Project DA3: Datacompressie

Rien Maertenis
3de Bachelor Informatica

11 mei 2016

1 Standaard algoritme

1.1 Keuze van het algoritme

De keuze van het algoritme is vooral gebaseerd op redeneringswerk. We vatten de eigenschappen van de mogelijke compressie-algoritmen samen:

LZW Compressiealgoritme waarin een woordenboek wordt bijgehouden die een codewoord van variabele lengte mapt op n of meerdere karakters. Dit algoritme is vooral efficient waneer vaak dezelfde (lange) reeks karakters in de data voorkomt. Omdat voor deze volledige reeks maar n codewoord nodig is.

LZ77 Bij dit compressie-algoritme wordt er met een sliding window gewerkt. Hier krjgen we dus ook efficinte compressie wanneer langere reeksen karakters vaak voorkomen.

Huffman Deze encodering bepaalt het aantal bits van een codewoord aan de hand van het aantal voorkomens in de gegeven data. Karakters die vaak voorkomen krijgen een kortere code, waardoor een goeie compressie wordt bereikt wanneer een klein aantal karakters vaak voorkomen.

Burrows-Wheeler Dit algoritme maakt een tekst niet kleiner. Het is een transformatie die het gemakkelijker maakt om bestanden beter te comprimeren. Na deze transformatie wordt vaak move-to-front toegepast en vervolgens huffman encoding. Dit heeft enkel echter een invloed wanneer vaak dezelfde opeenvolging van karakters in de data zitten.

Huffman encoding leek mij de beste manier om de gegeven data te comprimeren. De gegeven data heeft maar een alfabet van dertien karakters: de tien cijfers, de komma, en de twee vierkante haakjes. De cijfers, die het vaakst voorkomen, worden door huffman omgezet naar codewoorden van drie tot vier bits. Hierdoor verwachten we minstens een halvering van de originele bestandsgrootte.

Ik heb niet gekozen voor de LZW of LZ77 omdat deze algoritmes langere codewoorden nodig hebben voor opeenvolgende karakters zonder structuur (zoals de minst significate cijfers van een groot getal). Terwijl dat Huffman voor ieder karakter in ons alfabet een kortere code gebruikt.

Naast Huffman heb ik ook Burrows-Wheeler in combinatie met move-to-front gemplementeerd. Maar dit bleek achteraf geen goede keuze te zijn. De compressiewinst woog niet op tegen de extra rekentijd die nodig was om de transformatie te verwezenlijken. Daarnaast waren er ook wat problemen met het implementeren van de decompressie waardoor ik besloten heb de Burrows-Wheeler implementatie niet af te werken. De code en de tests zijn nog altijd terug te vinden in het project.

1.2 Implementatie

Bij mijn compressieprogramma heb ik een module die zich enkel bezighoud met het bitgewijs lezen en schrijven van data: `bitcode.h` en `bitcode.c`. Deze maken het mogelijk om n enkele bit of byte te lezen of te schrijven en om een bitcode te concateneren aan de ander. Om dit zo efficient mogelijk te maken worden er zo weinig mogelijk bitoperaties gebruikt. Bijvoorbeeld: twee keer een OR-operatie om een niet-gealigneerde byte te schrijven in plaats van acht OR-operaties met de individuele bits. De gealloceerde array van data groeit ook exponentieel (zoals een arraylist zou doen) om toevoegen in constante tijd te laten doorgaan.

1.2.1 Huffman

Mijn implementatie van het Huffman-algoritme volgt grotendeels dat die in de cursus bescheven staat. Dit zijn nog enkele implementatiedetails die het vermelden waard zijn:

De prioriteitswachtlijn die wordt gebruikt in mijn Huffman-implementatie is te vinden in `priorityqueue.c` en `priorityqueue.h`. De implementatie daarvan is redelijk naief: de items worden van groot naar klein gesorteerd, waardoor het kleinste element verwijderen in constante tijd gebeurt. Een nieuw element invoegen gebeurt door het item op het einde van de lijst toe te voegen en de hele lijst te sorteren. Aangezien de lijst een maximale grote heeft van 256 elementen (n per byte) en deze priorityqueue enkel wordt gebruikt bij het opbouwen van de Huffman-boom loont het niet de moeite om dit te optimaliseren.

Het opslaan van de Huffman-boom zelf gebeurt als volgt[1]: we overlopen de boom in pre-orde en bij iedere top schrijven we uit:

- De bit `0` wanneer we een inwendige top hebben.

[1]Het idee van dit formaat komt van het volgende StackOverflow-antwoord: `http://stackoverflow.com/a/759766/4424838`

- De bit `1` wanneer we een blad hebben, gevolgd door het karakter dat door die top wordt voorgesteld.

