Abstract - In Computer Science and Information Theory, data compression, or source coding, or bit rate reduction involves encoding the information using fewer bits than original representation. Compression is useful because it helps reduce resources usage, such as data storage space or transmission capacity. Not many Data Compression softwares for Android mobile phone are available in the market. Onavo Extend is one amongst the few that plays the lead in this regard. Onavo was initially created for iOS Mobiles but now it has extended its reach to the Android mobile also. It lacks some connectivity and quality issues, and this project intends to overcome all these problems. This project intend to create a faster and quality compliant data compression product for the Android Mobile Industry using Adaptive Huffman code to compress and Decompress the data thus reducing the byte intake.

Index terms - Adaptive Huffman coding, Android, Data compression, Huffman Coding

I. INTRODUCTION

Compression is the art of representing the information in a compact form rather than its original or uncompressed form. In other words, using the data compression, the size of a particular file can be reduced. Smartphones and tablet PCs are making big change in our life these days. The most popular operating systems for smart devices are Apple’s iOS and Google’s Android. Because Android is open source software, and offers developers free platform to make their own applications, lots of hardware vendors adopt Android and market share is also increasing.

Motivation and Problem Statement

Not many Data Compression softwares for Android mobile phone are available in the market. Onavo Extend is one amongst the few that plays the lead in this regard. Onavo was initially created for iOS Mobiles but now it has extended its reach to the Android mobile also. It lacks some connectivity and quality issues. This project is intended to overcome all such problems. Main objective of this project is to reduce the amount of data storage space required and thereby reducing length of data transmission time over the network and to provide faster data transmission.

II. VARIOUS DATA COMPRESSION ALGORITHMS

Lossless data compression is used to compact files or data into a smaller form. It is often used to package up software before it is sent over the Internet or downloaded from a web site to reduce the amount of time and bandwidth required to transmit the data. Lossless data compression has the constraint that when data is uncompressed, it must be identical to the original data that was compressed. Graphics, audio, and video compression such as JPG, MP3, and MPEG on the other hand use lossy compression schemes which throw away some of the original data to compress the files even further. So many researches were carried out in the area of compression. In the paper [1] ‘A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture’ (2007) several types of lossless compression and decompression techniques are described. In the paper ‘Comparison Of Lossless Data Compression Algorithms For Text Data’(2009),[2] comparison of different compression techniques are available, it is estimated that Huffman coding is having high code efficiency. In the paper ‘A Study on the Performance of Android Platform’, (2012), [3] it is said that, as the Android platform is widely used for embedded systems including smart mobile devices, the needs for systematic performance analysis have significantly increased. System performance is usually measured by benchmarks and profiler software. The performance of Android platform using a benchmark application and public profile software was analyzed.
In the paper, ‘New Approach For Time Efficient Backup and Restore on on-line Server using Android Platform’{4} a system is developed which incorporated the implementation of backup and restore of mobile data on android platform with reduce in time using some compression and encoding method for reducing data size of mobile device and there will be benefits of this encoding and compression method in time required to backup and restore of mobile device data on on-line server. In the paper ‘SMS Text Compression through IDBE (Intelligent Dictionary based Encoding) for Effective Mobile Storage Utilization’, {5} it is described that effective storage utilization is the key concept for better working of any operating system. Even operating systems used for mobile phones are not an exception for this fact. This paper proposes a technique for maximizing the utilization of the storage space present in mobile phones. Thus it is important to utilize the space occupied by SMS files in phone’s memory, which take maximum space. In the paper ‘Efficient Test-Data Compression for IP Cores Using Multilevel Huffman Coding’ {6}, a new test-data compression method for IP cores with unknown structure is introduced. The proposed method encodes the test data provided by the core vendor using a new, very effective compression scheme based on multilevel Huffman coding. In the paper ‘Ternary Tree and Memory-Efficient Huffman Decoding Algorithm’ {7}, the focus was on the use of ternary tree over binary tree. Here, a new one pass Algorithm for Decoding adaptive Huffman ternary tree codes was implemented. To reduce the memory size and fasten the process of searching for a symbol in a Huffman tree, this paper exploits the property of the encoded symbols and proposed a memory efficient data structure to represent the code word length of Huffman ternary tree. In the paper ‘Comparative Study Between Various Algorithms of Data Compression Techniques’ {8}, the focus is on the most prominent data compression schemes, particularly popular .DOC, .TXT, .BMP, .TIF, .GIF, and .JPG files. By using different compression algorithms, some results are obtained and regarding to these results the efficient algorithm to be used with a certain type of file to be compressed taking into consideration both the compression ratio and compressed file size is suggested. In the paper ‘Design of Modified Adaptive Huffman Data Compression Algorithm for Wireless Sensor Network’ {9}, a simple algorithm namely modified adaptive Huffman algorithm which does not require prior knowledge of the statistics of sensor data is proposed for data compression in the sensor network. The data compression is performed adaptively based on the temporal correlation in the sensor data. This modified adaptive Huffman encoding algorithm effectively combines the advantages of static and adaptive Huffman algorithms to provide effective compression by reducing the number of levels in the binary tree. In the paper {10}, various techniques for compression are described and are compared on the basis of use in different applications and their advantages and disadvantages. It was concluded that arithmetic coding is very efficient for more frequently occurring sequences of pixels with fewer bits and reduces the file size dramatically.

III. ABOUT ANDROID

Android is a Java-based mobile operating system that runs on the Linux 2.6 kernel. It was originally developed by a start up of the same name, Android, Inc. In 2005, Google purchased Android and took over its development work. Google wanted Android to be open and free; hence, most of the Android code was released under the open source Apache License, which means that anyone who wants to use Android can do so by downloading the full Android source code. The main advantage of adopting Android is that it offers a unified approach to application development. Android Versions: Android has gone through quite a number of updates since its first release. The version history of the Android mobile operating system began with the release of the Android beta in November 2007. The first commercial version, Android 1.0, was released in September 2008. Android is under on going development by Google and the Open Handset Alliance (OHA), and has seen a number of updates to its base operating system since its initial release. Android versions have been developed under a codename and released according to alphabetical order:

- Beta
- Cupcake (1.5)
- Donut (1.6)
- Eclair (2.0–2.1)
- Froyo (2.2–2.2.3)
- Gingerbread (2.3–2.3.7)
- Honeycomb (3.0–3.2.6)
- Ice Cream Sandwich (4.0–4.0.4)
- Jelly Bean (4.1–4.3)
- KitKat (4.4)

IV. HUFFMAN CODING

Huffman codes are optimal prefix codes generated from a set of probabilities by a particular algorithm, the Huffman Coding Algorithm. The algorithm is now probably the most prevalently used component of compression algorithms, used as the back end of GZIP, JPEG and many other utilities. The Huffman algorithm is very simple and is most easily described in terms of how it generates the prefix-code tree.

- Start with a forest of trees, one for each message. Each tree contains a single vertex with weight wi = pi
- Repeat until only a single tree remains

Select two trees with the lowest weight roots (w1 and w2). Combine them into a single tree by adding a new root with weight w1+w2, and making the two trees its children. It does not matter which is the left or right child, but our convention will be to put the lower weight root on the left if w1 = w2. For a code of size n this algorithm will require n−1 steps since every complete binary tree with n leaves has n−1 internal nodes, and each step creates one internal node. If we use a priority queue with O(logn) time insertions and find mins (e.g., a heap) the algorithm will run in O(nlogn) time. The key property of Huffman codes is that they generate optimal prefix codes. Huffman coding requires prior knowledge of the probabilities of the source sequence. If this knowledge is not available, Huffman coding becomes a two pass procedure: the statistics are collected in the first pass and the source is encoded in the second pass.

BUILDING A HUFFMAN TREE

The easiest way to see how this algorithm works is to work through an example. The algorithm is as follows: Find the two binary trees in the list that store minimum frequencies at their nodes. Connect these two nodes at a newly created common node that will store NO character but will store the
sum of the frequencies of all the nodes connected below it. So
our picture looks like follows:

```
9 12 'a' 13 'd' 14 'e' 85 'f'
/ \  
2 'b' 7 'c'
```

Now, repeat this process until only one tree is left:

```
21
/
9 12 'a' 13 'd' 14 'e' 85 'f'
/ \  
2 'b' 7 'c' 48
```

Once the tree is built, each leaf node corresponds to a letter
with a code. To determine the code for a particular node, walk
a standard search path from the root to the leaf node in
question. For each step to the left, append a 0 to the code and
for each step right append a 1. Thus for the tree above we get
the following codes:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>001</td>
</tr>
<tr>
<td>'b'</td>
<td>0000</td>
</tr>
<tr>
<td>'c'</td>
<td>0001</td>
</tr>
<tr>
<td>'d'</td>
<td>010</td>
</tr>
<tr>
<td>'e'</td>
<td>011</td>
</tr>
<tr>
<td>'f'</td>
<td>1</td>
</tr>
</tbody>
</table>

Another algorithm called Adaptive Huffman coding
(Dynamic Huffman coding) is found to have better
advantages than the above mentioned ones. In an adaptive
coding technique based on Huffman coding, building the
code as the symbols are being transmitted, having no initial
knowledge of source distribution, that allows one-pass
encoding and adaptation to changing conditions in data. The benefit of one-pass
procedure is that the source can be encoded in real time.

V. SIMULATION RESULTS

In the First Phase of this Project, a simple Huffman encoding
is done. The code is written in Java and simulated in
NetBeans IDE 7.2.1. Here, the String test is taken as "this is
an example for huffman encoding". After scanning the string
text, we read the characters and record the character
frequencies. Input is an array of frequencies, indexed by
character code. Initially we have a forest of trees. Then we
take two trees with lowest frequency, Put it into a new node
and re-insert into queue. Finally the tree is build and the code
is noted. Simulation results are shown below.

![Output - Java (run) %]

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>WEIGHT</th>
<th>HUFFMAN CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>1</td>
<td>0000</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
<td>0001</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td>0010</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>0011</td>
</tr>
<tr>
<td>b</td>
<td>5</td>
<td>0100</td>
</tr>
<tr>
<td>f</td>
<td>6</td>
<td>0101</td>
</tr>
<tr>
<td>d</td>
<td>7</td>
<td>1000</td>
</tr>
<tr>
<td>i</td>
<td>8</td>
<td>1001</td>
</tr>
<tr>
<td>t</td>
<td>9</td>
<td>1010</td>
</tr>
<tr>
<td>s</td>
<td>10</td>
<td>1011</td>
</tr>
<tr>
<td>e</td>
<td>11</td>
<td>1100</td>
</tr>
<tr>
<td>u</td>
<td>12</td>
<td>1101</td>
</tr>
</tbody>
</table>

BUILD SUCCESSFUL (total time: 0 seconds)

INFERENCES

After implementing simple Huffman coding, some inferences
are obtained. Some of the advantages of Huffman Coding
were

- Algorithm is easy to implement
- Produce a lossless compression of images

Main disadvantages found are

- Efficiency depends on the accuracy of the statistical model
  used and type of image.
- All codes of the encoded data are of different sizes (not of
  fixed length). Therefore it is very difficult for the decoder to
  know that it has reached the last bit of a code.

It is found that Adaptive Huffman Coding can provide better
results than Huffman Coding. Adaptive huffman coding
(Dynamic Huffman coding) is an adaptive coding technique
based on Huffman coding, building the code as the symbols
are being transmitted, having no initial knowledge of source
distribution, that allows one-pass encoding and adaptation to
changing conditions in data. The benefit of one-pass
procedure is that the source can be encoded in real time.
Hence, it is concluded that, in the 2nd phase of this project,
Adaptive Huffman Coding will used.

VI. ADAPTIVE HUFFMAN CODING

There are a few shortcomings to the straight Huffman
compression. First of all, the need to send the Huffman tree at
the beginning of the compressed file, or the decompressor
will not be able to decode it. This can cause some
overhead.Also, Huffman compression looks at the statistics
of the whole file, so that if a part of the code uses a character
more heavily, it will not adjust during that section. The
solution to all of these problems is to use an Adaptive method.
The Concept:
The basic concept behind an adaptive compression algorithm is very simple:

Initialize the model
Repeat for each character
   Encode character
   Update the model

Decompression works the same way. As long as both sides have the same initialize and update model algorithms, they will have the same information. The problem is how to update the model. To make Huffman compression adaptive, just re-make the Huffman tree every time a character is sent, but that would cause an extremely slow algorithm. The trick is to only update the part of the tree that is affected.

The Algorithm:
The Huffman tree is initialized with a single node, known as the Not-Yet-Transmitted (NYT) or escape code. This code will be sent every time that a new character, which is not in the tree, is encountered, followed by the ASCII encoding of the character. This allows for the decompressor to distinguish between a code and a new character. Also, the procedure creates a new node for the character and a new NYT from the old NYT node. Whenever a character that is already in the tree is encountered, the code is sent and the weight is increased. In order to do this algorithm to work, some additional information to the Huffman tree should be added. In addition to each node having a weight, it will now also be assigned a unique node number.

VARIous ALGORITHMS IN ADAPTIVE HUFFMAN CODING
In the 2nd phase of the project, Adaptive Huffman Coding is implemented. Adaptive Huffman coding (also known as Dynamic Huffman coding) is an adaptive coding technique based on Huffman coding. It permits building the code as the symbols are being transmitted, having no initial knowledge of source distribution, that allows one-pass encoding and adaptation to changing conditions in data. The benefit of one-pass procedure is that the source can be encoded in real time, though it becomes more sensitive to transmission errors, since just a single loss ruins the whole code.

ALGORITHMS
There are a number of implementations of this method, the most notable are FGK (Faller-Gallager-Knuth) and Vitter algorithm.

FGK Algorithm
It is an online coding technique based on Huffman coding. Having no initial knowledge of occurrence frequencies, it permits dynamically adjusting the Huffman's tree as data are being transmitted. In a FGK Huffman tree, a special external node, called 0-node, is used to identify a newly-coming character. That is, whenever a new data is encountered, output the path to the 0-node followed by the data. For a past-coming character, just output the path of the data in the current Huffman's tree. Most importantly, Huffman tree must be adjusted if necessary, and finally update the frequency of related nodes. As the frequency of a datum is increased, the sibling property of the Huffman's tree may be broken. The adjustment is triggered for this reason. It is accomplished by consecutive swapings of nodes, subtrees, or both. The data node is swapped with the highest-ordered node of the same frequency in the Huffman's tree. All ancestor nodes of the node should also be processed in the same manner. Since the FGK Algorithm has some drawbacks about the node-or-subtree swapping, Vitter proposed another algorithm to improve it.

VITTER ALGORITHM
Code is represented as a tree structure in which every node has a corresponding weight and a unique number. Numbers go down, and from right to left. Weights must satisfy the sibling property, which states that nodes must be listed in the order of decreasing weight with each node adjacent to its sibling. To get the code for every node, in case of binary tree we could just traverse all the path from the root to the node, writing down (for example) "1" if we go to the right and "0" if we go to the left. Some general and straightforward method are needed to transmit symbols that are "not yet transmitted" (NYT). Encoder and decoder start with only the root node, which has the maximum number. In the beginning it is our initial NYT node. When transmitting an NYT symbol, we have to transmit code for the NYT node, then for its generic code. For every symbol that is already in the tree, we only have to transmit code for its leaf node. For every symbol transmitted both the transmitter and receiver execute the update procedure:

1. If current symbol is NYT, add two child nodes to NYT node. One will be a new NYT node the other is a leaf node for our symbol. Increase weight for the new leaf node and the old NYT and go to step 4. If not, go to symbol's leaf node.
2. If this node does not have the highest number in a block, swap it with the node having the highest number, except if that node is its parent.
3. Increase weight for current node
4. If this is not the root node go to parent node then go to step 2. If this is the root, end.

Example
Start with an empty tree.

For "a" transmit its binary code. NYT spawns two child nodes: 254 and 255. Increase weight for root. Code for "a", associated with node 255, is 1. For "b" transmit 0 (for NYT node) then its binary code. NYT spawns two child nodes: 252

ISSN: 2278 – 1323 All Rights Reserved © 2014 IJARCET 2263
VII. GETTING STARTED WITH ANDROID

Google provides two integrated development environments (IDEs) to develop new applications. The Android Developer Tools (ADT) are based on the Eclipse IDE. ADT is a set of components (plug-ins), which extend the Eclipse IDE with Android development capabilities. Google also supports an IDE called Android Studio for creating Android applications. The Android system uses a special virtual machine, i.e., the Dalvik Virtual Machine (Dalvik) to run Java based applications. Dalvik uses a custom bytecode format which is different from Java bytecode. Therefore you cannot run Java class files on Android directly; they need to be converted into the Dalvik bytecode format. Similar to the JVM, Dalvik optimizes the application at runtime. This is known as Just In Time (JIT) compilation.

DEVELOPING ANDROID APPLICATIONS

Android applications are primarily written in the Java programming language. During development the developer creates the Android specific configuration files and writes the application logic in the Java programming language. The ADT or the Android Studio tools convert these application files, transparently to the user, into an Android application. When developers trigger the deployment in their IDE, the whole Android application is compiled, packaged, deployed and started. The Java source files are converted to Java class files by the Java compiler. The Android SDK contains a tool called dx which converts Java class files into a .dex (Dalvik Executable) file. All class files of the application are placed in this .dex file. During this conversion process redundant information in the class files are optimized in the .dex file. Android contains a permission system and predefines permissions for certain tasks. Every application can request required permissions and also define new permissions. For example, an application may declare that it requires access to the Internet.

Installation: Download packaged Android. AVDs allow you to test your Android applications on different Android versions and configurations without access to the real hardware. During the creation of your AVD you define the configuration for the virtual device. This includes, for example, the resolution, the Android API version and the density of your display. After the download you may need to restart your development environment to be able to create an AVD with the intel emulator.

Create an Android Virtual Device (AVD)

Define a new Android Virtual Device (AVD) by opening the AVD Manager via Window → Android Virtual Device Manager and by pressing the New button.
VIII. CONCLUSION

EXPERIMENTS ON VARIOUS IMAGES

Various images were considered and compression is done on all the sample images. Compression Ratio is the ratio between the size of the compressed file and the size of the source file. Compression Ratio = Size after compression / Size before compression.

Compression Factor is the inverse of the compression ratio. That is the ratio between the size of the source file and the size of the compressed file. Compression Factor = Size before compression / Size after compression

Saving Percentage calculates the shrinkage of the source file as a percentage. Saving percentage = (size before compression - size after compression) / size before compression.

Image 1:

The uncompressed image has a size of 28. We obtained a decompressed image size of 27.9 KB

Image 2:

The uncompressed image has a size of 872 KB. We obtained a decompressed image size of 269 KB retained.

Image 3:

The uncompressed image has a size of 116 KB. We obtained a decompressed image size of 81.9 KB.
COMPARISON:

<table>
<thead>
<tr>
<th>SI No</th>
<th>Size of original image</th>
<th>Size of compressed image</th>
<th>Compression Ratio</th>
<th>Compression Factor</th>
<th>Saving Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>144 KB</td>
<td>29.8 KB</td>
<td>0.206</td>
<td>4.83</td>
<td>0.793</td>
</tr>
<tr>
<td>2</td>
<td>872 KB</td>
<td>269 KB</td>
<td>0.308</td>
<td>3.24</td>
<td>0.691</td>
</tr>
<tr>
<td>3</td>
<td>28.6 KB</td>
<td>27.9 KB</td>
<td>0.975</td>
<td>1.02</td>
<td>0.024</td>
</tr>
<tr>
<td>4</td>
<td>116 KB</td>
<td>81.9 KB</td>
<td>0.706</td>
<td>1.41</td>
<td>0.293</td>
</tr>
</tbody>
</table>

IX. SIMULATION RESULTS IN ANDROID

Initially, a login page will be displayed. Login using the specified username and password. Here, once we chose the ‘load image from URL, the image which is to be compressed is loaded’.

Once the image is loaded, click on the compress button. Then a page will appear, in which we should select the URL of the image from a drop down list. Once the compress button is pressed, image will be compressed and the compressed image will be stored in SDCARD.

In the case of Decompression, click on the Decompress Image button. Then a page will appear, in which we should select the URL of the compressed image from a drop down list. Once the decompress button is pressed, image will be decompressed and the decompressed image will be stored in SDCARD.

X. CONCLUSION

In this project, implementation of Adaptive Huffman coding is done in Java. The central contribution of this project is to provide a method to implement Adaptive Huffman coding in compressing image in Android. In the first part of this paper,
detailed description about data compression and the various data compression algorithms are provided. From the comparative study, it is observed that Huffman Coding is the better among all the algorithms. In the first part of this paper, data compression using Huffman Coding is done in Java. The results suggest to improve the compression ratio and it is expected that, Adaptive Huffman algorithm provided better compression ratio than the other data compression algorithms. In the second part of this paper, the size of image is compressed using the Adaptive Huffman coding technique. Thus an app is developed which can compress the image and store it and decompress it when required.

XI. FUTURE WORK

Future work will include the investigation of implementation aspects related to the usage of the proposed simplified algorithm for image compression. Modified Adaptive Huffman encoding algorithm can be used as an extension for future work, since it may effectively combines the advantages of static and adaptive Huffman algorithms to provide effective compression by reducing the number of levels in the binary tree. The implementation of this algorithm may show better compression ratio than Adaptive Huffman algorithm. Since the number of levels is restricted, this algorithm may require less computation than Adaptive Huffman, which is an important requirement for wireless sensor nodes. Moreover, it can be extended to real time application also.

REFERENCES