COMPRESSION LIBRARY FOR PALM OS PLATFORM

By

NEBOJŠA ĆIRIĆ

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003
Copyright 2003

by

Nebojša Ćirić
Dedicated to my family.
ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Douglas Dankel II. It was a great pleasure to work with him and I gained valuable experience under his direction. I would also like to thank Dr. Sanjay Ranka and Dr. Joachim Hammer for being on my committee. Many thanks go to Mr. John Bowers for being a very helpful graduate secretary. He was always there to answer my questions and offer suggestions. Finally I would like to thank my wife Ivana for giving me support when I needed it the most.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Description of the Problem</td>
<td>1</td>
</tr>
<tr>
<td>Overview of CLP</td>
<td>2</td>
</tr>
<tr>
<td>Organization of Thesis</td>
<td>4</td>
</tr>
<tr>
<td>2 COMPRESSION THEORY AND ALGORITHMS</td>
<td>5</td>
</tr>
<tr>
<td>Compression Theory</td>
<td>5</td>
</tr>
<tr>
<td>Static and Adaptive Modeling</td>
<td>6</td>
</tr>
<tr>
<td>Static Modeling</td>
<td>6</td>
</tr>
<tr>
<td>Adaptive Modeling</td>
<td>8</td>
</tr>
<tr>
<td>Lossy and Lossless algorithms</td>
<td>9</td>
</tr>
<tr>
<td>Compression Algorithms</td>
<td>10</td>
</tr>
<tr>
<td>Specialized Algorithms</td>
<td>10</td>
</tr>
<tr>
<td>Run length encoding (RLE)</td>
<td>10</td>
</tr>
<tr>
<td>JPEG and MPEG algorithms</td>
<td>11</td>
</tr>
<tr>
<td>MP3 and WMA algorithms</td>
<td>11</td>
</tr>
<tr>
<td>Generic Algorithms</td>
<td>11</td>
</tr>
<tr>
<td>Dictionary based algorithms</td>
<td>11</td>
</tr>
<tr>
<td>Statistical model based algorithms</td>
<td>12</td>
</tr>
<tr>
<td>3 DESCRIPTION OF THE IMPLEMENTED ALGORITHMS</td>
<td>14</td>
</tr>
<tr>
<td>The Huffman’s Coding Algorithm</td>
<td>14</td>
</tr>
<tr>
<td>Building a Huffman’s Tree</td>
<td>14</td>
</tr>
<tr>
<td>The Compression Procedure</td>
<td>16</td>
</tr>
<tr>
<td>The Decompression Procedure</td>
<td>16</td>
</tr>
<tr>
<td>Summary</td>
<td>16</td>
</tr>
<tr>
<td>The Arithmetic Coding Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>The Arithmetic Coding Procedure</td>
<td>17</td>
</tr>
<tr>
<td>Implementation with integer arithmetic</td>
<td>18</td>
</tr>
<tr>
<td>Summary</td>
<td>20</td>
</tr>
</tbody>
</table>
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1. Input alphabet for Huffman’s coder</td>
<td>15</td>
</tr>
<tr>
<td>3-2. Arithmetic encoding process for alphabet S and sequence a b a EOS.</td>
<td>19</td>
</tr>
<tr>
<td>3-3. Arithmetic decoding process for alphabet S and bit sequence 011012</td>
<td>19</td>
</tr>
<tr>
<td>5-1. Compression results</td>
<td>34</td>
</tr>
<tr>
<td>5-2. Decompression results</td>
<td>34</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1.</td>
<td>Static data compression diagram.</td>
<td>7</td>
</tr>
<tr>
<td>2-2.</td>
<td>Static data decompression diagram.</td>
<td>7</td>
</tr>
<tr>
<td>2-3.</td>
<td>Adaptive compression diagram.</td>
<td>8</td>
</tr>
<tr>
<td>2-4.</td>
<td>Adaptive decompression diagram.</td>
<td>9</td>
</tr>
<tr>
<td>2-5.</td>
<td>Lossy compression example. A) High quality JPG image. B) Low quality JPG image.</td>
<td>10</td>
</tr>
<tr>
<td>3-1.</td>
<td>Huffman’s tree and code words generated from input alphabet S.</td>
<td>15</td>
</tr>
<tr>
<td>4-1.</td>
<td>The UML compression CLP library diagram.</td>
<td>23</td>
</tr>
<tr>
<td>4-2.</td>
<td>The UML bit input/output BitIO class diagram.</td>
<td>24</td>
</tr>
<tr>
<td>4-3.</td>
<td>The UML sequence diagram for the compression process using the StatHuffman class.</td>
<td>27</td>
</tr>
<tr>
<td>4-4.</td>
<td>The UML sequence diagram for the decompression process using the StatHuffman class.</td>
<td>28</td>
</tr>
<tr>
<td>4-5.</td>
<td>The UML sequence diagram for the compression process using the StatAritmetic class.</td>
<td>29</td>
</tr>
<tr>
<td>4-5.</td>
<td>The UML sequence diagram for the decompression process using the StatAritmetic class.</td>
<td>30</td>
</tr>
<tr>
<td>5-1.</td>
<td>Symbol frequency distribution.</td>
<td>33</td>
</tr>
</tbody>
</table>
Abstract of Thesis Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Master of Science

COMPRESSION LIBRARY FOR PALM OS PLATFORM

By

Nebojša Ćirić

May 2003

Chair: Dr. Douglas Dankel II
Major Department: Computer and Information Science and Engineering

This thesis presents the design and implementation of a compression library (CLP) for the Palm Operating System (OS) platform. Data compression is a well-researched field in information theory and there are numerous programs and libraries available for almost every platform and OS in the market.

Porting an existing library from the UNIX or MS Windows OS to the Palm OS is a nontrivial task because of constraints imposed by the handheld platform’s memory size and organization. That is why CLP implemented simple, yet effective algorithms with small memory requirements.

CLP contains two algorithms: the static Huffman compression and the static arithmetic compression. It would be easy to expand the library with new algorithms, if needed, because it was designed and implemented using an object oriented approach.

