Project Datacompressie

Jarre Knockaert

30 november 2016

1 Standaard algoritme

1.1 Bespreking verschillende algoritmes

1.1.1 LZ77

Het grote voordeel van LZ77 is dat het gebruik maakt van afhankelijkheden tussen symbolen. Dit zorgt ervoor dat vaak voorkomende patronen korter kunnen gecodeerd worden. Bij het gegeven bestandsformaat echter omvat de inhoud stijgende (random) cijfers gescheiden door komma's, hier zit weinig patroon in, de symbolen zijn onafhankelijk van elkaar. Er zijn weinig lange exacte herhalingen (lange matches) in de tekst.

1.1.2 LZW

LZW zal hier slecht presteren met dezelfde reden als LZ77. LZW maakt ook gebruik van patronen in de inhoud van de tekst. In tegenstelling tot LZ77 gebeurt het coderen efficïenter, aangezien het dure opzoeken van de langst mogelijke substring niet hoeft te gebeuren.

1.1.3 Burrows-Wheeler

Net als LZ77 en LZW maakt Burrows-Wheeler gebruik van bepaalde patronen die terugkomen in teksten. Als er bepaalde prefixen zijn die vaak terugkomen zal dit algoritme goed presteren. Maar ook hier zijn deze patronen beperkt en zullen dus nauwelijks profijt leveren.

1.1.4 Huffman

In tegenstelling tot de vorige algoritmes, maakt Huffman codering geen gebruik van bepaalde patronen in de input. Het maakt enkel gebruikt van de frequenties van de tekens in de input. Aangezien er in het gegeven formaat maar een beperkt aantal tekens zijn (komma, 2 vierkante haakjes en '0' tot en met '9'), zullen deze tekens met een kleiner aantal bits kunnen voorgesteld worden en zal de lengte van de geëncodeerde code veel kleiner zijn dan de originele lengte. Doordat dit algoritme het beste lijkt voor het gegeven probleem, koos ik voor Huffman codering bij het standaard algoritme.

1.2 Uitvoerige bespreking Huffman

1.2.1 Verwachtingen

Initieel verwachtte ik dat het algoritme zou werken, maar zonder zeer veel compressie vanwege de overhead van de Huffman boom. Deze is echter zeer klein ten opzichte van de winst die gemaakt wordt door de kleinere codes. Ook verwachtte ik dat het algoritme traag zou coderen en decoderen omdat vaak met individuele bits gewerkt wordt. Het algoritme werkte dan ook

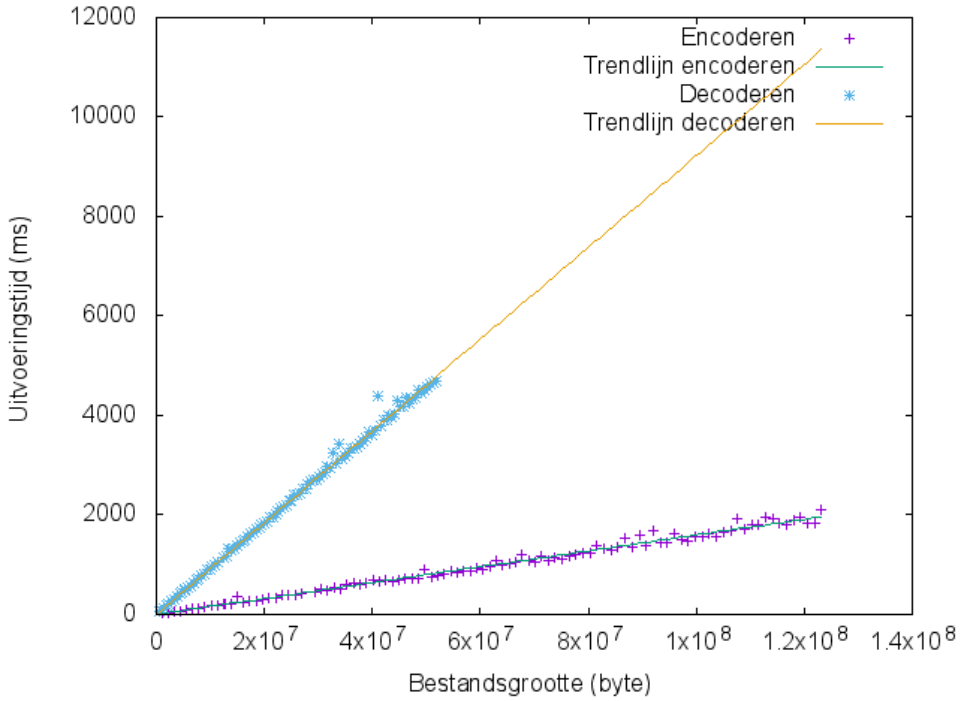
volgens de verwachtingen zeer traag initieel, maar met enkele optimalisaties zoals I/O operaties beperken, minimaliseert het snelheidsprobleem.

1.2.2 Performantie

Hier zal ik de snelheid van het algoritme bespreken. Op figuur 1 ziet u de performantie van mijn algoritme. Hier kan je duidelijk zien dat de uitvoeringstijd lineair stijgt in functie van de bestandsgrootte bij zowel het encoderen en het decoderen. De rechte $\frac{uitvoertijd}{bestandsgrootte}$ stijgt veel sneller bij het encoderen dan bij het decoderen, wat betekent dat bij een zeer groot bestand, het verschil in decodeertijd en encodeertijd altijd groter zal worden. Merk op dat de grootte van het geëncodeerde bestand werd gekozen als bestandsgrootte bij het decoderen zodat de encodeer- en decodeersnelheid enkel afhangt van de invoer. Het encoderen verloopt veel sneller dan het decoderen omdat bij het encoderen groepen van bits (voor elk karakter) weggeschreven worden naar de output, en bij het decoderen moeten deze bits telkens bit per bit gelezen worden om zo de Huffman boom te doorlopen. Het bit per bit doorlopen van de Huffman boom voor elk karakter kost echter veel tijd wat de langere uitvoeringstijd verklaart. Neem bijvoorbeeld dat elke code gemiddeld lengte 4 heeft en stel n het aantal bytes in de input bij het decoderen. Het encodeeralgoritme overloopt 2 keer alle karakters, éénmaal om de frequenties te bepalen en één keer om elke code uit te schrijven. Het aantal bewerkingen zal hier dus $2n$ zijn. Bij het decoderen zal het aantal bewerkingen $4n$ zijn. Voor elk karakter moeten namelijk 4 bits bekeken worden. De rechten hebben een verschillende richtingscoëfficient omdat het verschil tussen $4n$ en $2n$ altijd groter zal worden.

