

COMPUTER SCIENCE 123

Foundations of Computer Science

24. Huffman codes

Summary: This lecture describes the construction of a Haskell program that uses Huffman codes to compress text.

You should also have: Tutorial sheet 12
Solutions to Tutorial sheet 11

Reference: Thompson Chapter 15

Data compression

- Billions of electronic messages are sent around the world every day (every hour?)
 - the number is growing rapidly as use of the WWW (the “World-Wide Wait”) expands
- One of the principal factors affecting the delivery times of these messages is their size
- **Data compression** aims to represent a given message using as few bits as possible
- The ISO standard code for characters uses 8 bits for each of the 256 characters ($2^8 = 256$)
 - so a message with n characters requires $8n$ bits
 - this is known as a **fixed-length code**
 - the code for every character has the same length
 - fixed-length codes are very simple, and it is easy to encode and decode messages
- We can get better compression for messages in a given language by using a **variable-length code**
 - e.g. in English text, 'e' occurs more often than 'x', so give 'e' a short code and 'x' a long code
 - e.g. Morse code
- We can get even better compression for a given message by using a code that is based on the structure of the message itself
 - characters that appear often in the message are given short codes
 - characters that appear rarely in the message are given long codes
- To be precise, if each character c_i appears f_i times and is given a code of length l_i , we want to minimise the sum

$$\sum_{i=0}^{255} f_i l_i$$

Huffman trees

- One method for constructing an optimal code for a given message is called **Huffman coding**
 - invented by David Huffman
- The code to be used for a message is represented as a tree
 - e.g. for the message `battat` from the bottom of Page 286 in Thomson

- Each character that appears in the message is stored at one of the leaves of the tree
- The encoding for each character is given by the path from the root of the tree to the appropriate leaf
 - so for the above tree: L = t, RL = a, RR = b
- Note that:
 - frequent character → short path → short code
 - rare character → long path → long code

Decoding using Huffman trees

- Decoding a message involves tracing the paths through the tree
 - use the encoded message to trace a path to the appropriate leaf, then start again from the root
- e.g. RRLLLLRLL

⇒ bRLLLLRLL

⇒ b aLLRLL

⇒ b atLRLL

⇒ b attRLL

⇒ b att aL

⇒ b att at

Datatypes

- We assume that a **message** is just a string

```
type Message = String
```

- The **encoding** of a message is a sequence of **bits**

```
data Bit      = L | R
              deriving (Show, Read)
```

```
type Encoding = [Bit]
```

- The **frequency** of a character is the number of times that it occurs in the message

```
type Frequency = Int
```

- The **tree** holds each character in the message
 - the path from the root of the tree to the `Leaf` holding the character gives its encoding
 - the `Frequency` attached to each `Tree` is the total of the frequencies of the characters in the `Tree`

```
data Tree = Leaf Frequency Char
          | Node Frequency Tree Tree
          deriving (Show, Read)
```

- The **table** holds the encodings for the characters explicitly
 - exactly as in an `Environment` (Lecture 23)

```
type Table = [(Char, Encoding)]
```

Decoding a message

- The complete encoding for a message contains both the tree and the sequence of bits

```
decode :: String -> Message
-- pre: s has exactly two lines, with a
-- pre: Huffman tree on the first line
-- pre: and the encoded message
-- pre: on the second line
-- decode s returns the message
-- encoded in s
decode s = decodeMessage (read t)
                        (getEncoding s')
  where [t, s'] = lines s
```

```
getEncoding :: String -> Encoding
-- getEncoding s returns the encoding in s
getEncoding s = [read [c] | c <- s]
```

- To decode a message, we repeatedly traverse the tree, as described above

```
decodeMessage :: Tree->Encoding->Message
-- pre: t is a Node
-- decodeMessage t m returns the message
-- encoded in m, using t
decodeMessage t = decodeByt t
  where
    decodeByt :: Tree->Encoding->Message
    -- decodeByt x h returns the message
    -- encoded in h, using x for the
    -- initial bit and t for the remainder
    decodeByt (Leaf f c)      rest
      = c : decodeByt t rest
    decodeByt (Node f l r) (L : rest)
      = decodeByt l rest
    decodeByt (Node f l r) (R : rest)
      = decodeByt r rest
    decodeByt (Node f l r) []
      = []
```

