter is er nog een addertje onder het gras die bij volgende analyse naar boven komt waaruit zal blijken dat **de eerste optie beter is**.
We testen deze 2 opties op het bestand "medium.txt" met een grootte van ongeveer 9700 kB.
In het bestand HuffmanSpecifiek.c is de macro `WRITE_DIFF_AS_STRING` gecommentarieerd. Als deze macro gedefinieerd is, zal de string representatie gebruikt worden (lees zeker de documentatie bij de macro). Dit heb ik gebruikt voor het vergelijken.
Het tussenbestand bij gebruik van de eerste, tweede optie heeft een grootte van respectievelijk 7094 kB en 3907 kB. Dat is bijna de helft kleiner!
Na stap 2 zijn de bestanden respectievelijk 3114kB en 3280 kB. Nu is de string-representatie alsnog beter. Het addertje onder het gras is dat bij de string representatie er maar 11 mogelijke karakters/bytes zijn. Bij de raw long long bytes kan er in principe elke van de 256 mogelijkheden voorkomen. Hierdoor wordt bij de tweede optie onze Huffman-boom groter en bijgevolg onze codewoorden langer. Dit verklaart waarom optie 1 ondanks het grote verschil in de grootte van het tussenbestanden beter is dan optie 2.

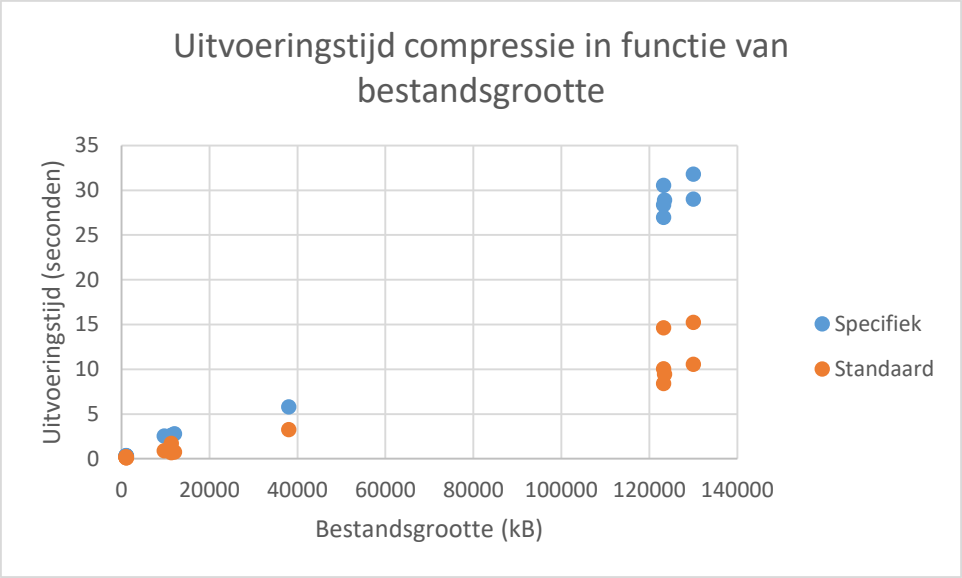
Extra algoritmen (eventueel korte uitleg op verdediging):

- Bit array: Voor alle getallen binnen 2^i en 2^{i+1} hoeven we maar 2^i bits te gebruiken, wat minder is dan hun string representatie. We houden tevens per logaritmisch interval bij hoeveel getallen vanuit het invoerbestand er zich in bevinden.
- Vaste prefixcode: Met een vaste prefixcode hoeven we geen frequenties te tellen waardoor we een stuk sneller kunnen comprimeren

