Huffman

Programmering II - Elixir

Kenan Dizdarevic

6 mars 2023

Inledning

Datakompression är något vi har försökt bemästra ända sedan datorernas begynnelse. Målet är att koda om data på ett vis som innefattar att färre bitar används. Fördelarna med detta är att man sparar lagringsutrymme samt att man uppnår en snabbare överföringshastighet över en länk.

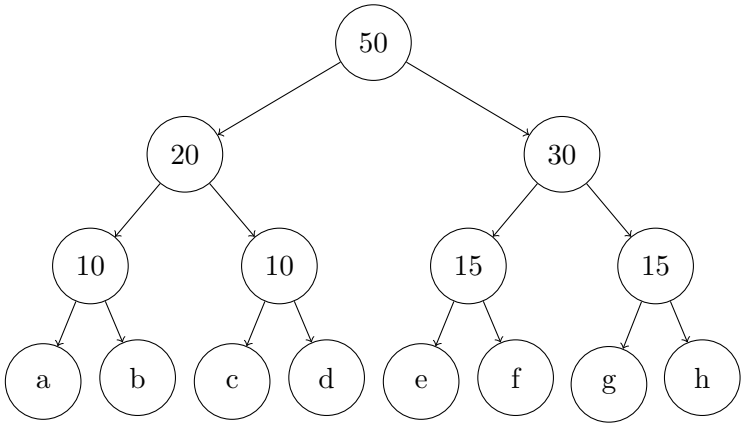
Vid komprimering av video och bild behöver vi inte vara noga med att få med allt. Vår hjärna klarar ändå av att tolka bilden om några pixlar saknas. När vi komprimerar text är det vitalt att vi får med varje bokstav. Annars uppstår det problem och det korrekta budskapet framförs inte.

En algoritm som komprimerar data är *Huffman-kodning*. Den fungerar på så vis att varje tecken omvandlas till binär kod. Detta implementeras i form av en trädstruktur som genererar en unik kod för varje tecken.

Huffman-trädet konstrueras genom att man skapar en lista över alla tecken och dess frekvens i datamängden. Sedan väljer man de två tecken som har lägst frekvens och skapar en nod som har dessa två barn. Den nya nodens frekvens är summan av barnens frekvenser. Noden läggs sedan tillbaka och processen upprepas tills trädet är färdigt.

Med ett färdigställt träd har vi möjligheten att skapa en tabell som visar vilken kod varje tecken har. Denna tabell fungerar på så vis att vi traverserar trädet från roten ända ner till lövnoden. Varje gång man går till höger i trädet lägger man till en etta i koden, och varje gång man går till vänster i trädet lägger man till en nolla.

Ett Huffman-träd presenteras i figur 1. I exempelträdet representeras bokstaven a med den binära koden 000.



Figur 1: Huffman-träd

Implementering

Frekvens

Principen som *David A. Huffman* myntade bygger på att vi har frekvensen för varje tecken i datamängden. Innan vi har möjlighet att konstruera trädet behöver vi analysera datan och skapa en tabell som innehåller varje teckens frekvens, detta implementeras i funktionerna `freq/1` och `freq/2`.

Funktionen `freq/1` anropar endast `freq/2` med datan som skall analyseras och en tom lista. Implementeringen av `freq/2` presenteras nedan:

```
def freq([], freq) do freq end
def freq([char | tail], freq) do
    freq(tail, add(char, freq))
end
```

Frekvenserna kommer att representeras som tupler. Dessa tupler innehåller tecknet i form av ett *ASCII-nummer* och frekvensen i form av ett tal.

När vi når basfallet har vi traverserat över hela datan som representeras som en lista och det finns inga tecken kvar returnerar vi tabellen som vi har skapat. I det andra fallet utför vi ett rekursivt anrop där vi lägger till det nuvarande tecknet i vår tabell.

För att lägga till tecken i vår tabell implementerar vi funktionen `add/2` som tar emot ett tecken och en tabell med frekvenser. Funktionen implementeras på följande vis:

```
def add(char, []) do [{char, 1}] end
def add(char, [{char, n} | tail]) do
    [{char, n + 1} | tail]
end
def add(char, [head | tail]) do
```

```
    [head | add(char, tail)]
end
```

I basfallet har vi endast ett tecken, vi returnerar tabellen med tecknet och frekvensen 1. I de två andra fallen kollar vi om vårt tecken matchar. Om det matchar ökar vi frekvensen med 1 annars utför vi ett rekursivt anrop på resterande del av listan.

Träd

Frekvenserna fungerar korrekt detta innebär att vi har möjligheten att implementera ett Huffman-träd. Vi kommer att representera en lövnod med endast ett tecken. En vanlig nod kommer att representeras som en tupel med två grenar {`höger, vänster`}.

Vi skall implementera principen som vi nämnde i inledningen. En ny nod skall skapas där dess barn är de tecken med lägst frekvens, nodens frekvens är summan av barnens frekvenser. Denna process skall ske rekursivt tills vi erhåller ett fullständigt Huffman-träd.

Trädet konstrueras med hjälp av funktionerna `huffman_tree/1` och `insert/2`. Den första funktionen tar emot en sorterad lista med frekvenser och returnerar ett Huffman-träd. Koden för detta presenteras nedan:

```
def huffman_tree([{root, _}]) do root end
def huffman_tree([{left, lfreq}, {right, rfreq} | tail]) do
    parent_node = {{left, right}, lfreq + rfreq}
    huffman_tree(insert(parent_node, tail))
end
```

I basfallet har vi endast en nod kvar, vi skall endast returnera den noden. Det andra fallet behandlar när vi har två eller fler noder kvar. Vi summerar nodernas frekvenser och skapar en ny nod åt dem. Denna funktion anropas rekursivt tills vi har ett färdigställt träd. Värt att notera är att detta även går att implementera med hjälp av `reduce/2` som återfinns i *Enum-modulen*, men lösningen är inte rekursiv.

