Information and Knowledge Management
ISSN 2224-5758 (Paper) ISSN 2224-896X (Online)
Vol 2, No.2, 2012

The Novel Lossless Text Compression Technique Using Ambigram Logic and Huffman Coding

Suresh Kumar, Madhu Rawat, Vishal Gupta*, Satendra Kumar

Department of Computer Science, G. B. Pant Engineering College, Pauri, Uttarakhand, INDIA

* E-mail of the corresponding author: vishalgupta87@gmail.com

Abstract

The new era of networking is looking forward to improved and effective methods in channel utilization. There are many texts where lossless data recovery is vitally essential because of the importance of information it holds. Therefore, a lossless decomposition algorithm which is independent of the nature and pattern of text is today's top concern. Efficiency of algorithms used today varies greatly depending on the nature of text. This paper mainly brings in the idea of using an art form called ambigram to compress text which is again compressed by Huffman coding with consistency in the efficiency of the compression.

Keywords: Ambigrams, Huffman coding, Lossless compression, Steganography, Embedded algorithms, Encryption.

1. Introduction

There are many algorithms exist in this world for compressing the data, some of them carries lossy techniques which sometimes destroy some important data also. Our technique is carries lossless compression using ambigram and Huffman coding which compress the data more than 60%. The ambigram technique is known to this world from decades earlier but not to be used for compressing the data. Our main concern is in this technique which can be further used with the much known compression technique Huffman coding without the loss of any data.

A. Ambigram - Definition

The word ambigram was firstly describe by Douglas R. Hofstadter, a computer scientist who is best known as the Pulitzer Prize winning author of the book Godel, Escher, Bach. In Hofstadter defines what he means by an ambigram.

"An ambigram is a visual pun of a special kind: a calligraphic design having two or more (clear) interpretations as written words. One can voluntarily jump back and forth between the rival readings usually by shifting one's physical point of view (moving the design in some way) but sometimes by simply altering one's perceptual bias towards a design (clicking an internal mental switch, so to speak). Sometimes the readings will say identical things; sometimes they will say different things."

B. Huffman Coding:

In computer science and information theory, Huffman coding is an entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol.

2. Types of Ambigrams

A. Half Turn Ambigrams

Half-tum ambigrams have two different readings and to switch from one to the other we simply have to rotate the ambigram 180 degrees in the plane it is living in as shown in Fig. 1.

B. Quarter Turn Ambigrams

Quarter Tum Ambigrams have three different readings and to switch from one to another we simply have to rotate 90 degrees in the clockwise or anti-clockwise direction as shown in Fig. 2.

C. Wall Reflection Ambigrams

Wall Reflection Ambigrams have two different readings and to switch from one to another you simply have to reflect through a vertical line in the plane as shown in Fig. 3.

D. Lake Reflection Ambigrams

Lake reflection ambigrams have two different readings and to switch from one to the other you simply have to reflect through a horizontal line in the plane.

E. Dissection Ambigrams

Another type ambigram that does not fall into either of the categories is the dissection ambigram. The example below illustrates that the circle can be squared after all as shown in Fig. 4.

3. Types of Huffman coding

Many variations of Huffman coding exist, some of which use a Huffman-like algorithm, and others of which find optimal prefix codes (while, for example, putting different restrictions on the output). Note that, in the latter case, the method need not be Huffman-like, and, indeed, need not even be polynomial time. An exhaustive list of papers on Huffman coding and its variations is given by "Code and Parse Trees for Lossless Source Encoding".

A. *n*-ary Huffman coding

The *n*-ary Huffman algorithm uses the {0, 1, ... , *n* − 1} alphabet to encode message and build an *n*-ary tree. This approach was considered by Huffman in his original paper. The same algorithm applies as for binary (*n* equals 2) codes, except that the *n* least probable symbols are taken together, instead of just the 2 least probable. Note that for *n* greater than 2, not all sets of source words can properly form an *n*-ary tree for Huffman coding. In this case, additional 0-probability place holders must be added. This is because the tree must form an *n* to 1 contractor; for binary coding, this is a 2 to 1 contractor, and any sized set can form such a contractor. If the number of source words is congruent to 1 modulo *n*-1, then the set of source words will form a proper Huffman tree.