1.2.3 Implementatie

Encoderen Mijn implementatie van het Huffman algoritme deelt de input op in verschillende blokken met een bepaalde grootte. Dit biedt als voordeel dat zeer grote bestanden kunnen verwerkt worden en zich niet compleet in het werkgeheugen moeten bevinden. Een positief neveneffect hiervan is dat de compressie verbetert. De reden hiervan is dat bepaalde bytes meer voorkomen in bepaalde blokken en minder voorkomen in andere blokken. Neem als voorbeeld JPEG-bestanden. Deze eindigen met een heel lange reeks bytes met waarde 0. Deze grote hoeveelheid bytes kunnen zeer efficiënt gecodeerd worden als ze in een apart blok bevinden zodat hun code lengte 1 heeft. (Het is het enige karakter in het blok, dus de code heeft lengte 1.) Een negatief neveneffect hiervan is dat er extra overhead is, voor elke blok moet een aparte Huffman boom gemaakt worden. De boom is echter zeer klein en dit is niet opvallend in het grote plaatje, de winst door het positieve neveneffect is veel groter dan het verlies door de bijkomende overhead. De volgende stappen worden genomen bij het encoderen tot het



Figuur 1: Deze grafiek beschrijft de performantie van Huffman. De rechtes doorheen de curves zijn de trendlijnen.

volledige bestand geëncodeerd is.

1. Lees het volgende blok in.
2. Overloop het blok om de frequenties van elk karakter vast te leggen.
3. Bouw een Huffman boom m.b.v. de frequentielijst uit de vorige stap.
4. Schrijf de lengte van het te encoderen blok weg.
5. Schrijf de geëncodeerde boom weg.
6. Schrijf de codes weg van iedere byte uit het huidige blok.

Stap 2: Ik hou een lijst bij met grootte 256, waarvan elke index de waarde van een byte voorstelt. De waarde bij een bepaalde index is dan de frequentie van die byte.
Stap 3: Deze stap gebeurt op basis van algoritme 15 uit de cursus. Dit is het eigenlijke Huffman codering algoritme. Om dit algoritme eenvoudig te implementeren wordt gebruik gemaakt van een prioriteitswachtlijn. Deze is geïmplementeerd met een binaire hoop. De wachtlijn bevat een lijst van Huffman bomen. Initieel heeft elke Huffman boom juist 1 top, de wortel. Deze wortel heeft de waarde van een karakter uit het huidige blok en zijn frequentie in dit blok. De boom met de laagste frequentie is telkens de wortel van de prioriteitswachtlijn. Het algoritme wordt dan geïmplementeerd met het volgende stuk code: (De parameters en functienamen zijn hier vereenvoudigd.)

```
while (queue->length > 1) {
        push(merge(pop(), pop()));
}
```

Dus zolang er minstens 2 elementen in de prioriteitswachtlijn zijn, worden de eerste 2 elementen samengevoegd en opnieuw geplaatst in de prioriteitswachtlijn. Een laatste pop zorgt dan voor de uiteindelijke Huffman boom.
Stap 4: De decoder moet weten tot waar de code loopt om het bestand juist te kunnen decoderen. (Anders zou het niet duidelijk zijn wanneer de nieuwe boom juist begint). Een mogelijkheid zou zijn om een terminator toe te voegen aan de boom die aangeeft waar een blok eindigt, dit zou het programma overbodig ingewikkeld maken. Het eenvoudigst, en zeker ook niet slecht, is het schrijven van het aantal karakters dat een geëncodeerd blok bevat. Dit is gelijk aan het aantal karakters in het huididige blok. De groter de bestanden, de vaker dit gelijk zal zijn aan de maximaal grootte van de buffer. Dit is een constante in het programma. Om te zorgen voor een minimale overhead schrijf ik 1 bit die aanduidt of de lengte gelijk is aan de maximale buffergrootte. Als deze bit 0 is, is de lengte niet gelijk en schrijf ik nogmaals 32 bits die de werkelijke grootte voorstellen. Deze

32 bits zullen maar één keer weggeschreven worden. Dit is ideaal aangezien bij nagenoeg alle blokken er maar 1 bit bijkomstige overhead zal zijn om de lengte te encoderen.
Stap 5: Om de overhead van de boom te minimaliseren schrijf ik deze recursief uit. Dit gebeurt als volgt: (Opnieuw is de code aangepast om het meer leesbaar te maken.)

```
if (node->left && node->right) {
        write_bits(0, 1);
        write_tree_recurse(node->left);
        write_tree_recurse(node->right);
}
else {
        write_bits(1, 1);
        write_bits(node->value, 8);
}
```

Telkens wordt 0 geschreven zolang geen blad bereikt is. Zodra een blad bereikt is, wordt een 1 uitgeschreven gevolgd door de waarde van het blad. De boom wordt zo in een minimaal aantal bits uitgeschreven.
Stap 6: In de laatste stap worden de codes van alle bytes uitgeschreven naar het bestand. Ook dit gebeurt met buffers. Pas zodra de buffer vol zit of het volledige bestande is geëncodeerd wordt de buffer uitgeschreven. Om de codes eenvoudig te kunnen uitschrijven (en dus niet telkens in de boom te moeten zoeken), wordt een lijst met codes gemaakt zodat de code op een bepaalde index de code voorstelt van de byte gevormd door de index. (Bijvoorbeeld code op index 97 is de code van waarde 'a'.) Deze lijst wordt gemaakt door recursief de boom te doorlopen en zodra een blad bereikt wordt deze code toe te voegen aan de lijst met codes. Er is nog één addertje onder het gras. De maximale lengte van een code is 255 bits. Dit kan bijvoorbeeld gebeuren bij een bestand waarvan de frequenties gevormd zijn door de reeks van Fibonacci. Hiermee wordt ook rekening gehouden in de implementatie (desondanks dat dit probleem zich niet kan voordoen bij een relatief kleine buffergrootte, wat het geval is in mijn implementatie). Om toch dit scenario uit te sluiten wordt gebruik gemaakt van een lijst van integers om een code voor te stellen. Bij het uitschrijven van de codes wordt telkens de lijst overlopen en elke integer toegevoegd aan een buffer. De aparte bits worden telkens toegevoegd aan de buffer en zodra de grootte van buffer maximaal is wordt deze uitgeschreven. Uiteindelijk wordt nog éénmaal de buffer uitgeschreven om de restende bytes weg te schrijven. Een andere optie zou zijn telkens uitschrijven na het coderen van een blok. Maar het is mogelijk dat het geëncodeerde deel langer wordt dan het originele blok, zodat in dit geval vroeger zou moeten weggeschreven worden, de meeste gevallen echter zullen zorgen voor een kleinere geëncodeerde grootte, in dit geval zullen er minder I/O operaties zijn. In beide gevallen is het dus

eenvoudiger en voordeliger om pas uit te schrijven zodra de buffergrootte bereikt is.