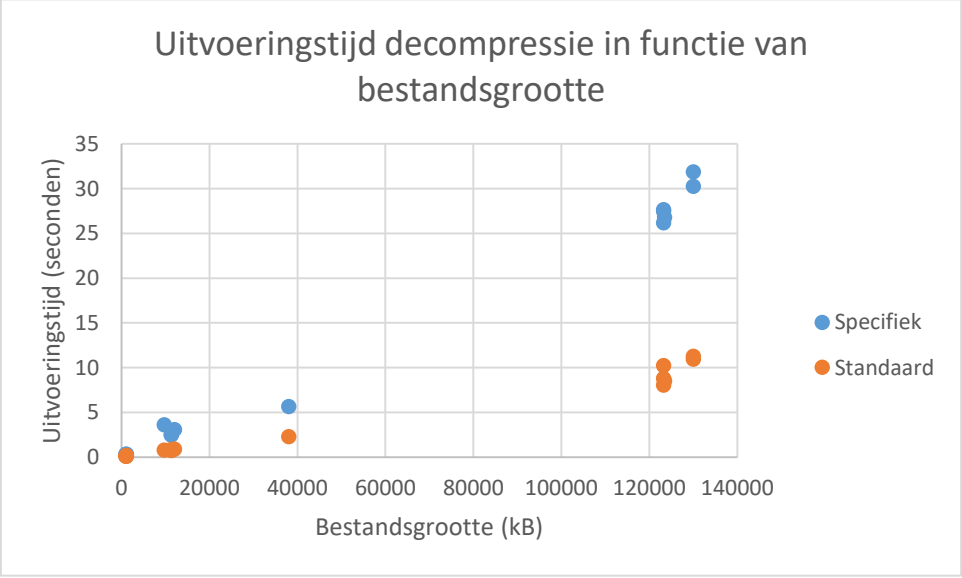
Vergelijking van de algoritmen

We verwachten dat beide algoritmen lineair uitvoeren, op een constante na. We verwachten dat het specifieke algoritme het langst zal duren om te comprimeren en te decomprimeren, want deze bevat volledig het standaard algoritme en daarbovenop een stap om de verschillen te berekenen en weg te schrijven. Zoals eerder al vermeld, zou men in de praktijk beter het specifieke algoritme optimaliseren door de frequenties te tellen tijdens het berekenen van de verschillen.

In figuur 1 zien we het verband het comprimeren van bestanden met willekeurige gekozen getallen.

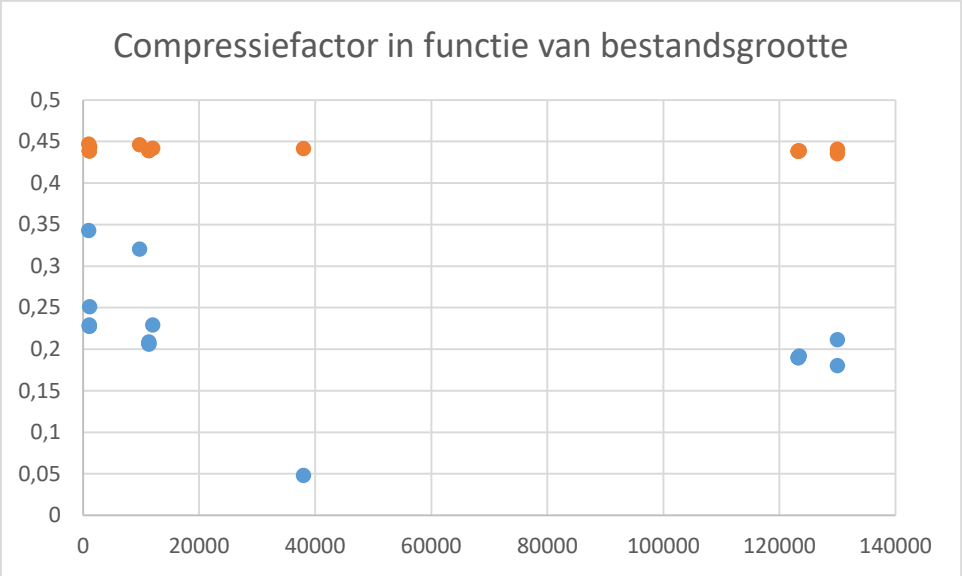


Figuur 1



Figuur 2

Onze verwachting klopt. Aangezien beide algoritmen lineair zijn en er relatief gezien weinig verschil is, zullen we ons focussen op het verschil in compressiefactor. Voor de compressiefactor merken we een verschil in grootteorde. Het standaard algoritme heeft steeds ongeveer dezelfde compressiefactor, maar bij het specifiek algoritme varieert dit zoals te zien in Figuur 3. Het blauwe datapunt bij 37 kB is zo laag, omdat het een “toevallig” best case scenario is.