B. Adaptive Huffman coding

A variation called adaptive Huffman coding involves calculating the probabilities dynamically based on recent actual frequencies in the sequence of source symbols, and changing the coding tree structure to match the updated probability estimates.

C. Huffman template algorithm

Most often, the weights used in implementations of Huffman coding represent numeric probabilities, but the algorithm given above does not require this; it requires only that the weights form a totally ordered commutative monoid, meaning a way to order weights and to add them. The Huffman template algorithm enables one to use any kind of weights (costs, frequencies, pairs of weights, non-numerical weights) and one of many combining methods (not just addition). Such algorithms can solve other minimization problems, such as minimizing $\max[w_i + \text{length}(c_i)]$, a problem first applied to circuit design.

D. Length-limited Huffman coding

Length-limited Huffman coding is a variant where the goal is still to achieve a minimum weighted path length, but there is an additional restriction that the length of each code word must be less than a given constant. The package-merge algorithm solves this problem with a simple greedy approach very similar to that used by Huffman's algorithm. Its time complexity is $O(nL)$, where L is the maximum length of a code

word. No algorithm is known to solve this problem in linear or linearithmic time, unlike the presorted and unsorted conventional Huffman problems, respectively.

4. Working model

In this model, the text to be compressed is got from the user which is stored in a temporary memory. First step is to calculate the position of white spaces in the entered text and store the same in a file. Similarly, the positions of special characters are stored in a separate file, after which the white spaces and special characters are removed from the original text. Then the number of alphabets in the text is calculated and the text is divided into two equal parts. The first part is taken and for each letter present, a symbol from the font file is chosen in such a way that when the text is rotated by 180 degrees, the second part of the text can be read. In this way the text can be compressed to about 50%. After this we compressed this text with Huffman coding which further compressed the data and the final data is compressed more than 60%. [Refer Fig. 5].

5. Implementation

A. Creating font file

Creating a font file for ambigram would require 26 symbols for each character. For example, 'a' alone requires 26 symbols for it has look like all possible letters of alphabet when rotated. An example for this is given below:

A true type font file containing about 676 ambigram symbols is created and each symbol is given a code as follows:

• Each of the letters in the English alphabet set is given an index from 0 to 25. For example, letter 'a' is given an index o. Under each alphabet index, a set of 26 symbols is created. For example, under 'a', i.e. under 0, 26 ambigram symbols are created by combining 'a' with all the 26 alphabets in such a way that when rotated 180 degrees, every other letter from a to z can be formed following which the code for each symbol is assigned to be

$$\text{code} = (\text{first alphabet's index} * 26) + \text{second alphabet's index} \quad (1)$$

For example, the code of the symbol which represents 'db' is calculated as (3*26) + 1 = 79. Thus the font file with 676 ambigram symbols with each one mapped to a user defined code is created.

B. Text Compression

During first phase of the compression, the first letter of the first part on rotating should be the last letter of the second part and the second letter of the first part of the first part must be the last but one letter of the last part. Thus the first letter of the first part and the last letter of the second part are taken and their corresponding indices are found out and assigned to i and j respectively. Then the code of the symbol for representing these two letters is found out using (1). The corresponding symbol is fetched from the font file and stored in a file and the two letters are removed from the original file. The process is repeated till there are no more letters left. If the total count of letters in the original file is odd, then a single letter will be left out, which will be copied as it is without any replacement in the compressed file containing symbols [8].