CLP uses a simple interface, exposing only a few methods to the user, thus decreasing deployment time and increasing productivity.
The library was specially designed with the Palm Database (PDB) format in mind. It allows a user to easily manipulate data in separate records, which is the preferred mode of operation on Palm OS.
CHAPTER 1
INTRODUCTION

This chapter begins with a description of the problem addressed within this thesis, followed by the overview of the compression library for Palm OS platform (CLP) that was developed to solve this problem. The chapter concludes with a summary of the remaining chapters in this thesis.

Description of the Problem

The number of handheld devices is increasing rapidly every year because of constant price drop, increased functionality, and increasing demand. They are used everywhere—from hospitals to supermarkets and IT companies. The largest share, 75%, of the personal data assistants (PDA) market is held by Palm OS based PDAs. Second, a 25% market share is PocketPC PDAs based on MS Windows CE or Linux operating systems. Companies producing Palm PDAs are Palm, Visor, Handera, Sony, and Handspring. Companies producing MS Windows based PocketPC PDAs include Hewlett-Packard, Compaq, and Toshiba, while Sharp produces a Linux based system.

The Palm platform covers a wide range of devices. The oldest or cheapest ones have only 2MB of main memory, a black and white screen, an RS232 serial link to a PC, and a 16MHz Motorola DragonBall processor. The newest and the most expensive ones have up to 16MB of main memory, CompactFlash (CF) or SmartMedia (SD) memory expansion slots, a color screen, a USB link to PC, and a 133MHz Intel StrongArm CPU [1, 2]. They all share “beaming” capability. Beaming is process of transferring data between PDA devices, laptops, or cell phones using the Infrared Data Association.
Protocol (IrDA) for short distances (usually less than 1m). Beaming speed is comparable with an RS232 serial connection speed (i.e., up to 115200 bits/second) [1, 2].

PDAs are mostly used as personal organizers and for reading e-documents, e-mails, and e-books downloaded from a PC. These documents can be several megabytes in length, requiring significant time to download through an RS232 or IrDA connection. Large documents can also decrease the amount of free main memory by 20-30% per document.

Text documents, as opposed to binary files, have good compression properties because of non-random word and character repetition. Using compression can reduce memory consumption and download time, thus reducing battery consumption and improving the overall response time of the system.

The top compression algorithms can reduce the size of text documents by up to 90%, but with the tradeoffs of increased algorithm complexity/processing time and memory consumption. This thesis proposes an algorithm with similar performance for a 16MHz Palm Vx with 4MB of memory or a 133MHz Tungsten device with 16MB of main memory.

**Overview of CLP**

CLP, the compression library for Palm, is a general compression library developed to solve the problem described above. It provides a simple interface for application developers to easily compress and decompress data on PDA devices or a PC. CLP can be effortlessly extended with new compression algorithms because of its modular, object-oriented design. Using interfaces hides implementation complexity from the application and allows library changes and improvements without a need to change the application itself.
A Palm OS device keeps its data in a specific file format known as a Palm Database File (PDB) [1, 2]. This is record based, random access file residing in main memory or on a memory card. The PDB format has a limit of 65535 records per file. Each record can be up to 64KB long and of variable size [1, 2]. The new Palm devices are introducing regular files without this limitations but the huge number of legacy applications slows the transition.

Two general categories of compression techniques are used: adaptive and statistical. In this environment adaptive compression techniques do not yield good results because the size of each record is too small for these methods to effectively gather statistics about data before compressing. As a result, CLP uses only a statistical compression method.

Palm OS devices, especially older ones, have a small amount of dynamic memory (on the order of few kilobytes) that is shared with the TCP/IP stack, global variables, etc. [1, 2] This limits the number and the size of the statistical tables that a compression algorithm can hold in memory at one time. As a result, CLP implements only the $0^{th}$ – order statistical compression methods. But even with all these limitations, good compression ratios and compression speeds are obtainable using the current implementation of CLP.

The CLP library can be used on any device, including PCs, provided that the C++ compiler and Standard Template Library are available. This enables the user to preprocess data on more a powerful platform and just transfer them to the PDA device for later use.
Organization of Thesis

This thesis consists of six chapters. Chapter 2 introduces compression theory and algorithms. Chapter 3 gives an in-depth description of the implemented algorithms. Chapter 4 discusses the implementation details from a programmer’s perspective. Chapter 5 addresses tests and results of compression/decompression capabilities of the CLP library on the basis of test data and sample application. Finally, Chapter 6 presents future work and conclusion.
This chapter introduces compression theory and explains the difference between static and adaptive modeling. Lossy and lossless data compression approaches are explained. Algorithms that are currently used in commercial compression applications or libraries are reviewed.

**Compression Theory**

Compression Theory is tightly related to Information Theory. Information Theory is a branch of mathematics started in 1948 from the work of Claude Shannon at Bell Labs [3]. It deals with various questions about information.

Data compression is interested in information redundancy. Data containing redundant information takes extra bits to store. Eliminating the extra bits reduces data size, hence freeing more memory or communication channel bandwidth.

To find the amount of redundancy in the data, Information Theory introduces the concept of entropy as a measure of how much information is encoded in the data. High entropy identifies that the data set has low redundancy while low entropy points to a highly redundant data set, which is consistent with the thermodynamics’ definition of entropy.

Let \((S, P)\) be some unspecified finite probability space. If symbol/event \(Y \subseteq S\), the self-information or information contained in \(Y\) is \(I(Y) = -\log P(Y) = \log \frac{1}{P(Y)}\). This equation also defines the number of bits needed to encode symbol \(Y\) and its entropy. If the probability of \(Y\) is high then the number of bits needed to encode \(Y\) is low. The
number of bits needed to encode the whole message $M$ is simply a sum of the code
lengths for each symbol contained in the message, or
\[
I(M) = \sum_i C_i I(Y_i) = -\sum_i C_i \log P(Y_i),
\]
where $C_i$ is a count of the symbol $Y_i$ in the
message $M$ [4].

