EE1-08/EE2-18 Soft. Eng. I – Algorithms & Data Structures

Coursework Assignment 2 – Spring 2019

Sahbi Ben Ismail (`s.ben-ismail@imperial.ac.uk`)

Text compression using Huffman's coding algorithm

This assignment deals with an important text processing task: **text compression**.

Huffman coding (1952) is a statistical coding algorithm used for data compression. It can be used with various data types such as texts, images (JPEG), audios (MP3), and videos (DivX).

In this assignment we focus on *text* compression (encoding) and uncompression (decoding) using the Huffman coding (see Figure 1 below). The main objective is to write two algorithms:

- one that *encodes* an input text message M (string of characters) into an compressed message M' (string of bits),
- and one that *decodes* the compressed message M' (string of bits) into an output message M" (string of characters).

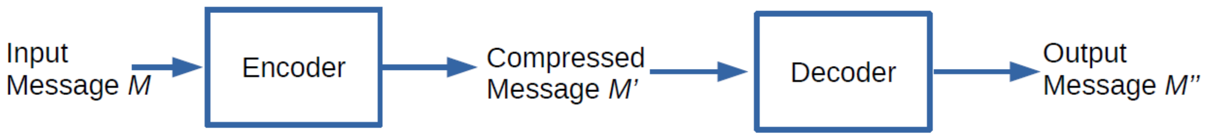

Figure 1: Problematic of encoding/decoding.

Huffman coding is a *lossless* compression method: the input message M and the output message M" are identical.

Such text compression/uncompression can be useful in telecommunications for example, in order to minimise the time needed to transmit a text message over a communication channel.

From ASCII coding to Huffman coding

In the ASCII (the *American Standard Code for Information Interchange*) encoding scheme, a character is encoded in a *fixed-length* binary string (8 bits[1]).

A Huffman code, on the other hand, uses a *variable-length* encoding optimised for the string M. The optimisation is based on the use of character frequencies, where we have, for each character c, a count $f(c)$ of the number of times c appears in the string M (occurrences frequency).

[1]Originally, ASCII used 7 bits.

The Huffman code saves space over a fixed-length encoding by using *short bitstrings* to encode *high-frequency characters* and *long bitstrings* to encode *low-frequency characters*.

Huffman coding, an illustration example

Let's consider the input string M = "go go gophers".

Table 1 shows the *frequencies table* for the input string M.

Character	'␣'	'e'	'g'	'h'	'o'	'p'	'r'	's'
Frequency	2	1	3	1	3	1	1	1

Table 1: Frequency table

Figure 2 and Table 2 show an example of Huffman code with the *Huffman tree* T and its associated *Huffman encoding table* for the input string M.

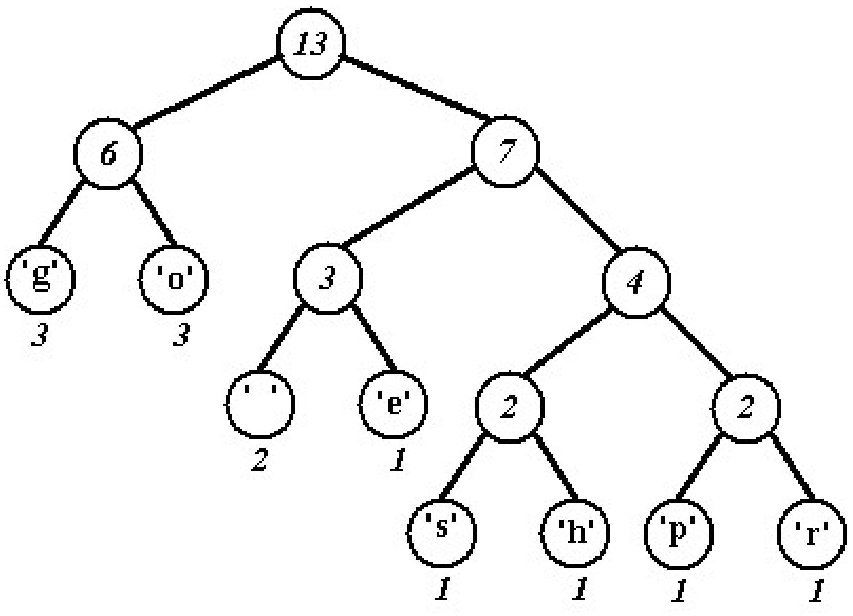

Figure 2: Huffman tree T

(Further details about the construction of the Huffman tree T will be given below.)

