Tinpro01-8 Eindopdracht: compressie

W. Oele

22 april 2021

Inleiding

In deze opdracht ga je m.b.v. Haskell een klein onderzoek uitvoeren naar de effectiviteit van twee bekende compressie algoritmen. Onder *datacompressie* verstaat men het representeren van een gegeven hoeveelheid informatie in een kleiner aantal bits of symbolen. Datacompressie kent veel toepassingen, bijvoorbeeld:

- bestanden inpakken: winzip, rar, arj, gzip, etc.
- audio: mp3, flac, etc.
- foto: jpeg
- video: h264, avi, mpeg, mp4, mkv, etc.

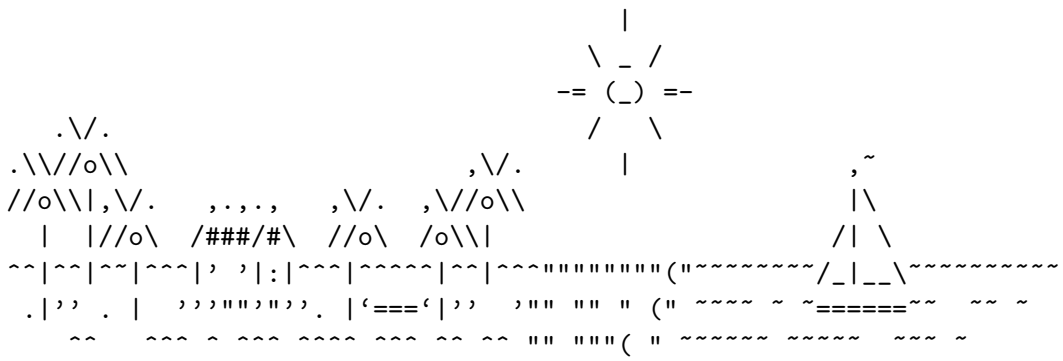
In deze opdracht ga je twee programma' s schrijven die doen aan informatiecompressie. We houden het hierbij eenvoudig en werken alleen met tekstbestanden. In het ene programma implementeer je het *run-length* algoritme en in het andere gebruik je de zogeheten *Huffman codering*.

Het run-length algoritme

Het run-length algoritme is een eenvoudig algoritme. Stel dat we in een tekstbestand het volgende tegenkomen:

```
aaabbcbbccccbbccccccaaaaabbbb
```

Een dergelijke opeenvolging van characters is niet ongewoon. Zo zitten er in onderstaand stukje ASCII art nogal wat zich herhalende characters:



Let erop dat in bovenstaand stukje ASCII art nogal wat spaties zitten en ook die zijn voor een computer "gewoon" een symbool. Andere voorbeelden zijn er natuurlijk ook: zo zitten er in diverse foto' s nogal wat opeenvolgende pixels met dezelfde kleur, etc.

Het run-length algoritme werkt als volgt: Tel het aantal zich herhalende symbolen en noteer dit met een getal, gevolgd door het betreffende symbool. De eerder genoemde string wordt zodoende als volgt gecomprimeerd:

```
aaabbcbbccccbbccccccaaaaabbbb
```

```
3a2bc2b5c2b6c5a4b
```

Het aantal symbolen in de string bedraagt 30 en het aantal symbolen na compressie is slechts 17. Van de oorspronkelijke grootte is nog maar $\frac{17}{30} \times 100 = 57\%$ over.

Het decomprimeren werkt uiteraard in de omgekeerde volgorde: `3a` wordt omgezet in `aaa`, `5c` in `ccccc`, etc.

Opdracht 1a: Run-Length compressie

Schrijf een programma dat een eenvoudig tekstbestand comprimeert. Voorwaarden:

- Noem het programma rlcompress
- Het programma draait stand-alone, dus zonder interpreter.
- Het programma krijgt op de commandline twee parameters mee: De naam van het te comprimeren bestand, gevolgd door de naam van het bestand, waarin de gecomprimeerde gegevens moeten worden opgeslagen.
- Het programma berekent de compressiefactor zoals boven beschreven en print deze op het scherm.
- Test je programma met een eenvoudig ASCII textbestand dat minstens 500 woorden bevat. Let erop dat in het tekstbestand geen cijfers voorkomen (waarom is dat een probleem?).
- Test je programma ook op een stuk ASCII art. Ook hier geldt dat de tekst geen cijfers mag bevatten.

Een uitvoer van het programma ziet er, bijvoorbeeld, als volgt uit:

```
wessel@digitalsnail: ./rlcompress text.txt compressed.txt
length of text.txt: 4881 characters
length of compressed file compressed.txt: 2354 characters
factor: 2354/4881*100=48%
done...
```

Opdracht 1b: Run-Length decompressie

Schrijf een programma dat een eenvoudig tekstbestand decomprimeert. Voorwaarden:

- Noem het programma rldecompress.
- Het programma draait stand-alone, dus zonder interpreter.
- Het programma krijgt op de commandline twee parameters mee: De naam van het te decomprimeren bestand, gevolgd door de naam van het bestand, waarin de gedecomprimeerde gegevens moeten worden opgeslagen.
- Test je decompressie programma met de eerder gecomprimeerde tekst en ASCII art en controleer of de gedecomprimeerde bestanden hetzelfde zijn als hun originelen.