Op die manier kan er op een recursieve manier door de boom gelopen worden om ze op te slaan en kan ze ook recursief ingelezen worden, terwijl het codewoord voor ieder karakter wordt ingesteld (met de methodes `huffman_build_dictionary()` en `huffman_reconstruct_tree()` in `huffman.c`).

Uiteindelijk is er een compressieratio van $2,3$ bereikt.

1.2.2 Burrows-Wheeler

Na het implementeren van Burrows-Wheeler was het snel duidelijk dat dit algoritme niet voldeed aan mijn verwachtingen: de compressiewinst was te klein en het snelheidsverlies te significant. Dit is ook logisch, de Burrows-Wheeler transformatie is pas efficient wanneer de tekst vaak voorkomende opeenvolgingen van karakters heeft. Wat niet het geval is in onze testdata. Omdat ik besloten heb de Burrows-Wheeler implementatie niet af te werken heb ik hier helaas geen testresultaten van.

De volgende implementatiedetails zijn het vermelden waard:

- Bij de encodering wordt er gebruikt gemaakt van circulaire strings (`circular_string.c`) die een pointer bevatten naar de originele string en een offset die hun rotatie aangeeft. Dit maakt het sorteren en roteren efficinter.
- Het sorteren van de strings gebeurt momenteel met `qsort`, wat hoogstwaarschijnlijk de grootste bottleneck was. Omdat er gebruik wordt gemaakt van functiepointers is het moeilijker om daar te optimaliseren. Handmatig het sorteeralgoritme schrijven zou waarschijnlijk een significate performantiewinst als resultaat hebben.
- De index van het originele eerste karakter in de string wordt niet meegecodeerd met huffman, maar op voorhand uitgeschreven.
- Bij het decoderen wordt er gewerkt met pointers naar karakters die worden gesorteerd. Aan de hand van die pointers en het laastst gedecodeerde karakter wordt de positie van het volgende karakter bepaald.

2 Specifiek algortime

2.1 Algoritme

Mijn specifiek compressie-algoritme past de volgende stappen na elkaar toe:

Binaire voorstelling $\rightarrow$ Delta encoding $\rightarrow$ Variable-length integers

2.1.1 Omzetten naar binaire voorstelling

Als eerste stap worden de getallen in de inputfile omgezet van ASCII-waardes naar hun binaire voorstelling. Hierdoor moeten de kommas en haakjes niet meer opgeslagen worden. De grootste winst wordt echter geboekt bij het voorstellen van de getallen zelf: ze hebben nu altijd 8 bytes nodig, in plaats van gemiddeld 19 bytes (voor uniforme verdeling).

2.1.2 Delta encoding

In plaats van de volledige getallen op te slaan, wordt het verschil met het vorige getal opgeslagen (of zelfs dit verschil -1 aangezien de getallenreeks strict stijgend is). Dit heeft als voordeel dat de getallen kleiner zijn, en dus beter kunnen gecomprimeerd worden bij de volgende stap.

2.1.3 Variable-length integers

De laatste stap is het omzetten naar integers met een variabele lengte. Deze voorstelling begint met 3 bits die een getal n tussen 1 en 8 voorstellen (het getal 0 wordt niet gencodeerd), gevolgd door de n minst significante bytes van het gencodeerde getal.

Dit zorgt ervoor dat grote getallen in het slechtste geval een lengte krijgen van 67 bits. Maar aangezien we in de vorige stap enkel de verschillen tussen opeenvolgende getallen hebben opgeslagen, zullen de getallen gemiddeld veel kleiner zijn, waardoor een redelijk goede compressie wordt behaald.

Uiteindelijk verkrijgen we een compressieratio van ongeveer $3,5$.

2.1.4 (Huffman)

Origineel werd na de laatste stap nog Huffman-encoding toegepast, maar later heb ik die weggehaald omdat geen significante compressiewinst meer werd verkregen. Integendeel, sommige (grotere) bestanden werden minder goed gecomprimeerd hierdoor.

3 Vergelijking algoritmes

Zoals verwacht is het specifieke algoritme beter in het comprimeren van de gegeven bestanden. We kunnen namelijk een aantal assumpties maken die we bij het standaard algoritme niet kunnen (omdat we alle mogelijke inputbestanden moeten kunnen comprimeren).

