AMAR BESSEDIK
Project2: Huffman Codes Using Common Lisp Programming Language
CSC540
03/06/2017

Project Description

In this project, the Huffman coding algorithm, which is a divide and conquer approach, is used to encode a message into binary code and decode the binary back to the original message. In order to achieve this, a Huffman Tree is constructed using the frequency list of the content of the message. Generating the frequency list requires counting the occurrence of each symbol (or its weight) in the original message and pair each count with its corresponding symbol. After this first phase, every pair is put in a list and the entire list in a list itself; then sort the whole thing in increasing order of weight. This way, we now have a list of Huffman sub-trees (sub-htrees). The objective is to compress data using variable length coding rather than fixed length coding all in a lossless manner.

Building the Huffman Tree:

In order to construct the entire Huffman tree, we proceed to merging sub-trees. This is done by starting from the first two lowest frequencies by putting the two symbols in question together into a list along with the sum of their frequencies into another list then add the two sub-trees to form the first tree merge. At the end of this operation, we end up with a reduced list of sub-trees by one. The list of sub-trees is sorted again and the process is repeated until we get one node only. This node is the root of the Huffman tree containing all symbols and the sum of all their respective weights. The Common Lisp functions involved in this phase are the following in pseudo-code: (Each one them uses its immediately below result)

- *(make-frequency-list out-of-message) , (sort (copy-list frequency-list) #'< :key #'second) ,*
  (initialize sorted-list), (merge-htrees initialized-list), (trim-htrees resulting-from-merge-list),
  *(sort (copy-list after-merge-list) **#'htree-less**) >>> (repeat the process recursively until the list becomes of length 1) >>> (generate huffman-tree out-of-message)*

Encoding The message:

In order to encode the message in question, we take the symbols one by one and test whether they are present in the root of the tree. If yes, then generate ('CONS' function in Common lisp) a '0' or '1' depending on whether the symbol belongs to the left sub-tree or to the right one respectively. The Common lisp functions involved in this operation are the following in pseudo-code:

- *(encode-symbol symbol using-huffman-tree)*
  > (next-encoding-branch symbol using-huffman-tree)
  *>> (if current-branch is a leaf return 0/1) **otherwise***
  *>> (cons '0 **if** (next-encoding-branch symbol using-left-subhtree) **is True***
  *>> (cons '1 **if** (next-encoding-branch symbol using-right-subhtree) **is True***
  >> (error " ERROR" otherwise)
- *(encode all-symbols hffman-tree) using 'encode-symbol' function.*

Decoding The Binary:

Decoding the binary requires taking each bit in the binary list and go towards the left branch of the tree or towards the right branch depending on whether a bit is '0' or '1' respectively. Before we move to the branches, we need to make sure that we are indeed using binary bits, otherwise it's an error. After that, each time we advance, we need to test if the current node is a leaf or not. If yes, then return the symbol; otherwise continue advancing until a leaf is reached. Once a symbol is returned, we restart over from root until there no bits left. The Common Lisp functions involved are the following in pseudo-code:

- (decoder binary current-branch huffman-tree)
  *> (next-decoding-branch bit current-branch) – (if current-branch is a leaf return symbol) **otherwise***
  *>> (continue advancing **if** (next-decoding-branch next-bit using-left-subhtree) **is True***
  *>> (continue advancing **if** (next-decoding-branch next-bit using-right-subhtree) **is True***
  >> (error " ERROR" otherwise)
  (decode binary hffman-tree) using 'decoder' function.