After that Huffman compression starts working which is the second phase of the compression and as follows:

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, *n*. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain symbol weight, links to two child nodes and the optional link to a

parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and $n - 1$ internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process essentially begins with the leaf nodes containing the probabilities of the symbol they represent, and then a new node whose children are the 2 nodes with smallest probability is created, such that the new node's probability is equal to the sum of the children's probability. With the previous 2 nodes merged into one node (thus not considering them anymore), and with the new node being now considered, the procedure is repeated until only one node remains, the Huffman tree.

The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
    1. Remove the two nodes of highest priority (lowest probability) from the queue
    2. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
    3. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Since efficient priority queue data structures require O(log n) time per insertion, and a tree with n leaves has $2n-1$ nodes, this algorithm operates in O(n log n) time, where n is the number of symbols.

If the symbols are sorted by probability, there is a linear-time (O(n)) method to create a Huffman tree using two queues, the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues:

1. Start with as many leaves as there are symbols.
2. Enqueue all leaf nodes into the first queue (by probability in increasing order so that the least likely item is in the head of the queue).
3. While there is more than one node in the queues:
    1. Dequeue the two nodes with the lowest weight by examining the fronts of both queues.
    2. Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.
    3. Enqueue the new node into the rear of the second queue.
4. The remaining node is the root node; the tree has now been generated.

Although this algorithm may appear "faster" complexity-wise than the previous algorithm using a priority queue, this is not actually the case because the symbols need to be sorted by probability before-hand, a process that takes O(n log n) time in itself.

In many cases, time complexity is not very important in the choice of algorithm here, since n here is the number of symbols in the alphabet, which is typically a very small number (compared to the length of the message to be encoded); whereas complexity analysis concerns the behavior when n grows to be very large.

It is generally beneficial to minimize the variance of code word length. For example, a communication buffer receiving Huffman-encoded data may need to be larger to deal with especially long symbols if the tree is especially unbalanced. To minimize variance, simply break ties between queues by choosing the item in the first queue. This modification will retain the mathematical optimality of the Huffman coding while both minimizing variance and minimizing the length of the longest character code.

C. Decompressing Text

While decompressing, the process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value). Before this can take place, however, the Huffman tree must be somehow reconstructed. In the simplest case, where character frequencies are fairly predictable, the tree can be pre-constructed (and even statistically adjusted on each compression cycle) and thus reused every time, at the expense of at least some measure of compression efficiency. Otherwise, the information to reconstruct the tree must be sent a priori. A naive approach might be to prepend the frequency count of each character to the compression stream.

This initial decompressed text with Huffman coding is read from the end of the file. If the end contains any letter it is copied as it is to a file. As and when a symbol is encountered, the code of the symbol is obtained on comparison with the font file. The indices of the two letters are calculated from the code as follows:

• Perform the operation (code / 26).

• The quotient gives the index of the first character (i)

And the remainder gives the index of the second character 0). Once the indices are calculated, the ambigram symbol is replaced by the corresponding pair of alphabets in the new file, by appending the alphabet corresponding to index 'i' to the beginning of the file and the alphabet corresponding to index 'j' to the end of the new file. After decompressing all the symbols with respective characters, the files with the position of white spaces and special characters are read and the white spaces and special characters are inserted accordingly in the new file thereby getting back the original text.

6. Applications in the field of steganography

The purpose of steganography is to hide the very presence of communication by embedding messages into innocuous-looking cover objects, such as digital images [9]. To accommodate a secret message, the original cover image is slightly modified by the embedding algorithm to obtain the stego image. Our compression method (which is the combination of Huffman coding and ambigrams) is applicable over a variety of data. For example-confidential data of Indian Army, navy and Air Force, confidential letters of the CEO of the companies, etc. where privacy is the most key issue. This method enhances the security and decreases detectability manifold as the original font set is made available to only the receiver. Therefore the secret message cannot be tracked by any external agent. The output of this technique is embedded by an image and then suitably encrypted and sent to the receiver with the corresponding stego key.

7. Conclusion and future work

We concluded here by saying that this technique presented here compressed text by around 60% which is comparable to other methods in existence. Moreover, unlike many other algorithms, this method does not restrict the user to give only specific types of inputs. Also, this is a lossless compression technique which involves no data loss while decompressing.