3.1 Best- en worst-case gevallen

3.1.1 Standaard

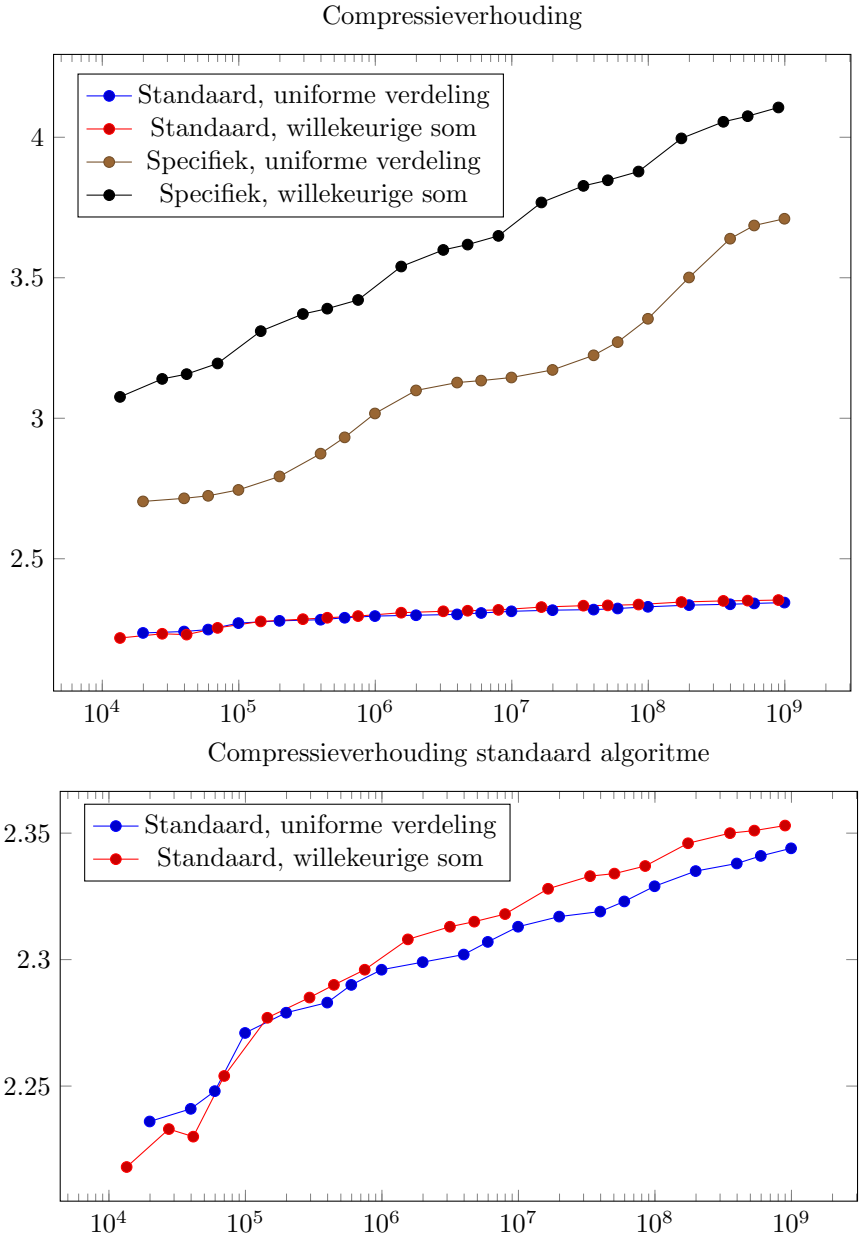
Het worst-case scenario bij Huffman-encodering doet zich voor wanneer het aantal voorkomens van de symbolen voldoet aan de fibonacci-reeks. Deze resulteerd in een volledig ongebalanceerde Huffman-boom die de data encodeerd met meer bits dan in het originele bestand. De restrictie die onze data oplegt maakt het echter moeilijk om hieraan te voldoen. In de praktijk is de kans dat dit voorkomt zo goed als 0.

In het beste geval zijn er nog een aantal symbolen die ontbreken in de gegeven data. In dat geval zijn er nog minder bits nodig om de resterende symbolen voor te stellen. Natuurlijk is ook dit een geval dat in de praktijk niet zal voorkomen.