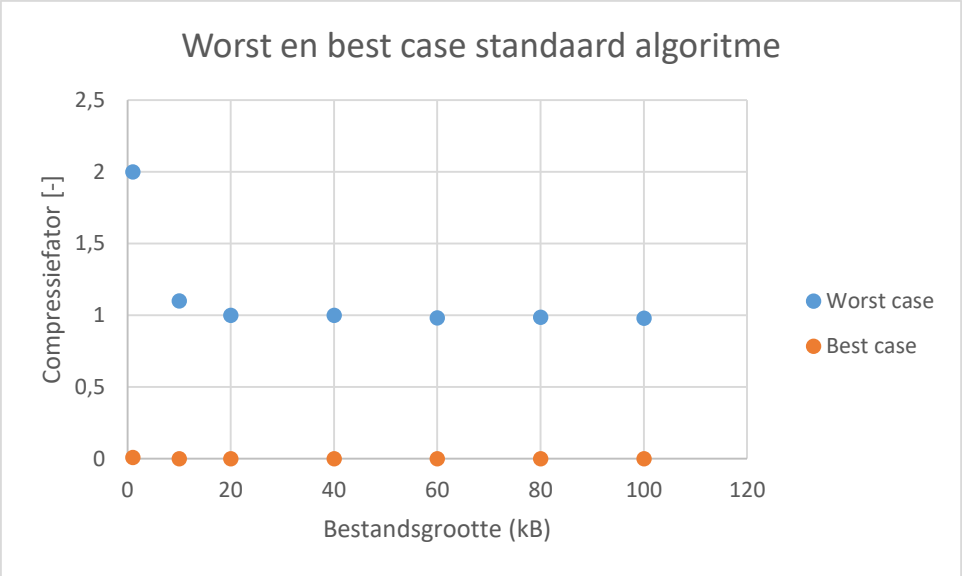
Figuur 3

Algemeen

De Huffman tree wordt steeds mee opgeslagen. De tree neemt gemiddeld gezien ongeveer 20 bytes in beslag. Voor heel korte bestanden is dit relatief veel en is de kans groot dat het gecomprimeerd bestand langer wordt. Als we bovendien nog eens weinig redundantie laten voorkomen in het bestand, kan het comprimeren heel nadelig worden. Bv. een bestand met alle 256 mogelijke combinaties van een byte na elkaar is klein en bevat geen redundantie. Ik heb een pythonscript geschreven en meegeleverd genaamd "AlgemeenGenerateWorstCase.py". Het verwacht respectievelijk de lengte van het te generen bestand en de naam als argumenten. Het script genereert dan een bestand van n tekens als volgt: 0x00 0x01 0x02 … . Na 0xFF begint hij terug met 0x00. Voor een grote lengte verwachten we dat de compressiefactor zal verbeteren aangezien we dan al enkele keren terug naar 0x00 gesprongen zijn en er dus voldoende redundantie is om effectief te verkleinen. In figuur 4 ziet u de compressiefactor uitgedrukt in de meegegeven parameter in het script. We plotten voor veelvouden van 10 kB.

De beste case bekomen we als het gemiddeld aantal bits exact gelijk is aan de Shannon-entropie. Hiertoe moeten de frequenties van de symbolen onderling verschillen met een factor 0.5^k, $k \in \mathbb{N}$. De toppen met dezelfde ouder zullen hierdoor exact dezelfde frequentie hebben.

