Huffman

Programmering II

Axel Karlsson

VT22 4 mars 2022

Inledning

I denna rapport ska jag redovisa min lösning för uppgift 12, Huffman. Jag har skapat ett program som kan komprimera text med huffmankodning och sedan testat programmets prestanda med olika typer av texter.

Frekvens av chars

Denna funktion skulle hitta hur många gånger varje unik `char` förekom i en textsträng. Jag implementerade detta genom funktionen `freq/1`. Funktionen går igenom varje index i en sträng (alltså en lista av `chars`) och sparar antalet gånger ett tecken förekommer som en tuppel i en ackumulatorlista. Jag har använt funktioner från `List` modulen (även i den utelämnade koden), så det är inte självklart vilken komplexitet funktionen har.

```
def freq(sample) do freq(sample, []) end # Anropa med ackumulator
def freq([], freq) do freq end # Alla chars är kollade
def freq([char | rest], freq) do
  case List.keymember?(freq, char, 0) do
    true ->
      # <Inkrementera indexet i freq>
    false ->
      # <Lägg till {char, 1} i freq>
  end
end
```

Funktionen returnerar en lista av tupplar, se nedan.

```
[
  {100, 1}, # {<asciivärde>, <antal förekomster>}
  ...
  {115, 1}
]
```

Huffmanträd

Nästa del av uppgiften var att skapa ett huffmanträd utifrån resultatet av `freq/1`. Den givna algoritmen för att göra detta var att ta de två lägsta värdena i frekvenslistan, kombinera dem till ett och sedan stoppa in dem igen. Detta upprepas tills det endast finns ett element i listan, som då kommer vara roten till ett träd.
Jag implementerade detta i funktionen `huffman/1`. Funktionen tar en lista av tupplar, som den från `freq/1`, och sorterar den med minsta elementet först.

```
def huffman(freq) do
  freq = List.keysort(freq, 1) # Sortera listan av chars efter frekvens
  hufftree(freq)
end
```

Denna lista bearbetas sedan i funktionen `hufftree/1`, som bygger trädet med den tidigare nämnda algoritmen. Återigen använde jag `List` modulen.

```
def hufftree([root]) do root end # När det endast finns ett element
def hufftree(tree) do
  {el1 = {_, n1}, tree} = List.pop_at(tree, 0) # Första elementet
  {el2 = {_, n2}, tree} = List.pop_at(tree, 0) # Andra elementet
  comb = {{el1, el2}, n1+n2} # Kombinera dem
  tree = insert_sorted(comb, tree) # Stoppa in på rätt plats
  hufftree(tree)
end
```

Kodningstabeller

Det sista steget som krävdes för att koda text var en funktion som skapar en kodningstabell utifrån ett huffmanträd. Denna tabell innehåller information om hur man ska navigera trädet för att nå en viss `char`.
Funktionen implementerades i `encode_table/1`. Den använder två ackumulatorlistor, en för tabellen och en för vägen. Funktionen går rekursivt igenom hela trädet och varje gång ett löv, alltså en `char`, nås sparas vägen till det i tabellen.

```
def encode_table(tree) do encode_table(tree, [], []) end
# Matchar mot en gren
def encode_table({{el1, el2}, _n}, table, path) do
  table = encode_table(el1, table, path ++ [1])
  table = encode_table(el2, table, path ++ [0])
  table
```

```
  end
  # Matchar mot ett löv
  def encode_table({el, _n}, table, path) do table ++ [{el, path}] end
```

Detta resulterar i en datastruktur som den nedan.

```
[
  {122, [1, 1, 1]},
  ...
  {101, [0, 0, 0]}
]
```

Kodning och avkodning

De tidigare nämnda funktionerna kombinerades i funktionerna `encode/2` och `decode/2`. Den första funktionen går igenom varje `char` i den givna texten, kollar upp det motsvarande värdet i kodningstabellen och sparar det i en ackumulatorlista som sedan returneras.

```
  def encode([], _table) do [] end
  def encode([char | rest], table) do
    {_, path}  = List.keyfind!(table, char, 0)
    path ++ encode(rest, table)
  end
```

Funktionen för avkodning var i princip given i uppgiften så jag har valt att inte inkludera koden.