Decoderen Het decoderen is heel wat eenvoudiger dan het encoderen. Ook hier wordt met blokken gewerkt, het decoderen begint met het eerste blok met de gegeven buffergrootte (of kleiner als het bestand niet zo groot is), en leest een nieuwe blok in zodra het huidige blok is verwerkt. Dit staat los van het decoderen en gebeurt intern bij het lezen van de bits. Ook het uitschrijven van de gedecodeerde bytes gebeurt in blokken, zodra het volledige decoderen gedaan is of de maximale grootte bereikt is, wordt de buffer uitgeschreven. Het decoderen van één blok gebeurt als volgt:

1. Lees de lengte.
2. Lees de boom op recursieve wijze.
3. Start in de wortel en zolang dat de index kleiner is dan de lengte:
   (a) Als de huidige top geen blad is, lees een bit en ga naar het linkerkind als de bit 0 is of naar het rechterkind als de bit 1 is.
   (b) Lees 8 bits en schrijf deze weg, ga naar de wortel.

Stap 1: Lees 1 bit, als deze 0 is, is de grootte van het volgende blok de maximum buffergrootte, zo niet volgen er 32 bits die de grootte specificeren. Dit geval doet zich maximaal 1 keer voor.
Stap 2: De boom wordt recursief gelezen. Indien een 0 gelezen wordt, worden 2 kinderen aangemaakt voor de huidige top, en wordt een recursieve oproep uitgevoerd op kinderen. Als een 1 gelezen wordt, krijgt de huidige top de waarde gespecificeerd door volgende 8 bits.
Stap 3: In de while-loop wordt de inhoud van een geëncodeerd blok gelezen door het doorlopen van de boom (0: neem linkerkind, 1: neem rechterkind) en telkens wanneer men een blad bereikt, specificeren de volgende 8 bits de waarde. Deze wordt dan weggeschreven naar de buffer.

I/O Om eenvoudig bits en bytes te kunnen lezen en schrijven van en naar een buffer, heb ik 2 aparte bestanden gemaakt die alle I/O operaties verzorgen. Zo kunnen deze operaties eenvoudigweg gebruikt worden in het algoritme. Deze bestanden verzorgen 4 belangrijke functies: read_bits, write_bits, read_block en write_byte. Deze bestanden bevatten elk 2 structs (bytereader, bytewriter en bitreader, bitwriter) die variabelen voor deze functies bijhouden.

De bytereader wordt gebruikt voor de functie read_block en de bytewriter voor write_byte. Deze houden onder andere een buffer bij met de gegeven buffergrootte. De write_byte samen met de bytewriter functie zorgt ervoor

dat het lezen en schrijven van blokken dynamisch kan gebeuren. Telkens een volledige blok verwerkt is, zorgt de buffer er zelf voor dat de huidige buffer weggeschreven wordt (bytewriter). De read_block functie zorgt ervoor dat telkens blok per blok kan verwerkt worden. Het gelezen blok bevindt zich dan in de bytereader. Aangezien het Huffman algoritme geen functie read_byte nodig heeft, werd deze niet geïmplementeerd. Het algoritme leest enkel de individuele bits, zodat enkel de functie read_bits nodig is.

De bitreader wordt gebruikt in de functie read_bits. Deze leest individuele bits (of meerdere indien mogelijk) uit een byte en geeft deze terug. Elke keer een volledig byte gelezen is, wordt de volgende opgeslagen in de bitreader. Zodra een volledig blok gelezen is, wordt een nieuw blok gelezen met behulp van de bytereader.

Als laatste wordt de bitwriter gebruikt om de individuele bits (bv. de Huffman boom) weg te schrijven. Hier wordt de write_byte functie gebruikt met de bytewriter om een byte weg te schrijven zodra de byte volledig is.

2 Specifiek algoritme

Algoritme In het specifieke algoritme maak ik gebruik van delta encoding in combinatie met Huffman encoding. Aangezien we een lijst van stijgende getallen als input hebben, is het ideaal om enkel het verschil met het vorig getal telkens op te slaan. Zo worden de getallen veel kleiner en zijn er bijgevolg minder bits nodig om het getal voor te stellen als het verschil klein genoeg is.

Er is ook een methode nodig om elk getal zo klein mogelijk voor te stellen. Een optie zou zijn om elk getal als een individueel decimaal teken voor te stellen. Voor elk getal is er dan compressie vanaf het verschil met het vorige getal +/- 10 keer kleiner is dan het originele getal. Er is namelijk 1 getal minder nodig om het verschil voor te stellen. Een getal kan echter beter voorgesteld worden in binaire voorstelling. Daarom maak ik gebruik van VLQ (Variable-Length Quantity) voor het encoderen van elke delta. Elk getal wordt voorgesteld in groepen van 7 bits. De meest significante bit van elke byte geeft aan indien nog een groep volgt. Deze methode bevat dus slechtst 1 bit overhead per byte. Na het encoderen van de getallen als verschillen (delta encoding), en het toepassen van VLQ op elk getal, pas ik vervolgens Huffman toe voor extra compressie.

Het decoderen gebeurt dan door het uitvoeren van de volgende stappen: decodeer een blok met Huffman, decodeer de individuele getallen uit dit blok door telkens een getal te vormen met alle groepen van 7 bits tot de

groep van 7 bits waarvan de byte begint met 0. Tel vervolgens dit getal op met het vorige getal om het originele getal te bekomen.

Als laatste is het ook mogelijke om alle komma's (en [,]) uit het bestand weg te laten in het gecomprimeerde bestand. Door VLQ is het namelijk mogelijk om te weten wanneer een getal eindigt.