Echter kunnen we het ons makkelijker maken om een best case bestand op te stellen vanwege het feit dat het standaard algoritme run length encoding zal toepassen indien er maar 1 verschillend karakter is en voldoende groot is, want RLE encoding levert ons steeds een bestand van 0.5 (header) + 8(long long voor de teller) + 1(karakter) ≤ 10 bytes. Het bijhorend script heet AlgemeenGenerateBestCase.py. Dit script neemt de argumenten aan analoog als het vorige. Het genereert een bestand met allemaal ene (als karakters). Opnieuw plotten we de compressiefactor in functie van het aantal eenheden in figuur 4.



Figuur 4

De best case is inderdaad steeds 10 bytes (zie ook Excel bestand "Metingen.xlsx", tabblad "factorAlgemeen"). In het geval van de worst case is het gecomprimeerd bestand bijna even groot als het origineel, want de compressiefactor is 1. Deze daalt echter zeer langzaam zoals we ook verwachtten, want er ontstaat meer redundantie. Voor zeer kleine bestanden is de factor zelfs 2 vermoedelijk vanwege de constante overhead van de header en de boom.

Specifiek

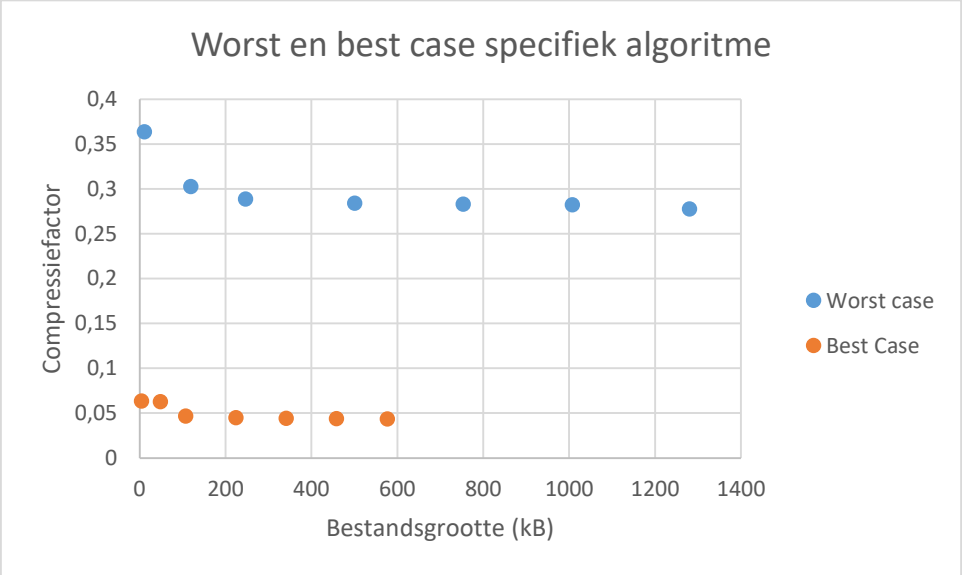
Voor de worst en best cases baseren we ons op de formaten vermeld in de vorige sectie.
We werken nu echter verplicht volgens het opgegeven formaat.
We hier voor de duidelijkheid van optie 1 uit dat we de verschillen als string schrijven in het tussenbestand (cfr. Antwoord op vraag 4.3).
Extra: De best case van de vorige sectie, zal hier niet werken want we zijn beperkt door ons formaat. Het hoogst mogelijk invoerbestand met 1 karakter voor dit formaat is [1 111 111 111 111 111 111] (haakjes worden niet mee gecodeerd). De compressieratio is dus 10/19 > ½.