In order to produce an optimal variable-length code for the input string M, Huffman coding is based on the construction of a binary tree T that represents the code. Each node in T, except the root, represents a bit in a code word, with each left child representing a '0' and each right child representing a '1'.

Each leaf node v is associated with a specific character, and has a frequency $f(v)$, which is simply the frequency in M of the character associated with v. In addition, we give each non-leaf node v in T a frequency $f(v)$, that is the sum of the frequencies of its two sub-trees.

Character	'␣'	'e'	'g'	'h'	'o'	'p'	'r'	's'
Huffman encoding	100	101	00	1101	01	1110	1111	1100

Table 2: Huffman encoding table (constructed from the Huffman tree T)

How to encode?

The code for a character c is obtained by tracing the path from the root of T to the leaf node where c is stored, and associating a left child with '0' and a right child with '1'.

For example, the code for 'g' is 00 (left→left), and the code for 'h' is 1101 (right→right→left→right).

Finally, the compressed string M' is obtained by concatenating all the consecutive code words for all the characters in M.

For example, using the Huffman encoding table in Table 2, the 13-characters (104-bits) input string M = "go go gophers" will be encoded as the following 37-bits compressed string:

M' = 0001100000110000011110110110111111100.

(M' can also be first seen as 00.01.100.00.01.100.00.01.1110.1101.101.1111.1100 where the dots separate the concatenated code words of the characters 'g', 'o', '␣','g', 'o', '␣', 'g', 'o', 'p', 'h', 'e', 'r', 's'. These separators are useful for the encoding comprehension, but are actually not useful for the decoding. See the "How to decode?" sub-section below.)

Such a compression has a ratio r = 37/104 = 35.5769231 %: the compressed message M' requires 35% of the space required by the input message M.

The *compression ratio* depends on a the frequency of characters in M. Ratios around 50% are common for English messages.

How to decode?

The Huffman encoding table constructed from the Huffman tree T has an important property called the *prefix property*: no code word in the encoding table is a prefix of another code word.

This property simplifies the decoding of M' in order to get back to M: by following the root-to-leaf paths in the Huffman tree T.

In the previous example, the compressed string

M' = 0001100000110000011110110110111111100

is decoded left→left (i.e. the character 'g'), then left→right (i.e. the character 'o'), then right→left→left (i.e. the character '␣') etc..., which leads finally to the output string

M" = "go go gophers" = M.

Implementation of the Huffman coding/decoding in `C++`

Consider the following definition for an appropriate data structure in `C++`:

```
struct hufftreenode {
  char character;
  // this will contain:
  // the characters in leaf nodes
  // nothing in non-leaf nodes

  int frequency;
  // this will contain:
  // the characters frequencies in leaf nodes
  // the cumulated frequencies in non-leaf nodes

  hufftreenode* left;
  // this will conventionally represent the 0 branch

  hufftreenode* right;
  // this will conventionally represent the 1 branch

  // (as usual, leaf nodes will have both left and right pointing to NULL↩
     )
};

typedef hufftreenode* hufftreeptr;
```

Write an implementation for the two following functions:

```
std::string huffencode(const std::string& message,
            hufftreeptr& hufftree,
            std::map<char, std::string>& hufftable);
// encodes an input message (a string of characters) into an encoded ↩
   message (string of bits) using the Huffman coding
// builds the Huffman tree and its associated encoding table

std::string huffdecode(const std::string& encodedmessage,
            const hufftreeptr& hufftree);
// decodes an encoded message (a string of bits) into the original ↩
   message (a string of characters) using the Huffman tree built during ↩
   the encoding
```

