Reconstruction of Huffman Tree From Serialized Form

Huseyin Can Ercan

6 November, 2017

1 Introduction

Huffman encoding is a variable length encoding method for lossless data compression. Huffman method uses symbol frequencies or symbol ratios for tree generation. Main purpose of the Huffman tree is to place high frequency symbols to upper level of the tree. With this approach we can generate shorter prefix codes for common symbols. Method places the low frequency symbols at the lower level of the tree, these codes are larger than upper layer codes. After the tree generation and encoding of the data we face another problem, storage of tree. Simply we can store the frequency values of the symbols but in some cases this method has some drawbacks. Higher symbol variety means a larger header for tree. To overcome this problem we can use serialized binary tree bit arrays and symbol arrays.

1.1 Algorithm and Data Format

Huffman tree generation needs symbol frequencies or symbol ratios, also to place symbols on tree, algorithm needs symbols. In our case we are going to use ASCII characters as text files for input.

Tree construction algorithm uses queued node arrays to connect tree nodes. For every lowest symbol frequency pair we generate a new node from frequency sum. Huffman tree nodes are connected via total frequency sums to parent nodes.

Huffman tree construction:

1.Create a leaf node for each symbol and add it to the node array and sort the array.
2.While there is more than one node in the array:
    3.Remove the two nodes of highest priority (lowest probability) from the array
    4.Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities, new node symbol is not important.
    5.Add the new node to the array and sort the array.
6.The remaining node is the root node and the tree is complete.

Listed steps of the Huffman tree generation describes a specific implementation for Python 3. Normally heap is used for low frequency node selection but native Python heap implementation does not let the usage of same value elements in heapified

arrays. Thus an alternative implementation has been selected, used sorted arrays for low frequency selection.

After the construction of tree we can generate bit array form of the tree structure with left-node-right travel. In same travel order we need to store symbols.

Table 1 Tree serialization examples

File	Symbols and counts:	Tree and symbols:
sample_text_1.txt	'a': 2, 'd': 6, 'g': 12, 'f': 4, 's': 3	00001111 safdg
sample_text_3.txt	'z': 2, 'k': 7, 'e': 14, 'n': 4, 'a': 25, 'm': 8	0000101111 knzmea
sample_text_4.txt	'f': 3, 'a': 1, 'r': 2, 'n': 1, '0': 1, 's': 1, '7': 3, '4': 6, 'j': 2, '9': 7, '-': 2, 'k': 1, '6': 3, '8': 11, '3': 1, 'd': 1, '5': 4	0000100110111000111010011000111 53dsk9j-f4876n0ar
sample_text_5.txt	'o': 14, 'g': 1, 'x': 2, 'f': 5, ' ': 26, 'T': 1, 'n': 8, 'i': 9, 's': 17, 'p': 6, 'h': 4, 'a': 11, '.': 1, 'm': 4, 'v': 1, 'd': 7, 'u': 3, 'r': 11, 'e': 18, 't': 10, 'y': 2, 'C': 2, 'l': 7, 'c': 7	000000001101101101110011011010110001010111010101 .vgTyChmodlcp artfuxesin
sample_text_6.txt	'l': 1, 'C': 2, 'u': 1, ' ': 2, 'e': 1, 'y': 1, 'o': 1, 'a': 2, 'p': 1, 'g': 1, 'h': 1, 's': 1, 'T': 1, 'r': 2	0000111001101100011011 0101 aCeylughoprsT

Reconstruction of binary from bit expression of left-node-right traversal provides tree structure. After empty tree generation we can place the symbols at the leaf nodes. For reconstruction stack is used, because “0” and “1” effects on reconstructed tree levels and new nodes. Stack provides the required traversal logic. For every “0” a new left node is required, “1” has a more complex mechanism. For every “1” we need to go last empty right node in the tree and add a new right node after we need to push this new node, fortunately poping the last two nodes provides this parent node(empty right child).

Lastly added child can be a left node, when we receive a “1” after pop of two nodes we get the parent of this left node which has an empty right slot.

Lastly added child can be a right node, when we receive a “1” after pop of two nodes we reach the closest upper layer empty right slot node. First pop eliminates the lastly added right node. Second pop directly takes us the closest empty right node.

Tree reconstruction:
1.Push “ROOT” node to stack
2.While stack is not empty iterate on bit array

3.If next bit is 0 pop a node store locally push again, generate new node, set stored node’s left child to new node

If next bit is 1 pop a node, pop a node again and store locally, don’t push any, generate new node, set stored node’s right child to new node, push new node.

Symbol placing:
1.Push “ROOT” node to stack
2.While stack is not empty

3. Pop a node.

4.If node is not leaf push left child, push right child, else place symbol to node, increase symbol pointer

For reconstruction examples steps displays the empty tree generation, for every new node a new unique value is used, later this values are replaced with symbols for leaf nodes.

Table 2 Reconstruction example

Serialized data:	0000101111 knzmea
Reconstruction steps:	New-Parent Node-New Node left, parent: ROOT child: 1 left, parent: 1 child: 2 left, parent: 2 child: 3 left, parent: 3 child: 4 right, parent: 3 child: 5 left, parent: 5 child: 6 right, parent: 5 child: 7 right, parent: 2 child: 8 right, parent: 1 child: 9 right, parent: ROOT child: 10
Tree:	(((((k)3((n)5(z)))2(m))1(e))ROOT(a))

Table 3 Reconstruction example

Serialized data:	00001111 safdg
Reconstruction steps:	left, parent: ROOT child: 1 left, parent: 1 child: 2 left, parent: 2 child: 3 left, parent: 3 child: 4 right, parent: 3 child: 5 right, parent: 2 child: 6 right, parent: 1 child: 7 right, parent: ROOT child: 8
Tree:	(((((s)3(a))2(f))1(d))ROOT(g))

1.2 Implementation Details

For implementation Python 3.5.2 is used. For data input text files are used. Used samples are stored under “sample_test” folder under GitHub repository.

References

1. https://en.wikipedia.org/wiki/Huffman_coding