Tips, werkwijze en aanwijzingen

Command line parameters

Het in een programma opvangen van parameters die je op de command line meegeeft is eenvoudig. De library `System.Environment` kan gebruikt worden om command line parameters op te vangen in een lijst. Experimenteer zo nodig met een klein programma om dit te testen.

I/O en core algorithms

In Haskell zorgt het datatype `IO` voor een harde scheiding tussen code die aan IO doet (de "inpure" code) en code die dat niet doet (de "pure" code).

Het is daarom verstandig het inlezen van bestanden en verwerken van command line parameters in eerste instantie te negeren en je te concentreren op de core algorithms: run-length compressie. Een simpele manier om dit te doen is door in je source code een variabele neer te zetten met daarin een niet al te lange string die je gebruikt om er je run-length algoritme op te testen. Je kunt

dan experimenteren en in de interpreter werken zonder dat IO en andere zaken in de weg zitten.

Zodra het algoritme volledig in "pure" code is geprogrammeerd kun je je gaan richten op de communicatie met de buitenwereld.

Bestanden lezen en schrijven

In de library `System.IO` zitten een aantal handige functies, waarmee je bewerkingen op bestanden kunt uitvoeren:

- de inhoud van een textbestand lezen en opvangen in een `String`.
- Een `String` wegschrijven naar een bestand.
- etc.

Huffman compressie

In de vorige opdracht heb je kunnen zien dat het run-length algoritme niet altijd even efficiënt is. In deze opdracht ga je het Huffman algoritme programmeren. Enkele eigenschappen van dit algoritme:

- aanzienlijk efficiënter dan het run-length algoritme.
- heeft geen problemen met getallen.
- behaalt vaak een compressie factor die zeer dicht tegen het theoretische maximum aanzit.
- aanzienlijk moeilijker te programmeren dan het run-length algoritme.

Werking

Om de werking van het Huffman algoritme te illustreren gaan we een kort stukje tekst comprimeren, te weten onderstaande string:

```
aaabbcbbcccddccbbcccdecccaaaaabbbb
```

Huffman compressie stap 1: letters tellen

In de gegeven string tellen we van elke letter hoe vaak deze voorkomt in de tekst. Zodoende krijgen we de onderstaande tabel:

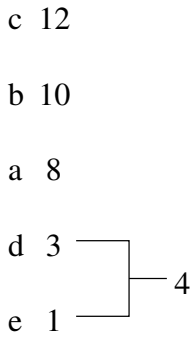
8	10	12	3	1
a	b	c	d	e

We sorteren deze tabel aflopend op het aantal keren dat de letter voorkomt in de tekst:

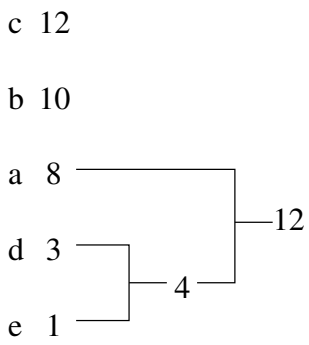
12	10	8	3	1
c	b	a	d	e

Huffman compressie stap 2: de coderingsboom bouwen

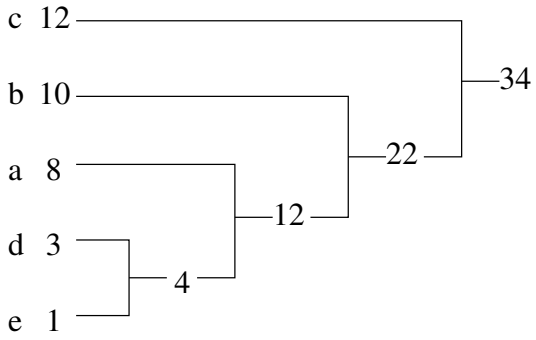
We kiezen uit de lijst met letters de twee letters die het *minst* in de tekst voorkomen en tellen hun getalswaarde bij elkaar op. Zo ontstaat:



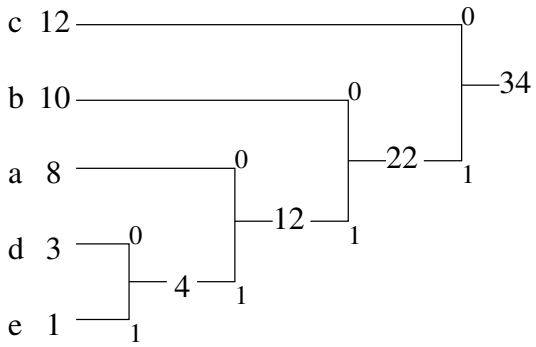
We herhalen deze stap...