Write also an implementation for the following utility function:

```
bool valid_hufftree(const hufftreeptr hufftree);
 // returns true if the input hufftree is a valid Huffman tree
 // i.e. all the terminal nodes (leaves) have characters,
 // all the non-leaf nodes have two sub-trees each,
 // and the occurrence frequency of a non-leaf node equals
 // the sum of the occurrence frequencies of its two sub-trees.
```

So, a full encoding/decoding program might be:

```
int main() {

  // an example of basic encoding/decoding
  std::string message, encodedmessage, decodedmessage;
  hufftreeptr hufftree;
  std::map<char, int> freqtable;
  std::map<char, std::string> hufftable;

  message = "go go gophers";

  // 1) encoding (compression)
  encodedmessage = huffencode(message, freqtable, hufftree, hufftable);
  // freqtable should be as in Table 1.
  // hufftree might be as in Figure 2.

  // hufftable should correspond to hufftree, and might be as in Table 2 ↩
      <' ',”100”>, <'g',”00”>, <'o',”01”>, <'p',”1110”>, <'h',”1101”>, <'e↩
      ',”101”>, <'r',”1111”>, <'s',”1100”>.

  // encodedmessage might then be the 37-bits string ↩
      ”0001100000110000011110110110111111100”

  if(valid_hufftree(hufftree))  {
    std::cout << "valid Huffman tree." << std::endl;
  }
  else {
    std::cout << "not valid Huffman tree." << std::endl;
  }

  // 2) decoding (uncompression)
  decodedmessage = huffdecode(encodedmessage, hufftree);
  // should be ”go go gophers”

  if(decodedmessage == message) {
    std::cout << "decoding OK." << std::endl;
  }
  else {
    std::cout << "decoding not OK." << std::endl;
  }

  return 0;
}
```

Explanation of function `huffencode`

First, function **`huffencode`** examines the input text **`message`**, computes the frequency of each character in it, and builds the frequency table **`freqtable`** (input/output parameter).

Then, it builds the Huffman tree **`hufftree`** (input/output parameter).

Then, it builds the associated encoding table **`hufftable`** (input/output parameter).

Finally, it encodes the input **`message`** (a string of characters) into a compressed message (string of bits) and returns that compressed message.

The tables **`freqtable`** and **`hufftable`** can be built using the **`std::map`** data structure (see the Appendix for more details).

The Huffman tree **`hufftree`** is built as follows:

1. **For** each character c in the string **`message`** **do**:
   (a) Create a single-node binary tree T storing the character c and it frequency $f(c)$.
   (b) Insert T in a collection of trees[2], called a *forest*, $\mathcal{F}$.
2. **Repeat** this step until there is only one tree left in the forest $\mathcal{F}$:
   (a) Choose two trees T1 and T2 in $\mathcal{F}$ with the smallest frequencies f_1 and f_2.
   (b) Merge T1 and T2 into one single tree with T1 as a left sub-tree, T2 as a right sub-tree, and $f_1 + f_2$ as a frequency.
3. The single tree T left in $\mathcal{F}$ after the previous step is a Huffman encoding tree.

The following Figures 3 to 10 illustrate the evolution of the forest $\mathcal{F}$ using the previous example (M = "go go gophers").

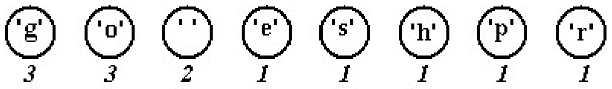

Figure 3: Forest $\mathcal{F}$ evolution - step 1. T1 and T2 can be chosen among 5 trees with the minimal frequency 1. (It doesn't matter which two are picked.)