Encoding a message—analysing the message

- A message is a list of characters
- To determine the optimal encoding, we need to know the frequency of each character
 - we sort the list so that similar elements are consecutive, then
 - count the occurrences of each element, then
 - sort the resulting list by increasing frequency
 - e.g. "battat"
 - ⇒ "aabttt"
 - ⇒ [(2, 'a'), (1, 'b'), (3, 't')]
 - ⇒ [(1, 'b'), (2, 'a'), (3, 't')]

```
frequencies :: Ord a =>
              [a] -> [(Frequency, a)]
-- frequencies s returns the characters
-- of s, each with its frequency,
-- sorted by increasing frequency
frequencies = sort . collate . sort
```

- note that these two uses of sort work on lists of different types!

```
collate :: Ord a =>
          [a] -> [(Frequency, a)]
-- pre: s is sorted
-- collate s returns the characters
-- of s, each with its frequency
collate [] = []
collate (x : xs) = (length ys + 1, x) :
                  collate zs
                  where (ys, zs) = break (/= x) xs
```

- note that, e.g., `break (/= 'm') "mmmoopqxx"`
= ("mmm", "oopqxx")

Encoding a message—building the tree

- To build the tree from the analysed message:
 - we make each character into a Leaf, then
 - repeatedly combine the two “least frequent” trees into one (`iterate`), until
 - there is only one tree left on the list (`dropWhile`)

```
makeTree :: Message -> Tree
-- makeTree s returns a tree describing
-- the encoding for s
makeTree s = makeCodes [Leaf f c
                        | (f, c) <- frequencies s]
```

```
makeCodes :: [Tree] -> Tree
-- pre: not (null ts)
-- pre: ts is sorted by increasing freq.
-- makeCodes ts returns a single tree
-- holding all of the characters from ts
makeCodes = head . head .
            dropWhile ((> 1) . length) .
            iterate combine
```

```
combine :: [Tree] -> [Tree]
-- pre: length xs >= 2
-- pre: xs is sorted by increasing freq.
-- combine xs returns xs with its
-- first two elements combined,
-- sorted by increasing frequency
combine (t : t' : ts)
    = insertTree
      (Node (freq t + freq t') t t')
      ts
```

```
insertTree :: Tree -> [Tree] -> [Tree]
-- pre: ts is sorted by increasing freq.
-- insertTree t ts returns t : ts
-- sorted by increasing frequency
```

```
freq :: Tree -> Frequency
-- freq t returns the frequency from t
```

Encoding a message—turning the tree into the table

- The tree holds the encoding for each character implicitly in its structure
 - the encoding for a character is given by the path from the root of the tree to the appropriate `Leaf`
- The table holds the encodings explicitly
- For a given `Node`, every character in the left sub-tree has an encoding that starts with an `L`
 - similarly for the right sub-tree

```
makeTable :: Tree -> Table
-- makeTable t returns a table with the
-- path for each character on t
makeTable (Leaf f c)
    = [(c, [])]
makeTable (Node f l r)
    = map (add L) (makeTable l) ++
      map (add R) (makeTable r)
    where add d (c, p) = (c, d : p)
```

Encoding a message

- The complete representation of the message is its encoding plus the tree
 - we won't explore optimising the representation of the tree

```
encode :: Message -> String
-- encode s returns the encoding
-- and tree for s
encode s
  = unlines
    [show t,
     concat (map show
                (encodeMessage (makeTable t) s))]
  where t = makeTree s
```

- To generate the encoding for the message, we just look-up the encoding for each of the characters and concatenate them

```
encodeMessage :: Table->Message->Encoding
-- pre: every character on s
-- pre: has an entry in tbl
-- encodeMessage tbl s returns the
-- encoding for s, using tbl
encodeMessage tbl s =
  concat [fromJust (lookup c tbl) | c <- s]
```

- fromJust is required because lookup returns a Maybe, allowing for the possibility that a character isn't in the table