Project Datacompressie

Jarre Knockaert

1 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 vaakvoorkomende patronen korter kunnen gecodeerd worden. Echter bij het gegeven bestandsformaat 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. Echter ook hier zijn deze patronen beperkt en zullen dus nauwelijks profijt leveren.

1.1.4 Huffman

In tegenstelling tot de vorige algoritmes, maakt huffman 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 ('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. Aangezien dit algoritme het beste lijkt voor het gegeven probleem, koos ik dit 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 prefixboom. 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

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 niet compleet in het werkgeheugen zich moeten bevinden. Een positief neveneffect hiervan is dat de compressie verbeterd. De reden hiervan is dat bepaalde bytes meer voorkomen in bepaalde blokken en minder voorkomende in andere blokken. Neem als voorbeeld JPEG-bestanden. Deze eindigen met een heel lange reeks 0 bytes. Deze grote hoeveelheid bytes kunnen zeer efficient gecodeerd worden indien als ze in een apart blok bevinden zodat hun code lengte 1 heeft. (Het is de enige value in het blok, dus de code heeft lengte 1.) Een negatief neveneffect hiervan is dat er exta 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 volledige bestand geëncodeerd werd.

1. Lees het volgende blok in.
2. Overloop het blok om de frequenties van elk karakter vast te leggen.
3. Bouw een prefixboom m.b.v. de frequentielijst uit 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, het eigenlijke Huffman codering algoritme. Om dit algoritme eenvoudig te implementeren wordt gebruik gemaakt van een priority queue. Deze is geïmplementeerd met een binaire boom. Deze priority queue bevat een lijst van prefixbomen. Initieel heeft elke prefixboom juist 1 top, de wortel, de wortel heeft als waarde, de waarde van een byte en als frequentie, de frequentie van de byte in het huidige blok. De boom met laagste frequentie is telkens de wortel van de priority queue. 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 2 elementen in de priority queue zijn: voeg ze samen en plots ze in de priority queue. Een laatste pop zorgt dan voor de uiteindelijke prefixboom.

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 een terminator toevoegen aan de boom, dit maakt het programma echter overbodig ingewikkeld. Het eenvoudigst, en zeker ook niet slecht, is het schrijven van het aantal karakters 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 minimale overhead schrijf ik, in de plaats van telkens 32 bits met de grootte, 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 voorstelt, deze 32 bits zullen maar één keer weggeschreven worden. Dit is ideaal aangezien bij quasi alle blokken er maar 1 bit bijkomstige overhead zal zijn vanwege de lengte.

Stap 5: Om de overhead van de boom te minimaliseren schrijf ik deze recursief uit. Dit gebeurt als volgt: (Opnieuw is de code lichtjes 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 indien nog geen blad bereikt is. Zodra een blad bereikt is, wordt een 1 uitgeschreven gevolgd door de waarde van het blad. De boom wordt 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 bestanden 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. (bijvoorbeeld bij een bestand waarvan de frequenties gevormd zijn door de fibonacci reeks, hierover later meer.) Hiermee wordt ook rekening gehouden in de implementatie (desondanks dat dit probleem zich niet kan voordoen bij een relatief kleine buffersize, 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. Ook het uitschrijven gebeurt via een buffer. De aparte bits worden telkens toegevoegd aan de buffer en zodra de grootte van buffer maximaal is wordt deze uitgeschreven. Uiteindelijk wordt nogmaals 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. Het is vooral belangrijk dat de blokken precies en blok per blok verwerkt worden, er sluipen namelijk eenvoudig kleine foutjes bij het decoderen als er een fout is in de code. Ook hier wordt met blokken gewerkt, het decoderen begint met het eerste blok van de maximale buffergrootte (of kleiner als het bestand niet zo groot is) en lees een nieuwe blok in zodra het huidige blok is gelezen. 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 een ga naar het linkerkind als als de bit 0 is, anders ga naar het rechterkind.
   (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 specifieren. 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 de recursieve

oproep uitgevoerd op kinderen. Als een 1 gelezen wordt, krijgt de huidige top de waarde gespecifieerd 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 betekent neem linkerkind, 1 betekent neem rechterkind), en telkens wanneer men een blad bereikt, specifieren de volgende 8 bits de waarde. Deze wordt dan weggeschreven naar de buffer.