Funktionen `insert/2` lägger in en nod i en sorterad lista baserat på frekvensen. De tecken med lägst frekvens återfinns i början på listan. Funktionen tar emot två argument, noden som skall läggas till och listan. Funktionen implementeras på följande vis:

```
def insert({node, freq}, []) do [{node, freq}] end
def insert({node, freq}, [{current_node, c_freq} | tail]) when freq <  c_freq do
    [{node, freq}, {current_node, c_freq} | tail]
end
def insert({node, freq}, [{current_node, current_freq} | tail]) do
    [{current_node, current_freq} | insert({node, freq}, tail)]
end
```

I det första fallet har vi endast en tom lista, vi lägger till noden i listan. I det andra fallet kollar vi om frekvens som skall läggas till i listan är mindre än den första frekvensen i listan, om frekvensen är mindre lägger vi det först i listan. Detta fall har vi för att hela tiden ha listan sorterad. I det sista fallet utför vi ett rekursivt på hela listan förutom den första noden. Detta görs för att kunna placera noden på korrekt plats.

Teckentabell

Med ett färdigställt Huffman-träd har vi möjligheten att extrahera de binära koderna för varje tecken. När vi går till vänster i trädet lägger vi till en nolla, när vi går till höger lägger vi till en etta till den binära koden. Denna princip skall vi följa för varje tecken. Funktionen `binary_encode/2` implementeras på följande vis:

```
def binary_encode({left, right}, path) do
    left_code = binary_encode(left, [0 | path])
    right_code = binary_encode(right, [1 | path])
    left_code ++ right_code
end
def binary_encode(char, path) do
    [{char, Enum.reverse(path)}]
end
```

Vi traverserar över trädet samtidigt som vi utökar vår binära sekvens för varje nod. Slutligen använder vi oss av `reverse/1` för att erhålla korrekt ordning på den binära sekvensen.

Kodning

Vi har en text som består av tecken, representerad i form av en lista. Dessa tecken representerar vi med en sekvens bitar som finns i tabellen. Vi skall alltså implementera funktionen `encode/2` som omvandlar en text till bitar. Argumenten som funktionen tar emot är en lista med tecken och vår tabell som beskriver varje tecken.

I det första fallet returnerar vi endast en tom lista, eftersom fuktionen tar emot en text utan tecken. Det andra fallet presenteras nedan:

```
def encode([char | tail], table) do
    {_, code} = List.keyfind(table, char, 0)
    code ++ encode(tail, table)
end
```

Vi nyttjar funktionen `keyfind/3` för att matcha vårt nuvarande tecken med en binär sekvens. Slutligen gör vi ett rekursivt anrop samtidigt som vi sammanfogar det rekursiva anropen och den binära sekvensen som vi fick från mönstermatchningen.

Avkodning

För att kodningen som vi implementerade skall fylla någon mening måste vi kunna avkoda den binära sekvensen och generera en text. Problemet som uppstår är att vi inte vet hur många bitar som representerar varje tecken. Tabellen som vi implementerade med hjälp av trädet medför att det endast finns ett sätt att avkoda sekvensen.

Vi nyttjar samma tabell som när vi kodade sekvensen. Det vi skall göra är att studera en bit i taget tills vi lyckas matcha med ett tecken i tabellen. Funktionen `decode_char/3` tar emot en sekvens med bitar, en integer och tabellen med tecken som argument. Funktionen presenteras i koden nedan:

```
def decode_char(seq, n, table) do
    {code, rest} = Enum.split(seq, n)
    case List.keyfind(table, code, 1) do
      {char, _} ->
        {char, rest}
      nil ->
        decode_char(seq, n + 1, table)
    end
end
```

Vi delar upp den binära sekvensen i två delar, de första n bitarna och den resterande delen. Sedan kollar vi i tabellen om de n stycken bitarna matchar med ett tecken i vår tabell. Om matchningen lyckas skall vi returnera tupeln `{char, rest}`. Om den binära sekvensen inte matchar mot något tecken skall vi utföra ett rekursivt anrop och öka antalet bitar som vi matchar.

Prestandamätning

Vi skall nu mäta tiden det tar att utföra varje steg i Huffmans algoritm. Vi kommer att använda Karin Boyes dagboksroman *Kallocain* som data.

Vi kommer att konstruera vårt träd och tabell med romanen. Detta innebär att de tecken vi kan nyttja är de som finns i texten. Lyckligtvis ska vi koda och avkoda samma text, detta medför således inget problem.

Träd [ms]	**Tabell [ms]**	**Kodning [ms]**	**Avkodning [ms]**
44	0	66	1300

Tabell 1: Exekveringstid för de olika funktionerna.

Det går relativt snabbt att komprimera data och sedan dekomprimera den. Den största tidsboven vi har är funktionen som avkodar en binär sekvens till text.

Vi kan förbättra funktionen `decode/2` genom att konstruera den som en svansrekursiv funktion. Vi behöver exempelvis en ackumulator som sparar de tecken som vi redan har avkodat. Detta innebär att vi inte behöver nyttja en ny stackram vid varenda rekursivt anrop, vilket i sin tur leder till bättre prestanda.

Slutsats

Slutligen ser vi att Huffmankodning är en otroligt kraftfull och användbar algoritm. Algoritmen underlättar för oss när vi komprimerar datamängder.

Det är viktigt att kunna komprimera data, eftersom det medför flertal fördelar. Huffmankodning är en algoritm som får oss att inse hur viktigt det är att spara på de resurser vi har tillgång till. Vi vill ständigt försöka använda så få bitar som möjligt.