I/O Ook hier moet de I/O verzorgd worden. Ik heb de volgende functies gemaakt in de encoder en decoder:

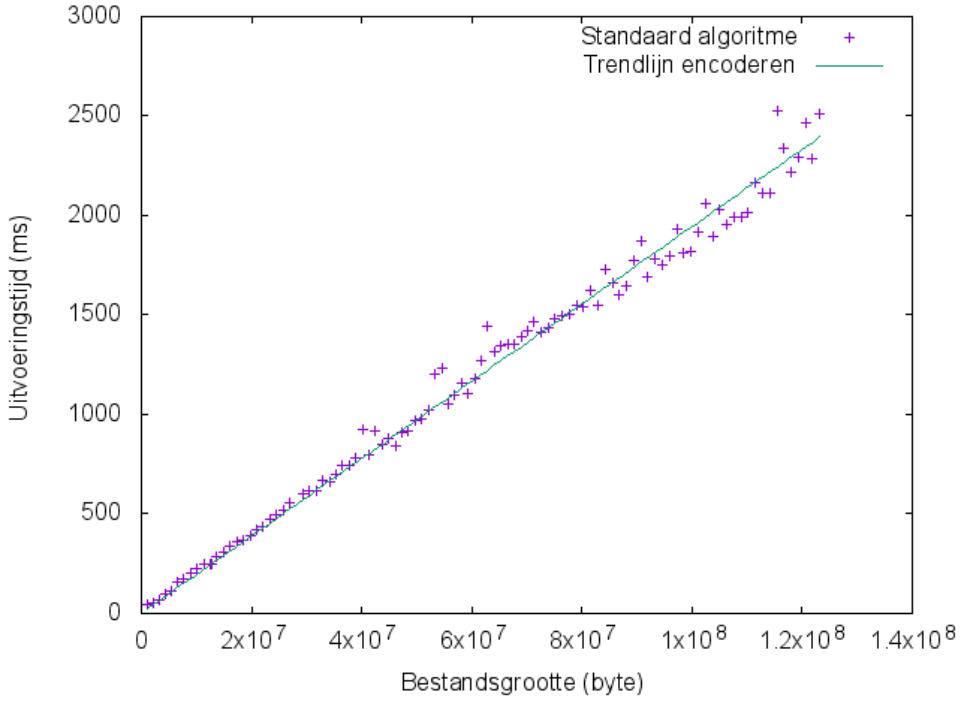
- De functie read_long geeft de volgende unsigned long long weer uit de inputbuffer. Intern wordt de functie strtoll hiervoor gebruikt. Aangezien de buffer vaak niet de volledig inhoud van het bestand inslaat, is het mogelijk dat geen volledig getal gelezen werd met strtoll. Hierdoor ontstonden enkele moeilijkheden in de implementatie maar dit werd opgelost door het inlezen van het volgende blok zodra de buffer leeg is. De delta wordt vervolgens aangevuld met eventueel startende nullen uit het nieuwe blok en hierna wordt de rest van het getal geconcateneerd, die ook gelezen werd met strtoll.

- De functie write_number encodeert een getal met VLQ en schrijft het naar de outputbuffer. Eens de outputbuffer volledig is, wordt deze geëncodeerd met Huffman codering.

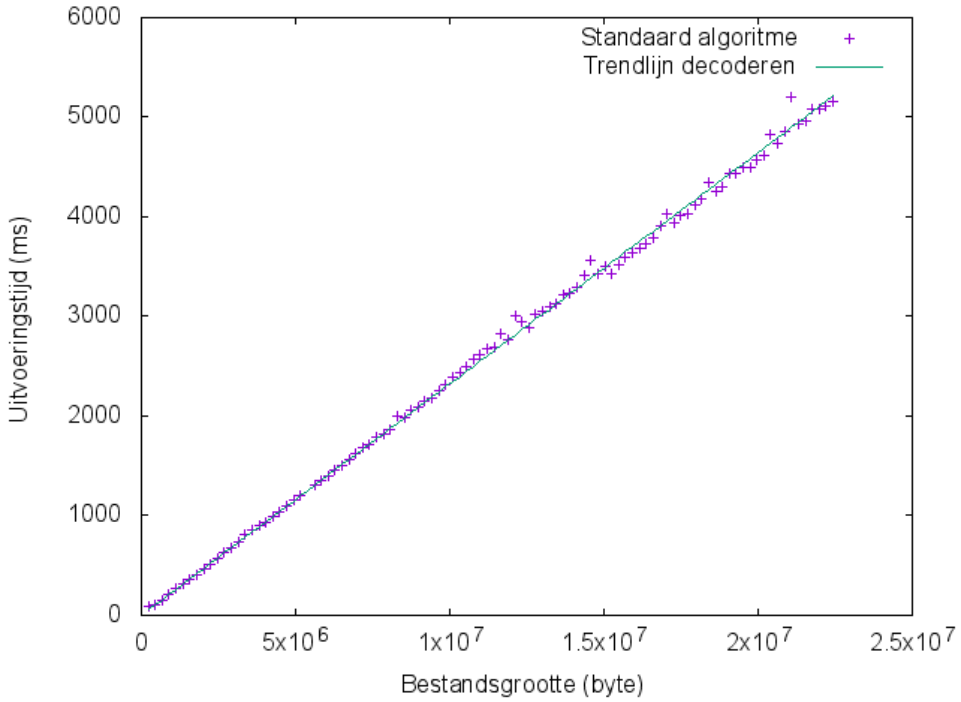
- De functie read_number leest een getal geëncodeerd met VLQ en geeft deze terug.

- De functie write_long maakt gebruik van een bytewriter om de uitvoer uit te schrijven naar een bestand. Een unsigned long long kan uitgeschreven worden met snprintf. Ook hier werd rekening gehouden met het feit dat de buffer eventueel niet het volledige getal kan opslaan.

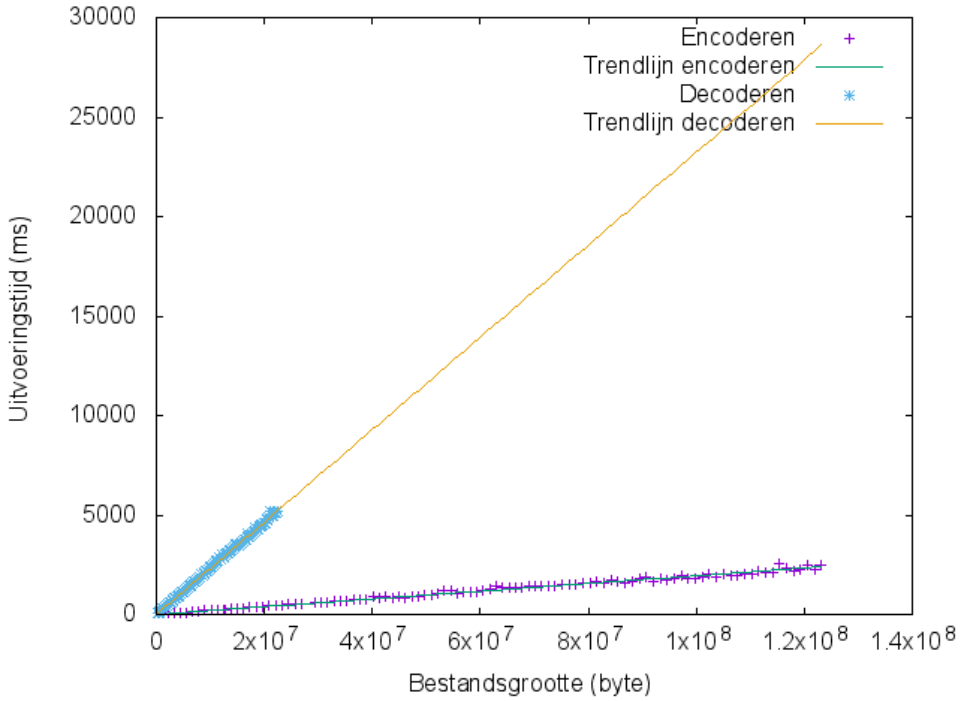
Performantie Hier bespreek ik de performantie van het specifieke algoritme. De performantie van het encodeeralgoritme wordt getoond op figuur 2, het decodeeralgoritme op figuur 3 en de vergelijking van beide op figuur 4. Er is net als in het standaardalgoritme een duidelijk verschil tussen de snelheid van het encoderen en decoderen. De verhouding $\frac{uitvoertijd}{bestandsgrootte}$ stijgt veel sneller bij het decoderen. De reden hiervoor is hetzelfde als vermeld in de bespreking van het Huffman algoritme. Aangezien ook hier Huffman toegepast wordt, zal het decoderen, net als bij Huffman, ook trager werken dan het encoderen.



