Huffman Encoding in Prolog

Sadok Dalin

October 10, 2018

1 Problem Description

Huffman coding is a lossless compression algorithm used primarily to encode text. The compression works by assigning binary codes of variable length to individual characters in an input text based on character frequency. Characters with a low frequency are assigned longer codes while the most frequent characters are assigned short codes.

Similar to ISO Latin-1 which is used in Prolog, Huffman codes are prefix codes. This means that no whole code word can be the prefix of another, simplifying the decoding process.

The main goal of this project was to write a Prolog program that could compress and decompress a given text using Huffman coding. The problem was divided into several subproblems:

- Count the frequency of all unique characters in a given text.
- Generate a Huffman-tree using the frequencies of distinct characters in the text.
- Traverse the Huffman tree and construct a binary code for each unique character in the input text.
- Compress a text given a set of Huffman codes and store it in a binary sequence.
- Decompress an encoded binary sequence and return the original text.

2 Solution

The Huffman encoding was implemented by writing 1-2 predicates for each subproblem. Since the steps in Huffman coding were discrete, dividing the implementation in this way seemed logical.

2.1 Counting the frequency of unique letters

The predicates letter_frequency/2, letter_frequency_helper/5 and freq_to_leaves/2 were responsible counting the frequency of each unique character in the text and also for converting lists of character frequencies to leaves.

Frequencies of each character were calculated by counting the duplicates of each element in a sorted list. letter_frequency/2 was used to detect the first occurrence of a unique character while letter_frequency_helper/5 was used to count and strip the remaining duplicates of that element from the input text. Each character and corresponding frequency was then converted into a leaf node.

2.2 Constructing the Huffman tree

The predicate construct_tree/2 was used to construct the Huffman Tree using by taking a list of leaves and outputting a labeled binary tree. There were two types of nodes in the tree, leaves and nodes. Individual characters and frequencies were stored in leaves while summaries of character frequencies were stored in nodes.

construct_tree/2 took a list of leaves sorted in ascending frequency order. The first two elements of the list were then aggregated in a node which was appended to the list. This continued until only a single node remained in the list, the Huffman Tree. Because the list was already sorted, the computational cost of appending the node to the resulting tree was minimized. An accumulating parameter was also used to optimize the recursion.

2.3 Constructing Huffman codes

The predicate huffman_code/2 was used in conjunction with the findall/3 predicate to find all Huffman codes(i.e. all paths in the tree leading to a leaf). huffman_code/2 located the code for a single character by going down the left branch recursively until a leaf was found. At each recursive call a 0 or a 1 was appended to an accumulator depending on which branch was explored. When

a leaf was found, the character(stored in the leaf) was returned along with the Huffman code in a list with 2 elements.

2.4 Encoding/Decoding a text

The predicates encode/3, decode/3 and search_letter/3 were used to encode and decode an input text using a list of characters to Huffman codes. The encoding went through each individual character in a text and searched for the corresponding code in the list of Huffman codes, accumulating the encoded text as a binary sequence.

Encoded texts were decoded by trial and error using a continuously growing buffer of bits. When a character corresponding to the binary sequence in the buffer was found, the buffer was emptied and the character was appended to a result list. search_letter/3 was used to either find the Huffman code or a character given a list of Huffman codes.

2.5 Test predicates

Two test predicates test/0 and test/2 were implemented to debug and print each step in the encoding and decoding process. test/0 took a static sample text and encoded/decoded it. Each step in the process was then printed. test/2 was implemented similarly but only prints the encoding process.

3 Conclusion

The code was not hard to write, especially since many of the steps in Huffman coding process are suited for recursive implementation. Most problems that were encountered were because of simple typos or wrong use of built in predicates. The most difficult part of the project was finding a way to incorporate the strengths of Prolog into the project in a natural way(such as using the backtracking for finding all the Huffman codes given a Huffman tree).

The tracer was helpful throughout the project to track recursions that never reached the base case. This was especially relevant for the predicates counting the frequencies of characters, which took the most time to write correctly.

The program can be improved in several ways:

- An implementation which could generate Huffman codes given one text

and encode/decode a different text. The current implementation is not designed to handle characters missing from the Huffman tree.

- The append predicate could be optimized to append bits in the front of the buffer instead of the end in decode/3.
- The decoding predicate could be made to be bidirectional so that it can guess a tree and possible combinations of letters from an encoded text.
- The Huffman encoding process could be changed to handle words of varying lengths in addition to single characters, compressing the text even further.