3.1.2 Specifiek

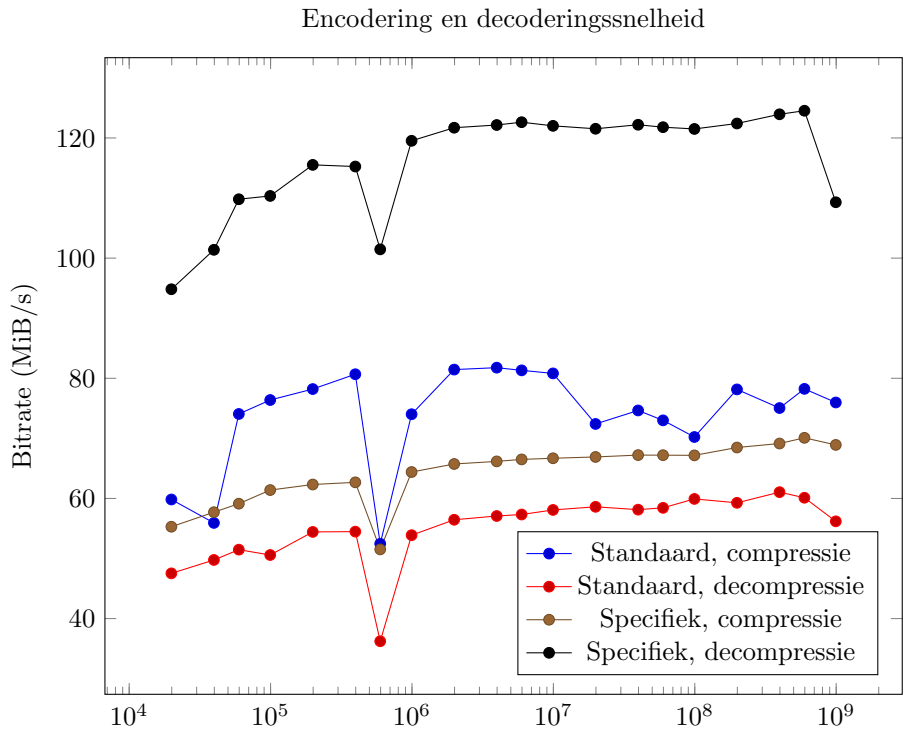
Het worst-case scenario van het specifieke algoritme, is wanneer er een zo gelijke, zo groot mogelijke afstand tussen elk opeenvogend paar getallen zit. Door de delta-encoding en de variabele lengte van de integers zal het verschil dan meer bits nodig hebben om opgeslagen te kunnen worden. De stijgende aard van de getallen en de bovengrens zorgen er echter voor dat dit niet uit de hand kan lopen: als de afstand zo groot mogelijk moet zijn, dan zit ieder getal gelijk verdeeld over het totale bereik. Maar hoe meer getallen er gecodeerd worden (en dus hoe groter het bestand), hoe minder ruimte er kan zijn tussen opeenvolgende getallen. Dit verklaart waarom een stijging in bestandsgrootte ook een stijging in de compressieratio als gevolg heeft.

Het beste geval doet zich dan weer voor wanneer er tussen ieder getal een zo klein mogelijk verschil is. Wanneer het verschil tussen twee getallen kleiner is dan 256 hebben we maar 11 bits nodig om it voor te stellen (de 3 bits die aanduiden hoeveel bytes er nodig zijn + de minst significante byte van het verschil). Als dus uitgaan van een gemiddelde lengte van 19 karakters per getal (in uniforme verdeling) dan bereiken we een compressieratio van $(19 \cdot 8)/11 \approx 13,8$.



Figuur 1: Vergelijking van de compressieratio van de twee compressiealgoritmen. Met het verschil tussen de uniforme verdeling en willekeurige som van het standaard algoritme uitvergroot.

4 Versturen over internet



Figuur 2: De encodering en decoderingssnelheid van de algoritmes. Dit wordt berkend door de grootte van het originele bestand te delen door de tijd die nodig was om het bestand te comprimeren.

Door een bestand te comprimeren voor we ze over het internet versturen kunnen we onze data vlugger doorsturen. Onze data zit namelijk compacter verpakt, en aan eenzelfde transmissiesnelheid wordt er meer informatie verstuurd. Wanneer we bijvoorbeeld een bestand van 100 Mebibyte kunnen versturen aan 10 MiB/s zal dit oorspronkelijk ongeveer 10 seconden duren. Comprimeren we dit echter met een compressieratio van 2, dan zal dit maar 5 seconden duren. De nieuwe snelheid kunnen we berekenen aan de hand van de volgende formule:

$$datasnelheid = transmissiesnelheid \cdot compressieratio$$

Om de data naar de ontvanger te sturen moet die eerst gecodeerd worden, doorgestuurt worden over het internet, en vervolgens gedecodeerd worden. Omdat n van die drie stappen de data slechts kan verwerken aan de snelheid die ze binnekrijgt of kan uitsturen, is de totale snelheid van deze ketting gelijk aan de traagste van deze drie stappen. Optimaal zou zijn wanneer alledrie de snelheden gelijk zijn, dan is $compressiesnelheid = datasnelheid = decompressiesnelheid$.

We nemen nu het slechte geval van onze metingen: daar was de decompressiesnelheid van het standaard algoritme gelijk aan 40 MiB/s. Nemen we nu

een compressieratio van 2,3 dan hebben we een transmissieratio van minstens 17,4 MiB/s nodig om de data aan een optimale snelheid binnen te krijgen. De gemiddelde downloadsnelheid van een internetverbinding in Belgie ligt een pak lager dan dat. (Een speedtest op eduroam levert doorgaans een snelheid van 40Mib/s = 5 MiB/s)

Pas wanneer de transmissiesnelheid hoger is dan de encoderings- of decoderingssnelheid loont het niet meer de moeite om de data te comprimeren. Met de huidige telecommunicatiemarkt zal dat niet het geval zijn.