Figuur 2: Deze grafiek beschrijft de performantie van het specifieke encodeeralgoritme. De rechte doorheen de curve is de trendlijn.



Figuur 3: Deze grafiek beschrijft de performantie van het specifieke decodeeralgoritme. De rechte doorheen de curve is de trendlijn.



Figuur 4: Deze grafiek beschrijft de performantie van het specifieke encodeer - en decodeeralgoritme. De rechtes doorheen de curves zijn de trendlijnen.

3 Vergelijking algoritmes

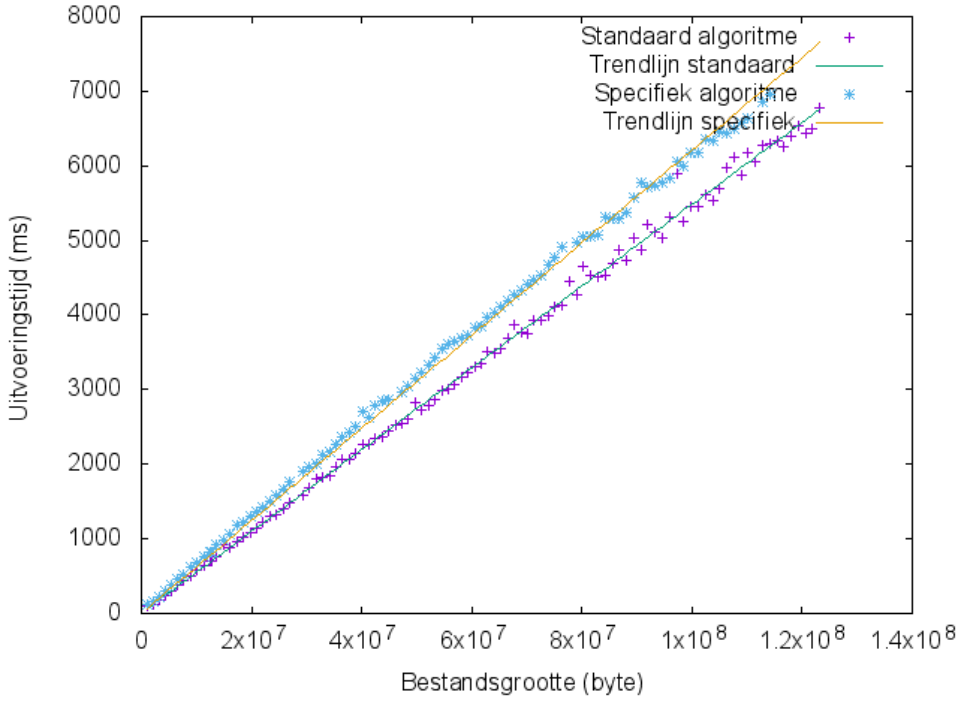
3.1 Vergelijking snelheid

Op figuur 5 kunnen we de snelheid zien van het encoderen en decoderen van een volledig bestand met de beide algoritmes. De verhouding $\frac{uitvoertijd}{bestandsgrootte}$ stijgt sneller bij het specifieke algoritme. Dit is logisch aangezien het specifieke algoritme enkel een toevoeging is voor het standaard algoritme. Uit de punten op de grafiek wordt het duidelijk dat er slechts een klein verschil is bij uitvoeringstijd tussen het standaardalgoritme en het specifieke algoritme. De reden hiervoor is dat Huffman veel trager presteert dan de toegevoegde delta-codering waardoor deze toevoeging slechts een kleine rol zal meespelen in de uitvoeringstijd. Het nieuwe algoritme werkt byte per byte, maar het Huffman algoritme werkt met de individuele bits.