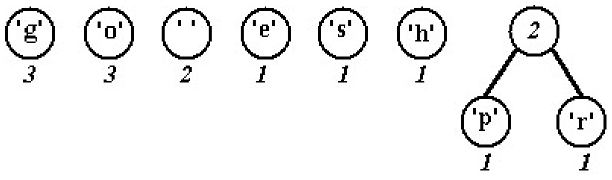

Figure 4: Forest $\mathcal{F}$ evolution - step 2. T1 and T2 can be chosen among 3 trees with the minimal frequency 1. (It doesn't matter which two are picked.)

[2]For example, $\mathcal{F}$ can be implemented as an `std::list` or an `std::vector` of trees.

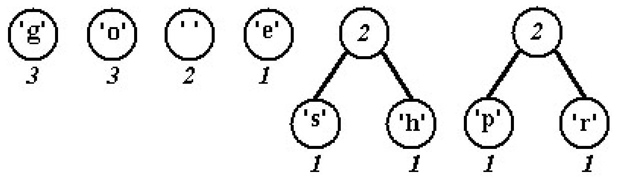

Figure 5: Forest $\mathcal{F}$ evolution - step 3. The minimal frequencies are 1 (1 tree) and 2 (3 trees).

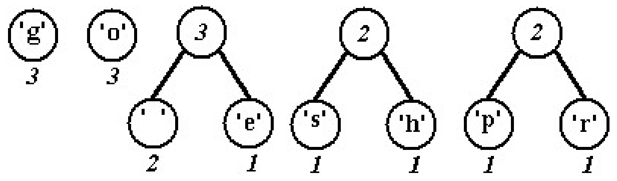

Figure 6: Forest $\mathcal{F}$ evolution - step 4. T1 and T2 are the two trees with the minimal frequency 2.

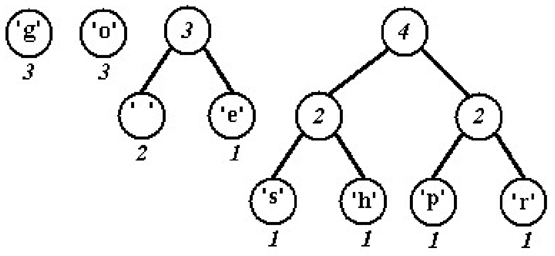

Figure 7: Forest $\mathcal{F}$ evolution - step 5. T1 and T2 can be chosen among 3 trees with the minimal frequency 3. (It doesn't matter which two are picked.)

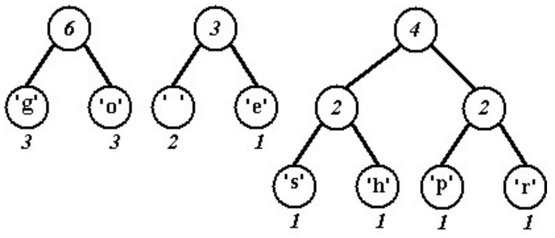

Figure 8: Forest $\mathcal{F}$ evolution - step 6. T1 and T2 are the two trees with the minimal frequencies 3 and 4.

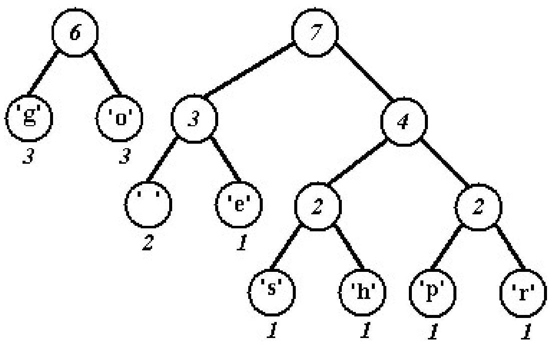

Figure 9: Forest $\mathcal{F}$ evolution - step 7. T1 and T2 are the (last) two trees with the minimal frequencies 6 and 7.