In future work, this proposed idea can be implemented and further be extended by embedding this technique in any other compression technique. Thereby the overall efficiency of compression can be further increased.

References

Douglas R. Hofstadter, Ambigrammi (in Italian), Hopeful-monster Editor, Firenze, 1987.

D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., September 1952, pp 1098–1102. Huffman's original article.

Burkard Polster, Les Ambigrammes-I'art de symetriser les mots, Editions Ecritextes, 2004.

John Langdon, "Wordplay Ambigrams and Reflections on the art of Ambigrams, Harcout" Brace Jovanovich, 1992.

Scott Kim, Inversions: A Catalog of Calligraphic CartweeIs, Byte Books, McGraw-Hili, 1981.

Ken Huffman. Profile: David A. Huffman, Scientific American, September 1991, pp. 54–58

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 16.3, pp. 385–392.

Gowtham S., Iyshwarya G., Kaushik Veluru, Tamarai Selvi A., Vasudha J., Text Compression Using Ambigrams, IEEE Conf. ICETE 2010,V4 page 509-511

Faisal AIturki, and Russell Mertsereau, "A Novel Approach for Increasing Security and Data Embedding Capacity in Images for Data Hiding Applications", *International Conference on Information Technology: Coding and Computing*, 2011. Page(s):228-233

Figure 1: Half Tum Ambigram of the word "ambigram"

Figure 2: Quarter Turn Ambigram of the word ”OHIO”

Figure 3: Wall Reflection Ambigram of the word “GEOMETRY”

Figure 4: Dissection Ambigram of the word “CIRCLE”

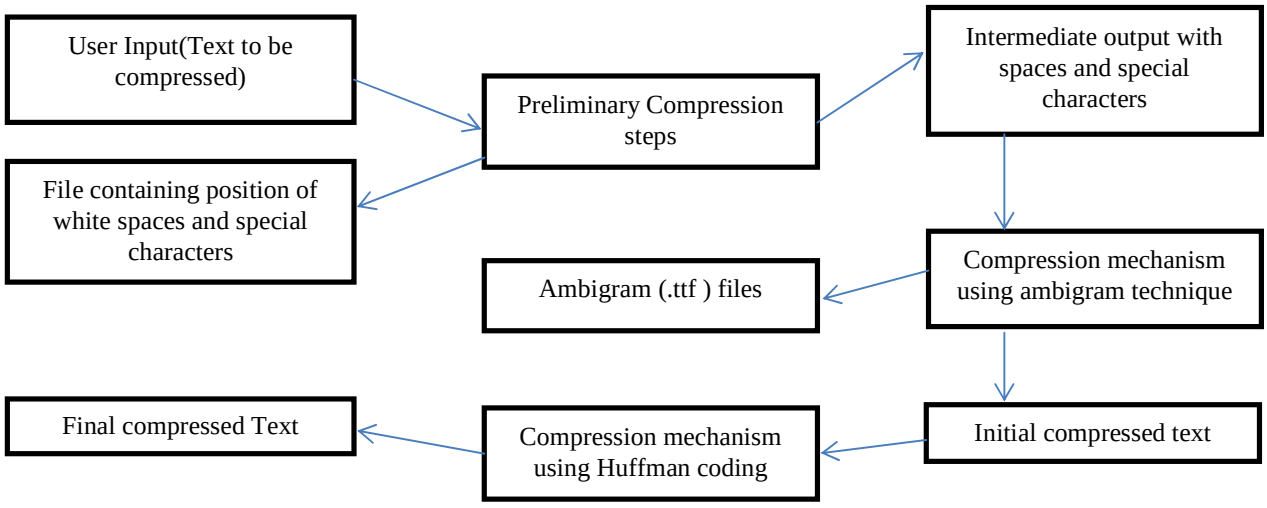

Figure 5: Text Compression Mechanism