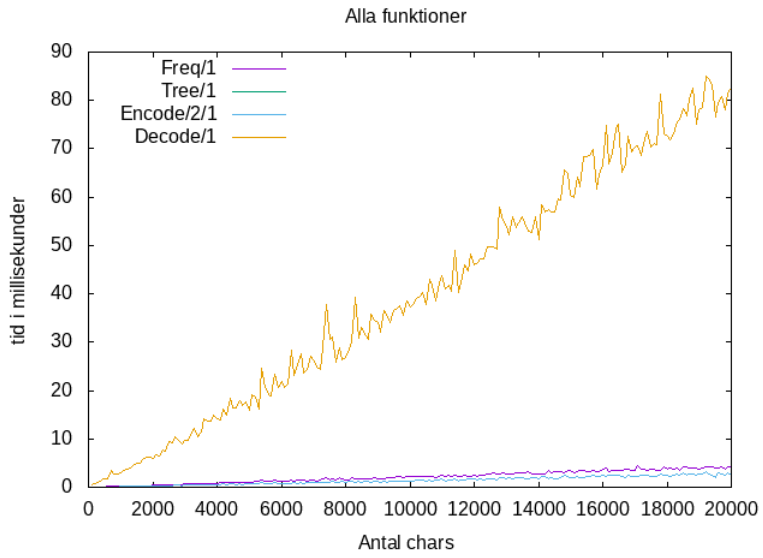
Benchmarks

Jag testade funktionerna `freq/1`, `huffman/1`, `encode/2` och `decode/2` för olika textlängder och med vissa tecken filtrerade. Funktionen `bench/1` väljer `i * size` tecken ur en given text. Denna text skickas sedan till `bench_huff/1` som mäter exekveringstiden på de fyra tidigare nämnda funktionerna med texten. Denna mätning görs 100 gånger för varje `i`, och genomsnittet av resultaten returneras.
Texten jag använde var en engelsk version av Mary Shelleys Frankenstein på cirka fyrahundratusen tecken. Jag gjorde även mätningar där jag filtrerade ut alla tecken utom de 10, 8, 5 och 3 vanligaste tecknen.

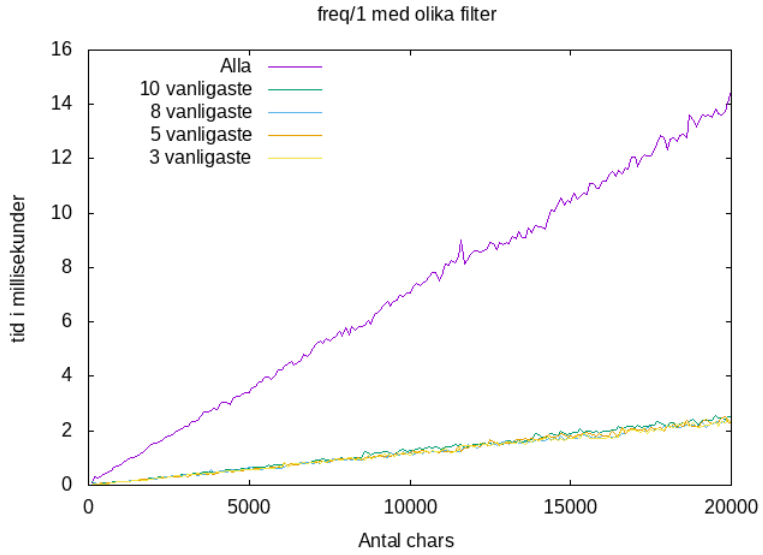
```
  def bench(file) do
  text = read(file, :all)
  ... # Här filtreras texten
    for i <- 1..200 do
```

```
      # <Välj i*100 tecken på en slumpmässig plats i texten>
      {t_fr, t_tr, t_en, t_de} = loop(fn() -> bench_huff(text) end)
      ...
    end
    :ok
  end
```