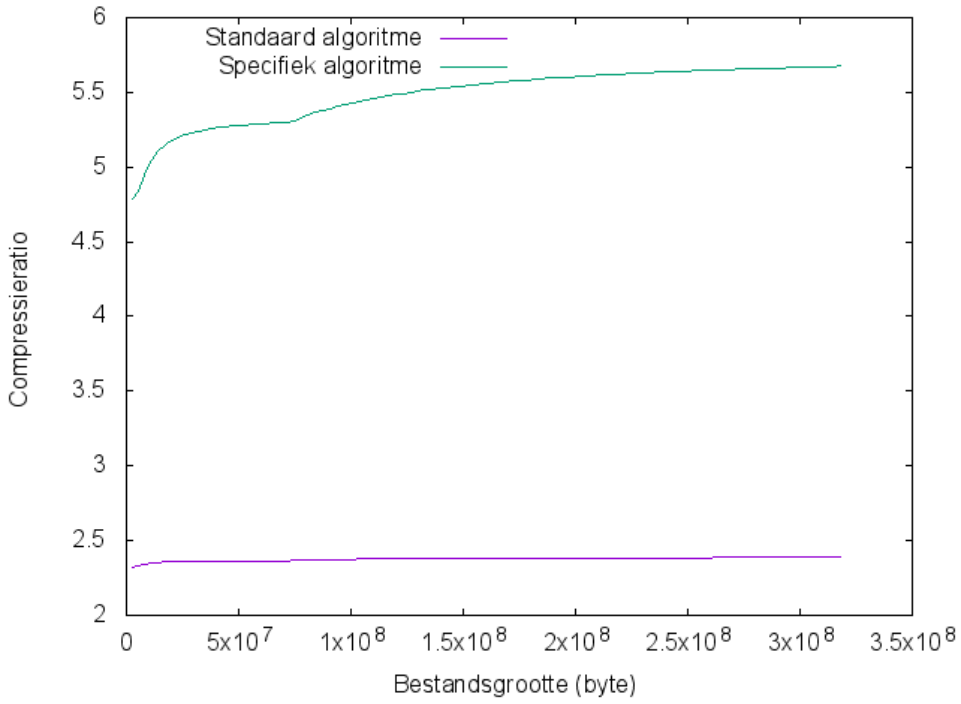
3.2 Vergelijking compressieratio

Als volgt bespreek ik het compressieratio van beide algoritmes bij groeiende bestandsgroottes. Hier verwacht ik voor beide algoritmes een constante functie. Bij Huffman verwacht ik een constante functie omdat elk blok apart gedecodeerd wordt, en elk blok bevat willekeurige getallen. Dus gemiddeld gezien zou elk blok dezelfde grootte moeten hebben na het encoderen. Bij verschillende bestanden zet deze trend zich verder waardoor het voorgaande moet gelden en dus moet het compressie-ratio onafhankelijk zijn van de bestandsgrootte. Bij het specifieke algoritme verwacht ik een gelijkaardige trend. We kunnen op figuur 6 waarnemen dat dit nagenoeg het geval is voor het standaardalgoritme, we merken enkel een kleine stijging in compressieratio aan de linkerkant van de grafiek omdat de overhead van het algoritme een grotere rol zal spelen bij zeer kleine bestanden. Bij grotere bestanden echter is deze overhead verwaarloosbaar klein.

Voor het specifieke algoritme merken we eerdere een logaritmisch stijgende functie desondanks de verwachtingen. Dit valt ook te verklaren, bij kleine bestanden zullen de getallen over het algemeen kleiner zijn dan bij grote bestanden. De delta's zullen dus dichter bij de originele waarde liggen, waardoor delta-encoding zorgt voor minder compressie. Bij grotere bestanden daarentegen zullen de delta's veel kleiner zijn ten opzichte van de inputgetallen, waardoor het compressieratio stijgt. Het initieel snel stijgen van de functie valt te verklaren door de manier waarop de bestanden opgebouwd zijn. De bestanden worden opgebouwd door telkens een willekeurige waarde, gegenereerd door de functie rand() in C, op te tellen bij het vorige getal. Dit wordt gedaan voor elk getal tot het bestand volledig is. De maximum waarde verkregen door rand() is 32767, dit zorgt ervoor dat de delta's relatief veel groter zijn in het begin van de file. De kleinste bestanden bevatten enkel/vooral deze relatief grote verschillen t.o.v. de originele getallen en



Figuur 5: Deze grafiek beschrijft de performantie van het standaardalgoritme en het specifieke algoritme.



Figuur 6: Deze grafiek beschrijft de compressieratio ten opzichte van de bestandsgrootte van het standaardalgoritme en het specifieke algoritme.

zullen dus veel slechter comprimeren.

Wanneer we beide algoritmes vergelijken kunnen we duidelijk het effect zien van het delta encoderen. Het compressieratio wordt ongeveer 2-3 keer beter door het toevoegen van de delta-encodering in combinatie met VLQ. Al deze algoritmes samen zorgen dus voor een compressieratio van maar liefst 5,6.

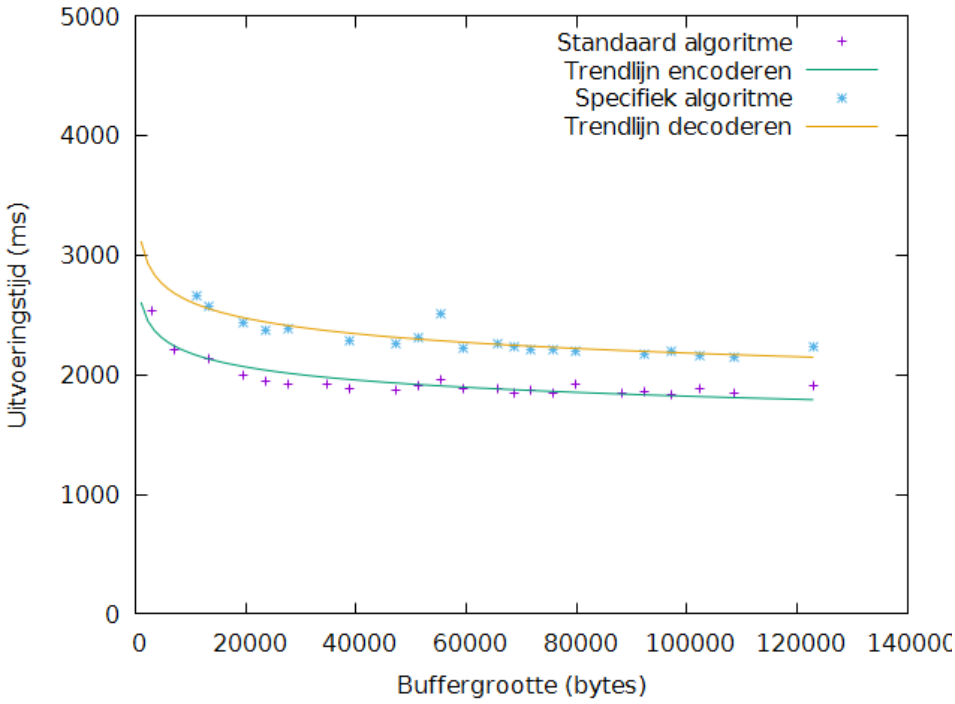
3.3 Verschillende buffergroottes

Ik zal hier het effect nagaan van verschillende buffergroottes bij beide algoritmes op uitvoertijd en compressieratio. Vervolgens zoek ik op deze manier het punt waarvoor de algoritmes optimaal werken. Hierbij kiezen we een vaste bestandsgrootte van 126 MiB (10^7 getallen).