The probability of the symbol depends on the model we choose to describe the data
set. This means that the entropy of a message or symbol is not an absolute, unique value.
As a result, models that predict symbols with high probability are good for a data
compression system.

After data is modeled it is encoded with a proper number of bits. If the entropy of
the symbol was 3.5 bits then that symbol should be encoded with 3.5 bits. Some encoding
schemes (e.g., the Huffman scheme) decrease compression by rounding up number of
bits to boost the processing speed.

**Static and Adaptive Modeling**

There are two approaches to model data: static and adaptive. The static method was
developed first but now is abandoned in favor of the adaptive method. A description of
each of the methods follows [5].

**Static Modeling**

The static model first gathers statistical information about each symbol in a table,
by scanning the data once and counting the symbol frequencies in the data set. The
resulting model is used for data encoding and decoding. For the encoder and decoder to
be compatible they must share the same model, which means that the table has to be
transmitted to both the encoder and the decoder. Data compression and decompression
with the static model are shown in Figure 2-1 and Figure 2-2.
Depending on nature of the data set, the probability of the adjacency of two or more specific symbols in the message can be independent, for binary files, or dependent with high or low values, for text files. The encoding model can try to predict such probabilities, thus increasing the compression ratio. The number of adjacent symbols defines the order of the model. A 0<sup>th</sup>-order model assumes that the symbol position in the message is independent of the position of other symbols. A 1<sup>st</sup>-order model assumes that two adjacent symbols are dependent, etc.

For a 0<sup>th</sup>-order model there is only one table with symbol counts. For the standard ASCII character set, the table would have 256 entries. With the 0<sup>th</sup>-order model each symbol is assumed independent from the other symbols.

As the order of the model increases the number of tables increases. In the case of the 1<sup>st</sup>-order model there are 256 tables with 256 entries each. With the 1<sup>st</sup>-order model some relation between symbols is assumed, for example there is high probability of the
character “u” after the character “q” in English language, but very low probability for “w” after “b.” This approach yields a better compression but the overhead of keeping a larger table is often too expensive.

The requirement of keeping the model table with the data to decode that data is the reason why the static modeling has been practically abandoned in modern compression theory.

**Adaptive Modeling**

The adaptive algorithm does not have to scan the data to generate statistics. Instead, the model is constantly updated as new symbols are read and coded. This is true for both the encoding and decoding processes and means that the table does not have to be saved with the data to be decoded. The compression and decompression models are shown in Figure 2-3 and Figure 2-4.

![Figure 2-3. Adaptive compression diagram.](image-url)
The problem with the adaptive model is that it starts with an empty table, so the compression is very low in the beginning. Most adaptive algorithms adjust to the input stream after a few thousands symbols resulting in a good compression ratio. Also, they are able to adapt to changes in the nature of the input stream, like when the data changes from text to an image.

**Lossy and Lossless algorithms**

Data can be compressed with or without loss of information. Lossy compression is mostly used for drastically reducing size of images and sound files, because the human senses are not very sensitive towards small changes in quality. The JPG format is able to reduce an image size by examining adjacent pixels and making ones that appear similar the same, thus reducing the entropy and increasing the compression. Images A and B in Figure 2-5 are saved using the highest and lowest JPG quality. Note that while image A is twice as large as B, the quality of picture B is not drastically different.

Lossless compression is used for all other types of data where accuracy is mandatory. Some of the most widely used lossless algorithms are described below, in the generic algorithms section.
Compression Algorithms

Compression algorithms can be divided into two groups: generic and specialized. Generic algorithms are able to compress any type of information with good but not perfect results. Specialized algorithms are very good at compressing some types of information, like images or sound, but have poor results for other types of data.

Specialized Algorithms

Some well-known specialized algorithms are reviewed in this section.

Run length encoding (RLE)

Run length encoding algorithm [5] is mostly used with bitmap images (e.g., black and white images) where symbols (pixels) with the same value are often found in contiguous streams. The stream can be then replaced with <Count, Value> pair, thus decreasing the image size. This method must be cleverly implemented to avoid data expansion, which can happen if the streams are short or the symbols alternate.
**JPEG and MPEG algorithms**

The Joint Photographic Experts Group (JPEG) [6] and the Moving Pictures Expert Group (MPEG) committees proposed two lossy algorithms for image and moving picture compression, respectively. These algorithms make two passes over the data. First pass converts the data into the frequency domain, using FFT-like algorithms. Once transformed, the data are “smoothed” by rounding off the peaks, resulting in a loss of information. In the second pass, the data are compressed using one of the lossless algorithms. Compression ratios can be very high with acceptable quality degradation as shown in Figure 2-5.

**MP3 and WMA algorithms**

MP3 and Windows Media Audio (WMA) [7] are lossy algorithms used for audio signal compression. Compression ratios are high with a reduction in size by a factor of 10 or more with low distortion of the audio signal. These algorithms use the fact that the human ear cannot hear some frequencies if they are masked by other frequencies. They eliminate those hidden frequencies, hence reducing the size of the signal. The encoding procedure is highly CPU intensive but decoding is not. This property is one of the reasons why the MP3 format is widely accepted for storing audio files.

**Generic Algorithms**

There are a few generic algorithms currently in use, like the (1) Dictionary, (2) Sliding Window [4, 5], (3) Huffman, and (4) arithmetic coding algorithms. They are all implemented as lossless and adaptive.

**Dictionary based algorithms**

This family of algorithms encodes variable-length strings into fixed length codes, which are called tokens. The most popular algorithm in this group is the Lempel-Ziv-
Welch (LZW) algorithm [3, 4, 8, 9], which is used in almost every commercial compression utility or library. It parses the input stream and for each new phrase that encounters, it adds a <Token, Phrase> pair to the dictionary. The algorithm is fairly simple but its implementation is usually complex because of the dictionary maintenance tradeoffs. If the dictionary becomes full one of three actions occurs: (1) the dictionary could be flushed, which affects compression, (2) the dictionary could be frozen, which affects compression if data changes over time, or (3) the dictionary could be expanded, which could lower the compression ratio because the token size increases.