We zullen trachten een tussenbestand te vormen met maar 2 verschillende karakters: namelijk de komma en een cijfer. We kiezen "1" als cijfer zodat de opeenvolgende getallen niet te grote sprongen maken. De tussenbestanden kiezen we als volgt: 1,1,1 …. 1,. Invoerbestanden zijn dus gewoon [1,2,3,4…]. Huffman zal dan 1 bit per karakter gebruiken. Dit kan natuurlijk voor ieder cijfer. Voor een invoerbestand van n karakters is de compressiefactor **ten opzichte van het tussenbestand** exact gerekend $\frac{(n*1+4+(3+8*2))}{(n*8+2)}$. Voor grote n is dit ongeveer $1/8$ wat relatief gezien uitstekend is.
Dit is nog maar voor het tussenbestand waar de verschillen steeds 1 byte zijn. Voor het effectieve invoerbestand stijgt het aantal karakters per getal naargelang n stijgt. Aangezien we elk getal zijn duale (dit is het verschil op dezelfde in als in het tussenbestand) kunnen voorstellen als 1 bit zal de compressiefactor per getal naar 0 naderen naargelang n stijgt en bijgevolg ook de compressiefactor van het volledige bestand. In figuur 3 ziet u de plot op elke invoerbestanden gegeneerd met SpecifiekGenerateBestCase.py

De worst case verkrijgen we als de Huffman boom perfect gebalanceerd is en dan het "liefst" met zoveel mogelijk verschillende karakters zodat de diepte van de boom toeneemt.
We zullen het **tussenbestand** zodanig opstellen opdat hij aan deze eigenschap voldoet, het werkelijk invoerbestand kan men dan vervolgens gewoon afleiden door de optelsom te maken.

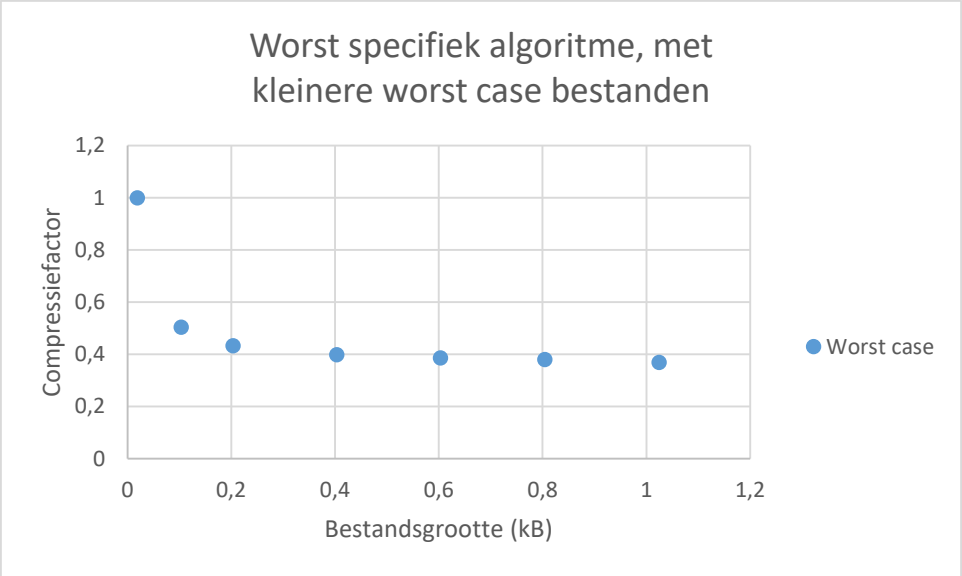
Elk karakter heeft dan een even lang codewoord. Het gecodeerde “bericht” heeft dan namelijk een grootte van (zonder de Huffman boom meegerekend) $n * log_2 k$ bits met k het aantal verschillende karakters. We kunnen via het tussenbestand maximum 11 karakters meegeven (haakjes zijn gefilterd en tellen dus niet mee). Een eenvoudig formaat voor het tussenbestand met ongeveer gelijke verdeling van de karakters: 123456789, 123456789,… Alleen moeten we nu de bestanden hierboven beschouwen alsof ze verschillen zijn. De effectieve worst/best case bestanden vinden we dan gewoonweg door de paarsgewijze optelsom te maken. De cijfers hebben dezelfde verdeling, maar de komma niet dus is dit niet de effectieve worstcase. Zulke bestanden opstellen kan zeer complex worden voor grotere bestandsgroottes. We verwachten echter nog steeds een zeer slechte compressiefactor. Het bijhorend script heet SpecifiekGenerateWorstCase.py.