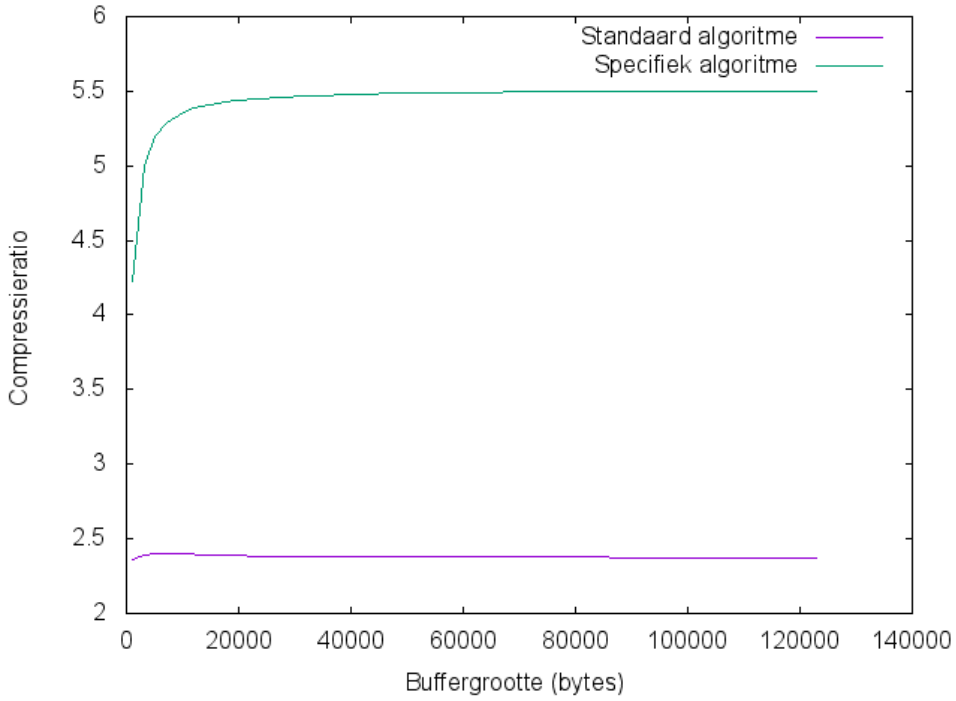
Uitvoertijd Op figuur 7 vinden we de uitvoertijd op een bestand van 126 MiB met een variabele buffergrootte. We krijgen als trendlijn een logaritmisch dalende functie voor beide algoritmes. Per blok wordt het algoritme uitgevoerd. Alle acties die onafhankelijk zijn van de grootte van de buffer zullen vaker gebeuren bij kleine buffers. Voorbeelden hiervan zijn het inlezen en uitschrijven van de buffer en het correct instellen van variabelen. Bij het specifieke algoritme is er weinig bijkomstige overhead per blok. Dit verklaart waarom de beide functies even snel dalen in uitvoeringstijd.

Compressieratio Op figuur 8 zien we het compressieratio van de beide algoritmes bij een constante bestandsgrootte van 126 MiB en een variabele buffergrootte. Neem eerst de functie van het specifieke algoritme. Initieel zien we een kleine stijging in compressie bij zeer lage buffergroottes. Vervolgens zien we over de hele lijn een zeer traag dalende compressieratio.

Het compressieratio van Huffman wordt bepaald door een afweging van overhead en lokaliteit (lokaal hogere frequenties van karakters). Bij kleinere buffergroottes zal elk individueel buffer kleiner gecodeerd worden. Neem bijvoorbeeld 2 opeenvolgende buffers van elk 1024 bytes groot, waarbij in de eerste buffer er een groot aantal karakters '1' zijn en weinig karakters '2', en in de tweede buffer er een groot aantal karakters '2' zijn en weinig karakters '1'. Nu zal elke buffer van 1024 bytes een korte code geven aan het karakter waarbij de frequentie het hoogst is, en een lange code aan het karakter met een lagere frequentie. Echter bij een buffer van 2048 zullen de frequenties van beide karakters ongeveer gelijk zijn, en zullen de codes gemiddeld gezien langer zijn. Dit betekent dat het gecomprimeerde bestand groter zal zijn en dus het compressieratio lager. Dit verklaart het dalende compressieratio bij groeiende buffergroottes. Een uitzondering hierop is een zeer kleine buffergrootte. Elke buffer krijgt een zekere overhead door de Huffman boom. Deze overhead wordt kleiner als een grotere buffergrootte genomen wordt.



Figuur 7: Deze grafiek beschrijft de uitvoertijd ten opzichte van de buffergrootte van het standaardalgoritme en het specifieke algoritme voor een bestand van 126 MiB.



Figuur 8: Deze grafiek beschrijft het compressieratio ten opzichte van de buffergrootte van het standaardalgoritme en het specifieke algoritme voor een bestand van 126 MiB.

Deze overhead is zeer klein en valt enkel op bij zeer kleine buffergroottes waardoor de lokaliteit vooral bepalend is voor het compressieratio vanaf een zekere buffergrootte. Een optimum vond ik op een buffergrootte 5120 bytes, hier verkreeg ik een compressieratio van 2,4 bij het Huffman encoderen van een bestand van 126 MiB.

Het compressieratio van de delta-encoder stijgt over de hele lijn. Initieel is er een heel sterke stijging in compressieratio. De reden hiervoor is dat de overhead een grotere rol zal spelen dan de lokaliteit besproken bij het Huffman-algoritme. De overhead zal groter zijn dan de winst verkregen door lokaliteit waardoor er een betere compressieratio is bij grotere buffers.

We kunnen dus besluiten uit deze resultaten dat we best een kleine buffergrootte nemen voor Huffman en een grote buffergrootte voor het specifieke algoritme.

3.4 Best- en worst-case scenarios

Standaardalgoritme De grootte na compressie van een blok met de Huffman encodering is afhankelijk van 2 zaken: het aantal karakters en de balancering van de prefixboom. De balancering van de Huffman boom is afhankelijk van de frequenties van de verschillende karakters. Als alle karakters een gelijke frequentie hebben, en er een even aantal verschillende karakters zijn, dan is de Huffman boom optimaal gebalanceerd (men kan dit nagaan met een simpel voorbeeld). In dit geval zullen alle karakters een code hebben met gelijke lengte. De lengte is dan steeds gelijk aan log2(x) met x het aantal verschillende karakters. Als alle 256 karakters gebruikt worden zal elke code lengte 8 hebben. Dit levert echter geen compressie aangezien een karakter altijd voorgesteld kan worden met 8 bits. Indien niet alle karakters gebruikt worden, zal de lengte van elke code kleiner worden, en zal er compressie zijn.

Een optimaal geval wordt verkregen met een Huffman boom waarbij alle karakters kunnen voorgesteld worden met 1 bit. Dit is een Huffman boom met diepte 1. Hierin kunnen maximaal 2 karakters voorgesteld worden. Ik zal nu het compressieratio nagaan bij dergelijk optimaal geval. Een bestand met een gelijk aantal nullen en enen met originele grootte 200 MiB werd gecomprimeerd tot 25.0005 MiB (0.0005 MiB overhead). Het compressieratio is nu 8. Dit resultaat is correct, elk karakter van 8 bits, kan nu voorgesteld worden met juist 1 bit.

