Improvement in Compression Efficiency of Huffman Coding

Mohd. Faisal Muqtida
Computer Science, M.Tech, SRMSCET
Bareilly, India

Raju Singh Kushwaha
Assistant Professor, SRMSCET
Bareilly, India

ABSTRACT
Compression helps in reducing the redundancy in the data representation so as to reduce the storage requirement of it. The task of compression consists of two components, an encoding algorithm that takes a message and generates a "compressed" representation and a decoding algorithm that reconstructs the original message or some approximation of it from the compressed representation. Many algorithms are available for compressing the data. Some of the algorithms help in achieving lossless compression and some are good at lossy compression. In this paper, the proposed technique has improved the better compression ratio and compression efficiency on the Huffman Coding on data. The proposed technique works even with image file also but conventional Huffman Algorithm cannot do this. This paper also outlines the use of Data Normalization on the text data so as to remove redundancy for more data compression.

Keywords
Huffman Algorithm, Compression Ratio, Data normalization, Type Casting.

1. INTRODUCTION

1.1 DATA COMPRESSION
Compression helps in reducing the redundancy in the data representation so as to reduce the storage requirement of it. So we are bound to compress data. Compression is helpful because it helps reducing the data length, such as hard disk space. On the downside, compressed data must be decompressed to be used, and this extra processing may be harmful to some applications. A data compression feature can help reduce the size of the database as well as improve the performance of I/O intensive workloads. However, the type casting and data normalization technique is applied to get the maximum compression efficiency over the data. Reducing the amount of data required to represent a source of information while preserving the original content as much as possible. The main objectives of this paper are to achieve better compression efficiency by applying a proposed technique in a Huffman Algorithm.

2.CONVENTIONAL DATA COMPRESSION METHODS
Two type of compression exists in real world. One is Lossless compression and another is Lossy compression. Lossless and lossy compression are terms that describe whether or not, in the compression of a file, all original data can be recovered when the file is uncompressed.

2.1 Lossless Compression Vs. Lossy Compression
Lossless compression algorithms usually exploit statistical redundancy in such a way as to represent the sender's data more concisely, but nevertheless perfectly. Lossless compression is possible because most real-world data has statistical redundancy. For example, in English text, the letter 'e' is much more common than the letter 'z', and the probability that the letter 'q' will be followed by the letter 'z' is very small.
Another kind of compression, called lossy data compression, is possible if some loss of fidelity is acceptable. For example, a person viewing a picture or television video scene might not notice if some of its finest details are removed or not represented perfectly (i.e. may not even notice compression artifacts). Similarly, two clips of audio may be perceived as the same to a listener even though one is missing details found in the other. Lossy data compression algorithms introduce relatively minor differences and represent the picture, video, or audio using fewer bits.
Lossless compression schemes are reversible so that the original data can be reconstructed, while lossy schemes accept some loss of data in order to achieve higher compression. However, lossless data compression algorithms will always fail to compress some files; indeed, any compression algorithm will necessarily fail to compress any data containing no discernible patterns. Attempts to compress data that has been compressed already will therefore usually result in an expansion, as will attempts to compress Encrypted data. In practice, lossy data compression will also come to a point where compressing again does not work, although an extremely lossy algorithm, which for example always removes the last byte of a file, will always compress a file up to the point where it is empty. A good example of lossless vs. lossy compression is the following string -- 888883333333. What you just saw was the string written in an uncompressed form. However, you could save space by writing it 8[5]3[7]. By saying "5 eights, 7 threes", you still have the original string, just written in a smaller form. In a lossy system, using 83 instead, you cannot get the original data back (at the benefit of a smaller file size).

3. HUFFMAN ALGORITHM
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.
Huffman`s procedure creates the optimal code for a set of symbols and probabilities' subject to the constraints that the symbols be coded one at a time .
After the code has been created coding or Decoding Is accomplished in a simple look up table manner .the code itself is an instantaneous uniquely decodable block code. It is called a block code because each source symbol is mapped into a fixed sequence of code symbols.
codes.