**Statistical model based algorithms**

This family of algorithms encodes symbols into bit-strings of variable length using a statistical model. These algorithms are based on a Greedy Approach where symbols with the high probabilities are given the shortest codes (bit-strings). The model has to accurately predict the probabilities of symbols to increase the compression.

The Huffman coding algorithm [4, 5, 10] achieves the minimum amount of redundancy possible if the bit-strings are limited to an integer size. As a result, it is not an optimal method, just a good approximation. It is a very simple and fast algorithm, suitable for devices with slow CPUs. Memory consumption depends on the order of the model used.

The arithmetic coding algorithm [4, 5, 11, 12] does not produce a single code for each symbol. It produces a code for the whole message. Each symbol added to the message incrementally modifies the output code. This approach allows a symbol to be encoded with a fractional number of bits instead of an integer number, thus exactly representing entropy of the symbol. In theory, arithmetic coding is optimal for a given model, but a real-world implementation has to make some tradeoffs related to floating
point or integer arithmetic, thus decreasing the compression ratio. The arithmetic coding
algorithm needs a powerful CPU for both the encoding and decoding process, which
makes it less desirable on PDA devices than the Huffman coding algorithm.

Both of these algorithms are covered in depth in next chapter.
CHAPTER 3
DESCRIPTION OF THE IMPLEMENTED ALGORITHMS

This chapter describes the two algorithms used to implement CLP. These are the static Huffman and Arithmetic algorithms. As mentioned in the previous chapter, both belong to a class of Greedy algorithms. Each tries to assign the shortest bit-sequence to the symbol with the highest frequency or the lowest entropy. We start by examining the Huffman Algorithm.

The Huffman’s Coding Algorithm

Assume an input alphabet, s_1, s_2 ... s_m, with corresponding frequencies, \( f_1 \geq f_2 \geq \ldots \geq f_m \), for each symbol. To compress or decompress data using Huffman coding, symbols and frequencies must be transformed into code words. This can be accomplished by building a Huffman’s tree.

Building a Huffman’s Tree

Each leaf node in the Huffman’s tree is assigned one of the symbols from the input alphabet. Symbols s_m and s_{m-1} become siblings in the tree, with a common parent node, S_p, and frequency \( f_m + f_{m-1} \). In each iteration of the algorithm, the two nodes from the set of leaf and intermediate nodes with the least weights that have not yet been used as siblings are paired as siblings and assigned to a parent node whose weight becomes the sum of the siblings’ weights. The algorithm stops when the last two siblings are paired. The final parent, with a weight 1, is the root of the tree.
Table 3-1. Input alphabet for Huffman’s coder

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>1/3</td>
</tr>
<tr>
<td>S₂</td>
<td>1/5</td>
</tr>
<tr>
<td>S₃</td>
<td>1/6</td>
</tr>
<tr>
<td>S₄</td>
<td>1/10</td>
</tr>
<tr>
<td>S₅</td>
<td>1/10</td>
</tr>
<tr>
<td>S₆</td>
<td>1/10</td>
</tr>
</tbody>
</table>

To illustrate this process consider the following example. In Figure 3-1, the input alphabet S, defined in Table 3-1, is transformed into a Huffman’s tree. The average code word length is defined as \( \bar{l} = \sum_{i=1}^{m} f_i w_i \), where \( f_i \) is frequency and \( w_i \) is code length of symbol \( s_i \). For the input alphabet \( S \), \( \bar{l} \) has decreased from 3 to 2.47. In an average case, for this simple alphabet, the compression ratio is 17.7%.

Figure 3-1. Huffman’s tree and code words generated from input alphabet S.
The Compression Procedure

The compression algorithm is fairly simple. It consists of four steps:

1. Scan the input stream and obtain the frequencies for each symbol. Save that information.
2. Build the Huffman’s tree and extract the code words.
3. Input a symbol from the stream and output its code.
4. Repeat step 3 until the end of stream.

The Decompression Procedure

The decompression algorithm is also simple, consisting of five steps:

1. Build the Huffman’s tree from the saved information.
2. Move the pointer to the root node.
3. For each input bit follow the corresponding label down the tree, until a leaf node is reached.
4. Output the symbol at that leaf node.
5. Repeat the algorithm from step 2, until the end of the stream.

All code words produced from the Huffman’s tree have the property that no code word is the prefix of another code word. The decompression procedure uses this property to traverse tree from the root node to the leaf and to find the corresponding symbol without having to resolve any ambiguities.

Summary

For a given model, the Huffman coding is optimal among the probabilistic methods that replace source symbols with an integer number of bits. However, it is not optimal in the sense of entropy. As an example, consider an alphabet consisting of two symbols. Regardless of the probabilities, the Huffman coder will assign a single bit to each of symbols, giving no compression.
The Arithmetic Coding Algorithm

Assume an input alphabet, \(s_1, s_2 \ldots s_m\), with corresponding frequencies, \(f_1 \geq f_2 \geq \ldots \geq f_m\), for each symbol. Using the arithmetic coding algorithm, the entire source text, composed of symbols from the alphabet \(S\), is assigned a code word determined by the process described below.

Each source symbol, \(s_i\), is assigned a subinterval, \(A(i)\), in the interval \([0, 1)\). The subintervals, \(A(1), A(2) \ldots A(m)\), are disjoint subintervals of \([0, 1)\). The length of \(A(i)\) is proportional to \(f_i\).

Having determined the interval \(A\), the arithmetic coder chooses a number \(r \in A\) and represents the source text with some finite segment of the binary expansion of \(r\).

The smaller the interval \(A(i)\) is, the farther out the decoder has to go with binary expansion to decode the source symbol. For this reason, symbols with a higher frequency or longer interval have shorter binary expansions, resulting in better compressed.

The Arithmetic Coding Procedure

The arithmetic coding algorithm is more complex than the Huffman coding algorithm, both in the number of steps and in the calculation complexity. The steps are:

1. A current interval \([L, H]\) is initialized as \([0, 1)\) and is maintained at each step. An underflow count is initialized at 0 and is maintained to the end of the file.

2. (Underflow condition) If the current interval satisfies \(\frac{1}{4} \leq L \leq \frac{1}{4} \leq H \leq \frac{1}{2}\), then expand it to \(\left[2(L - \frac{1}{4}), 2(H - \frac{1}{4})\right]\) and add 1 to the underflow count.

3. (Shift condition) If \([L, H] \subseteq \left[0, \frac{1}{2}\right]\), then output 0 and any pending underflow bits (which are all 1), and expand the current interval to \([2L, 2H]\). If \([L, H] \subseteq \left[\frac{1}{2}, 1\right]\), then output 1 and any pending underflow bits (which are all 0), and expand the current interval to \([2L - 1, 2H - 1]\). In either case, reset the underflow count to 0.

4. If none of the conditions in steps 2 or 3 hold, then divide the current interval into disjoint subintervals \([L_i, L_{i+1})\) corresponding to the symbol \(s_i \in S\), with lengths
determined by the probabilities. If $s_i$ is the next source letter, assign $L \leftarrow L_i$ and $H \leftarrow L_{i+1}$.

5. Repeat steps 2-4 until the end of the stream and none of the conditions in steps 2 or 3 hold. At this stage, the final interval satisfies $L < \frac{1}{4} < \frac{1}{2} < H$ or $L < \frac{1}{2} < \frac{3}{4} < H$, and sequence “01” is output if the first condition holds and “10” is output otherwise. Any leftover underflow bits are output after these bits.

**Implementation with integer arithmetic**

The previously described algorithm cannot be directly applied in practice for two reasons. First, floating point arithmetic is much slower than integer arithmetic on any computer platform. Second, the result $r$ can become long, hence demanding a very high precision that is not available with current technology.

To solve these problems, the interval $[0, 1)$ can be replaced with $[0, M)$, where $M$ is a positive integer with a value of at least $4(|S|-2)$. The value of $M$ is usually chosen as some power of 2, which can improve integer arithmetic performance. Each subinterval in $A$ is expanded, which corresponds to a value of $M$ and a subinterval length. Also, $L$ and $H$ values can be represented as 0x0000… and 0xFFFF… in the initial case, where the ellipsis means that 0s or 1s can be shifted in when needed, thus enabling 16 or 32-bit register arithmetic.

The interval $[0, M)$ is just an approximation of the interval $[0, 1)$, with imprecise symbol frequencies, which leads to lower compression ratios. But using integer arithmetic drastically decreases the processing speed.

To illustrate encoding and decoding process using the integer approximation algorithm, consider the following example. The source alphabet is $S = \{a, b, EOS\}$, the frequencies for each symbol are $f_a=6/10, f_b=3/10$, and $f_{EOS}=1/10$, and the cumulative counts are $C_a=6, C_b=9$ and $C_{EOS}=10$. The input sequence is: a b a EOS. To obtain an
interesting, non-trivial, example, \( M \) should be \( 2^4 \). The lowest possible value for \( M \) can be obtained from following inequality \( M \geq 4(|S| - 2) \).

The EOS symbol is used to terminate the input message. Adding an artificial symbol to the alphabet changes the real symbol frequencies, thus lowering the compression ratio, but by using it, the message size does not have to be appended to the message and incremental transmission becomes possible.

The encoding and decoding processes are given in Tables 3-2 and 3-3.

Table 3-2. Arithmetic encoding process for alphabet \( S \) and sequence a b a EOS.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Current interval</th>
<th>Subintervals</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>[0, 16)</td>
<td>[0, 9)</td>
<td>[9, 14)</td>
</tr>
<tr>
<td>a</td>
<td>[0, 9)</td>
<td>[0, 5)</td>
<td>[5, 8)</td>
</tr>
<tr>
<td>b</td>
<td>[5, 8)</td>
<td>Expand ( x \rightarrow 2x ), where ( x ) is current interval</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>[10, 16)</td>
<td>Expand ( x \rightarrow 2(x-M/2) )</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>[4, 16)</td>
<td>[4, 11)</td>
<td>[11, 14)</td>
</tr>
<tr>
<td>a</td>
<td>[4, 11)</td>
<td>Expand ( x \rightarrow 2(x-M/4) )</td>
<td>Underflow</td>
</tr>
<tr>
<td>EOS</td>
<td>[12, 14)</td>
<td>Expand ( x \rightarrow 2(x-M/2) )</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>[8, 12)</td>
<td>Expand ( x \rightarrow 2(x-M/2) )</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>[0, 8)</td>
<td>Expand ( x \rightarrow 2x )</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>[0, 16)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3-3. Arithmetic decoding process for alphabet \( S \) and bit sequence \( 011012 \).

<table>
<thead>
<tr>
<th>Value</th>
<th>Current interval</th>
<th>Subintervals</th>
<th>Output symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0110_2 )=6</td>
<td>[0, 16)</td>
<td>[0, 9)</td>
<td>[9, 14)</td>
</tr>
<tr>
<td></td>
<td>[0, 9)</td>
<td>[0, 5)</td>
<td>[5, 8)</td>
</tr>
<tr>
<td></td>
<td>[5, 8)</td>
<td>Expand ( x \rightarrow 2x )</td>
<td>a</td>
</tr>
<tr>
<td>( 1101_2 )=13</td>
<td>[10, 16)</td>
<td></td>
<td>Expand ( x \rightarrow 2(x-M/2) )</td>
</tr>
<tr>
<td>( 1010_2 )=10</td>
<td>[4, 16)</td>
<td>[4, 11)</td>
<td>[11, 14)</td>
</tr>
<tr>
<td></td>
<td>[4, 11)</td>
<td></td>
<td>Expand ( x \rightarrow 2(x-M/4) )</td>
</tr>
<tr>
<td>( 1100_2 )=12</td>
<td>[0, 14)</td>
<td>[0, 8)</td>
<td>[8, 12)</td>
</tr>
</tbody>
</table>

For a given model, the arithmetic coding algorithm is optimal and outperforms the Huffman method. It is important to note that implementations of the arithmetic coding
algorithm are less than optimal, due to integer arithmetic and other compromises. Since
the Huffman coding algorithm is nearly optimal in many cases, the choice between the
methods is not as simple as the theory suggests.

**Summary**

The two algorithms used for development of CLP are described in this chapter. An
in-depth theoretical analysis [4] and detailed implementation [5] can be found in the
literature. The Huffman’s coding algorithm is lightweight and fast, but it is non-optimal.
The arithmetic coding algorithm is CPU intensive, but in its “pure” form, it is optimal. In
Chapter 5 compares the results of both algorithms and identifies which one is better for
PDA application.
CHAPTER 4
CLP LIBRARY IMPLEMENTATION

This chapter discusses the implementation details from a programmer’s perspective. These include (1) the class hierarchy, (2) the interface exposed by the compression library, (3) the proper call sequence to the class methods, (4) error handling, and (5) instructions on how to include the CLP library into a Palm OS application.

Class Hierarchy

The CLP library consists of two separate sets of classes. One consists of the main compression hierarchy, while the other holds the simple bit-stream class.

Compression Class Hierarchy

Different compression algorithms use different techniques for compressing data, but they all have a similar interface [13] to the application that uses them. It is a common practice to enforce such behavior by using a common virtual base class, in the case of the CLP library that base class is the BaseCompress class.

Static and adaptive compression algorithms update the coding model in a different way. The static compressor gathers statistical information before compression, while the adaptive does that during the compression phase. As a result, there are two new classes inherited from the BaseCompress class: StatisticCompress and AdaptiveCompress. Both are virtual classes with the AdaptiveCompress class not being implemented.

The Huffman and arithmetic compression classes are inherited from the StatisticCompress class. While they share the same interface, their internal
implementations are different, reflecting the differences in their corresponding algorithms.

A modular OO, C++ [14] approach makes changes and additions to the library easy. If a new algorithm is needed, then its class can be inherited from StatisticCompress or AdaptiveCompress, without changing the existing code.

Each class is declared in a separate header file, and its implementation is placed in a corresponding .cpp file, thus reducing the size of each module.

The complete UML [15] compression class diagram is given in Figure 4-1.

**BitIO Class Hierarchy**

The BitIO class is a helper class. It handles single or multiple bit stream operations. The OS file or memory functions are developed for byte access, but for compression purposes, the ability to store an arbitrarily sized bit sequence in memory is needed.

The UML representation of the BitIO class is given in Figure 4-2.

**CLP Library Interface**

The compression library should provide a simple interface, enabling the user to easily send data to the library and receive responding results from it. Each class in the hierarchy either expands the interface or implements its methods.
Figure 4-1. The UML compression CLP library diagram
Figure 4-2. The UML bit input/output BitIO class diagram.

**BaseCompress Class Interface**

The BaseCompress class provides the behaviors detailed below by exposing the following methods in its interface:

- **Initialize** – Initializes the compression/decompression engine for a given class. This is a pure virtual method [14].

- **Compress/Decompress** – Sends data to the CLP library for compression/decompression. These are pure virtual methods.

- **GetResult** – Extracts the result from the CLP library. This method is implemented in the BaseCompress class.

- **FlushBuffer** – Flushes the internal buffers and prepares the CLP library for the next compression sequence. This method is implemented in the BaseCompress class.

**StatisticCompress Class Interface**

The StatisticCompress class expands the base interface with methods operating on the statistical data and resulting header:

- **Vectorizer** – Returns the vector with statistical information that was generated by Initialize call. This is a pure virtual method.
- GetHeaderSize – Returns the size of the header, so that the application can allocate enough memory to save the header. This is a pure virtual method.

- SetHeader – Data can be preprocessed or even compressed on the PC. The header information generated through preprocessing can be read from the file and used for compression/decompression on Palm platform thus decreasing the overhead and increasing speed. This is a pure virtual method.

**StatHuffman and StatArithmetic Classes**

The StatHuffman and StatArithmetic classes implement the interface declared by the StatisticCompress parent class. They implement the corresponding Huffman and Arithmetic algorithms.

**BitIO Class Interface**

The BitIO class implements a simple interface that allows the user to read/write a sequence of bits from/to a stream.

The proper call sequence is outlined below:

1. OpenIO – Opens a new bit I/O stream connection.
2. Input/Output Bit/Bits – Inputs/outputs the bit sequence from/to the stream.
3. FlushIO – Flushes the bits remaining in the buffer to the stream.
5. CloseIO – Closes the I/O stream.

Step 2 is repeated as long as there are bits to input or output.

**Methods Interaction**

To properly use the CLP library, the application must follow this method’s calling sequence for both the compression and decompression process:

1. Initialize

2. Set/GetHeader – Set the header if is needed later, or get it if it was generated before.
3. Compress/Decompress – This call passes data to the library and returns the size of the result.

4. GetResult

5. FlushBuffer

   Steps 3-5 are repeated for each record that the application intends to compress/decompress.

   A detailed calling sequence for the public and private functions, for both types of compression/decompression, is given as a UML sequence diagrams in Figures 4-3, 4-4, 4-5, and 4-6.

**Error Handling**

The CLP library uses the C++ exception handling mechanism [14] to handle errors. There are only three types of exceptions that CLP throws:

- **out_of_range** is thrown if the array index is out of range.

- **length_error** is thrown if the array resize operation fails (i.e., there is insufficient memory).

- **unspecified exception** is thrown if an unspecified exception happens (i.e., some system exception).

   All three types of exceptions are inherited from the C++ Standard Template Library (STL). The CLP library uses the STL’s vector container type [14, 16] to implement the dynamic arrays used for holding the results generated during compression and decompression.