Een worst-case kan worden gemaakt door de boom zeer slecht te balanceren en elk mogelijk karakter te gebruiken. Zo zullen de codes langer worden dan 8 bits. Het slechtst mogelijk geval verkrijgen we door het genereren van een bestand waarvan de frequenties van elk karakter gelijk zijn aan de Fibonaccireeks. Deze boom zal op elke diepte juist één karakter bevatten waardoor een code maximaal lengte 256 zal hebben. De code van dit karakter is dan maar liefst 32x langer dan de grootte van het originele karakter. Echter bij het encoderen van dergelijk bestand krijg ik nog steeds een zeer hoge compressieratio. De reden hiervoor is dat de slecht gebalanceerde boom zich bevindt in een van de blokken. De overige blokken daarentegen worden zeer goed geëncodeerd omdat hun frequenties zeer hoog zijn, en elk blok weinig karakters bevat.

Specifiek algoritme Het compressieratio hier hangt af van de grootte van de delta's ten opzichte van de originele getallen. Neem bijvoorbeeld als input [1,2,3,4,5]. Na delta-encodering zal het bestand eruit zien als [1,1,1,1,1]. Aangezien ik VLQ gebruik zal elk getal minstens voorgesteld worden met 8 bits. Hier zal er dus geen compressie zijn (behalve na het weglaten van komma's). Nemen we nu als voorbeeld [10,11,12,13,14], kan dit worden gecomprimeerd als [10,1,1,1,1]. Initieel werd elk getal voorgesteld met 16 bits

(2 karakters). Nu zal elk getal echter in een groep van 7 bits kunnen opgeslagen worden (aangevuld met 1 bit overhead). We krijgen dus in theorie een grootte van 5*8=40 bits in de plaats van een grootte 5*16=80 bits. De compressieratio is nu 2. Hierdoor wordt het duidelijk dat de compressie het grootst is als de delta's veel kleiner zijn dan de originele getallen. Deze verschillen mogen maximaal $2^8 - 1$ zijn om optimale compressie te verkrijgen. Ik bespreek de volgende 2 gevallen, waarbij we telkens als startwaarde 2^{62} nemen bij compressie:

1. Als ik telkens bij de vorige waarde, beginnend bij de 2^{62} het getal 127 optel, dan krijg ik een compressieratio van 160 (200 MiB $\rightarrow$ 1.25 MiB). (Er is +/- gelijke compressie bij het nemen van een ander getal in $]0, 2^8[$).

2. Als ik telkens bij het vorige getal een willekeurige getal optel in het interval $]0, 2^8[$ krijg ik een compressieratio van 22,8. (200 MiB $\rightarrow$ 8.76 MiB).

Hier wordt de combinatie van Huffman encodering, delta-encodering en VLQ goed weergegeven. In geval 1 zal elke delta dezelfde vorm aannemen, namelijk 0111111. Door VLQ is het mogelijk om dit getal met 8 bits op te slaan i.p.v. 64 bits. Aangezien enkel deze delta voorkomt, zal Huffman deze delta's zeer compact kunnen opslaan en verkrijgen we een zeer hoog compressieratio. Dit is het beste geval. In het tweede geval zal Huffman niet zeer effectief zijn, aangezien eender welke waarde in het interval $[0, 127]$ mogelijk is als input voor dit algoritme. Het is echter nog steeds het beste geval voor de delta-encoder alleen, waardoor we nog steeds het hoge compressieratio van 22,8 krijgen. Merk op dat het compressie ratio van geval één ongeveer 8 keer hoger is dan geval 2. Hierdoor kunnen we zien dat Huffman inderdaad elk getal als 1 bit opslaat in het eerste geval, en als 8 bits in het tweede geval.

Het slechste geval is een bestand waarbij de verschillen heel groot zijn, en de verschillen niet op elkaar gelijken (zodat Huffman deze niet efficiënt encodeert). Hiervoor genereerde ik willekeurige getallen van 61 bits en telde deze telkens bij de vorige op. Bij een invoerbestand van 127 bytes (het bestand is zeer klein omdat de verschillen zeer groot moeten zijn), is het gecomprimeerde bestand na compressie nog 107 bytes groot. Het compressieratio is hier slechts 1,19.

4 Versturen over internet

Als eerst zou ik zeker en vast de getallen delta-encoderen. Dit is zeer snel en levert veel compressie op. Huffman daarentegen werkt traag, het moet eerst een heel blok doorlopen om de frequenties te bepalen en moet vervolgens nog eens alle karakters doorlopen uit het blok om de codes uit te schrijven.

Het decoderen duurt nog langer omdat het bit per bit het bestand moet doorlopen. Nu zijn er 2 opties:

1. Delta-encodeer de invoer en pas geen Huffman toe. De winst van Huffman bij het delta-encoderen is niet zeer groot aangezien de bytes alle vormen kunnen aannemen. Enkel bij bepaalde bestanden waarbij de delta's in een klein interval liggen zorgt Huffman voor een grote verbetering in compressie. Dit voordeel weegt echter niet op tegen de tijd nodig om de bestanden hierdoor te comprimeren. Een optie is dus om Huffman gewoonweg achterwege te laten.

2. Delta-encodeer de invoer en pas adaptieve Huffman toe. Bij adaptieve Huffman codering wordt de boom online opgebouwd en wijzigt de boom als de frequenties wijzigen. Dit zorgt ervoor dat de invoerstroom niet eerst volledig doorloopt moet worden om de Huffman boom te bepalen waardoor het algoritme veel sneller zal werken. Bij het decoderen doen we hetzelfde. Zoals eerder vermeld zorgt Huffman voor weinig extra compressie bij het encoderen waardoor het weinig nut heeft om dit toe te passen. Dit zal echter wel voordelig zijn als er geen delta-encodering is.

Bij het specifieke algoritme passen zou ik dus enkel delta codering toepassen, en bij het standaardalgoritme zou ik adaptieve Huffman codering toepassen.

Mijn huffman algoritme heeft een compressiesnelheid van gemiddeld 58 MB/s en dus 464 Mbit/s. Het is dus voordelig om eerst Huffman toe te passen tot een bitrate van 464 Mbit/s, bij een hogere bitrate zal het sneller zijn om de bestanden zonder compressie door te sturen. Het specifieke algoritme heeft gemiddeld een compressiesnelheid van 44 MB/s en dus 352 Mbit/s. Hier is het voordelig om compressie toe te passen tot een bitrate van 352 Mbit/s.