3.1 Procedure of Huffman algorithm:

The Huffman algorithm generates the most efficient binary code tree at given frequency distribution. Prerequisite is a table with all symbols and their frequency. Any symbol represents a leaf node within the tree.

3.1.1 Compression

The following general procedure has to be applied:

- search for the two nodes providing the lowest frequency, which are not yet assigned to a parent node
- couple these nodes together to a new interior node
- add both frequencies and assign this value to the new interior node

The procedure has to be repeated until all nodes are combined together in a root node.

Example: "abracadabra"

Symbol	Frequency
a	5
b	2
r	2
c	1
d	1

According to the outlined coding scheme the symbols "d" and "c" will be coupled together in a first step. The new interior node will get the frequency 2.

STEP 1:

Symbol	Frequency	Symbol	Frequency
a	5	a	5
b	2	b	2
r	2	r	2
c	1 ---------->	1	2
d	1		

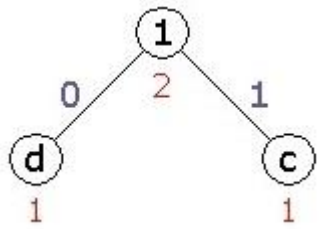


Fig.1: Code tree after the 1st step

STEP 2:

Symbol	Frequency	Symbol	Frequency
a	5	a	5
b	2	b	2
r	2 ---------->	2	4
1	2		

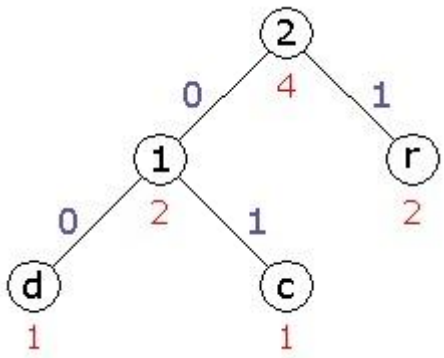


Fig.2: Code tree after the 2nd step:

STEP 3:

Symbol	Frequency	Symbol	Frequency
a	5	a	5
2	4 ---------->	3	6
b	2		

Code tree after the 3rd step:

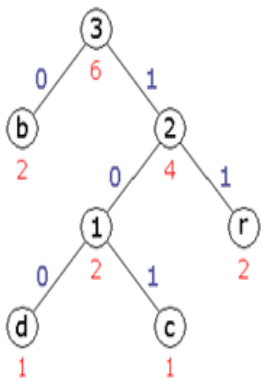


Fig.3: Code tree after the 3rd step.

STEP 4:

Symbol	Frequency	Symbol	Frequency
3	6 ---------->	4	11
a	5		

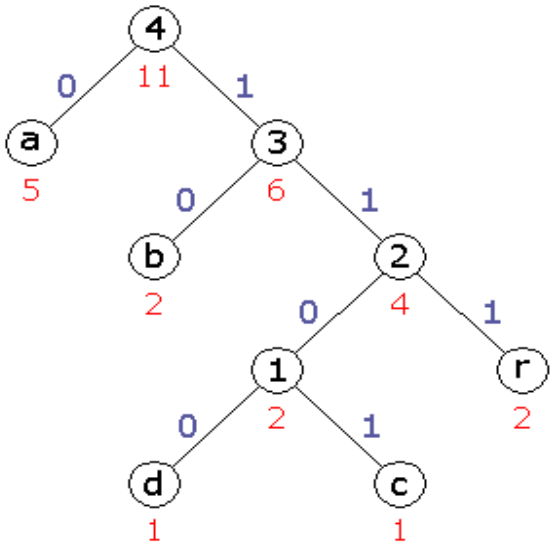


Fig.4: Code tree after the 4th step:

CODE TABLE

If only one single node is remaining within the table, it forms the root of the Huffman tree. The paths from the root node to the leaf nodes define the code word used for the corresponding symbol:

Symbol	Frequency	Code Word
a	5	0
b	2	10
r	2	111
c	1	1101
d	1	1100

Complete Huffman Tree:

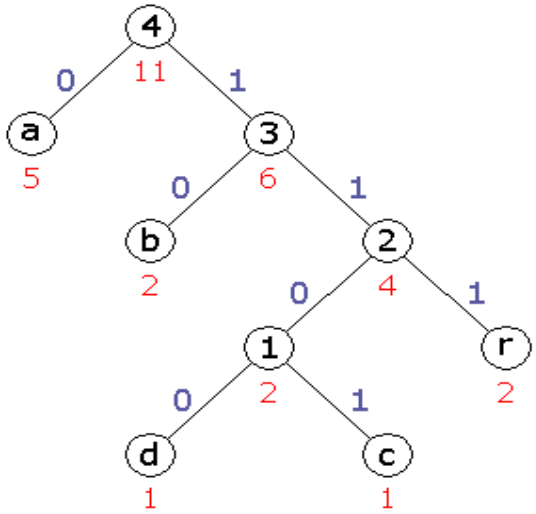


Fig.5: Complete Huffman Tree Encoding

The original data will be encoded with this code table as follows:

Symbol	Frequency	Code Word
a	5	0
b	2	10
r	2	111
c	1	1101
d	1	1100

a b r a c a d a b r a

0 10 111 0 1101 0 1100 0 10 111 0

Encoded data: 23 Bit

Original data: 33 Bit

3.1.2 DECOMPRESSION

For decoding the Huffman tree is passed through with the encoded data step by step. Whenever a node not having a successor is reached, the assigned symbol will be written to the decoded data.

0101110110101100010111 0

Encoded	decoded
0	a
10	b
111	r
0	a
1101	c
0	a
1100	d
0	a
10	b
111	r
0	a

4. TYPE CASTING

Type conversion, **Typecasting**, and **coercion** refer to different ways of, implicitly or explicitly, changing an entity of one data type into another. This is done to take advantage of certain features of type hierarchies or type representations. One example would be small integers, which can be stored in a compact format and converted to a larger representation when used in arithmetic computations. In object-oriented programming, type conversion allows programs to treat objects of one type as one of their ancestor types to simplify interacting with them.

Each programming language has its own rules on how types can be converted. In general, both objects and fundamental data types can be converted. In most languages, the word **coercion** is used to denote an implicit conversion, either during compilation or during run time. A typical example would be an expression mixing integer and floating point numbers (like 5 + 0.1), where the integers are normally converted into the latter. Explicit type conversions can either be performed via built-in routines (or a special syntax) or via separately defined conversion routines such as an overloaded object constructor.

In most Algol-based languages with nested function definitions, such as Pascal, Delphi, Modula 2 and Ada, conversion and casting are distinctly different concepts. In these languages, conversion refers to either implicitly or explicitly changing a value from one data type to another, e.g. a 16-bit integer to a 32-bit integer. The storage requirements may change as a result of the conversion. A loss of precision or truncation may also occur. The word cast, on the other hand, refers to explicitly changing the interpretation of the bit pattern representing a value from one type to another. For example 32 contiguous bits may be treated as an array of 32 Booleans, a two character Unicode string, an unsigned 32-bit integer or an IEEE single precision floating point value. While the storage requirements are never changed, it still requires knowledge of low level details such as representation format, byte order, and alignment requirements in order to be meaningful.

5. Data Normalization

Data normalization() is a process in which data attributes within a data model are organized to increase the cohesion of entity types. In other words, the goal of data normalization is to reduce and even eliminate data redundancy, an important consideration for application developers because it is incredibly difficult to stores objects in a relational database that maintains the same information in several places. **Table 1** summarizes the three most common forms of normalization (**First normal form (1NF)**, **Second normal form (2NF)**, and **Third normal form (3NF)**) describing how to put entity types into a series of increasing levels of normalization. Higher levels of data normalization are beyond the scope of this article. With respect to terminology, a data schema is considered to be at the level of normalization of its least normalized entity type. For example, if all of your entity types are at second normal form (2NF) or higher then we say that your data schema is at 2NF.

Table 1. Data Normalization Rules.

Level	Rule
First normal form (1NF)	An entity type is in 1NF when it contains no repeating groups of data.
Second normal form (2NF)	An entity type is in 2NF when it is in 1NF and when all of its non-key attributes are fully dependent on its primary key.
Third normal form (3NF)	An entity type is in 3NF when it is in 2NF and when all of its attributes are directly dependent on the primary key.

First normal form:

- Eliminate repeating groups in individual tables.
- Create a separate table for each set of related data.
- Identify each set of related data with a primary key.

Second normal form:

- Create separate tables for sets of values that apply to multiple records.
- Relate these tables with a foreign key.

Records should not depend on anything other than a table's primary key (a compound key, if necessary). For example, consider a customer's address in an accounting system. The address is needed by the Customers table, but also by the Orders, Shipping, Invoices, Accounts Receivable, and Collections tables. Instead of storing the customer's address as a separate entry in each of these tables, store it in one place, either in the Customers table or in a separate Addresses table.

Third normal form:

- Eliminate fields that do not depend on the key.

Values in a record that are not part of that record's key do not belong in the table. In general, any time the contents of a group of fields may apply to more than a single record in the table, consider placing those fields in a separate table.

For example, in an Employee Recruitment table, a candidate's university name and address may be included. But you need a complete list of universities for group mailings. If university information is stored in the Candidates table, there is no way to list universities with no current candidates. Create a separate Universities table and link it to the Candidates table with a university code key.

It may be more feasible to apply third normal form only to data that changes frequently. If some dependent fields remain, design your application to require the user to verify all related fields when any one is changed.

Normalizing an Example Table:

These steps demonstrate the process of normalizing a fictitious student table.

1. Unnormalized table:

Student#	Advisor	Adv-Room	Class1	Class2	Class3
1022	Jones	412	101-07	143-01	159-02
4123	Smith	216	201-01	211-02	214-01

2. **First Normal Form:** No Repeating Groups

Tables should have only two dimensions. Since one student has several classes, these classes should be listed in a separate table. Fields Class1, Class2, and Class3 in the above records are indications of design trouble.
Spreadsheets often use the third dimension, but tables should not. Another way to look at this problem is with a one-to-many relationship, do not put the one side and the many side in the same table. Instead, create another table in first normal form by eliminating the repeating group (Class#), as shown below:

Student#	Advisor	Adv-Room	Class#
1022	Jones	412	101-07
1022	Jones	412	143-01
1022	Jones	412	159-02
4123	Smith	216	201-01
4123	Smith	216	211-02
4123	Smith	216	214-01

3. **Second Normal Form:** Eliminate Redundant Data
Note the multiple Class# values for each Student# value in the above table. Class# is not functionally dependent on Student# (primary key) , so this relationship is not in second normal form. The following two tables demonstrate second normal form:

Students:

Student#	Advisor	Adv-Room
1022	Jones	412
4123	Smith	216

Registration:

Student#	Class#
1022	101-07
1022	143-01
1022	159-02
4123	201-01
4123	211-02
4123	214-01

4. **Third Normal Form:** Eliminate Data Not Dependent On Key
In the last example; Adv-Room (the advisor's office number) is functionally dependent on the Advisor attribute. The solution is to move that attribute from the Students table to the Faculty table, as shown below:

Students:

Student#	Advisor
1022	Jones
4123	Smith

Faculty:

Name	Room	Dept
Jones	412	42
Smith	216	42

6. PROPOSED TECHNIQUE

In Matlab we have implemented Huffman coding in a two step procedure the first step is converting normal file to Huffman file and getting the compressed code outside. In next step or second step we convert that compressed code to back to original text code.

6.1 Proposed Compression Method

The proposed technique can be clearly understood by the block diagram below:

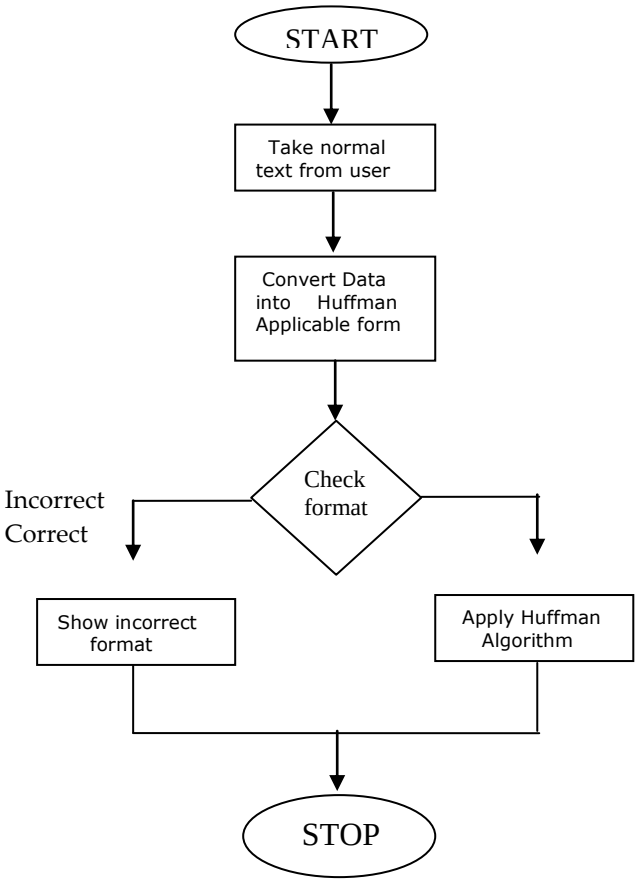


Fig:6 Compression method using proposed tehchnique

In the diagram we can see that first our Matlab code is taking data from the user. After that it is converting that data to UINT8 format which can be compressed using Matlab if by

any chance data is not in UINT8 format then it cannot be compressed. After it is converted and we apply Huffman compression and get a compressed code.

6.2 PROPOSED DECOMPRESSION METHOD

The proposed technique can be clearly understood by the block diagram below:

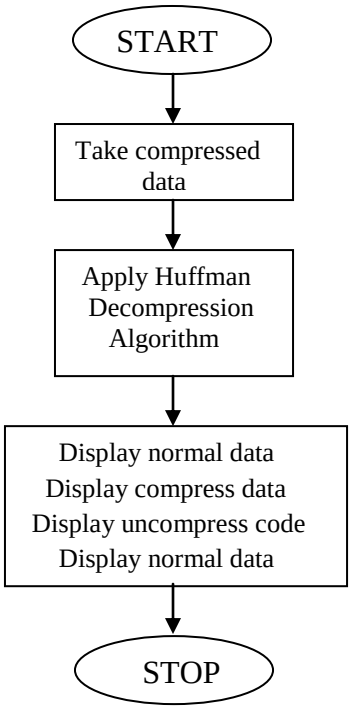


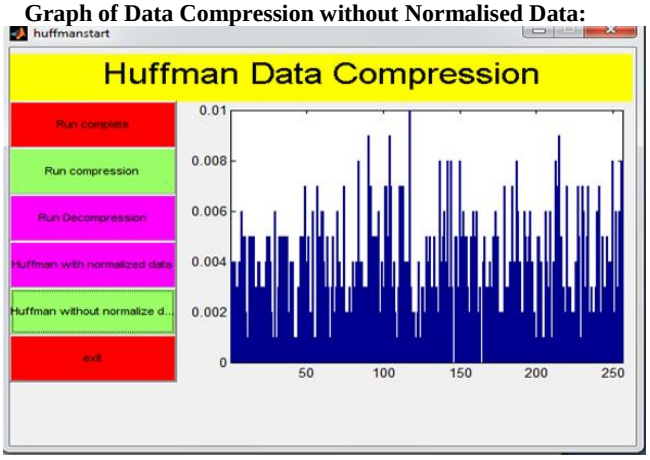
Fig.7 Decompression using proposed techinique

In above block diagram we can see that the code is converting compressed data back to uncompressed data after that it is typecasting the uncompressed data so that we can again read the text file and see if it has been properly converted.

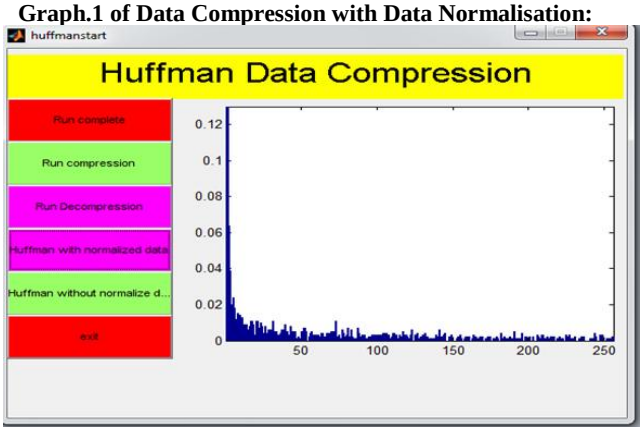
7. IMPROVEMENT IN COMPRESSION EFFICIENCY

The above proposed compression and decompression method is developed in MATLAB, and we have used the various types of file formats. Also from the tool we have analysed that the compression efficiency of Huffman coding is improved in comparison to the conventional Huffman coding by using the proposed technique by 40%. By using Data normalization before compression we are improving the compression efficiency of Huffman Algorithm. This can be more cleared with the following graph:

Graph of Data Compression without Normalised Data:



Graph.1 of Data Compression with Data Normalisation:



The compression efficiency is improved a lot and is been verified by applying on different formats files. The table below shows that how much file has been compressed using the proposed technique:

Tabels For Compression & Decompression:

File Format	Size of File to be Compressed (Kb)	Size of Compressed File (Kb)	Size of File after Decompression (Kb)
.txt	30	19	30
.doc	386	159	386
.ppt	66	32	66

8. CONCLUSION

Data compression is most consideration thing of the recent world. We have to compress a huge amount of data so as to carry from one place to other or in a storage format. That is why data has to compress. This proposed compression technique has improved the efficiency of compression of Huffman Algorithm and also the proposed compression technique for Huffman Algorithm is compressing image in a file also but conventional Huffman Algorithm cannot do this. So by including the existence of Typecasting and Data Normalization we had improve the compression efficiency.

9. REFERENCES

[1] Ternary Tree & A new Huffman Decoding Technique, IJCSNS International Journal of Computer Science and Network Security, Vol.10 N0.3, March 2010.

[2] A Study and implementation of the Huffman Algorithm based on Condensed Huffman Table, 2008 International Conference on Computer Science and Software Engineering.

[3] A Method for the Construction of Minimum-Redundancy Codes David A. Huffman, Associate, IRE.(1952).

[4] Typecasting, Legitimation: A Formal Theory Greta Hsu Univ. Of California at Davis Michael t. Hannan Stanford University.

[5] Database Normalisation: Korth Refrences:A CS2 assignment.

[6]Database System Concepts By SILBERSCHATZ, KORTH, SUDARSHAN McGraw-Hill Higher Education, ISBN NO. 0-07-120413-X

[7] A Method for the Construction of Minimum-Redundancy Codes DAVID A. HUFFMAN, ASSOCIATE,IRE.

[8] A study and implementation of the Huffman Algorithm based on condensed Huffman table, 2008 Software Engineering.

[9] Owen L.Astrachan,2004,Huffman coding:A CS2 assignment.

[10]Typecasting, Legitimation , and Form Emergence: A Formal Theory Greta Hsu Univ. of California at Davis Michael T. Hannan Stanford University László Pólos Durham University Running head: Typecasting, Legitimation, and Form Emergence March 24, 2010