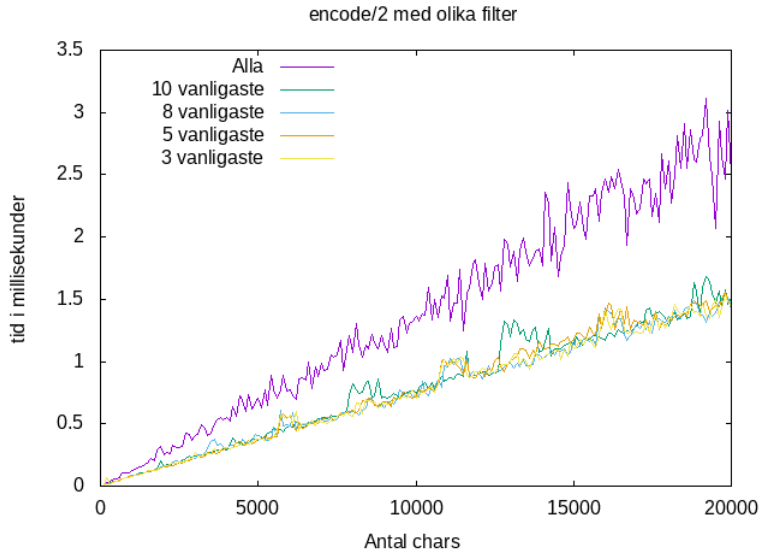
Resultat



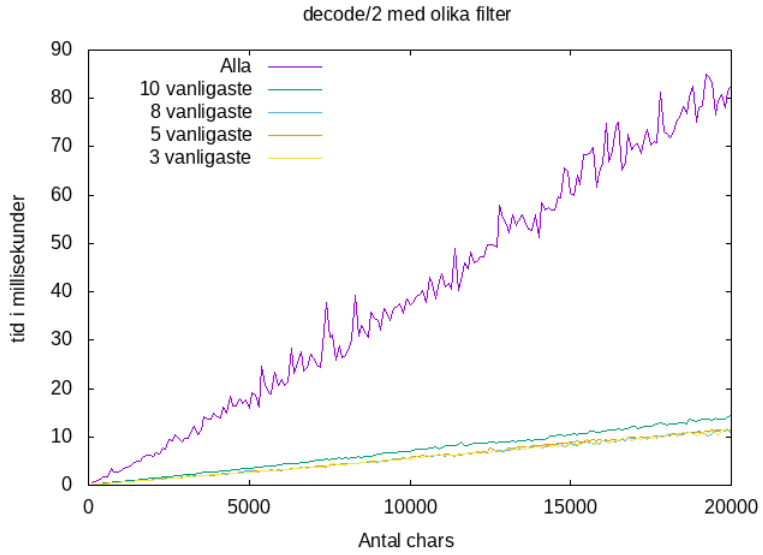
Figur 1: Benchmark av alla funktioner



Figur 2: Benchmark av freq/1. De olika graferna är olika filtreringar av texten.



Figur 3: Benchmark av encode/2. De olika graferna är olika filtreringar av texten.



Figur 4: Benchmark av decode/2. De olika graferna är olika filtreringar av texten.

Analys

I figurer 2, 4 och 3 syns ett starkt samband mellan antal unika element i texten och exekveringstid. Alla tre funktioner som mäts i graferna innehåller ett anrop till `List.keyfind/3` i en lista vars längd beror på antal unika element, vilket jag skulle gissa är anledningen till dessa tre liknande resultat. I figur 1 ser vi att `decode/2` är den långsammaste funktionen med stor marginal. Jag tror att det beror på det tidigare nämnda anropet till `List.keyfind/3`. I `decode/2` sker ett sådant anrop för varje bit i den huffmankodade texten, medan det i `freq/1` och `encode/2` sker ett anrop för varje `char` i texten. Då varje `char` kodas till åtminstonne en bit blir det totalt många fler anrop.