We plotten analoog als bij het standaard algoritme de best en de bijna-worst case. De lengteparameters zijn steeds 1000,10000,20000,40000,60000,80000 en 100000. De invoerbestands groottes variëren echter voor de beide cases. De plot ziet u in Figuur 5.



Figuur 5

Onze best case is perfect zoals we voorspelden. Onze bijna-worst case is niet wat we verwachtten. Aangezien de worst case minder slecht wordt naarmate het bestand groter, starten we beter vanaf 1 eenheid in plaats van 1000. We doen dit opnieuw voor 1,10,20,40,60,80 en 100 eenheden. Een eenheid is hier één stringblok “12345678”. Het resultaat ziet u in figuur 6.



Figuur 6

Onze bijna-worst case toont al een iets slechtere compressiefactor van ½. Dit verwachtten we ook, omdat we 11 symbolen hebben die gelijk voorkomen (op de komma na). 11 symbolen vereist 4 bits per symbool. Dat is dus de helft minder bits dan het origineel bestand.
Als we bestanden gebruiken waar ook de komma's evenveel voorkomen, kunnen we vermoedelijk nog een slechtere compressiefactor bekomen in de zin dat hij ongeveer ½ blijft ongeacht de grootte.

5 Versturen over internet

Stel dat we een transmissiekanaal hebben met een bit snelheid van b bits/seconde. Stel dat we een bestand van 1 Mb willen versturen. De tijd nodig om dit ongecomprimeerd door te sturen is 1Mb/b seconden.

Het standaard algoritme comprimeert en decomprimeer respectievelijk ongeveer 8kB en 12kB per seconden. Het comprimeren en decomprimeren van een bestand van 1Mb duurt respectievelijk ongeveer $\frac{1MB}{8kB} = 125$ en $\frac{1MB}{12kB} = 83$ seconden. Voor het standaard heeft een bestand van 1Mb een compressiefactor van ongeveer 0.45. Het gecomprimeerd bestand heeft bijgevolg een grootte van 450 kB. Het sturen van dit bestand duurt 450kB/b seconden.
Dit levert ons een vergelijking op voor b:

$$125s + 83s + \frac{450kB}{b} < \frac{1MB}{b}$$

Dit is waar voor alle b kleiner dan 2,644 kB/s.

Analoog voor het specifieke algoritme waar de compressie en decompressiesnelheid respectievelijk 4kB en 4,8 kB. Het comprimeren en decomprimeren van een bestand van 1Mb duurt respectievelijk ongeveer $\frac{1MB}{4kB} = 250$ en $\frac{1MB}{4{,}8kB} = 208$ seconden. Voor het standaard heeft een bestand van 1Mb een compressiefactor van ongeveer 0.20. Het gecomprimeerd bestand heeft bijgevolg een grootte van 200 kB. Het sturen van dit bestand duurt 200kB/b seconden.
Dit levert ons opnieuw een vergelijking op voor b:

$$250s + 208s + \frac{200kB}{b} < \frac{1MB}{b}$$

Dit is waar voor alle b kleiner dan 1,647 kB/s.

6 Testen

In de map tests vindt u de testcode. De testbestanden werd niet allemaal meegeleverd, want sommige zijn heel groot.

Standaard

In TestServices.c staan enkele vaste testfuncties waaronder ook document- en afbeeldingsbestanden. De debug() methode heb ik gebruikt om alle andere 43 testbestanden te testen op correctheid met behulp van cmp_files(). Als ultieme test werd een bestand van 9 GB gecomprimeerd en gedecomprimeerd met succes.

Specifiek

TestSpecifiek.c werd gebruikt voor het specifieke algoritme te testen. De methode test_full() comprimeert en decomprimeert een gegeven bestand. Dit deed ik opnieuw manueel voor alle 43 testbestanden. Voor het vergelijken werd gebruikt van het FC-commando (Windows).

Referenties

1. *Burrows, Michael; Wheeler, David J. (1994), A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation*