Master Thesis
Computer Science
Thesis no: MCS-2004:27
January 2005

A study in compression algorithms

Mattias Håkansson Sjöstrand

Department of
Interaction and System Design
School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the Department of Interaction and System Design, School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Mattias Håkansson Sjöstrand
Address: Holmgatan 12 a
371 38 Karlskrona
Sweden
E-mail: mahasj@affv.nu

University advisor(s):
Göran Fries
Department of Interaction and System Design

Department of
Interaction and System Design
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 102 45

To my mother

ABSTRACT

Compression algorithms can be used everywhere. For example, when you look at a DVD movie a lossy algorithm is used, both for picture and sound. If you want to do a backup of your data, you might be using a lossless algorithm. This thesis will explain how many of the more common lossless compression algorithms work. During the work of this thesis I also developed a new lossless compression algorithm. I compared this new algorithm to the more common algorithms by testing it on five different types of files. The result that I got was that the new algorithm was comparable to the other algorithms when comparing the compression ratio, and in some cases it also performed better than the others.

Keywords: compression algorithms, probability, dictionary, BCCBT

CONTENTS

ABSTRACT ..... **I**

CONTENTS ..... **II**

1 INTRODUCTION **1**

2 MODELS **3**

3 LOSSLESS COMPRESSION TECHNIQUES **6**

3.1 PREDICTIVE TECHNIQUES 6
3.1.1 RLE *6*
3.2 PROBABILITY TECHNIQUES 7
3.2.1 Huffman coding *7*
3.2.2 Arithmetic coding *11*
3.3 DICTIONARY TECHNIQUES 13
3.3.1 LZ77 ..... *13*
3.3.2 LZ78 ..... *15*
3.3.3 LZW *16*
3.4 TRANSFORMING THE DATA 18
3.4.1 MTF *19*
3.4.2 BWT *20*
3.5 BCCBT 22
3.6 ADAPTIVE BCCBT 28

4 LOSSY COMPRESSION TECHNIQUES **30**

4.1 SCALAR QUANTIZATION 30
4.2 VECTOR QUANTIZATION 31

5 USING LOSSY ALGORITHMS IN LOSSLESS COMPRESSION **34**

6 RESULTS **35**

7 CONCLUSION **49**

A PSEUDOCODE OF THE BCCBT ALGORITHM **50**

B PSEUDOCODE OF THE ADAPTIVE BCCBT ALGORITHM **51**

REFERENCES **52**

BOOKS 52
WEB ADDRESSES 52

1 INTRODUCTION

The purpose of this thesis is to examine how well different compression algorithms work and see if it is possible to change or combine them to achieve higher compression ratio. I will explain all algorithms in detail that I am using for the tests, so no previous knowledge about compression algorithms is necessary. During the work of this thesis I also developed a new compression algorithm, which I have chosen to call BCCBT, and compare this new algorithm against some of the more common ones.

In chapter 2 I will discuss different ways on how we can look at the data. If we want to create an algorithm for a special kind of data, for example pictures, we need to know the properties of the data in order to create an algorithm that fits our needs. Chapter 3 will be a discussion on how the different lossless algorithms work. The reason why I have chosen to test those algorithms that are explained in this chapter, is because they can be used on all kinds of data, that is, they are not specialized on certain kinds of data, although they will treat the data differently and thereby achieving different results. In this chapter we will also look at the BCCBT algorithm and the Adaptive BCCBT algorithm. The next chapter, chapter 4, I will briefly explain how lossy algorithms work. The reason for this is that you need to know a little about lossy algorithms when I am about to explain how we can use lossy algorithms in lossless compression, which I will discuss in chapter 5. Chapter 6 will show the results that I got when I compared the different algorithms to each other. I will also explain some of the results shown in the tables and figures. The last chapter, chapter 7, will be the conclusion of everything. There are also two appendices, appendix A and appendix B, that will show the pseudocode of the BCCBT algorithm and the Adaptive BCCBT algorithm respectively.

So, what is compression all about? Well, simply speaking it is about making data smaller. By doing this, we can store more data to a disc or save bandwidth over a network among other things. This leads to that we can save money. If we can store more data on a disc by compressing the data, we do not need to upgrade as often as if we would have been forced to do without using compression. With bandwidth we can lower the cost, if we pay for the number of bits sent over the network, by compressing the data before sending it over the network. This in return can lead to that we are able to let more users access our server if we are running one, without the need for more fiber optics, and still be able to keep the bandwidth cost at the same level.

This may sound great, so why is not it used more often than it really is? The problem with many compression algorithms is that they are very CPU demanding[1]. If we have to compress the data in real-time then compression might be a bad idea. Let us say that a company is running a web server that is using a database that is constantly changing. Then the server has to compress this database every time some client is requesting the whole database. This may lead to a drastic decrease in transmission rate if there are many users trying to get access to the database at the same time, and in the worst case the server might have no other choice than to disconnect some of the users. On the other hand, if time is not a concern, then compression is the ideal solution to save for example space or bandwidth. In this thesis, we will only be focusing on how well the different compression algorithms compress, not how time and memory efficient they are.

Unfortunately there does not exist a single compression algorithm that is excellent on all kind of data. Instead there are a number of different algorithms, each specialized on some kind of data. For example, the Huffman algorithm is based on probabilities on each symbol in the data, while the LZ*-family (Liv Zempel) recognizes patterns in the data to be able to make, for example, the file smaller. But more on this in later chapters.

[1] http://compression.ca/

To make things even more complicated, compression algorithms can be divided into two groups: lossless ones, and lossy ones. Lossless compression is algorithms that do not change the data, that is, when one decompresses it, it is identical to the original data. This makes lossless algorithms best suited for documents, programs and other types of data that needs to be in its original form. Lossy algorithms do however change the data. So when one decompresses it, there are some differences between the decompressed data and the original data. The reason for changing the data before compressing it, is because one can achieve a higher compression ratio than if one had not changed it. This is why lossy algorithms are mostly used to compress pictures and sounds. Unfortunately, lossy algorithms are for the most part much more complex than lossless algorithms, and therefore this thesis will mostly focus on lossless algorithms and their properties.

2 MODELS

When you want to compress the data you have to decide what compression algorithm you should use. This may sound trivial, but it is actually harder than you might first think. Let us say that we have an alphabet with the letters *a*, *b*, *c*, *d* and the numbers 0-9. If we now have a sequence like *aaaaaaaaabbc* (twelve symbols) we could encode this as 9*a*2*b*1*c* (six symbols). In this case we have saved six symbols by compressing it in this way. On the other hand, if we have the sequence *adcbadcbadcbadcbadcbadcbadcbadcb* (32 symbols) then it is a bad idea to compress this sequence in the same way as we did with the one before. If we would, then we would end up with 64 symbols instead, resulting in an expansion of the data. But if we look closer on the sequence we can see that it contains the pattern *adcb*. So we could have written that sequence as 8*adcb* (five symbols) instead, saving 27 symbols, which is very good. These two examples are using a compression scheme called RLE. We will discuss this compression technique in more detail in the coming chapter.

Now let us take a more difficult sequence with the alphabet *a*, *b*, *c* and *d*. Since we have four different symbols we will need two bits $\left(2^2 = 4\right)$ for each symbol to be able to distinguish them from each other, see table below.

Symbol	Code
a	00
b	01
c	10
d	11

Let us look at the sequence *abacabadabacbabcab* (18 symbols, 36 bits). If we were to use the RLE technique for this sequence we would notice that there are no patterns, which in this case would result in an expansion instead. So for obvious reasons RLE is not ideal solution for this problem. But if we take the frequency of each symbol we will get the result shown in the table below.

Symbol	Frequency	Probability
a	8	0.44
b	6	0.33
c	3	0.17
d	1	0.06

As we can see from the table, *a* and *b* occur many times in the sequence, while the frequency of *c* and *d* is very low. Maybe we can change the code for each symbol so we can compress the sequence? And as a matter of fact, we can.

Symbol	Code
a	0
b	10
c	110
d	111

If we now look at this new table and replace each symbol in the sequence with its code instead, we will get the bit string 010011001001110100110100101100 10 (32 bits, 16 symbols). By changing the code for each symbol we have manage to compress the sequence by two symbols. The result may not be very impressive in this example, but depending on how long the sequence is and the probability of each symbol, this

compression technique can be very useful. Note that we can decompress this bit string by substituting each code with its symbol and thereby obtaining the original sequence again. This would not have worked if we had used, for example, the table below.

Symbol	Code
a	0
b	10
c	11
d	110

If we have the sequence *ca*, then we would encode this as 110 according to the table. If we now would try to decompress the bit string then we would not know if it is a *d* we should decode or first a *c* and then an *a*. So it is very important to give each symbol the right code so we always know how to decompress the data.

The technique we have used above is based on the probability of each symbol in some alphabet. Huffman, BCCBT and some other techniques are using this method in different ways to compress the data, which we will see in the next chapter.

A third way to compress some data is to use a dictionary. For example, if we are reading a good book about mathematic, we will probably encounter the words definition, theorem, proof, example and solution more often than other words. If we are really lucky, we will see these words at every page. Instead of coding these words as they are, we can use a static dictionary that contains these words. So when we compress the text, we just replace each word with its index in the dictionary instead.

Example 2-1

This text will contain the words definition, theorem, proof, example and solution. If you want a proof of this, just read this example and you will see the solution.

The dictionary may look something like this:

Index	Word
0	definition
1	theorem
2	proof
3	example
4	solution

If we now replace each word with its index using the table above, the text will change into this:

This text will contain the words 0, 1, 2, 3 and 4. If you want a 2 of this, just read this 3 and you will see the 4.

As you can see, by comparing the two texts, we have managed to compress the original text pretty much. Of course, if there are numbers in the text then we cannot use this approach. But then we can use an escape character instead and after that the index number. So instead of writing "…contain the words 0, 1…" we could write it like "…contain the words ~0, ~1…" where ~ is the escape character in this example.

The problem with a static dictionary is that it may work very well on some kind of data, while on others it may have no effect at all. Therefore we have techniques that are using dynamic dictionaries. Basically, what they do is that they build the dictionary

while reading the data. We will see some of the techniques that are using this approach to compress the data, for example LZ77, LZW.

If we cannot find a compression technique that works well on our data, then we can try to transform the data in some way so it may be more compression friendly. Let us look at a simple example.

Example 2-2

Value	3	5	7	9	11	13	15	17	19	21
Data block	0	1	2	3	4	5	6	7	8	9

As we can see, there are no patterns in this sequence, and the frequency of each symbol is the same. So using an RLE technique is not the best option, nor is a technique based on probability. Furthermore, a dictionary will not help us here. But if we transform the sequence by subtracting the value of data block $n+1$ with the value of data block n we will get the sequence

Value	3	2	2	2	2	2	2	2	2	2
Data block	0	1	2	3	4	5	6	7	8	9

We can still retrieve the original sequence by using the recursive formula $f(n+1)=f(n)+f(n+1)$. So in this example we will get

$$f(0)=3$$
$$f(1)=f(0)+f(1)=3+2=5$$
$$f(2)=f(1)+f(2)=5+2=7$$

and so on.

If we now look at this new sequence we will notice that it has a more friendly structure than the one before, and it is therefore much easier to find a compression technique, for example the RLE algorithm, that will be able to compress this new sequence much better than the sequence before.

We will see some examples of techniques that transform the data before compressing it in the next chapter, for example BWT and BCCBT.

3 LOSSLESS COMPRESSION TECHNIQUES

This chapter will introduce many of the more popular lossless algorithms. Lossless algorithms are algorithms that do not change the data. This means that when we decompress the data it will be the same as the original data was.

We will also need to know a little about the entropy.[1] The entropy tells us how much we are able to compress a source, that is, how many bits/symbol we can compress the source at best. Usually, it is not possible to calculate the entropy of a physical source, instead we have to approximate it. This is called the first-order entropy and we can use the following formula to calculate it:

$$-\sum_{i=1}^{n} P(x_i)\log P(x_i)$$

Here $P(x_i)$ is the probability of the symbol x_i. Since we will be working in bits we will use the formula

$$-\sum_{i=1}^{n} P(x_i)\log_2 P(x_i)$$

Formula 3-1

Note that since this formula is an approximation of the real entropy, we will not always get a correct value of the entropy. For example, if we have the sequence *aaaaaaaaaaaabbb*, we will have $P(a) = 0.8$ and $P(b) = 0.2$. If we now use Formula **3-1** on this sequence we will get the entropy 0.7219 bits/symbol. This means that we should not be able to find a compression scheme that is able to compress it better than 0.7219 bits/symbol. However, by using the RLE algorithm, which we will discuss in section 3.1, we can compress the sequence into 12*a*3*b*. This new sequence contains four symbols (we count 12 as one symbol) while the original one contains fifteen symbols. Calculating the number of bits/symbol in this case will give us the result $4/15 = 0.267$ bits/symbol. As we can see, this is far less than the value of the entropy, and the reason for that is because we are using an approximation. In most cases we will not have such an extreme case, and the first-order entropy will do just fine.

3.1 Predictive techniques

3.1.1 RLE

RLE stands for Run Length Encoding and is a very simple algorithm. However, there are numerous of different versions of this algorithm and I will discuss only a few of them.

Let us say that we have an alphabet A={*a, b, c, d, -9, -8,…, 8, 9*} and we would like to compress the sequence *ccccdddcadcaaaaabbbbb* (21 symbols). What we do is telling how many symbols there are in a row. So for our sequence we would encode it like 4*c*3*d*1*c*1*a*1*d*1*c*5*a*5*b* (16 symbols) and by that we have managed to compress the sequence by five symbols. On the other hand, we could have encoded it differently. Instead of coding 1*c*1*a*1*d*1*c* we could have encoded it like –4*cadc*. By doing so we would save another three symbols, a total of eight symbols. So when we decode the

[1] If you want a more thoroughly explanation of entropy, see "Sayood K., *Introduction to Data Compression*, pages 13ff"

data we know that when we meet a positive number we should repeat the symbol the number of times the positive number specifies, and when we meet a negative number we know how many symbols that are not alike, so we just decode the symbols as they are.

Another way to encode the sequence above is to encode repeated symbols as *aan,* where *a* is the repeated symbol and *n* specify how many times the symbol *a* should be repeated.[1] So in our case the compressed output would look like *cc*2*dd*1*cadcaa*3*bb*3, a total of 16 symbols.

3.2 Probability techniques

In this section I will show two techniques that uses the probability of each symbol to compress the data. The first one, Huffman coding, was developed by David Huffman in 1951[2]. The idea to the second one, arithmetic coding, came from Claude E. Shannon in 1948 and was further developed by Peter Elias and Norman Abramson in 1963.[3]

3.2.1 Huffman coding

The Huffman coding technique is probably the most well known compression algorithm out there. What it does is that it assigns a bit string to each symbol. Furthermore, the higher frequency a symbol has, a shorter bit string it will get. The two symbols with the least frequencies will have the same length of their bit strings and almost identical bit strings except for the last bit.

To know which bit string we should assign to each symbol we build a binary tree[4]. Let us say that we have the alphabet A={*a, b, c, d, e, f, g, h*}. From the table below we can see the frequency of each symbol from some file.

Symbol	Frequency
a	59
b	22
c	7
d	98
e	45
f	62
g	31
h	4

In order to make a binary tree of this table we start by sorting each symbol by its frequency. We will get the list

Symbol	*h*	*c*	*b*	*g*	*e*	*a*	*f*	*d*
Frequency	4	7	22	31	45	59	62	98

Our next step is to take the two symbols with the least frequencies and make a new symbol by combining the two symbols and their frequencies. In our case we will combine the symbols *h* and *c* and get the frequency $4+7=11$ for this new symbol,

[1] For more information about this technique see the homepage http://www.arturocampos.com/ac_rle.html

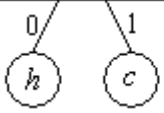
[2] http://www.anaesthetist.com/mnm/compress/huffman/

[3] Sayood K., *Introduction to Data Compression*, pages 79f

[4] For more information about binary trees I recommend reading the book "A. Standish Thomas, *Data structures in Java*, pages 242ff" or the book "Baase Sara and van Gelder Allen, *Computer algorithms Introduction to Design & Analysis*, pages 80ff".

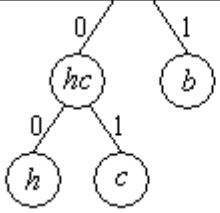
let us call it *hc*. The two symbols *h* and *c* will be child nodes for this new symbol. We now remove the two symbols *h* and *c* from the list and insert the symbol *hc* instead. After that we sort the list again. The new list will look like

Symbol	*hc*	*b*	*g*	*e*	*a*	*f*	*d*
Frequency	11	22	31	45	59	62	98

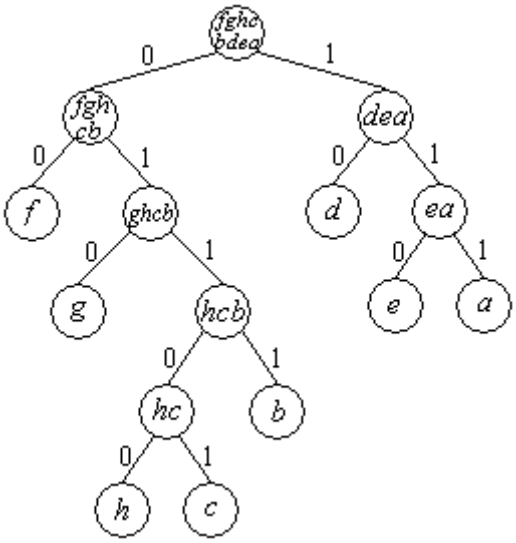


We now proceed as we did before. We take the two symbols with the least frequencies and create the symbol *hcb*, which will have the frequency 33, and after that we sort the list again. The new list will be

Symbol	*g*	*hcb*	*e*	*a*	*f*	*d*
Frequency	31	33	45	59	62	98



Continuing in this way we will finally get the binary tree



As we see from the tree, each symbol will get the bit string according to the table on the next page.

Symbol	Code	Length of bit string
d	10	2
f	00	2
a	111	3
e	110	3
g	010	3
b	0111	4
c	01101	5
h	01100	5

When we know the bit string for each symbol, it is very easy to compress the data. All we need to do is to substitute each symbol with its bit string.

Since the file contained eight different symbols, we need three bits for each symbol to be able to distinguish them from each other, in uncompressed mode. This makes the size of the file $3\times(59+22+7+98+45+62+31+4)=984$ bits large before compression. If we now calculate the new file size, according to the length of the bit string for each symbol listed in the table above, we will end up with a size of $2\times(98+62)+3\times(59+45+31)+4\times22+5\times(7+4)=868$ bits, meaning that we have saved a total of 116 bits (about 39 symbols) by compressing the file in this way.

As you might already have guessed, we always need to know the frequency of each symbol to be able to create a Huffman tree[1]. This makes the Huffman technique a two-step process. First we collect the probabilities of each symbol, and after that we build the tree. When we decompress, we also need some information to be able to decompress the data. The simplest way is to let the decoder know the frequency of each symbol. The only thing we need to do then is to create the Huffman tree in the same way as we did in the encoder, and after that just traverse the tree to find the symbol for some bit string. There is however a technique called "Canonical Huffman".[2] What it does is to apply some rules when one creates the Huffman tree. So instead of telling the frequency of each symbol to the decoder, you only need to let the decoder know the length of the bit strings for each symbol.

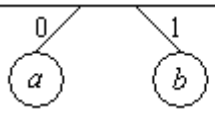
An interesting thing about the Huffman tree is that it is always optimal. What this actually means is that you cannot find another Huffman tree that is able to compress the data better.[3] But note this, there can be more than one optimal Huffman tree for the same dataset. The explanation for this is that depending on how we sort the list, we can create different Huffman trees. You can see an example of this in the two lists on the next page with the alphabet A={*a, b, c, d*}.

[1] There is a technique called Adaptive Huffman Coding. It is a one-step process and adapts the bit string for symbol (k+1) depending on the probabilities of the previous k symbols. See the book "Sayood K., *Introduction to Data Compression*, pages 55ff" for an explanation of this technique.

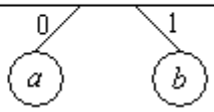
[2] For more information about Canonical Huffman I recommend the homepage http://www.anaesthetist.com/mnm/compress/huffman/

[3] You will find a proof of this in the book "Sayood K., *Introduction to Data Compression*, pages 45f".

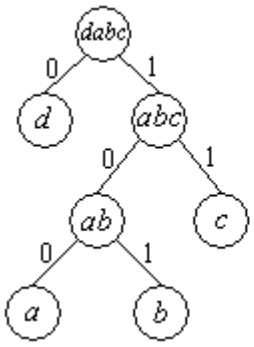
Symbol	*ab*	*c*	*d*
Frequency	14	14	14

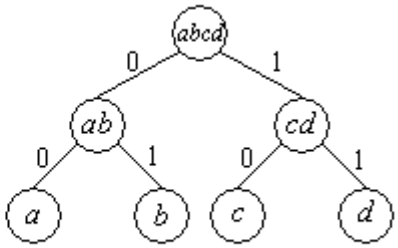


Symbol	*c*	*d*	*ab*
Frequency	14	14	14



The resulting trees will be





Let us say that the symbol *a* has the frequency two and the symbol *b* has the frequency twelve. Then according to the left tree, the number of bits per symbol would be $\frac{(3\times(2+12)+2\times14+1\times14)}{42}=2$ (so in this case we have not been able to compress the data at all). If we do the same calculation for the right tree we will also end up with 2 bits/symbol. This is not a coincidence. The answer is of course that the Huffman technique always creates an optimal Huffman tree.

The Huffman algorithm works very well on data with a large alphabet. However, if we have a small alphabet, or if the probability of the symbols is very skewed, it can do very poorly. For example, let us say that we have the alphabet A={*a*, *b*, *c*} and the probability of each symbol is $P(a)=0.55$, $P(b)=0.44$ and $P(c)=0.01$ from some source. If we now calculate the first-order entropy of this source we will get the value 1.06 bits per symbol. If we create a Huffman tree of this source, each symbol will get the bit string according to the table below.

Symbol	Code	Length of bit string
a	0	1
b	10	2
c	11	2

The number of bits per symbol in this case is $0.55\times1+0.44\times2+0.01\times2=1.45$ bits. As we can see, this is pretty far away from the first-order entropy. In order to lower the number of bits per symbol for a source, we can increase the size of the alphabet by grouping the symbols together. By doing this, we will get a value closer to the entropy. As we will see in the next section, the arithmetic algorithm is encoding whole sequences and not just one symbol at the time, and is therefore able to get closer to the entropy, even if the alphabet is small or the probability of the symbols is very skewed.

3.2.2 Arithmetic coding

While the Huffman algorithm assigns a bit string to each symbol, the arithmetic algorithm assigns a unique tag for a whole sequence. Since we are working with computers that are using bits, this tag will be a unique bit string in one way or the other.

The algorithm is dividing up an interval, usually between 0 and 1, to be able to assign a unique number for a certain sequence. How this interval is divided depends on the probabilities of the symbols. The higher probability a symbol has, the more space it will get in the interval. Let us say that we have an alphabet A={*a*, *b*, *c*, *d*} with the probabilities as shown in the table below:

Symbol	Probability
a	0.6
b	0.2
c	0.15
d	0.05

If we define the *cdf*[1] (cumulative distribution function) as $F(i) = \sum_{k=1}^{i} P(a_k)$ where $P(a_k)$ specifies the probability for the symbol a_k, we will get the values as shown in Table 3-1.

Table 3-1

F(*i*)	Value
F(1)	0.6
F(2)	0.8
F(3)	0.95
F(4)	1.0

As we see, the values range from 0 to 1. If we now define $F(0) = 0$ we can divide the interval [0.0, 1.0) into four subintervals: [*F*(0), *F*(1)), [*F*(1), *F*(2)), [*F*(2), *F*(3)) and [*F*(3), *F*(4)), that is: [0.0, 0.6), [0.6, 0.8), [0.8, 0.95) and [0.95, 1.0). Each symbol will have its own interval. Let us say that we have the sequence *acba* and we would like to encode this sequence. What we do is to see in what subinterval the first symbol in the sequence belongs to. In this case the first symbol is an *a*, which belongs to the subinterval [0.0, 0.6). We now divide this interval in the same way as we divided the interval [0.0, 1.0). The new subintervals will be: [0.0, 0.36), [0.36, 0.48), [0.48, 0.57) and [0.57, 0.6). The next symbol is a *c*, which belongs to the third subinterval [0.48, 0.57). We divide this interval in the same way as before and then read the next symbol in the sequence. Continuing in this way we will at the end have a unique number for the sequence. Let us look at an example to make things clear.

Example 3-1

First we have the alphabet A={*a*, *b*, *c*, *d*} with the corresponding probabilities $P(a) = 0.6$, $P(b) = 0.2$, $P(c) = 0.15$ and $P(d) = 0.05$. We also calculated the *cdf* (see Table 3-1). If we take a look at Figure 3-1, we will see the intervals at the beginning.

[1] For more information about the cumulative distribution function see "Sayood K., *Introduction to Data Compression*, pages 567f".

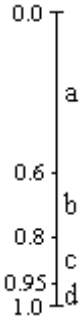


Figure 3-1

If we now continue to encode the sequence *acba* we can see the new intervals that will be created in Figure 3-2.

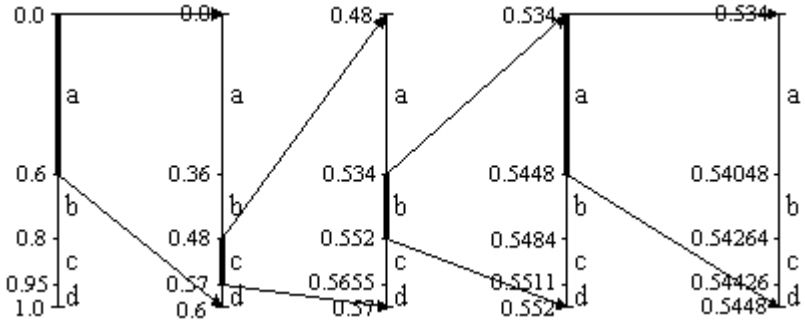


Figure 3-2

The last symbol is in the interval [0.534, 0.5448). So which number should we use as a tag? Well, actually we can choose any number in the interval [0.534, 0.5448). It does not matter when we are about to decode a tag. For example, we could choose the number 0.534 as our tag.

As we saw from Example 3-1, we chose the number 0.534 as our tag for the sequence *acba*. How do we decode this tag? There are two things we need to know: The tag itself of course, but also the probabilities of the symbols. The reason for this is that we need to calculate the *cdf* for the symbols as we did when we encoded the symbols in the first place. After we have this information, we proceed in the same way as we did in Example 3-1. See Example 3-2 for a demonstration of the decoding procedure.

Example 3-2

The first thing we need to do is to calculate the *cdf* for the symbols. Since the probabilities of the symbols are the same as when we encoded them, we will get the same values as Table 3-1 shows. We now divide the interval [0.0, 1.0) in the same way as we did in the encoding procedure. The result of this is shown in Figure 3-1. Now we have to check in what interval our tag is in. Since our tag is 0.534 we can see from Figure 3-1 that it belongs to the interval [0.0, 0.6), which is assigned to symbol *a*. Our first symbol to decode is therefore an *a*. Next we divide the interval [0.0, 0.6), and we will get the same values as shown in Figure 3-2. This time our tag is in the interval [0.48, 0.57), which is assigned to symbol *c*, so we decode a *c*. Continuing in this way we will at the end have decoded the sequence *acba* from our tag.

You might at first think: “Wow, this is great! This means that we can compress a whole sequence and after that we only need to store a number.” Well, this is true; we only need to store a number. Unfortunately, the computer has a finite number of bits,

and this number can have many decimals so we will not be able to store this number in an ordinary fashion. Even worse, when we do the calculations, the intervals will, sooner or later, be so small that the computer cannot handle the values and we will get the wrong tag value at the end. There are however tricks to come around this problem.[1]

3.3 Dictionary techniques

This section will discuss three compression techniques that are based on using a dynamic dictionary to be able to compress the data. They are: LZ77, LZ78 and LZW. LZ77 was developed in 1977 by Jacob Ziv and Abraham Lempel, while LZ78 was developed in 1978 by the same people.[2] The LZW algorithm is based on the LZ78 algorithm and was developed by Terry Welch in 1984.[3]

3.3.1 LZ77

The LZ77 algorithm is using a window that works like a dictionary. The window is divided into two sections: a search section, and a look-ahead section. See the picture below to understand what I mean.

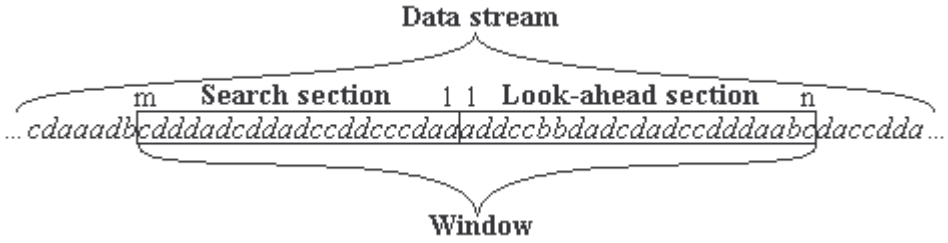


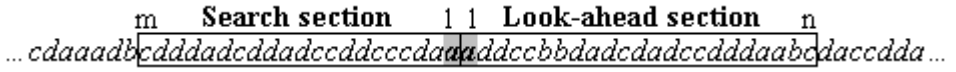
Figure 3-3

Let us say that we have an alphabet A={*a, b, c, d*} and we would like to continue to encode the sequence in Figure 3-3. The symbols in the search section, which has a size of m, have already been encoded, while the symbols in the look-ahead section, which has a size of n, are symbols that are about to be encoded. What we do, is to see if the first symbol in the look-ahead section exists in the search section. If it does not exist, we encode this as a triplet <0, 0, symbol>. If the symbol does exist in the search section, we take the offset of the symbol in the search section, and then find the length of the longest match of strings in the search section and look-ahead section. The triplet in this case would look like <offset, length, symbol>. The symbol in this case is the symbol that is after the match in the look-ahead section. The reason for having this symbol as the third field instead of the symbol of the last match, is because if we had not encoded it in the third field, there is a risk that the next triplet would be encoded as <0, 0, symbol>. So by including the symbol after the match, we are able to compress the data a bit more. An example of the encoding procedure will make everything clear.

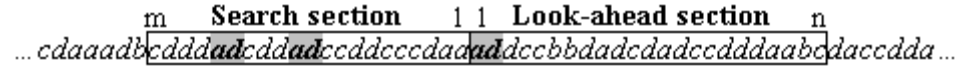
Example 3-3

If we look at Figure 3-3 we can see the status of the two sections. We have already encoded the symbols that are to the left of the look-ahead section. So our next step is to see if the symbol *a* exists in the search section. We start from right and read to the left. We find an *a* at the first position in the search section.

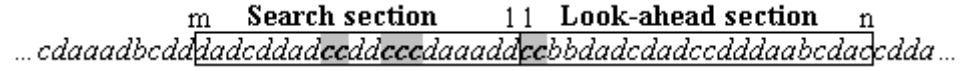
[1] For details on how to implement the arithmetic algorithm see the book "Sayood K., *Introduction to Data Compression*, pages 91ff"
[2] Sayood K., *Introduction to Data Compression*, page 120
[3] http://www.cs.cf.ac.uk/Dave/Multimedia/node214.html



However, this match has a length of only one. If we continue to look in the search section we will find two matches that have a length of two.



Which one we choose has no significance. Let us choose the first one we encountered. The triplet will then be encoded as <12, 2, *d*>. Since we have encoded three symbols we also move the window to the right by three symbols. So our next symbol to look for, in the search section, is a *c*.



This time we find three matches with a length of two. And as before, we choose the first match we encountered for our offset. So the triplet will be encoded as <8, 2, *b*>. We move the window to the right by three symbols, and the first symbol to look for next will then be a *b*. But as we can see in the search section, there are no *b*s, so we encode this triplet as <0, 0, *b*>. And after that we move the window by one symbol to the right. So our first three triplets will be <12, 2, *d*>, <8, 2, *b*> and <0, 0, *b*>.

The decoding procedure is very easy to do. We just read the first value in the triplet to see how many symbols we should go back in the sequence, and then the second value specifies how many symbols we should copy. The third value, the symbol, we just copy directly to the sequence. Naturally, the very first triplet we encounter will have both its offset and length set to 0, since the size of the search section will be 0 at first.

One thing that can be interesting to point out is that the length of the match can actually be greater than the size of the search section. This means that we have symbols that are in the look-ahead section, which will be part of the match, but will still be possible to decode. See Example 3-4.

Example 3-4

Figure 3-4

As we can see from Figure 3-4, we have a match that exceeds into the look-ahead section. The triplet in this case would be encoded as <5, 8, *d*>, and the window would be moved by nine symbols to the right. What will happen when we meet this triplet when we are about to decode the data? The sequence will look like this when we meet the triplet: …*bcbddabbacacbb*

The first value of the triplet says that we should go back by five symbols in the sequence, and then copy eight symbols. However, we have only five symbols that we can copy. On the other hand, when we copy the first symbol in the sequence, we will automatically have another symbol we can copy. The same thing for symbol two, the symbol three and so on. Therefore there will be no problems when the length of the match exceeds into the look-ahead section.

So how large should the size of the window be? Well, this is a hard question. The larger the search section is, the more patterns the algorithm will recognize since the search section actually is the dictionary. But if the search section becomes too large, then the algorithm can be very ineffective since we have to search a larger area. The look-ahead section should be large enough so we do not miss any symbols, or at least not many, that could have been recognized in the search section. One way to decide this size is to analyze the data first before compressing it.

Another thing to bear in mind is that the larger the window is, the more bits we need for the triplets. A triplet will at least need $\log_2\lceil m\rceil+\log_2\lceil m+n\rceil+\log_2\lceil A\rceil$ bits to be able to store all information, where *m* and *n* is the size of the search section and the look-ahead section respectively, and *A* is the size of the alphabet. The reason for $\log_2\lceil m+n\rceil$ is because the length of the match can exceed the size of the search section as we saw in Example 3-4.

We can do some modifications to the LZ77 algorithm so it will be more efficient in compressing the data. Instead of coding a single symbol as <0, 0, symbol>, we could get away with <0, symbol>. Since the first value of the triplet specifies the offset, we know that if this value is not 0, then the length field will be valid. On the other hand, if the offset value is 0, then we also know that the length field will be 0, so there is no reason for us to write this second field to the compression stream.

Another way is to use a single bit, which specifies if the coming data is a single symbol or not. For example, if the bit is 0, we could interpret this as that the next symbol is uncompressed. If the bit is 1 instead, we know that we had a match in the encoding procedure, so the next data will be an offset and a length field. We do not need the third field of the triplet in this case. This technique was developed in 1982 by James Storer and Thomas Szymanski and is called LZSS.[1]

3.3.2 LZ78

The LZ78 algorithm is a fairly simple technique. Instead of having a window as a dictionary, as LZ77 has, it keeps the dictionary outside the sequence so to speak. This means that the algorithm is building the dictionary at the same time as the encoding proceeds. When decoding, we also build the dictionary at the same time as we decode the data stream. Furthermore, the dictionary has to be built in the same way as it was built in the encoding procedure.

When decoding, the algorithm is using a double, <*i*, *s*>, to access the dictionary. The *i* is the index to the dictionary with the longest match, while the *s* is the symbol that is after the match. Let us go through an example to demonstrate how the technique works.

Example 3-5

Let say that we have an alphabet A={*a*, *b*, *c*, *d*} and we want to compress the sequence *abbcbbaabcddbccaabaabc*. What we do first is to see if the first symbol exists in the dictionary[2]. In this case the first symbol is an *a*, so we check if this symbol exists in the dictionary. Since the dictionary is empty at first, we will not find it. So what we do is that we add this symbol to the dictionary and write a double to the compression output. In this case we write <0, *a*>. The next symbol is a *b*, so we add this to the dictionary and write the double <0, *b*>. In this step the dictionary will have the following look:

[1] http://www.arturocampos.com/ac_lz77.html

[2] One way to search fast is to use a hash table that works as a dictionary. See the books "A. Standish Thomas, *Data structures in Java*, pages 324ff" and "Baase Sara and van Gelder Allen, *Computer algorithms Introduction to Design & Analysis*, pages 275ff" for an explanation of hash tables.

Dictionary		
Index	**String**	**Output**
1	*a*	<0, a>
2	*b*	<0, b>

The next symbol in the sequence is also a *b*, with the index 2 in the dictionary, so what we do now is to take the next symbol, which is a *c*, and combine the two symbols so we get the string *bc*. This string does not exist in the dictionary so we add it. The double that we write to the output will look like <2, *c*>. Continuing in this way the dictionary will at the end have the strings as shown in the table below.

Dictionary		
Index	**String**	**Output**
1	*a*	<0, *a*>
2	*b*	<0, *b*>
3	*bc*	<2, *c*>
4	*bb*	<2, *b*>
5	*aa*	<1, *a*>
6	*bcd*	<3, *d*>
7	*d*	<0, *d*>
8	*bcc*	<3, *c*>
9	*aab*	<5, *b*>
10	*aabc*	<9, *c*>

When we start to decode, we have to remember that we need to build the dictionary in the same way as we did in the encoding procedure. See Example 3-6.

Example 3-6

Let us take the same output as we got in Example 3-5. The first double, <0, *a*>, means that this symbol does not exists in the dictionary (remember that the dictionary is empty at this stage), so we add it to the dictionary and write the symbol *a* as the first symbol in the sequence. The next symbol is a *b* with the first field of the double set to 0. So we add the symbol to the dictionary and output *b* to the sequence. The third double is <2, *c*>. This means that we should output the string at index 2 of the dictionary combined with the symbol *c*. Furthermore, this string, *bc*, does not exists in the dictionary so we add it. At this stage, the dictionary will have the following appearance:

Dictionary		
Index	**String**	**Output**
1	*a*	*a*
2	*b*	*ab*
3	*bc*	*abbc*

If we continue in the same way we will at the end have the same dictionary as in Example 3-5, and also decoded the doubles to the original sequence.

3.3.3 LZW

The LZW algorithm is very similar to the LZ78 algorithm. Instead of having a double, <*i*, *s*>, the LZW is using some tricks to remove the need for the second field in the double. First, the dictionary contains the whole alphabet at the beginning of

encoding and decoding. Second, when building the dictionary, the last symbol in some string will always be the first symbol in the index below. An example will show better how it works.

Example 3-7

Let us use the same sequence, *abbcbbaabcddbccaabaabc*, and alphabet, A={*a, b, c, d*}, as in Example 3-5. The dictionary will at the beginning look like

Dictionary	
Index	**String**
1	*a*
2	*b*
3	*c*
4	*d*

The first symbol in the sequence is an *a*. This symbol does exist in the dictionary as index 1, so the next thing we do is to combine the symbol *a* with the next symbol in the sequence, in this case a *b*. We now have the string *ab*, which does not exist in the dictionary, so we add it to index 5, and encode a 1 to the output since the symbol *a* already exists in the dictionary. The next step we do is to take the symbol *b* in the string *ab*, and concatenate with the next symbol in the sequence, *b*. That way we create the string *bb*. This string does not exist in the dictionary so we add it, and encode a 2 to the output, since the symbol *b* is in the dictionary. The appearance of the dictionary is now

Dictionary	
Index	**String**
1	*a*
2	*b*
3	*c*
4	*d*
5	*ab*
6	*bb*

When we have encoded the whole sequence, we will have the dictionary

Dictionary			
Index	**String**	**Index**	**String**
1	*a*	11	*abc*
2	*b*	12	*cd*
3	*c*	13	*dd*
4	*d*	14	*db*
5	*ab*	15	*bcc*
6	*bb*	16	*ca*
7	*bc*	17	*aab*
8	*cb*	18	*ba*
9	*bba*	19	*aabc*
10	*aa*		

and the output 1 2 2 3 6 1 5 3 4 4 7 3 10 2 17.

If we compare the two outputs from Example 3-5 and Example 3-7 we can see that when we compressed the sequence using the LZ78 algorithm we needed 20 symbols,

while with the LZW algorithm we ended up with 15 symbols. So in this case we managed to save five symbols by using the LZW algorithm instead of the LZ78 algorithm.

Let us now see how the decoding procedure works.

Example 3-8

At the beginning we will have the dictionary with the whole alphabet in it. We also have the encoded sequence 1 2 2 3 6 1 5 3 4 4 7 3 10 2 17 which we got from Example 3-7. What we do now is to take the first symbol in the sequence, 1, and access the dictionary in that position. The string in that position is an *a*. We decode that string, and see if it exists in the dictionary. As we got it directly from the dictionary it exists. The next step is to take the second symbol in the sequence, 2 in this case, and decode the string at position two in the dictionary. Furthermore, we now concatenate this string, one symbol at the time, with the last symbol of the old string, and by that we create the new string *ab*. This string does not exist in the dictionary so we add it. We now use the last symbol in this new string and concatenate with the next decoded symbol in the dictionary, *b* since the next symbol in the sequence is 2, and the old string had no more symbols to concatenate. That way we have created the new string *bb*, which do not exist in the dictionary. The dictionary now looks like

Dictionary	
Index	**String**
1	*a*
2	*b*
3	*c*
4	*d*
5	*ab*
6	*bb*

with the decoded output *abb*. Continuing in this way we will end up with same dictionary and sequence as in Example 3-7.

An important thing to note when decoding is that if we have for example the symbol *c* in our hand, and our next step is to concatenate this symbol with the string *ad*, we can not just create the string *cad* and see if this string exists in the dictionary or not. We have to first create the string *ca*, and add it to the dictionary if necessary, and then see if the string *ad* exists in the dictionary before we continue to read from the encoded sequence.

There is one big problem with the decoding algorithm we used in Example 3-8. In some circumstances it can happen that we have to access the dictionary in a position where we actually are building the string. This is however not impossible to deal with, but it means that we need a special case in the decoding algorithm to handle this problem.[1]

3.4 Transforming the data

In this section I will introduce two algorithms that only change the data and not compressing it. The first one we will look at is MTF (Move To Front) and was developed by Jon Bentley, Daniel Sleator, Robert Tarjan and Victor Wei in 1986.[2] Part of the second algorithm, BWT (Burrows-Wheeler Transform), was developed in 1983

[1] For more information about this problem, see "Sayood K., *Introduction to Data Compression*, pages 130ff"

[2] http://www.data-compression.info/Algorithms/MTF

by David Wheeler, but was not published, to its full, until 1994 together with Michael Burrows.[1]

3.4.1 MTF

The MTF algorithm is a very simple transformation technique. Let us say that we have an alphabet A={*a, b, c, d, e, f, g, h*}. If we now create a list where each symbol gets a number assigned to it, depending on where in the list the symbol is, the list may look something like this

Symbol	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
Position	0	1	2	3	4	5	6	7

If we now wanted to transform the sequence *aabcbbbccbaafgaddeeehhggggcca* we would start with the first symbol and substitute it with the position it has in the list, in this case position 0. We then put that symbol to the top of the list. Since the symbol already is at the top, there will be no change to the list. The next symbol is also an *a* so we substitute that symbol with a 0, and there will still be no change to the list. The next symbol is a *b*. If we look at the list, we can see that this symbol has the position 1. So in this case we change the *b* to a 1 in the sequence. We now move that symbol to the top of the list. The new list will look like this

Symbol	*b*	*a*	*c*	*d*	*e*	*f*	*g*	*h*
Position	0	1	2	3	4	5	6	7

The next symbol is a *c* with the position 2. So we change the *c* in the sequence to a 2, and move that symbol to the top of the list. The list will be

Symbol	*c*	*b*	*a*	*d*	*e*	*f*	*g*	*h*
Position	0	1	2	3	4	5	6	7

Continuing in this way we will transform the old sequence to the new sequence 00121001012056250600704000705. As we can see, the new sequence has many low numbers. If we had only changed the symbols without affecting the list, we would have gotten the sequence 00121112210056033444776666220 instead. Let us see what the probability of each symbol is in the two sequences.

Affecting the list	
Symbol	**Probability**
0	0.48
1	0.14
2	0.10
3	0
4	0.03
5	0.10
6	0.07
7	0.07

Without affecting the list	
Symbol	**Probability**
0	0.21
1	0.17
2	0.17
3	0.07
4	0.10
5	0.03
6	0.17
7	0.07

If we now calculate the first-order entropy for the left and right table, we will get 2.26 bits/symbol for the left table, and 2.8 bits/symbol for the right table. By only transforming the sequence we have managed to change it in a way so we should be able to compress it better than before.

[1] http://dogma.net/markn/articles/bwt/bwt.htm

To decode the new sequence so we get the old sequence back is not very hard. We do exactly the same thing as we did when we transformed the sequence. The important thing to remember is that the list needs to be in the same way as it was when we transformed the sequence at first. After that we use the numbers in the sequence to access the list in the specified position and substitute that number with its symbol. If we read a 0, there will be no change to the list, we will only substitute the number with its symbol. If we read a number different from 0 in the sequence, we will substitute that number with its symbol in the list, and move that symbol to the top of the list.

3.4.2 BWT

This algorithm is a bit more complex than the MTF algorithm, but well worth the effort to learn. Let us say that we have a sequence of length *n*. What we do now is to create *n-1* more sequences, each cyclic shifted (to the left or right, does not matter). These *n* sequences we sort in lexicographical order. We take the last symbol in each sequence and create a new sequence of these last symbols, which, of course, will have length *n*. To be able to recover the original sequence again, we also need to know the position of the original sequence in the sorted list. Let us work through an example to demonstrate how the encoding procedure works.

Example 3-9

If we have, for example, the alphabet A={*a, b, c*} and the sequence *cabcbc* (length six) we start by creating five more sequences, each cyclic shifted. After that we sort these sequences and take the last symbol from each sequence. See the two tables below.

Table 3-2

Not sorted					
c	*a*	*b*	*c*	*b*	*c*
c	*c*	*a*	*b*	*c*	*b*
b	*c*	*c*	*a*	*b*	*c*
c	*b*	*c*	*c*	*a*	*b*
b	*c*	*b*	*c*	*c*	*a*
a	*b*	*c*	*b*	*c*	*c*

Table 3-3

Sorted					
a	*b*	*c*	*b*	*c*	*c*
b	*c*	*b*	*c*	*c*	*a*
b	*c*	*c*	*a*	*b*	*c*
c	*a*	*b*	*c*	*b*	*c*
c	*b*	*c*	*c*	*a*	*b*
c	*c*	*a*	*b*	*c*	*b*

We now have the new sequence *caccbb*. By only looking at it, we can see that this new sequence has a more compression friendly structure than the original one. If we had a longer sequence, this would be even more evident. The last thing we need to do is to send where in the list the original sequence is. In this case it is at position three (first row zero).

When one has used the BWT algorithm on a sequence, one usually does not start to compress it after that. Instead we try to transform it even more, for example with the MTF algorithm. It is hard to see the purpose of this on short sequences, but if we transform the sequence "*this_is_just_a_demonstration_for_the_bwt_algorithm*" we will get the new sequence "*tteanssr__r__hd_ltttth_r_aehooimfgotoiiunsw_miasjb*". As we can see, we have more letters that are together in this new sequence, while we had none in the original sequence. If we now were to use the MTF algorithm on these two sequences, with the list initialized to

_	*a*	*b*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*l*	*m*	*n*	*o*	*r*	*s*	*t*	*u*	*w*

we would get the sequence "16 8 9 16 4 2 2 2 11 17 3 6 4 7 1 9 10 14 16 16 8 8 17 9 2 12 6 6 10 15 3 7 3 6 14 11 3 15 18 5 3 11 18 18 10 10 13 7 11 15" for the original one, and "16 0 5 3 13 16 0 16 6 0 1 1 0 11 9 2 14 9 0 0 0 4 3 5 1 8 9 4 16 0 14 16 14 15 4 10

1 5 0 17 15 15 18 13 10 6 13 5 18 18" for the new one. Calculating the first-order entropy, we get 4.02 bits/symbol and 3.83 bits/symbol respectively. By using the BWT algorithm we have managed to transform the sequence so it is easier to find more structure in the data, and by that we have also lowered the entropy, in this case by 0.19 bits/symbol, for the data.

Ok, so now we have transformed the sequence to a new sequence, let us call this new sequence for *L*, and we also know where in the list the original sequence was. This is the only information available to the decoder. Note that the sequence *L* contains all the symbols as the original sequence contained. So how do we transform the transformed sequence back to its original form? Well, the sequence *L* contains the last symbols of each cyclic shifted sequence, sorted in lexicographical order, see Table 3-3. Since the sequence *L* contains all the symbols from the original sequence, we also know the first symbol in each sorted sequence, again see Table 3-3. All we have to do is to sort the sequence *L* to be able to get the first symbol from each sorted sequence, let us call this sequence for *F*. By knowing *L* and *F* and the position of the original sequence in *L*, we can create a transformation *T* that will tell us in what order we should read the symbols in *L* to be able to get the original sequence back.

Table 3-4

F					
a	*b*	*c*	*b*	*c*	*c*
b	*c*	*b*	*c*	*c*	*a*
b	*c*	*c*	*a*	*b*	*c*
c	*a*	*b*	*c*	*b*	*c*
c	*b*	*c*	*c*	*a*	*b*
c	*c*	*a*	*b*	*c*	*b*

Table 3-5

L					
c	*a*	*b*	*c*	*b*	*c*
a	*b*	*c*	*b*	*c*	*c*
c	*b*	*c*	*c*	*a*	*b*
c	*c*	*a*	*b*	*c*	*b*
b	*c*	*b*	*c*	*c*	*a*
b	*c*	*c*	*a*	*b*	*c*

If we take a look at Table 3-4 we can see that this table is the same as Table 3-3. Table **3-5** however is created by cyclically shift every row of Table 3-4 one step to the right. As we can see, row 0 in Table 3-5 is the same as row 3 in Table 3-4. Let us use this information as our transformation *T*, that is, $T(0)=3$. Our *T* in this example will then be *T* = [3 0 4 5 1 2]. If we now define two operators L_F and L_L, where $L_F(j)$ is the first symbol in row *j* of Table 3-4, and $L_L(j)$ is the last symbol in row *j* of the same table, we have the following equality: $L_F(T(j)) = L_L(j)$

Since the sequence $T(j)$ of Table 3-4 is the same as the sequence *j* of Table 3-5, and the first symbol in each row in Table 3-5 is the same as the last symbol in each row in Table 3-4, the equality above is true. Furthermore, $L_L(j)$ precedes $L_F(j)$ cyclically; see for example the first row of Table 3-4 and Table 3-5. Therefore $L_L(T(j))$ precedes $L_F(T(j))$ and since $L_F(T(j)) = L_L(j)$, $L_L(T(j))$ precedes $L_L(j)$, $L_L(T(T(j)))$ precedes $L_L(T(j))$ and so on. Knowing this fact, we can recover the original sequence that we had before the transformation. Since the sequence *L* contains the last symbol of every cyclically shifted sequence, we build the original sequence by using the fact that $L_L(T(j))$ precedes $L_L(j)$. See Example 3-10 for a demonstration of this.

Example 3-10

We know that the original sequence is located at position three. Our last symbol in the original sequence is therefore $L_L(3)=c$. Our next symbol will be located at $L_L(T(3))=L_L(5)$ of the sequence *L*, which is the symbol *b*. So at this stage we have

decoded the sequence *bc*. The next symbol is $L_L(T(T(3))) = L_L(T(5)) = L_L(2) = c$. If we continue in this way we will have decoded the sequence *cabcbc* at the end.

As we can see by Example 3-10, it is very easy to recover the original sequence when we have the transformation *T*. So how do we create this transformation? When we created the transformation *T* before, we looked at the whole sequence. But we have only *F* and *L*. If we look at Table 3-5 we see that the first symbol of the first row is a *c*. If we now look at Table 3-4 we can see that we have three choices: row 3, 4 or 5. Since Table 3-4 is sorted by the first symbol, Table 3-5 is sorted based on the second symbol. Therefore we know that the order of the symbols in *F* and *L* is the same, meaning that the first *c* in *L* is the first *c* in *F*, that is, $T(0) = 3$, $T(2) = 4$, $T(3) = 5$ and so on.

3.5 BCCBT

BCCBT is a shortening for Bit Coding using a Complete Binary Tree.[1] It has some similarities with the Huffman algorithm; both are using a binary tree to decide what kind of bit code every symbol should get. Also, both are using the probability of each symbol to build the binary tree. What differs is that the Huffman algorithm is a two-step process, while BCCBT is a three-step process in most cases, but can be a two-step process too, but will then not be as efficient in that case. Furthermore, while the Huffman algorithm has one output stream (the bit codes), the BCCBT algorithm will have two output streams. We will see more of this later in this section.

I got the idea to the BCCBT algorithm when I studied the Huffman algorithm. I was thinking: "What would happen if one was to balance the Huffman tree?" I did not study this very much, but it did lead me to complete binary trees. So I made up a sequence of different symbols and created a complete binary tree of this sequence, where I let the symbol with the highest probability of occurrence be the root node of this tree. Then I took the symbol with the second highest probability of occurrence and put it as the left child of the root node. I continued in this way until I had no more symbols to put in the tree. After that I assigned a bit code to each symbol depending on where in the tree the symbol was located. It did not take long until I realized that I needed something more than just the bit codes to be able to decode a bit string. The answer was at what level the symbols was located in the tree. I encoded this sequence using the complete binary tree, and saw some interesting results of the encoded string, which I will write about later.

So how does the BCCBT algorithm work? Well, first we need to know the probability of each symbol, just as the case with the Huffman algorithm. Next we create a complete binary tree. In this tree, the symbol that has the highest probability will be the root node. The symbol with the second highest probability will be the left child of the root node; the symbol with the third highest probability will be the right child of the root node and so on. Let us see an example on how to create this binary tree.

Example 3-11

Imagine that we have an alphabet A={*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*} where each symbol has the frequency, from some dataset, as shown in Table 3-6.

[1] For a definition of complete binary trees, see the book "A. Standish Thomas, *Data structures in Java*, page 249"

Table 3-6

Symbol	Frequency
a	32
b	55
c	4
d	19
e	37
f	26
g	9
h	7

If we now were to create a complete binary tree of Table 3-6, we would get the following tree:

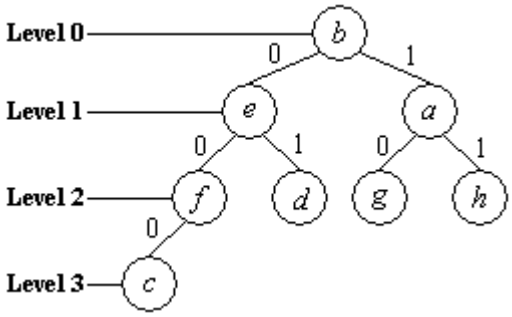


Figure 3-5 Complete binary tree

As we can see from Figure 3-5, each symbol will have its own bit code, except for the root node, which will have none. For example, the symbol *c* would get the bit code 000, while the symbol *g* would get the bit code 10. However, this information alone is not enough to be able to decode a bit sequence. If we have the bit sequence 01, we cannot know if this means the symbol *d*, or if it means the two symbols *e* and *a*. We need to know something more, and the answer is at what level the symbol is in, in the tree. For example, if we wanted to encode the sequence *feed*, using the tree in Figure **3-5**, we would encode it like [2]00[1]0[1]0[2]01, where the number between the [] specifies the level in the tree. Let us look at an example on how two decode this sequence using the tree in Figure 3-5.

Example 3-12

The first ”symbol” in the sequence [2]00[1]0[1]0[2]01 specifies at what level the symbol is in, in this case at level 2. The next two bits, 00, tell us how we should walk the tree, in this case to the left two times. We will end up at the symbol *f*. So our first symbol to decode is an *f*. The next “symbol” in the sequence is a 1, and the bit code is a 0. We walk to the left one step, and end up on level 1 where we find the symbol *e*. Continuing in this way we will at the end have decoded the sequence *feed*.

As we saw from Example 3-12, we had a sequence that looked like [level][bit code]…[level][bit code]. The level field does not only tell us at what level the symbol is located in the tree. It also tells us how long the bit code is, that is, how many bits we should read to be able to know how we should walk the tree.

If we take a look at Figure 3-5, we can see that the symbol *c* is alone at level 3. This means that as soon as we read a 3 in the level field, we will know that this is the symbol *c*. This in return means that the bit code field is unnecessary in this case. By skipping this bit code, we will be able to compress the data source even further. On the other hand, since the symbol *c* has the lowest probability, it is better to use this place in

the tree for another symbol, for example the symbol with the second highest probability, in Figure 3-5 the symbol *e*. We now move each symbol one step to the left in the tree. The new tree will be:

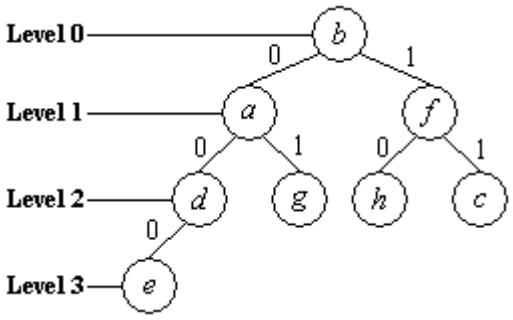


Figure 3-6

By building the tree in this way, we can skip the bit codes for the two most frequently occurring symbols, and thereby increasing the compression ratio. Furthermore, one symbol at each level, except for the symbol at level 0, will decrease the length of its bit code by one, since that symbol will move to a lower level. This means that we will decrease the length of the bit code sequence.

Can we always arrange the symbols in this way? Yes, if the size of the alphabet is 2^n. Let us make this into a theorem.

Theorem 3-1

If we have a complete binary tree with 2^n nodes, $n >= 0$, there will be only one node at the last level of the tree.

Proof

Let us say that we have a complete binary tree that has *n*-1 levels, and where each level is filled with nodes. Since each internal node in this binary tree will have two child nodes, the level *d* will have 2×2^k nodes if the level *d*-1 has 2^k nodes. In order to calculate how many nodes this binary tree has, we can use the formula

$$2^0 + 2^1 + 2^2 + ... + 2^{n-2} + 2^{n-1}$$

If we now add another node to this binary tree, that node will be the only one at level *n*. So the tree will now have $2^0 + 2^1 + 2^2 + ... + 2^{n-2} + 2^{n-1} + 1$ nodes. We can rewrite this formula to

$$2^1 + 2^1 + 2^2 + ... + 2^{n-2} + 2^{n-1} = 2 \times 2^1 + 2^2 + ... + 2^{n-2} + 2^{n-1} =$$
$$2^2 + 2^2 + ... + 2^{n-2} + 2^{n-1} = 2 \times 2^2 + ... + 2^{n-2} + 2^{n-1} =$$
$$2^3 + ... + 2^{n-2} + 2^{n-1} = ... = 2 \times 2^{n-2} + 2^{n-1} =$$
$$2^{n-1} + 2^{n-1} = 2 \times 2^{n-1} = 2^n$$

And we have proven that if we have a complete binary tree with 2^n nodes, there will be only one node at the last level.

Since we are using the frequency of each symbol to build the complete binary tree, the decoder will also need to know this information in one way or the other. In fact, it does not have to know the frequency of each symbol, but it needs to know the order of the symbols sorted by the frequency. For example, if we had the frequencies of the

symbols as in Table 3-6, we would send the symbols to the decoder in the following order: *b, e, a, f, d, g, h, c*

This means that it will be some overhead. How big overhead depends on the size of the alphabet. We can calculate this overhead by using the formula $A_S \times \lceil \log_2 A_S \rceil$, where A_S is the size of the alphabet. The result will tell us how many bits we need to store this overhead.

So how good is this compression algorithm (without the modification)? Well, the best case is of course if we have a source with an alphabet of size one. Then if we have a sequence of length *n*, we will need $L_S = n \times \lceil \log_2 (\lfloor \log_2 A_S \rfloor + 1) \rceil$ bits for the level sequence, while we need no bits for the bit code sequence. If we do the calculation we will note that we do not need any bits for the level sequence either in this case. The worst case is when the symbols in the source have the same probability of occurrence. Then for a sequence of length *n*, the level sequence would need L_S bits. The bit code sequence would need

$$\frac{n}{A_S}\left(2^0 \times \log_2 2^0 + 2^1 \times \log_2 2^1 + \ldots + 2^{m-1} \times \log_2 2^{m-1} + l \log_2 2^m\right)$$

bits. Here 2^x specifies the number of nodes at level *x*, $m(=\lfloor \log_2 A_S \rfloor)$ is the depth of the tree, and $l(=A_S - (2^m - 1))$ is the number of nodes at the last level *m*. We can rewrite this formula to

$$\frac{n}{A_S}\left(lm + \sum_{k=0}^{m-1} k \times 2^k\right)$$

Taking the derivative of this expression $\sum_{k=0}^{m-1} x^k = \frac{1-x^m}{1-x}$, we will get

$$\sum_{k=0}^{m-1} k x^{k-1} = \frac{(m-1)x^m - mx^{m-1} + 1}{(1-x)^2}$$

If we now multiply an *x* on both sides and set $x = 2$, we will get the expression

$$\sum_{k=0}^{m-1} k \times 2^k = (m-2)2^m + 2$$

So the number of bits the bit code sequence will need is

$$\frac{n}{A_S}\left(lm + (m-2)2^m + 2\right)$$

The total number of bits we will need is therefore

$$n\lceil \log_2 (m+1) \rceil + \frac{n}{A_S}\left(lm + (m-2)2^m + 2\right)$$

If we were not to compress the source, we would need $n\lceil \log_2 A_S \rceil$ bits. We also have $A_S = 2^k 2^m$ where $0 \le k < 1$. Furthermore

$$\begin{aligned}
&\lceil \log_2 A_S \rceil = \lceil \log_2 2^k 2^m \rceil \Leftrightarrow \\
&\lceil \log_2 A_S \rceil = \lceil k + m \rceil \Leftrightarrow \\
&\lceil \log_2 A_S \rceil = \lceil k \rceil + m
\end{aligned}$$

We also know that for some constant *c*, this expression holds true:

$$n\lceil \log_2 (m+1) \rceil + \frac{n}{A_S}\left(lm + (m-2)2^m + 2\right) \le cn\lceil \log_2 A_S \rceil$$

Solving *c* in this expression we will get

$$\begin{aligned}
&\lceil \log_2 (m+1) \rceil + \frac{1}{2^k 2^m}\left(\left(A_S - \left(2^m - 1\right)\right)m + (m-2)2^m + 2\right) \le c(\lceil k \rceil + m) \Leftrightarrow \\
&\frac{\lceil \log_2 (m+1) \rceil}{\lceil k \rceil + m} + \frac{\left(A_S - 2^m + 1\right)m}{(\lceil k \rceil + m)2^k 2^m} + \frac{m}{(\lceil k \rceil + m)2^k} - \frac{2}{(\lceil k \rceil + m)2^k} + \frac{2}{(\lceil k \rceil + m)2^k 2^m} \le c \Leftrightarrow \\
&\frac{\lceil \log_2 (m+1) \rceil}{\lceil k \rceil + m} + \frac{m2^k 2^m}{(\lceil k \rceil + m)2^k 2^m} + \frac{m}{(\lceil k \rceil + m)2^k 2^m} - \frac{2}{(\lceil k \rceil + m)2^k} + \frac{2}{(\lceil k \rceil + m)2^k 2^m} \le c \Leftrightarrow \\
&\frac{\lceil \log_2 (m+1) \rceil}{\lceil k \rceil + m} + \frac{1}{\left(\frac{\lceil k \rceil}{m} + 1\right)} + \frac{m}{(\lceil k \rceil + m)2^k 2^m} - \frac{2}{(\lceil k \rceil + m)2^k} + \frac{2}{(\lceil k \rceil + m)2^k 2^m} \le c
\end{aligned}$$

If we now let $m \to \infty$ the expression will be transformed to $1 \le c$. This means that we will not be able to compress the source, but we will be close to the original size of the source if we let $m \to \infty$. Of course, it is not possible to let $m \to \infty$ because that would mean that the size of the alphabet would be infinite. Now, set

$$f(x) = \frac{\lceil \log_2 (x+1) \rceil}{\lceil k \rceil + x} + \frac{x}{\lceil k \rceil + x} + \frac{x}{(\lceil k \rceil + x)2^k 2^x} - \frac{2}{(\lceil k \rceil + x)2^k} + \frac{2}{(\lceil k \rceil + x)2^k 2^x}$$

and

$$g(x) = \frac{\log_2 (2x)}{x} + \frac{3}{2^x} + 1$$

We can see that $f(x) \le g(x), x \ge 1$. We can also see that if we let $x \to \infty$ then $f(x) \to 1$ and $g(x) \to 1$. Furthermore, $g(x)$ is a decreasing function, that is $g(x_2) < g(x_1)$ if $x_2 > x_1$. We can prove this by calculating the following expression:

$$g(x \times d) - g(x) < 0, d > 1$$

We have

$$g(x\times d)-g(x)=\frac{\log_2(2xd)}{xd}+\frac{3}{2^{xd}}+1-\frac{\log_2(2x)}{x}-\frac{3}{2^{x}}-1<0\Leftrightarrow$$
$$\frac{\log_2(2xd)}{xd}-\frac{\log_2(2x)}{x}+\underbrace{\frac{3}{2^{xd}}-\frac{3}{2^{x}}}_{<0}<0$$

If we can show that $\frac{\log_2(2xd)}{xd}-\frac{\log_2(2x)}{x}\le 0$ then we have proven that $g(x)$ is a decreasing function.

$$\frac{\log_2(2xd)}{xd}-\frac{\log_2(2x)}{x}=\frac{\log_2(2xd)-d\log_2(2x)}{xd}=$$
$$\frac{\log_2(2xd)-\log_2(2^d x^d)}{xd}=\frac{\log_2\left(\frac{2xd}{2^d x^d}\right)}{xd}=\frac{\log_2\left(\frac{d}{2^{d-1}x^{d-1}}\right)}{xd}\le 0$$

And we have shown that $g(x)$ is a decreasing function. What this all means is that the larger *m* we have, the closer $f(x)$ will get to 1 since $g(x)$ is a decreasing function and $f(x)\le g(x)$. So the larger alphabet we have, the closer we will get to the original size of the source. On the other hand, the level sequence will have some special properties that will make us to compress the source even further as we will see next.

Let us take a real life example and compare the algorithm with the Huffman algorithm. The source that I have used is the text of the first chapter of this paper as it looked once. The result, in bytes (no overhead), that I got can be seen in Table 3-7.

Table 3-7

Original	Huffman	BCCBT
3414	1826	2462

As we can see from Table 3-7 the BCCBT algorithm is doing a pretty good job, although not as good as the Huffman algorithm. Let us see how the 64 first symbols of the output looks like in hexadecimal format.

Table 3-8

Huffman							
03	D9	F1	A8	31	D5	9F	4F
7C	DB	5F	CD	9B	6B	ED	7F
2B	9D	5C	C1	90	FD	D3	FD
F2	E7	75	B3	CF	11	29	9C
35	08	C4	DD	B5	27	C2	1D
A3	2B	6A	1B	EF	D0	EA	79
45	FB	27	B3	DD	B3	B5	F8
EA	ED	81	71	F9	5C	3B	94

Table 3-9

BCCBT							
49	33	00	7D	08	C1	F2	07
00	42	08	08	C9	43	01	19
81	94	30	28	60	22	02	12
89	34	51	70	06	48	40	04
08	36	6C	0E	14	82	83	06
20	B1	01	D2	0D	1F	31	03
19	44	16	02	83	A6	4B	29
42	85	03	04	02	0C	24	01

If we look at these two tables, we will note that the data is pretty much randomized, that is, we cannot see any structure in the tables. Remember that we saved the encoded symbols as [level][bit code] in the BCCBT algorithm. Let us see what will happen if we save the encoded symbols as [level][level]…[level][bit code][bit code]…[bit code], that is, we save the levels first, and after that the bit codes. In this case the 64 first symbols of the output will specify the level of each symbol in the tree. Furthermore, I

used an alphabet of size 256, so the levels will be between 0 and 8, meaning that we need 4 bits to store each level. The result can be seen in Table 3-10.

Table 3-10

Levels							
43	**80**	**34**	**33**	**21**	**80**	**24**	**01**
32	**10**	**13**	**81**	**21**	**02**	**10**	**12**
08	**42**	**32**	**28**	**03**	**24**	**04**	**83**
30	**32**	**44**	**83**	**82**	**10**	**32**	**33**
38	**11**	**22**	**20**	**18**	**33**	**22**	**54**
81	**04**	**23**	**40**	**22**	**30**	**18**	**80**
24	**02**	**10**	**21**	**03**	**21**	**12**	**43**
80	**12**	**03**	**32**	**23**	**80**	**23**	**03**

It is hard to see, but there are many numbers that occur many times, for example the numbers 80, 32 and 21. If we had looked at a greater range, we would have seen this more easily. This means that there are symbols that will occur more often than others. The explanation for this is that some symbols will be on the same level, or just a level below or above, in the binary tree. So if we have the tree in Figure 3-6, and we have a sequence that contains the symbols *d*, *g*, *h*, *c*, the level sequence will be 222…22 no matter how long the symbol sequence is. This means that we should be able to find more structure in the level sequence and thereby compressing it even more. We will see more of this in chapter 6.

Another thing worth noting is that we needed 4 bits for the level field, but we only used the numbers 0-8, that is, we have the numbers 9-15 that we can use for other things. For example, the number 9 could mean that we should use the RLE algorithm. So when we encode the data and we have long runs of a certain symbol, we will write the number 9 to the level sequence, and write the symbol and how many times it is repeated in the bit code field. When we decode and have read a 9 from the level field, we will read the symbol and how many times the symbol should be repeated from the bit code field. The drawback of this is that we will increase the combination of the numbers in the level sequence, and thereby loose some structure too. On the other hand, we will actually decrease the length of the level sequence by using the RLE algorithm. We will see a modification of the BCCBT algorithm that is using the RLE algorithm in chapter 6.

3.6 Adaptive BCCBT

The adaptive BCCBT works almost the same as the ordinary BCCBT. We are building the complete binary tree in the same way as we did before, we can also use the trick to decrease the bit code sequence as we did in Figure 3-6. However, what differs is while the BCCBT algorithm uses the probabilities of the whole source, the adaptive BCCBT adapts the binary tree depending on the sequence of the symbols. In fact, we are using the same approach to change the binary tree as the MTF algorithm is using to change a sequence. Let us see an example on how we encode a sequence using the tree in Figure 3-6.

Example 3-13

If we have the sequence *dbbhadh* we start by checking where the first symbol, *d*, is located in the tree. In Figure 3-6 it is located at level 2 with the bit code 00. So we write these values to the output. We now move this symbol to the root node. The tree will have the following look (in sequential representation[1]):

[1] For an explanation of sequential representation, see “A. Standish Thomas, *Data structures in Java*, pages 250ff”

	d	*b*	*a*	*f*	*g*	*h*	*c*	*e*

Our next symbol is a *b*, which is located at level 1 with the bit code 0. We move this symbol to the top of the tree. The sequential representation will now look like

	b	*d*	*a*	*f*	*g*	*h*	*c*	*e*

The next symbol is also a *b*, therefore we do not need to change the sequential representation of the tree. The symbol *b* is at this stage at level 0, and therefore has no bit code. Continuing in this way we will at the end have the level sequence [2][1][0][2][2][2][1] and the bit code sequence [00][0][][10][00][00][1]. The sequential representation of the tree will be

	h	*d*	*a*	*b*	*f*	*g*	*c*	*e*

As we can see from Example 3-13, the approach we are using does not differ from the approach that the MTF algorithm is using. When we decode, we do exactly the same thing as we did when we encoded the sequence. Let us see an example on how this works.

Example 3-14

At the start, we will have the same tree as the one in Figure 3-6. Let us also use the level sequence and the bit code sequence that we got from Example 3-13 as our data to decode. First we read a 2 from the level sequence. This tells us how many bits we should read in the bit code sequence. So we read two bits and get the bit code 00. We walk the tree, in this case two steps to the left, and end up at the symbol *d*. So we decode a *d*, and move that symbol to the root node. The sequential representation of the tree will be

	d	*b*	*a*	*f*	*g*	*h*	*c*	*e*

Next we read a 1 in the level sequence, meaning that we should read one bit in the bit code field, which will get us the bit code 0. So we walk one step to the left in the tree and decode the symbol *b*. We now move this symbol to the top of the tree and by that we will get the sequential representation

	b	*d*	*a*	*f*	*g*	*h*	*c*	*e*

The next field in the level sequence is a 0, that is, there are no bits to read in the bit code field (note that if we had read a 3 in the level sequence, this would also mean that we should not read any bits from the bit code field). This tells us that the symbol we looking for is at the root node, in this case a *b*. If we continue in this way we will at the end have decoded the sequence *dbbhadh*.

Since we do not need to know the frequency of each symbol, it is unnecessary for us to store this overhead. The only thing we need to think of is that both the encoder and decoder must have the complete binary tree initialized in the same way at the start.

In chapter 6 we will see more of this algorithm when I compare the different algorithms to each other.

4 LOSSY COMPRESSION TECHNIQUES

In this chapter I will discuss lossy compression techniques. Lossy compression techniques are techniques that change the data in some way so it will be more compression friendly. Note, this is not the same thing as when we only transform the data. We are actually changing the data, not transforming it. This means that when we decompress the data, the data will not necessary be the same as the original data was. There can be, and in most cases will be, some differences.

The first lossy algorithm I will discuss is called scalar quantization. It is probably the simplest form of the numerous lossy algorithms out there. The second one, vector quantization, is a bit more advanced, but still not very hard to implement.

4.1 Scalar quantization

Imagine that we want to save an audio stream digitally, using three bits, coming from an analog output. Furthermore, these values are in the interval [-7.0, 7.0], where the value 0 means silence. Since the interval [-7.0, 7.0] contains an infinite number of values we would need an infinite number of bits to be able to store all of them, which, of course, is not possible on a computer. What we do is that we map a subinterval to a discrete value or a code instead. For example, the subinterval [-7.0, -5.0] could be mapped to the bit code 000, while the subinterval (-1.0, 1.0] could be assigned to the bit code 101. Let us look at a simple example.

Example 4-1

If we have an analog audio stream whose values are in the interval [-7.0, 7.0] and we want to map these values to bit codes of length three, there are numerous ways we could do this. One way is to divide the interval as shown in Figure 4-1.

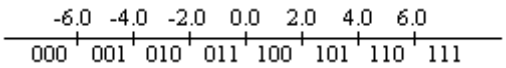


Figure 4-1

So if we have the value 0.73 it would be mapped to the bit code 100 according to Figure 4-1. We could also have used the intervals shown in Figure 4-2.

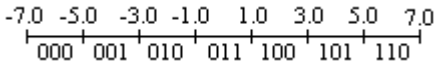


Figure 4-2

In this case the value 0.73 would be mapped to the bit code 011. Which one we choose depends on the situation, as we will see in the decoding procedure.

When we are about to decode a bit code we must choose what this bit code actually means. For example, if we want to decode the bit code 010 from Figure 4-1 we must decide what value it should have in the interval [-4.0, -2.0]. We could for example take the middle value in the interval, in this case –3.0, but we could choose any number we want in the interval. This means that we will in most cases loose some information when we decode the value. For example, if we had the value 4.07 before encoding, this will be mapped to the bit code 110 in the encoding procedure if we have the intervals divided as in Figure 4-1. If we choose the middle value in the decoding procedure, we will get the decoded value 5.0. As we see, we have an error of 0.93 from the correct value, and thereby we have lost some information. This is a typical phenomenon of lossy algorithms.

How we divide the interval is very important. If we were to use the middle value, Figure 4-1 would be a bad choice for the analog audio stream in Example 4-1. This is because if we have silence (the value 0), this value will be mapped to 011 (or 100). So when we decode the bit code 011, we will get the value –1.0, and by that we are unable to get absolute silence. In this case Figure 4-2 would be a much better choice since silence would be mapped to the bit code 011, and in the decoding procedure it would be decoded to the value 0 if we were to use the middle value.

Another important factor to consider is that our data may not be evenly spaced. For example, take a look at Figure 4-3.

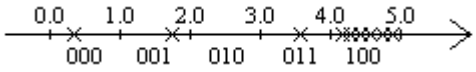


Figure 4-3

Here we can see that we have many values in the interval [4.0, 5.0] while we have very few or none values in the other intervals. In this case it would be a bad idea to map one interval to one value. Instead, it would be better to map the interval [4.0, 5.0] to many values, that is, we divide the interval [4.0, 5.0] to new intervals, as many as we need to. This may result in that we have to skip to map a few other intervals. Which ones depend on what kind of data we are working with. If we look at Figure 4-3 we see that the interval [2.0, 3.0] has no values. So this interval does not need to be mapped, and the bit code assigned to that interval could instead be assigned to the interval [4.0, 5.0]. This means that we have two bit codes assigned to the interval [4.0, 5.0], which in return will give us a closer match to the real values when we decode later on.

4.2 Vector quantization

Vector quantization works much like a dictionary, as those dictionaries we saw in chapter 3.3, except that the entries in this dictionary will always be of the same size. What we do is that we create a number of vectors of some size *D*, and use these vectors as a dictionary. The symbols in the data, we group into vectors of size *D*. We then compare these vectors to the vectors in the dictionary. The closest match we find, we substitute the symbols with its index value in the dictionary. An example will make it easier to grasp.

Example 4-2

Let us say that we have a picture, see Figure 4-4, that we would like to compress using vector quantization.

7	5	4	7
7	5	4	7
7	4	5	7
7	4	5	7

Figure 4-4

If we now were to use a 2-dimensional vector, we could, for example, go from left to right in Figure 4-4. We would then get the vectors (7, 5), (4, 7), (7, 4) and (5, 7), where each entry in the vectors is of size three bits. Furthermore, if we want a dictionary (codebook), containing two vectors, we could, for example, create the codebook

Codebook	
Index	**Vector**
0	(7, 5)
1	(4, 7)

The next step we do is to compare the vectors in the picture with the vectors in the codebook and substitute the vectors in the picture with the index value in the codebook. Since the vector (7, 5) is exactly the same as the vector in position 0 of the codebook, and (7, 4) are very close to the vector (7, 5), we change these vectors to a 0 in the picture. We do the same thing for the vectors (5, 7) and (4, 7), except that these vectors will be substituted to a 1. We can see the result in Figure 4-5.

0	1
0	1
0	1
0	1

Figure 4-5

The result is much smaller than the original data. Of course, to be able to decode these values, the codebook must also be available to the decoder. So if we have a codebook of size *N*, and each entry in the codebook is a vector of size *D*, the total number of bits the codebook needs will be $N \times D \times A_S$ bits, where A_S is the size of each entry in the vector in bits. To be able to access the codebook, each index value will need $\log_2 \lceil N \rceil$ bits. In our example the size of the codebook is $2 \times 2 \times 3 = 12$ bits, and each index value will need $\log_2 \lceil 2 \rceil = 1$ bit.

As we saw from Example 4-2, encoding is not a very difficult task with vector quantization. However, it can be very CPU demanding since we have to create the codebook. On the other hand, the decoding process can be done very quickly since we have the codebook from the beginning. All we need to do is to access the codebook in the place where the encoded values tell us to. So how does the decoding process works? This is best illustrated with an example.

Example 4-3

If we have the same codebook as in Example 4-2 and our data to decode looks like Figure 4-5, we start by replacing each value in Figure 4-5 with its vector in the codebook, that is, each value in Figure 4-5 specifies the index in the codebook. So the first value, 0, means that we should take the first vector in the codebook. The second value, 1, means that we should access the codebook in position two and so on. When we have decoded all the values we will get the result

7	5	4	7
7	5	4	7
7	5	4	7
7	5	4	7

Figure 4-6

As we can see, there are some differences between Figure 4-6 and the original picture, Figure 4-4. This is not surprisingly since vector quantization is a lossy algorithm.

A question that you might have is: "How do we create the codebook?" Well, there are numerous of different algorithms, and we will only be looking at one of them: the k-nearest-neighbor learning algorithm.[1] What we do first is that we create a set of vectors, of size *D*, from our data, for example in the same way as we did in Example **4-2**. The next step is to decide the size of the codebook, that is, how many vectors it

[1] For more information about this algorithm I recommend the book "Russell S., Norvig P., *Artificial Intelligence A Modern Approach*, pages 733ff"

should contain. For obvious reasons, the number of vectors in the codebook should be less than the total number of vectors we create from the dataset. If not, then we will not be able to compress the data, instead we will have an expansion. The larger codebook we have, the more closely we will be able to estimate the original dataset, and this in return will mean that we will need more bits to store the information. On the other hand, if we have a too small codebook, then we might destroy the data too much so we will not have any use for it. After we have decided the size of the codebook, we need to initialize the vectors in the codebook. This is an important step to do, as we will see when we are about to use the k-nearest-neighbor learning algorithm. The easiest way to initialize these vectors is to randomize them. When all this is done, we start to learn the codebook to recognize the structure in the data. We do this by taking the first vector in the codebook and see which vectors in the dataset it is closest to. The k-nearest vectors in the dataset will create a new vector by averaging the k-nearest vectors and the vector from the codebook together. We now take this new vector and replace it with the first vector in the codebook. We do this for all vectors in the codebook. After that, we repeat this process a few times until we have a good estimate of the original data. Let us look at an example.

Example 4-4

Imagine that we have a dataset in which we have created 2-dimensional vectors. We have also initialized the vectors in the codebook by randomizing them. If we would draw this on a plane, it may look something similar with Figure 4-7.

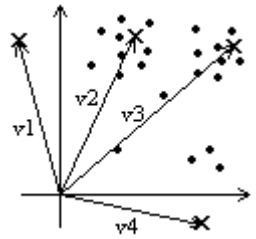


Figure 4-7

The vectors v1, v2, v3 and v4 belong to the codebook, while the dots are part of the dataset. If we now use the k-nearest-neighbor learning algorithm, in which we have chosen $k = 3$, we would get something like Figure 4-8.

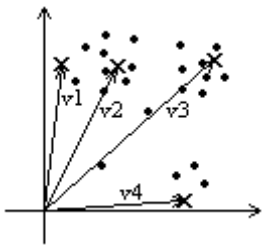


Figure 4-8

Continuing in this way, the vectors from the codebook will at the end have a good approximation of the original dataset. Note however that some dots in Figure 4-8 are very far from the approximation and will therefore be lost.

As I have said before, it is very important how we initialize the codebook at the beginning. For example, if we had chosen the four vectors in Figure 4-7 to be the same at the beginning, they would also be the same in the end of the k-nearest-neighborhood learning algorithm, no matter how many iterations we would do.

5 USING LOSSY ALGORITHMS IN LOSSLESS COMPRESSION

As I have written before, compression algorithms can be divided into two sections: lossless ones and lossy ones. On the other hand, we can combine them to achieve higher compression ratio. Remember that lossy algorithms removed some information from the source to be able to compress it better. This in return leads to files that are much smaller than we would be able to get with lossless algorithms. Furthermore, in chapter 2 we looked at an example in which we took the difference from each symbol and compressed that output to achieve higher compression. We can do the same thing using a lossy algorithm. What we do is that we first compress the data using a lossy algorithm. After that we take the difference between the original data and the uncompressed lossy data. This difference we compress using a lossless algorithm. So our compressed data will contain the lossy data, and the difference compressed using a lossless algorithm. An example on how this works will make it more understandable.

Example 5-1

Let us use the same figures and codebook as we used in Example 4-2 and Example 4-3 in chapter 4. Figure 4-4 shows the original picture, while Figure 4-6 shows the uncompressed picture after we have used a lossy algorithm on it. If we now take the difference between these two pictures, we will get the following table:

Table 5-1

0	0	0	0
0	0	0	0
0	-1	1	0
0	-1	1	0

As we can see, this table has many zeroes, which means that we should be able to find a lossless algorithm that can compress it pretty much. We now store the codebook, the lossy data (Figure 4-5), and the difference (Table 5-1). Since the codebook needs 12 bits, and the lossy data needs 8×1 bits, the total number of bits needed is 20 bits for the lossy part. The original picture took $3\times16=48$ bits, meaning that we have 28 bits that we can use for the lossless algorithm. How many bits we actually need, for the lossless part (Table 5-1), depend on which lossless algorithm we choose to use.

The result in Example 5-1 may not look too impressive. However, when dealing with a much larger dataset, we will see, in chapter 6, that using a lossy algorithm in lossless compression can help us to increase the compression ratio substantially.

6 RESULTS

This chapter will compare the different algorithms that I have described earlier with each other. The algorithms were implemented in the same way as I have described them in this thesis. Before we look how well they did against each other, I need to discuss some details on how I have chosen to implement the different algorithms and how I saved the overhead if the algorithm required such.

- **RLE**: We will look at all three versions of the RLE algorithm that I discussed in chapter 3, section 3.1.1. I will call them: RLE, RLESgn and RLEDbl. All three algorithms will be using one byte that specifies how many symbols that are repeated. None of them requires any overhead.

- **Huffman coding**: Here I am using an ordinary Huffman algorithm. However, the overhead is calculated as if I would have created the tree using the Canonical Huffman algorithm. The size of the alphabet is 256.

- **Arithmetic coding**: This algorithm is implemented using integers.[1] The overhead will require a total of $A_S \times \log_2 \lceil \max(freq) \rceil$ bits, where A_S is the size of the alphabet, which in this case is 256, and $\max(freq)$ is the frequency of the most occurring symbol.

- **LZ77**: I will not be using this algorithm for the test data since the LZSS algorithm is based on the LZ77 algorithm and works in the same way except that it stores the encoded symbols much better. That is, we know that the LZSS algorithm will perform better than the LZ77 algorithm.

- **LZSS**: The size of the search section and the look-ahead section will be 256 symbols each. No overhead is required.

- **LZ78**: This algorithm is using a dictionary with a capacity of 1000000 entries. Furthermore, the index value will need $\log_2 \lceil csd \rceil$ bits, where *csd* is the current size of the dictionary when writing the index value. For example, if we find a string at entry two in the dictionary, and the size of the dictionary is 59, we will need six bits to store this number. When the dictionary is filled, it will be emptied, and the process will be started over in filling it. No overhead is required.

- **LZW**: The LZW algorithm will have the same capacity of the dictionary as the LZ78 does. I will also be using the same technique of the index values as I used in LZ78. Furthermore, when the dictionary is filled, it will be emptied. No overhead is required.

- **MTF:** When I use this algorithm, I will assume that the alphabet is of size 256. The algorithm will always use the same initialization at the start of the test data. No overhead is required.

- **BWT:** In this algorithm I use a sequence length of 5000 symbols. No overhead is required.

[1] For details of this technique, see the book "Sayood K., *Introduction to Data Compression*, pages 97ff"

- **BCCBT:** The BCCBT algorithm that I developed in chapter 3, section 3.5, is using an alphabet of size 256. I will use a modified version of this algorithm that is using an implementation of the RLE algorithm. Furthermore, the overhead will be 256 bytes since the algorithm require us to store the order of the symbols in the alphabet.

- **Adaptive BCCBT:** This algorithm is implemented in the same way as I described it in chapter 3, section 3.6. The only thing I will add to it is an RLE algorithm. No overhead is required.

- **Scalar quantization:** This algorithm will not be used.

- **Vector quantization:** Nor will this algorithm.

The test data will be five different files: an application file (winlogon.exe for XP Pro SP2), one text file (a couple of pages from the book "Fellowship of the ring" by J.R.R. Tolkien), one audio file (Metallica – Nothing else matters, 44100 Hz, stereo, 16 bits) and two images: one fractal image (size: 1024x768 24 bits) and a postcard with some trees and flowers (size: 556x788 24 bits).

When I use the BWT algorithm I will always use the MTF algorithm after the transformation. Furthermore, the level sequence in the BCCBT algorithms will always be transformed using the BWT algorithm, unless I say something else. After that, the transformed level sequence will be compressed with the different algorithms. The bit code sequence however, will not be transformed or compressed.

All values in the tables on the next pages specify the file size, in bytes, after using the different compression algorithms.

WINLOGON.EXE (file size: 502 272 bytes)

Algorithm	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Size	869522	481591	483135	422029	419451	399580	391910	390148
Overhead	0	0	0	128	544	0	0	0
Total	869522	481591	483135	422157	419995	399580	391910	**390148**

Algorithm with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Size	684896	403510	406674	331806	328998	402560	370170	381805
Overhead	0	0	0	128	576	0	0	0
Total	684896	403510	406674	331934	**329574**	402560	370170	381805

BCCBT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	231866	231866	231866	231866	231866	231866	231866	231866
Level sequence	392540	230223	232404	142566	141789	227226	175985	165913
Overhead	256	256	256	416	768	256	256	256
Total	624662	462345	464526	374848	**374423**	459348	408107	398035

Adaptive BCCBT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	234661	234661	234661	234661	234661	234661	234661	234661
Level sequence	427034	237745	239033	155461	154714	244881	192698	182407
Overhead	0	0	0	160	480	0	0	0
Total	661695	472406	473694	390282	**389855**	479542	427359	417068

BCCBT with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	164764	164764	164764	164764	164764	164764	164764	164764
Level sequence	391204	207276	207228	153089	152376	224548	187883	184494
Overhead	256	256	256	416	704	256	256	256
Total	556224	372296	372248	318269	**317844**	389568	352903	349514

Adaptive BCCBT with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	196780	196780	196780	196780	196780	196780	196780	196780
Level sequence	389886	208075	208237	146991	146291	223859	182282	176546
Overhead	0	0	0	160	448	0	0	0
Total	586666	404855	405017	343931	**343519**	420639	379062	373326

If we look at the two tables at the top, we can see that the arithmetic algorithm together with BWT compressed this file better than the other ones of the ordinary algorithms. However, when I used the BCCBT algorithm together with BWT I manage to compress it with another 11730 bytes. Even when I did not use the BWT algorithm, we can see that the BCCBT algorithm performed better than the other algorithms. Let us take a look at the frequency of each symbol (Figure 6-1).

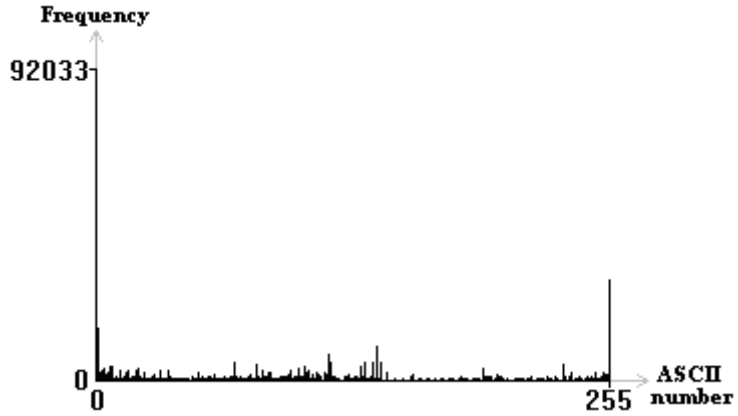


Figure 6-1

By looking at Figure 6-1 we see that the file contains many symbols with the ASCII numbers 0, 1 and 255, that is, we have a high probability of these symbols. But if we look at the first table we see that LZSS, LZ78 and LZW algorithms performed better than the other algorithms in that table. This is telling us that the file contains many patterns, and therefore the LZ*-family is able to recognize more structure in the data and by that managed to compress the file better. This is not a big surprise since the file is an executable file which probably contains many commands like “mov reg1, reg2”, “add reg1, reg2” and other frequently used commands. But after I used the BWT algorithm we can see that the algorithms based on probability of each symbol performed better than the algorithms based on patterns. Let us look at the frequency of each symbol after using the BWT algorithm (Figure 6-2)

Figure 6-2

We now see that the BWT algorithm (together with MTF) has changed the data to contain many more symbols with a lower ASCII number, and has also increased the probability of the symbol with the ASCII number 0 with a factor of more than two, meaning that the algorithms based on probability should be able to compress the data better than before, as we can see by looking at the second table from the top. Since the algorithm BCCBT is based on the probability of each symbol, this is also true for that algorithm, as we can see by comparing the third table with the fifth table. Furthermore, the adaptive BCCBT is doing pretty well, although not as good as the ordinary BCCBT algorithm. If we look at the RLE algorithms, we can draw the conclusion that there are symbols that are repeated, probably the symbol with the ASCII number 0 and maybe a few more. We can also see that the RLE algorithm is failing to compress the file at all, while the two others are doing a pretty good job, but usually not as good as the other algorithms.

TOLKIEN.TXT (file size: 33 018 bytes)

Algorithm	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Size	64588	33777	33740	18586	18420	28138	19759	15874
Overhead	0	0	0	160	416	0	0	0
Total	64588	33777	33740	18746	18836	28138	19759	**15874**

Algorithm with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Size	48000	30593	31118	15082	14832	27395	19984	17635
Overhead	0	0	0	160	448	0	0	0
Total	48000	30593	31118	**15242**	15280	27395	19984	17635

BCCBT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	7906	7906	7906	7906	7906	7906	7906	7906
Level sequence	30240	17110	17162	9889	9824	17531	13417	12257
Overhead	256	256	256	416	640	256	256	256
Total	38402	25272	25324	**18211**	18370	25693	21579	20419

Adaptive BCCBT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	11615	11615	11615	11615	11615	11615	11615	11615
Level sequence	31292	17225	17272	9681	9638	18057	13385	11983
Overhead	0	0	0	160	352	0	0	0
Total	42907	28840	28887	**21456**	21605	29672	25000	23598

BCCBT with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	5204	5204	5204	5204	5204	5204	5204	5204
Level sequence	29396	15687	15690	10392	10338	16935	13776	13011
Overhead	256	256	256	416	576	256	256	256
Total	34856	21147	21150	**16012**	16118	22395	19236	18471

Adaptive BCCBT with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	7719	7719	7719	7719	7719	7719	7719	7719
Level sequence	29188	15809	15832	9763	9715	16832	13178	12124
Overhead	0	0	0	160	320	0	0	0
Total	36907	23528	23551	**17642**	17754	24551	20897	19843

This file contains human language written in English. Since the human language is built on words, the dictionary algorithms should be able to compress it pretty good. Also, some characters are used more often than others, so the probability algorithms should also be able to decrease the file size. By looking at the two tables at the top, we can see that our statements hold true. The BCCBT algorithm is also doing a good job, but it is not the best of the different algorithms. The RLE algorithms are not very good to use for this file compared to the other algorithms. This means that the file does not contain many repeated symbols, although we get a few more repeated symbols after using the BWT algorithm. Let us look at the frequency of each symbol when not using the BWT algorithm (Figure 6-3).

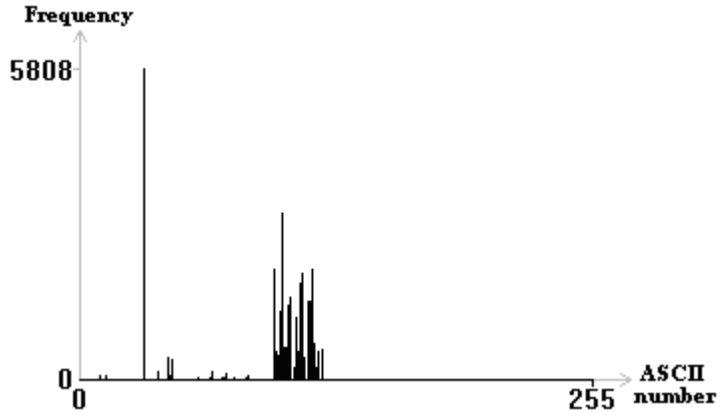


Figure 6-3

Here we can see that we are using only a fraction of the 255 different ASCII numbers available to us. Still, we can see that some characters are used more often than others, while some character are not used at all, or at least very rarely. Now, let us look at how the data has changed after using the BWT algorithm (Figure 6-4).

Figure 6-4

Again we see that the BWT algorithm (together with MTF) has increased the probability of the symbol with ASCII number 0, which is confirmed by looking at the second table from the top, since the probability algorithms performed much better after using the BWT algorithm. On the other hand, we can see that the LZW, for example, is not performing as well as it did when I did not use the BWT algorithm, meaning that the BWT algorithm has actually destroyed some structures in the data.

METALLICA.WAV (file size: 68 577 308 bytes)

Algorithm	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Size	136062906	69034236	68642029	63967081	63789997	76271204	68991554	74930466
Overhead	0	0	0	128	704	0	0	0
Total	136062906	69034236	68642029	63967209	**63790701**	76271204	68991554	74930466

Algorithm with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Size	133163940	69208296	69067872	65682731	65435235	75294491	72247453	75261123
Overhead	0	0	0	128	704	0	0	0
Total	133163940	69208296	69067872	65682859	**65435939**	75294491	72247453	75261123

BCCBT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	39123788	39123788	39123788	39123788	39123788	39123788	39123788	39123788
Level sequence	59829366	35072602	35494253	19578744	19418860	34092837	23455006	21530121
Overhead	256	256	256	416	992	256	256	256
Total	98953410	74196646	74618297	58702948	**58543640**	73216881	62579050	60654165

Adaptive BCCBT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	41261973	41261973	41261973	41261973	41261973	41261973	41261973	41261973
Level sequence	64017376	35458717	35608538	21192517	21056618	36551135	25516698	23611740
Overhead	0	0	0	160	704	0	0	0
Total	105279349	76720690	76870511	62454650	**62319295**	77813108	66778671	64873713

BCCBT with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	41748263	41748263	41748263	41748263	41748263	41748263	41748263	41748263
Level sequence	65088934	35251191	35317685	22596924	22482978	37520147	27375463	25809502
Overhead	256	256	256	416	960	256	256	256
Total	106837453	76999710	77066204	64345603	**64232201**	79268666	69123982	67558021

Adaptive BCCBT with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	43735326	43735326	43735326	43735326	43735326	43735326	43735326	43735326
Level sequence	64579354	35405051	35498595	21558842	21427637	37172492	26235626	24424520
Overhead	0	0	0	160	704	0	0	0
Total	108314680	79140377	79233921	65294328	**65163667**	80907818	69970952	68159846

For this file the BCCBT algorithm performed much better than the regular algorithms. Using the BCCBT algorithm and the arithmetic algorithm on the level sequence, we see that I was able to decrease the file size with another 5 247 061 bytes, approximately 5 MB. The RLE, RLESgn, RLEDbl, LZSS, LZ78 and LZW algorithms all caused an expansion of the file instead, meaning that it is hard to find any repeated symbols and patterns in the file. Let us look at the frequency diagram (Figure 6-5).

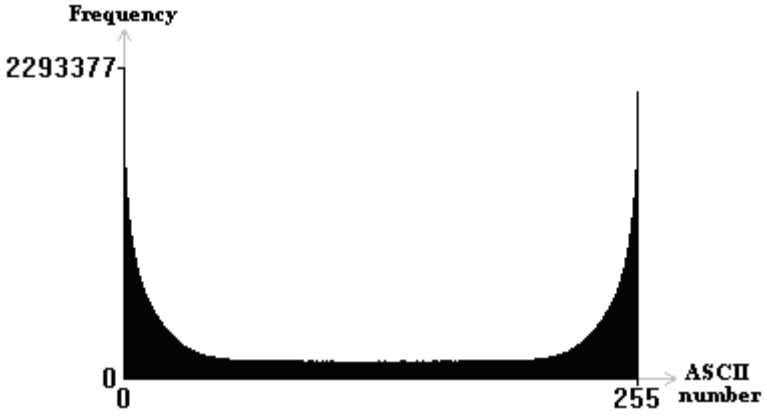


Figure 6-5

We can see that we have a high probability of low numbers and of high numbers, while we have a low probability of the values in-between, meaning that the algorithms based on the probability of each symbol should do quite well. And by looking at the tables we can see that this is true. An interesting thing is that when I use the BWT algorithm (together with MTF), it destroys the structure even more in the file. Furthermore, the BWT algorithm is changing the probability of each symbol to the worse, so the algorithms based on the probability of each symbol perform not as good as without using the BWT algorithm. Let us see how the frequency diagram looks like after using the BWT algorithm (Figure 6-6).

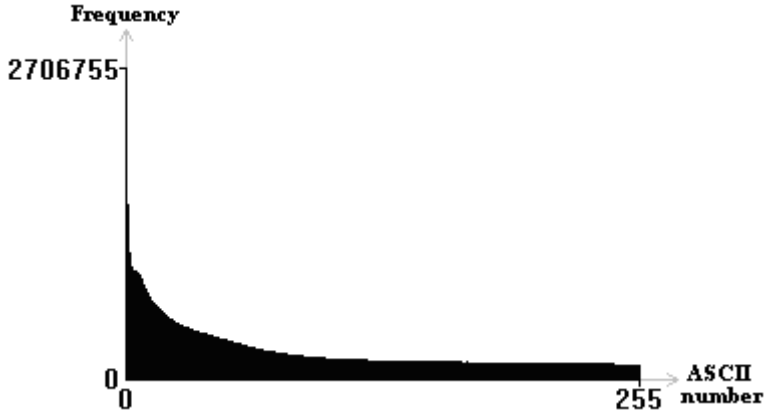


Figure 6-6

By looking at Figure 6-6, we see that the BWT algorithm has changed many of the symbols with a high ASCII number to a lower ASCII number. And if we look at Figure 6-5, we see that we had a high probability of the symbols whose ASCII number was greater than 250 before we used the BWT algorithm. So in this case the BWT algorithm actually changed the data to a state not favourable for the probability algorithms.

FRACTAL.BMP (file size: 2 359 350 bytes)

Algorithm	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Size	4713464	2379511	2361928	2021130	2012695	795064	490011	349155
Overhead	0	0	0	160	512	0	0	0
Total	4713464	2379511	2361928	2021290	2013207	795064	490011	**349155**

Algorithm with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Size	1083968	793178	812460	622180	520577	704248	572087	559217
Overhead	0	0	0	160	672	0	0	0
Total	1083968	793178	812460	622340	**521249**	704248	572087	559217

BCCBT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	1219256	1219256	1219256	1219256	1219256	1219256	1219256	1219256
Level sequence	820006	570507	597751	325130	303646	471605	337845	298769
Overhead	256	256	256	416	896	256	256	256
Total	2039518	1790019	1817263	1544802	1523798	1691117	1557357	**1518281**

Adaptive BCCBT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	615163	615163	615163	615163	615163	615163	615163	615163
Level sequence	873568	579907	594990	354247	338753	505947	365142	325224
Overhead	0	0	0	160	640	0	0	0
Total	1488731	1195070	1210153	969570	954556	1121110	980305	**940387**

BCCBT with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	273872	273872	273872	273872	273872	273872	273872	273872
Level sequence	739130	411667	413029	281605	279912	421852	338923	328298
Overhead	256	256	256	160	512	256	256	256
Total	1013258	685795	687157	555637	**554296**	695980	613051	602426

Adaptive BCCBT with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	324015	324015	324015	324015	324015	324015	324015	324015
Level sequence	750512	413356	414902	284522	283131	426380	344863	331916
Overhead	0	0	0	160	512	0	0	0
Total	1074527	737371	738917	608697	**607658**	750395	668878	655931

As we can see, the LZW algorithm managed to compress this file way better than the other algorithms. The reason for this is that this file is an image of a fractal, and since a fractal is created using mathematics, there are a lot of structure in the data that the LZW algorithm recognizes. We can also see that the adaptive BCCBT algorithm actually performs better than the ordinary BCCBT algorithm for the first time when not using the BWT algorithm. Let us take a look at Figure 6-7 that is showing the frequency of each symbol.

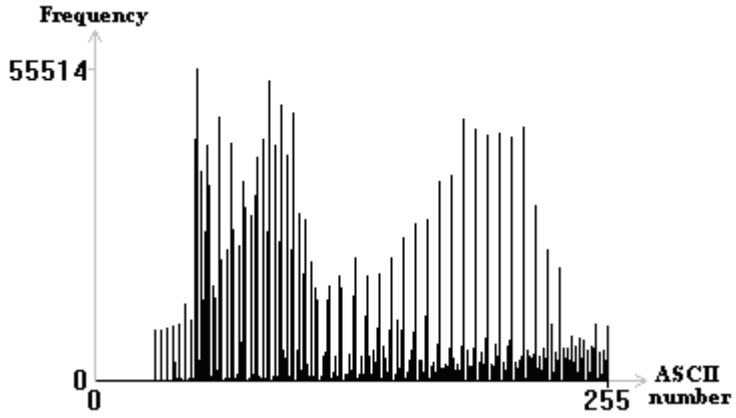


Figure 6-7

We see that we have many symbols that have almost the same probability, and this is the reason why the probability algorithms do not perform as well as, for example, the LZW algorithm. The frequency diagram after using the BWT algorithm (together with MTF) looks like Figure 6-8.

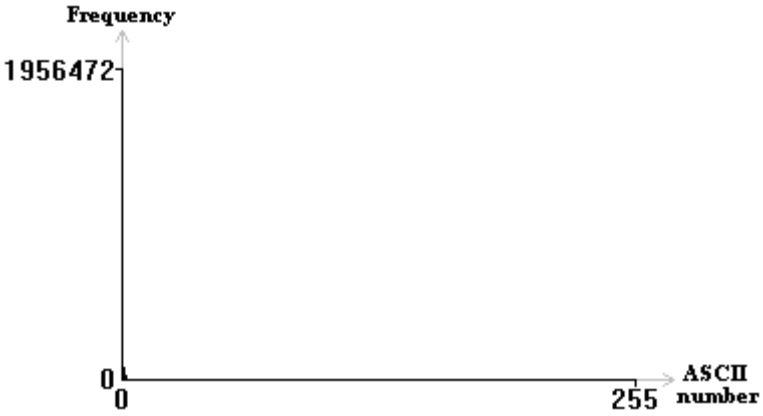


Figure 6-8

By looking at this diagram and looking at the result of the Huffman algorithm and the arithmetic algorithm in the second table from the top, we can see that they perform quite well, almost as good as when we used the LZW algorithm without using the BWT algorithm. However, we can also see that the arithmetic algorithm managed to decrease the size of the file with 101 091 bytes compared to the Huffman algorithm. Remember that I talked about skewed probabilities in chapter 3 when I discussed the Huffman algorithm? Well, this is an example of this, and by looking at Figure 6-8 we can see that the probabilities of the different symbols are very skewed. Note that even if we cannot see it in this diagram, there do exist symbols in the changed data that have an ASCII number greater than two.

FLOWERS.BMP (file size: 1 314 438 bytes)

Algorithm	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Size	2346492	1263010	1261390	1148062	1143352	1356896	1215550	1211607
Overhead	0	0	0	160	576	0	0	0
Total	2346492	1263010	1261390	1148222	**1143928**	1356896	1215550	1211607

Algorithm with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Size	2410832	1273922	1272910	1081002	1076175	1375573	1245244	1259737
Overhead	0	0	0	160	576	0	0	0
Total	2410832	1273922	1272910	1081162	**1076751**	1375573	1245244	1259737

BCCBT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	700116	700116	700116	700116	700116	700116	700116	700116
Level sequence	1089134	617478	623445	385287	382627	621977	466264	434495
Overhead	256	256	256	416	800	256	256	256
Total	1789506	1317850	1323817	1085819	**1083543**	1322349	1166636	1134867

Adaptive BCCBT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	673835	673835	673835	673835	673835	673835	673835	673835
Level sequence	1202560	635721	635850	442691	440358	691856	548043	521044
Overhead	0	0	0	160	480	0	0	0
Total	1876395	1309556	1309685	1116686	**1114673**	1365691	1221878	1194879

BCCBT with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	614105	614105	614105	614105	614105	614105	614105	614105
Level sequence	1225484	645102	644958	464785	462544	703913	572197	550339
Overhead	256	256	256	416	736	256	256	256
Total	1839845	1259463	1259319	1079306	**1077385**	1318274	1186558	1164700

Adaptive BCCBT with BWT	RLE	RLESgn	RLEDbl	Huff	Arith	LZSS	LZ78	LZW
Bit code sequence	690929	690929	690929	690929	690929	690929	690929	690929
Level sequence	1221142	647736	647939	438702	436259	703304	544949	514735
Overhead	0	0	0	160	512	0	0	0
Total	1912071	1338665	1338868	1129791	**1127700**	1394233	1235878	1205664

The best result I got from this file was when I used the arithmetic algorithm together with the BWT algorithm, although the BCCBT algorithm together with the BWT algorithm is not very far behind. But we can also see that I did not manage to compress this file very much. The reason for this is that this file contains an image with natural objects. Therefore it is very hard to find any structure in the file, and because of this the dictionary techniques will not be able to compress the file much.

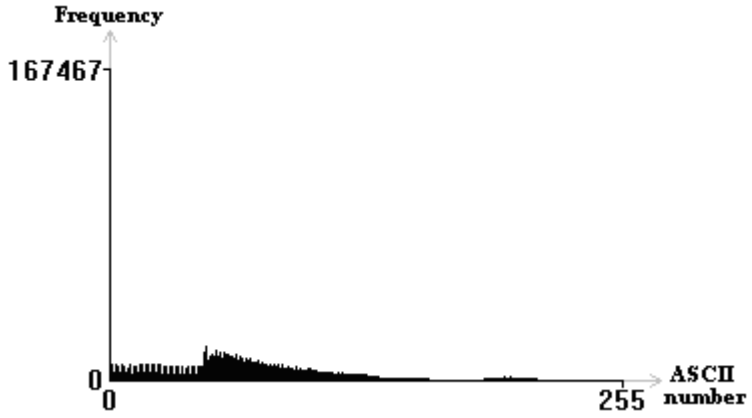


Figure 6-9

By looking at the frequency diagram (Figure 6-9), we see that we have almost the same probability for every symbol except for the symbol with the ASCII number 0. Since the RLESgn and RLEDbl algorithms managed to compress the file, we can be pretty sure that we have some repeated symbols with the ASCII number 0. The frequency diagram after using the BWT algorithm (together with MTF) looks like Figure 6-10.

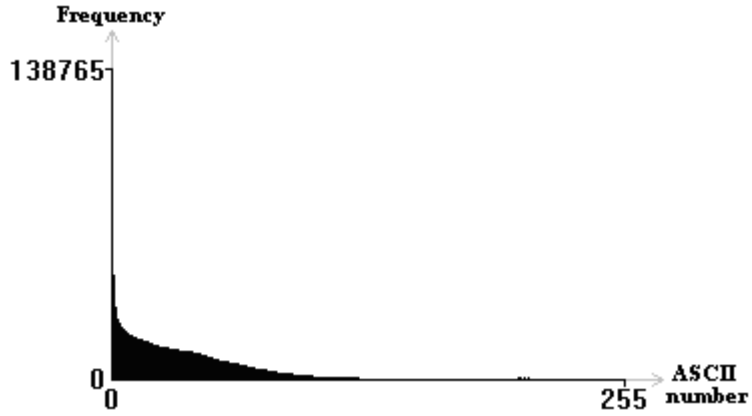


Figure 6-10

Studying the frequency diagram, we can see that the BWT algorithm is doing a pretty good job in changing the data so it contains more lower values. We can see the result of this by comparing the two tables at the top. Since the BWT algorithm is increasing the probability of symbols with a lower ASCII number, the algorithms based on the probability of each symbol is performing better after I have used the BWT algorithm. On the other hand, looking at the tables, we can see that the BWT algorithm is destroying the structure in the data so the dictionary techniques do not perform as well as they did before.

Let us now see if we can increase the compression ratio, by using the technique that I discussed in chapter 5, on the files flowers.bmp and metallica.wav.

First I used a jpeg algorithm to save the image flowers.bmp. The file size I got from this was 106 039 bytes. Then I took the difference between the pixel values of the original image and the "lossy" image. We can see the result of this in the frequency diagram (Figure 6-11).

Figure 6-11 The frequency of each symbol of the difference data

The result I got from this process I compressed with the BCCBT algorithm; the level sequence was compressed using the arithmetic algorithm, that is, I did not use the BWT algorithm. The size of this, with overhead and everything, was 863 752 bytes. So the total file size I got was 863 752+106 039=969 791 bytes. As we can see, by looking at the second table for the file flowers.bmp, I managed to decrease the file size with another 106 960 bytes using this technique.

For the file metallica.wav I used an mp3 algorithm. After that I converted it to a wav file again, and took the difference between the values of the original file and the "lossy" file. The result of taking the difference between the two files can be seen in Figure 6-12.

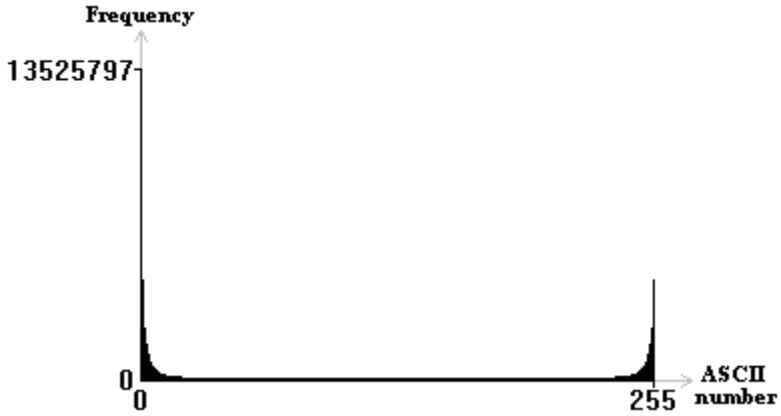


Figure 6-12 The frequency of each symbol of the difference data

The size of the mp3 file was 6 216 307 bytes. The difference I compressed in the same way I compressed the difference with the file flowers.bmp. The file size I got from this was 48 084 489. If we add these two numbers together we get a file size of 54 300 796 bytes. If we now compare this number with the best result we have in the tables for the file metallica.wav, we can see that I managed to decrease the size of the file with another 4 242 844 bytes, or approximately 4 MB.

As we see, using a lossy algorithm together with a lossless algorithm can help us to increase the compression ratio even further. The better a lossy algorithm can recreate the data to the original, the smaller values we will get when we take the difference between the “lossy” data and the original data, and thereby we are able to compress the difference even more using a lossless algorithm.

7 CONCLUSION

In this thesis I have discussed many of the more popular algorithms; I also explained the new compression algorithm, which I chose to call BCCBT. But I have only scratched the surface of this exciting field. There are literally hundreds (perhaps even thousands) of different compression algorithms out there. To discuss all of them is not possible in a thesis like this one, and was not the purpose to do so either. Instead, the purpose was to compare the more common algorithms against each other, and see if we could develop a new compression algorithm and compare this one to the more common algorithms. This new algorithm was explained in chapter 3, section 3.5, and I ran some test results of it in chapter 6. As we could see from the results, this new algorithm did quite well, and in a few cases it managed to compress the data much better than the other algorithms. I also discussed, in chapter 5, a way to increase the compression ratio by using lossy algorithms in lossless compression. In chapter 6 we could see that by using this technique we were able to increase the compression ratio even further on some of the test data.

When we are creating a new compression algorithm, we have to decide what the purpose is for the algorithm. Should it be specialized on one type of data, or should it be able to compress all kinds of data. This means that we have to know how the data looks like. We looked at a few examples of this in chapter 2 when I discussed techniques that were based on probabilities, dictionaries or transformations. For example, in chapter 6 I used two images. These images were in 24 bits colour format, saved as a bmp. The bmp format saves the colours as [BGR][BGR]…[BGR], where B means blue, G means green and R means red. When I used the algorithms, I looked at the data as [BGR][BGR]…[BGR]. A better way could have been to look at it like [BB…B][GG…G][RR…R] instead.

Future work might be, of course, to develop new algorithms than can compress the source even further than today's algorithms, or to change the algorithms, in one way or the other, that currently exist. But time and memory might also be two important factors. If an algorithm is compressing a source much better than all other algorithms, but it takes a forever to do so, then it is worthless to us. We do not only need algorithms that can compress the source, but we also need algorithms that are time and memory efficient, depending on where they are supposed to be used.

A PSEUDOCODE OF THE BCCBT ALGORITHM

ENCODING

```
Get the frequency of each symbol from the input stream
Set the frequency table to the frequency of each symbol
Create a complete binary tree using the frequency table
Set the bit codes according to where the symbols are in the tree
While more symbols to read from the input stream
      Read one symbol from the input stream
      Write the symbol's bit code to the bit code stream
      Write the length of the bit code to the level stream
Compress the level stream with a lossless algorithm
Write the frequency table to the output stream
Write the compressed level stream and the bit code stream to the output
stream
```

DECODING

```
Read the frequency table from the input stream
Create a complete binary tree using the frequency table
Read the compressed level stream from the input stream
Uncompress the compressed level stream
Read the bit code stream from the input stream
While more levels to read from the level stream
      Read one level from the level stream
      Read level bits from bit code stream
      Find the symbol in the complete binary tree using the level and
the bit code
      Write the symbol to the output stream
```

B PSEUDOCODE OF THE ADAPTIVE BCCBT ALGORITHM

ENCODING

```
Initialize the complete binary tree
While more symbols to read from the input stream
      Read one symbol from the input stream
      Get the symbol's bit code from the complete binary tree
      Write the symbol's bit code to the bit code stream
      Write the length of the bit code to the level stream
      Move the symbol to the root node of the complete binary tree
Compress the level stream with a lossless algorithm
Write the compressed level stream and the bit code stream to the output
stream
```

DECODING

```
Initialize the complete binary tree
Read the compressed level stream from the input stream
Uncompress the compressed level stream
Read the bit code stream from the input stream
While more levels to read from the level stream
      Read one level from the level stream
      Read level bits from bit code stream
      Find the symbol in the complete binary tree using the level and
the bit code
      Move the symbol to the root node
      Write the symbol to the output stream
```

REFERENCES

Books

Sayood, K. (2000) *Introduction to data compression*, 2nd edition, Morgan Kaufmann Publishers

Russell, S., Norvig, P. (2003) *Artificial Intelligence: A Modern Approach*, 2nd edition, Pearson Higher Education

Standish, T. (1997) *Data structures in Java™*, Addison-Wesley Pub Co

Baase, S., van Gelder, A. (1999) *Computers Algorithms: Introduction to Design & Analysis*, 3rd edition, Addison-Wesley Pub Co

Web addresses

Campos, A. (2000) 'Arturo Campos Home page', available from Internet <http://www.arturocampos.com> (15 November 2004)

van Schalkwyk, J.M. (2001) 'Huffman Encoding', available from Internet <http://www.anaesthetist.com/mnm/compress/huffman/> (15 November 2004)

Marshal D. (2001) 'Lempel-Ziv-Welch (LZW) Algorithm', available from Internet <http://www.cs.cf.ac.uk/Dave/Multimedia/node214.html> (28 November 2004)

Nelson M. (1998) 'Data Compression with the Burrows-Wheeler Transform', available from Internet <http://dogma.net/markn/articles/bwt/bwt.htm> (6 December 2004)

Dr. Jürgen A. (2004) 'Move To Front (MTF)', available from Internet <http://www.data-compression.info/Algorithms/MTF> (6 December 2004)

Gilchrist J. (2005) 'Jeff Gilchrist's compression.ca', available from Internet <http://compression.ca/> (19 January 2005)