Huffmankodning

Oscar Melin

2015-01-29

1 Uppgiften

Den har rapporten tar huffmankodning som exempel för att undersöka och analysera olika sätt att representera data. Jag kommer att titta på olika representationer av binära träd i array/tuple-form och dictionaries/hash tables för att kryptera och dekryptera text.

Huffmankodning är en ickeförsörande datakomprimeringsalgoritm där symboler av en bestämd längd[1] översätts och istället representeras utav en sekvens av tecken vars längd beror på symbolens relativa frekvens jämte andra behandlade symboler. För att hitta symbolernas relativa frekvens så analyserar jag hela filen som ska kodas för att på förhand kunna bygga upp en datatabell. Detta kallas **statisk huffmankodning** till skillnad från **adaptiv huffmankodning** vilket inte har använts till den här rapporten.

2 Ansats

Uppgiften är gjord i Erlang och alla exempel i den här rapporten kommer att vara i Erlang.

2.1 Kodning

Kodningen av en text har jag delat upp i fem steg illustrerat av *compress/1*

```
compress(Text) ->
    ChrFreq = lists:keysort(2, chr_freq(Text)),
    Tree = tree(ChrFreq),
    ChrCodes = chr_codes(Tree),
    Dict = dict:from_list(ChrCodes),
    EncodedText = encode(Text, Dict).
```

- *ChrFreq* är en sorterad min-till-max-lista av tuplar av formatet {SYMBOL, FREKVENS}.

[1] I den här rapporten kommer jag endast att behandla 8-bitars ASCII-kod.

- *Tree* är det binara trädet skapat av *tree/1*. Trädet är i formen av en tupel med två element där varje element antingen är en likadan tupel för att representera en nod i trädet eller en symbol som alltid kommer vara ett löv i trädet.

  t.ex. trädet {{a,b},c} är trädet [NOD ,NOD , c, a, b] i vanlig arrayrepresentation.

- *ChrCodes* är en osorterad lista av samma format som *ChrFreq* men med innehållet {SYMBOL, KOD}. Där KOD är den kodade versionen av SYMBOL som har genererats utav *chr_codes/1* som besöker varje löv i trädet och lagrar "0" för varje vänstergren den färdas över och "1" för varje högergren.

- *Dict dict:from_list/2* skapar en dictionairy från en lista av tuplar där nyckel och värde sätts till följande: {NYCKEL, VÄRDE}.

- *EncodedText* går igenom hela textsträngen och översätter varje symbol med hjälp av ordlistan.

2.2 Avkodning

Avkodningsprocessen skiljer sig inte avsevärt ifrån kodningen av texten och görs utav *decompress/2*.

```
decompress(Text, BitString) ->
    ChrFreq = lists:keysort(2, chr_freq(Text)),
    Tree = tree(ChrFreq),
    ChrCodes = chr_codes(Tree),
    Dict = dict:from_list(rev_tuple(ChrCodes)),
    DecodedText = decode(BitString, Dict).
```

Det som skiljer sig här är de två sista raderna.

- *rev_tuple/1* tar kodningstabellen ifrån tidigare och vänder på den, sedan så skapar vi en avkodningsordbok med hjälp utav *dict:from_list/1*.

- *decode/2* tar strängen av ettor och nollor samt avkodningsordboken och konverterar tillbaka strängen till dess ursprungliga form. Den gör detta genom att analysera en symbol ifrån strängen i taget och summera ihop konsekutiva symboler i *Chr* tills den hittar en matchning i ordboken.

  ```
  decode(BitString, Dict, Result) ->
      decode("", BitString, Dict, Result).
  ```

```
decode(A, [], _Dict, Result) ->
    Result;
decode(Chr, [H | Rest], Dict, Result) ->
    case dict:is_key(Chr ++ [H], Dict) of
        true ->
            decode("", Rest, Dict, Result ++ [dict:fetch(Chr ++ [H], Dict)]);
        false ->
            decode(Chr ++ [H], Rest, Dict, Result)
    end.
```

När den hittar en matchning så läggs den till i *Result*, och *decode/4* körs om igen med *Chr*=“”.

3 Utvärdering

I den här sektionen så testar jag programmets olika körningstider beroende på storleken på texten som ska kodas. Texterna tog jag ifrån lorem ipsum-generatorn på hemsidan http://sv.lipsum.com/

bytes	kodning	avkodning	kompression
1000	400ms	42ms	1.86
5000	842ms	663ms	1.85
10000	3780ms	3978ms	1.86

Tabell 1: Körningstider och kompressionsvärde på olika filstorlekar

Jag testade mitt program på tre olika filstorlekar; 1kB, 5kB och 10kB. Tiden för kodning och avkodning som visas är genomsnittstiden för 10 körningar. Det som förvånade mig mest med resultatet från de olika testen är att tiden för avkodning ökade mycket snabbare än tiden för kodningen.

Jag antog att avkodningstiden skulle hållas mycket kortare än kodningstiden eftersom programmet inte behöver bygga ett helt träd och sedan en ordlista därifrån. Vid avkodningen så har vi redan en tabell att översätta tillbaka med.

Om vi avstår ifrån tiden av träd och ordlistebyggandet vid kodningen och bara kollar på själva översättningen så går *encode/2* igenom varje element i textsträngen och gör en lookup för varje. I avkodningen så går *decode/2* igenom varje element i den kodade strängen och gör en lookup per genomsnittslängden för den kodade versionen av symbolerna.

4 Sammanfattning

Allt som allt så tyckte jag att det var en väldigt intressant uppgift att lösa. I början så tyckte jag att det var väldigt svårt att få fungerande kod på

papper så att säga. Mestadels eftersom det här är första gången som jag gör funktionell programmering på riktigt och det tog ett tag för mig att komma in i rekursionstänket utan objekt och loopar.

Ett stort problem jag hade som tog upp en stor del av tiden i början av projektet var att jag fixerade mig för mycket på att skriva Erlang så elegant och konventionsmässigt som möjligt vilket jag sedan ansåg inte genomförbart eftersom mina Erlangkunskaper inte var på den nivån än. Det slutade med att jag övergav mina överkomplicerade "list comprehensions" som jag knappt förstod själv och som inte alltid funkade som de skulle för simplare och mera intuitiva lösningar.

Hade jag haft mera tid så hade jag gjort en bättre analys och utvärdering av programmet och hur det uppför sig vid mera avvikande situationer. Jag hade också tagit fram en generell funktion för att uppskatta körningstiden för kodning och avkodning relaterar sig till olika variabler som alfabetsstorlek och filstorlek.

Sammanfattningsvis så är jag nöjd med mitt arbete och det jag har lärt mig inom funktionell programmering och behandling av de datastrukturer som använts.

5 Referenser

- D.A. Huffman, 'A Method for the Construction of Minimum-Redundancy Codes', Proceedings of the I.R.E., September 1952, pp 1098–1102. Huffman's original article.