...totdat we alle letters gehad hebben en de hele boom is opgebouwd:



Nu lezen we de boom van rechts naar links en bij elke vertakking plaatsen we een 0 en een 1:



Huffman compressie stap 3: de Huffman codering opstellen

We kunnen een Huffman codering opbouwen door de boom van rechts naar links te lezen en elke 0 of 1 te noteren die we onderweg tegenkomen. Zo ontstaat:

0	10	110	1110	1111
c	b	a	d	e

Huffman compressie stap 4: de tekst comprimeren

We nemen de oorspronkelijke tekst erbij en vervangen elke letter die we tegenkomen door het bitpatroon in de tabel. Zo ontstaat:

```
aaabbcbbcccddccbbcccdecccaaaaabbbb

1101101101010010100001110111000101000011101111000110110110110110101010101010
```

Wat zijn we nu opgeschoten? Welnu:

- Het aantal letters in de oorspronkelijke tekst bedraagt 34. Dat is $34 \times 8 = 272$ bits.
- Het aantal bits na Huffman compressie is 72.
- De compressiefactor is derhalve: $\frac{72}{272} \times 100 = 26\%$.

Huffman decompressie

Het decomprimeren van een gecomprimeerde/gecodeerde tekst bestaat uit het in omgekeerde volgorde doorlopen van bovenstaand proces:

- Lees een bit uit het gecomprimeerde bestand
- Neem de juiste afslag in de coderingsboom
- Herhaal bovenstaande stappen, totdat je in de coderingsboom bij een letter uitkomt.

Uit bovenstaande procedure kunnen wij beredeneren dat voor het decomprimeren van een tekst niet alleen de gecomprimeerde tekst nodig is, maar ook de coderingsboom, waarmee de oorspronkelijke tekst gecomprimeerd werd.

Opdracht 2a: Huffman compressie

Schrijf een programma dat een eenvoudig tekstbestand comprimeert m.b.v. Huffman compressie. Voorwaarden:

- Noem het programma huffmancompress
- Het programma draait stand-alone, dus zonder interpreter.
- Het programma krijgt op de commandline drie parameters mee: De naam van het te comprimeren bestand, gevolgd door de naam van het bestand, waarin de gecomprimeerde gegevens moeten worden opgeslagen, gevolgd door de naam van het bestand, waarin de coderingsboom wordt opgeslagen.
- Gebruik voor het opslaan van de coderingsboom de serialisatie techniek zoals uitgelegd in de les.
- Het programma berekent de compressiefactor zoals boven beschreven en print deze op het scherm.

- Test je programma met een eenvoudig ASCII textbestand dat minstens 500 woorden bevat.
- Test je programma ook op een stuk ASCII art.

Een uitvoer van het programma ziet er, bijvoorbeeld, als volgt uit:

```
wessel@digitalsnail: ./huffmancompress text.txt compressed.txt tree.txt
length of text.txt: 34 characters, 272 bits.
length of compressed file compressed.txt: 72 bits.
factor: 72/272*100=26%
file compressed.txt written to disk...
file tree.txt written to disk...
done...
```

Tips, werkwijze en aanwijzingen

Letters tellen

Voor het tellen van de aantallen voorkomens van letters in een string kun je uiteraard je eigen functies schrijven, maar wie even verder denkt ziet meteen dat het werken met een hashmap wellicht handig is. In Haskell bestaat de library `Data.HashMap.Lazy`. Je kunt deze library direct importeren in je programma, maar dan zullen er conflicten ontstaan daar zowel in de `Prelude` als in `Data.HashMap.Lazy` functies als `map`, `foldr` etc. voorkomen. De oplossing bestaat uit het gebruiken van de volgende constructie:

```
import Data.HashMap.Lazy as L
import Prelude as P
```

In je programma kun je nu onderscheid maken tussen de functies die in beide libraries aanwezig zijn:

```
P.map ...
L.map ...
```

Opdracht 2b: Huffman decompressie

Schrijf een programma dat een eenvoudig tekstbestand decomprimeert m.b.v. Huffman compressie. Voorwaarden:

- Noem het programma huffmandecompress

- Het programma draait stand-alone, dus zonder interpreter.
- Het programma krijgt op de commandline drie parameters mee: De naam van het te decomprimeren bestand, gevolgd door de naam van het bestand, waarin de gedecomprimeerde gegevens moeten worden opgeslagen, gevolgd door de naam van het bestand, waaruit de coderingsboom die voor het decomprimeren nodig is, wordt gelezen.
- Gebruik voor het lezen van de coderingsboom de serialisatie techniek zoals uitgelegd in de les.
- Test je programma met de gecomprimeerde bestanden en bijbehorende coderingsbomen uit de vorige opdracht.
- Controleer of de gedecomprimeerde tekst gelijk is aan de originele tekst.

Een uitvoer van het programma ziet er, bijvoorbeeld, als volgt uit:

```
wessel@digitalsnail: ./huffmandecompress compressed.txt decompressed.txt tree.txt
length of decompressed file: 34 characters, 272 bits.
file decompressed.txt written to disk...
done...
```