   A simple tester application shows the correct way of handling exceptions thrown by the CLP library in the application that uses it.
Figure 4-3. The UML sequence diagram for the compression process using the StatHuffman class.
Figure 4-4. The UML sequence diagram for the decompression process using the StatHuffman class.
Figure 4-5. The UML sequence diagram for the compression process using the StatAritmetric class.
Figure 4-5. The UML sequence diagram for the decompression process using the StatArithmetic class.

**How to Deploy the CLP Library**

To use the CLP library, the application programmer must (1) add the library file to the path accessible by the linker and (2) include StatHuffman.h or/and StatArithmetic.h files into his project. Both the StatHuffman and StatArithmetic header files will automatically include the necessary underlying header files and libraries into the project.
CHAPTER 5
COMPRESSION RESULTS

This chapter presents a summary of the CLP library’s performance on a Palm Pilot device based on a sample program using the library to compress and decompress drug reference data [5]. The compression ratios and speeds are measured for both the Huffman and Arithmetic coding algorithms. A comparison is made to determine which algorithm is more suitable for PDA use.

Test Environment

The following is a description of the sample application, scripts, utilities, source data, and testing procedure used.

Source Data

The initial data set was a text file containing ASCII strings delimited with new line characters. The size of the text file is approximately 2300 strings or 840KB.

Using the first PHP script given in Appendix, each string from the text file was inserted into a separate record in the PDB database.

The CLP implementation on the PC is used to generate a header with statistical data for the compression and decompression procedures. The second PHP script, also given in Appendix, is used to transfer binary information from the header file to the PDB database record on the PC.

All generated PDB files and the sample application are loaded into the Palm OS device emulator program (POSE).
Sample Application

A sample application was written in the C++ language and was linked with the CLP library for Palm OS. It first compresses the input PDB database using the Huffman and Arithmetic coding algorithms and logs compression times with an external profiler application. The compression results are written into new database files, one for each type of compression. The resulting databases are decompressed using both algorithms, and decompression times are logged with the profiler.

Scripts

Scripts for converting data to the Palm PDB format [1] were written using a php-pdb [17] module. PHP script language interpreters are available for almost all OS platforms, so these scripts are highly portable [17].

Utilities

Two utility programs are used for testing: POSE and Palm Reporter. They are both part of the Palm OS Software Development Kit (SDK) [1].

POSE is able to emulate any Palm OS device, if a ROM image for that device is available. It emulates the real speed of the device, so measurements taken on it are roughly equal to measurements taken on the real device.

Palm Reporter is a stand-alone program able to connect to POSE and receive log messages. It is used to obtain compression and decompression speed information from the sample program running on POSE.
Test Results

Symbol Distribution

The header files were generated from the source data by the CLP library for a PC. Both the Huffman and Arithmetic coder headers are identical with the symbol frequencies shown in Figure 5-1

![Character frequency distribution](image)

Figure 5-1. Symbol frequency distribution.

The data distribution in the graph meets expectations, because the source text consists of ASCII strings in the English language, thus the SPACE symbol, digits, and characters between a-z and A-Z have the highest frequency.

Compression Results

Palm devices use FLASH memory to store all databases and applications. FLASH memory is faster for reading than it is for writing (by a factor of 10). That is why there are two sets of results for the compression test. One includes writing results to storage memory and the other is without. They are both shown in Table 5-1.
Table 5-1. Compression results

<table>
<thead>
<tr>
<th>Compression method</th>
<th>Source data (in bytes)</th>
<th>Comp. data (in bytes)</th>
<th>Comp. ratio</th>
<th>Comp. time (write)</th>
<th>Comp. time (w/o write)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>878,090</td>
<td>878,090</td>
<td>0%</td>
<td>45s</td>
<td>2s</td>
</tr>
<tr>
<td>Huffman</td>
<td>878,090</td>
<td>544,136</td>
<td>38.03%</td>
<td>75s</td>
<td>62s</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>878,090</td>
<td>546,895</td>
<td>37.72</td>
<td>99s</td>
<td>83s</td>
</tr>
</tbody>
</table>

The compression time for both algorithms is comparable with the simple memory access and is around 0.02s/record.

Both algorithms have similar compression ratios. The expected result was that the arithmetic coder algorithm would have a better compression ratio than the Huffman coder algorithm, but in this case that is not true. There are many factors that can influence the compression results, with the main one being the integer arithmetic approximation.

**Decompression Results**

Decompression results are shown in Table 5-2.

Table 5-2. Decompression results

<table>
<thead>
<tr>
<th>Decompression method</th>
<th>Comp. data (in bytes)</th>
<th>Decomp. data (in bytes)</th>
<th>Decomp. time (write)</th>
<th>Decomp. time (w/o write)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>878,090</td>
<td>878,090</td>
<td>45s</td>
<td>2s</td>
</tr>
<tr>
<td>Huffman</td>
<td>544,136</td>
<td>878,090</td>
<td>128s</td>
<td>110s</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>546,895</td>
<td>878,090</td>
<td>812s</td>
<td>799s</td>
</tr>
</tbody>
</table>

Decompression is slightly slower than compression in the case of the Huffman coder algorithm. This result is expected, because the decoder must traverse the Huffman tree to lookup a symbol, which is a slower operation than finding a symbol-code pair in the compression procedure.

In the case of the arithmetic coder algorithm, the decompression is much slower than the compression, by a factor of 10. This is due to the more complex decompression algorithm. Also, some optimization could help in narrowing the gap.
Discussion

Users of PDA devices expect PDA applications to have a quick response time and a small size. The test results clearly show that the Huffman coder algorithm can provide both, while the arithmetic coder algorithm falls short when speed is taken into account.
This chapter reviews the key concepts addressed in this thesis. The important issues of the CLP library are discussed, and the future improvements over the current system are addressed as well.

**Overview of the CLP Library**

CLP is a simple, easily expandable compression library for PDA devices running Palm OS. It implements well-known algorithms that are used in most commercial compression applications because of their good performance.

The compression algorithms’ complex implementation is well hidden behind a simple interface exposed by the CLP library. This interface enables the application programmer to effortlessly deploy the CLP library.