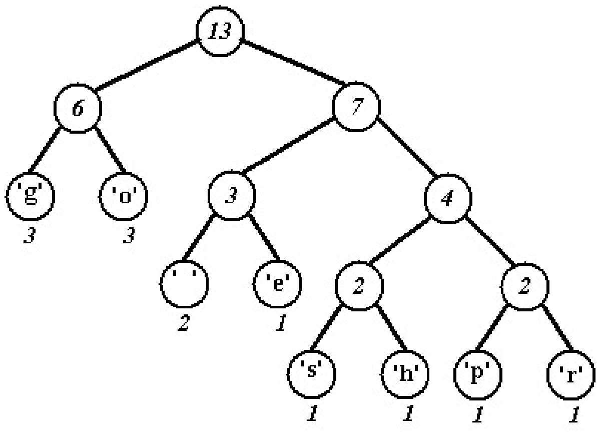

Figure 10: Forest $\mathcal{F}$ evolution - step 8. Final Huffman tree T.

Steps 1, 2, 3, and 5 show that the Huffman tree T is not unique, since T1 and T2 could be chosen among many (more than 3) trees.

There are other trees that use 37 bits: for example, by simply swapping any sibling nodes in the final Huffman tree T of Figure 10, we get a different encoding that uses the same number of bits.

But all these Huffman trees are guaranteed to be *optimal*: there are no other trees with the same characters that use fewer bits to encode the input message M = "go go gophers".

Explanation of function `huffdecode`

Function `huffdecode` decodes an encoded message `encodedmessage` using the Huffman tree `hufftree` (used to encode it), and returns the output message.

The output message can be be found by following the root-to-leaf paths in the Huffman tree `hufftree` (see the "How to decode?" sub-section above).

Explanation of the utility function `valid_hufftree`

The utility function `valid_hufftree` is a simple routine that checks if a Huffman tree `hufftree` is valid or not.

It might be used to check the tree built by function `huffencode` (and used by function `huffdecode`).

Appendix: Map data structure

A *map* is an associative data structure consisting in a collection of *key-value* elements, called entries. A map requires that each key be unique, so the association of keys to values defines a mapping.

The `C++` Standard Template Library (STL) provides an implementation of a map, simply called `map`[3] (`#include <map>`).

The following `C++` code illustrates a basic example of declaration and manipulation of two `std::maps` you might use in your Huffman encoding function `huffencode` in order to store the frequency table and the encoding table.

```
#include <iostream>
#include <string>
#include <map>

int main() {
  std::map<char, int> freqtable; //occurrence frequencies

  freqtable[' '] = 2; // space character
  freqtable['g'] = 3;
  freqtable['o'] = 3;
  freqtable['p'] = 1;
  freqtable['h'] = 1;
  freqtable['e'] = 1;
  freqtable['r'] = 1;
  freqtable['s'] = 1;

  std::cout << "freqtable['g'] = " << freqtable['g'] << std::endl;
  std::cout << "freqtable now contains " << freqtable.size() << " elements." <<↩
       std::endl;

  std::map<char, std::string> hufftable; //Huffman table
  hufftable[' '] = "100"; // space character
  hufftable['g'] = "00";
  hufftable['o'] = "01";
  hufftable['p'] = "1110";
  hufftable['h'] = "1101";
  hufftable['e'] = "101";
  hufftable['r'] = "1111";
  hufftable['s'] = "1100";

  // traversing the map (using an iterator)
  // and printing all the entries
  std::map<char, std::string>::iterator it;

  std::cout << "\nhufftable:"<< std::endl;
  for(it = hufftable.begin(); it != hufftable.end(); it++) {
  // iterator it will point to the map entries one by one
    std::cout << "(" << it->first << ", " << it->second << ")" << std::endl;
  }

  return 0;
}
```

[3] `C++` STL `map` documentation: http://www.cplusplus.com/reference/map/map/

It prints the following output:

```
freqtable['g'] = 3
freqtable now contains 8 elements.

hufftable:
(  , 100)
(e, 101)
(g, 00)
(h, 1101)
(o, 01)
(p, 1110)
(r, 1111)
(s, 1100)
```