New compression algorithms can be added to the CLP library without changing the existing library code. The C++ class hierarchy enforces the existing interface onto the new implementations, hence demanding only minor changes in the application code.

**Future CLP Library Improvements**

The current implementation of the CLP library leaves much room for improvement.

**Static vs. Shared Libraries**

The CLP library is implemented as a statically linked library (SLL). A copy of SLL is added to each application that uses it. Every application on the same PDA device must have its own separate copy of the library. This clearly wastes memory, which is not a good practice on a PDA device.
Shared or dynamically linked libraries (DLL) offer a fix to this problem by keeping the library in a separate file, which can be loaded on demand by the application needing it. Then, only one copy of the library is kept in the system thus reducing memory requirements. DLLs are not a perfect solution because:

1. They are highly system dependant.
2. They must provide backward compatibility with previous versions.

If condition 1 is acceptable and if the library is well implemented then a DLL approach should be chosen.

Compression libraries are usually in high demand in the system, thus they are shared by many applications at the same time. They are also system specific because they use low-level system properties to increase processing speed, hence the CLP library should be implemented as a DLL in its next version.

**C++ vs. C Language**

The C++ language offers a rich set of libraries and language elements (i.e., strict type checking, exception handling, strings, dynamic arrays, and classes) to the developer. These properties increase productivity by enabling the programmer to focus more on the problem than on its implementation. Also programs developed in an OO language can be easily expanded by adding new classes and reusing old ones. The problem with the C++ language is that this flexibility and power increases memory consumption and decreases speed, which can hurt performance on PDA devices.

The C language is the language of choice for embedded systems programmers, because of its small memory footprint and tight connection with the underlying system. It does not offer a rich set of language elements like C++, thus forcing the programmer to spend more time in the implementation and testing phases.
The current size of the CLP library is 145KB. If it was developed in the C language its size would decrease to approximately 30-40KB, and its dynamic memory consumption would be much smaller. This decrease in size happens because the C language does not use the STL library, which is quite large, and there is no object initialization overhead for virtual classes and methods.

With some decrease in flexibility and expandability the CLP library can be ported to the C language, hence reducing its size and memory footprint.

**1\textsuperscript{st}-order and Adaptive Algorithms**

Current, 0\textsuperscript{th}-order algorithms can be replaced with 1\textsuperscript{st}-order algorithms to increase compression ratios, though the problem with holding large statistical tables in memory must be solved first.

If record based compression/decompression is not needed then adaptive algorithms can be used. They would eliminate the need for header storage space and data preprocessing.
The source code for the CLP and the test application is not included in the thesis.

Only PHP scripts mentioned in the thesis are included.

```php
/*
 * header.php
 *
 * Makes test header PDB for compression
 *
 * It uses php-pdp module [18] to manipulate PDB files
 */

// Turn on all error reporting
ini_set('error_reporting', E_ALL);

$fp = fopen("SHuffman.bin", "rb");
for ($i=0; $i<256; $i++)
{
    $symbol = fread($fp, 1);
    echo ord($symbol);
    echo ",";
}
?>
```
<?PHP
/*
 * pdbbuild.php
 *
 * Make Test DB for compression
 *
 * It uses php-pdp module [18] to manipulate PDB files
 */

// Turn on all error reporting
ini_set('error_reporting', E_ALL);

include "./php-pdb.inc"

// Get data first
echo "Get data first and put it into SourceDataDB\n";
$fp = fopen("compressionTest.txt","rt");
if (! $fp)
{
    echo "Can't open compressionTest.txt\n";
    exit;
}

$PDBFile = new PalmDB('DATA','STRT','SourceDataDB');

$counter = 1;
$test = 0;

while(!feof($fp))
{
    $string = fgets($fp);
    $PDBFile->AppendString($string);
    $counter = $counter+1;
    $test = $PDBFile->GoToRecord($counter);
}

echo "Test = ", $test, "\n";

$PCPDBFile = fopen("SourceDataDB.pdb","wb");
if (! $PCPDBFile)
{
    echo "Can't create SourceDataDB.pdb\n";
    exit;
}

$PDBFile->WriteToFile($PCPDBFile);
fclose($PCPDBFile);

fclose($fp);

// Get first header (for Static Huffman Compression)
$fp = fopen("SHuffman.bin","rb");
if (! $fp)
{
    echo "Can't open SHuffman.bin\n";
    exit;
}

$Header = fread($fp, 256);
$PDBFile->AppendString($Header);

$PCPDBFile = fopen("SHuffmanDB.pdb","wb");
if (! $PCPDBFile)
{
    echo "Can't create SHuffmanDB.pdb\n";
    exit;
}
$PDBFile->WriteToFile($PCPDBFile);
fclose($PCPDBFile);

fclose($fp);

// Get second header (for Static Arithmetic Compression)
$fp = fopen("SArithmetic.bin","rb");
if (! $fp)
{
    echo "Can't open SArithmetic.bin\n";
    exit;
}

$Header = fread($fp, 256);
$PDBFile->AppendString($Header);

$PCPDBFile = fopen("SArithmeticDB.pdb","wb");
if (!$PCPDBFile)
{
    echo "Can't create SArithmeticDB.pdb\n";
    exit;
}
$PDBFile->WriteToFile($PCPDBFile);
fclose($PCPDBFile);

fclose($fp);
?>
REFERENCES


BIOGRAPHICAL SKETCH

Nebojša Ćirić was born in Belgrade, Republic of Serbia, Yugoslavia. He received his Bachelor of Science degree in computer science and engineering from the School of Electrical Engineering, Belgrade, Yugoslavia, in 1998. He worked at the Institute “Mihajlo Pupin” for seven months on his BS project as a programmer. In August 1999, he moved to Ljubljana, Slovenia, to work at the Hermes SoftLab, one of the largest software development companies in the country. He quit his job in Slovenia a year later to be with his wife in Gainesville, FL. He was accepted into the Computer and Information Science and Engineering Department at the University of Florida in August 2001. His research interests include object-oriented software development, artificial intelligence, and pattern recognition.