FAQ for the LZSS Compression

Nemesis - Terminus Traduction - 2003
Translated to English by DaMarsMan (Kojiro Translations 2005)
Revised by DaMarsMan June 06, 2006

Table of Contents :

What is it used with ? (Why are you reading this FAQ?)

First of all, this is the type of compression that is used mostly for text. It can be used in other cases but is more practical for compressing text and of course you'll understand why from reading on.

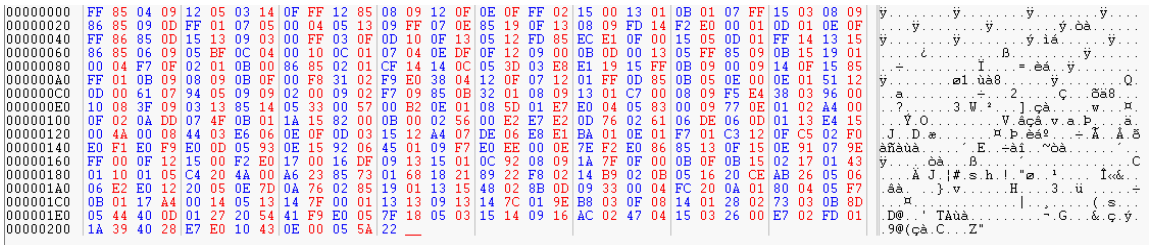
Let's use some sentences for an example.

- The dog sees the cat across the street.
- The dog crosses the street.
- The dog gets flattened.
- The cat benefits.

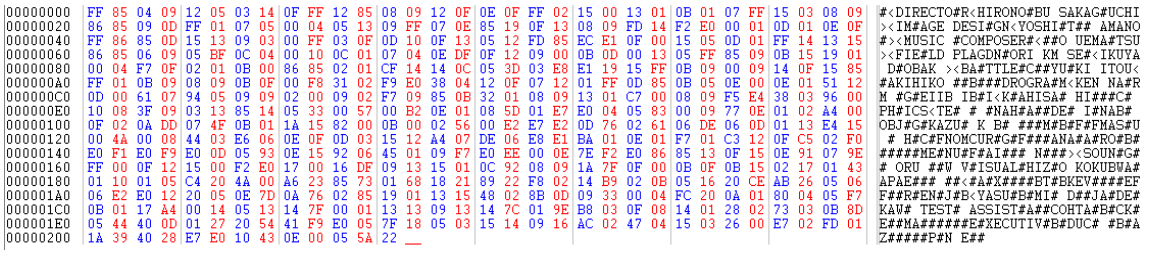
This is a pretty simple example but it does the job and gets the point across. One might observe that the "cat", "dog" and "street" are repeated several times. You can see a lot of repeated patterns in those sentences. The more times the word is repeated the better the compression. Take for example, the word " it" may appear far more often than "however".
This is how it is when it comes to data processing. You can compress everything or individual parts. Single characters can be compressed or even entire sentences.

How does this compression work ? (in general)

We will start with looking at decompression in its simplest form and that will help us to get an idea of why the final texts looks like it does.



Look at the text above. It's far from readable. Actually its the credits from Final Fantasy V for the SNES. This is before using the table file. You do know what tables are, right ? I mean, you are reading a document about LZSS. I would think you have worked with at least one table before. Aft er the table is applied the result looks like below.



A little better but still not completely readable ;). Look at the first character (It's made of 8 bits....re member this, it's important). Look at the pattern. You will see 8 readable characters and then an odd character. Then another 8 readable and so on....

Take a look at the second line where it says YOSHIT?? AMANO . If you know your Squaresoft the n you know that this is talking about Yoshitaka Amano. Now you may notice that it seems to have actually lost compression here. The compression added an extra character which seems useless.

Let's find an explanation for the extra character. Look at the characters. There is FF, FF and then FD.
In binary this is : 11111111 then 11111101 (8 bits for the next 8 characters or character strings).
When the byte is flipped you get this : 11111111 and 10111111
So, as you can see, a 1 corresponds to a character which is alone or decompressed we could say, which has been the case thus far. A 0 corresponds to compressed chain of characters that has already appeared somewhere in the text before. Do you remember seeing the sentences fro m the beginning. This is the missing AKA from SAKAGUCHI.
The binary codes, therefore, determine which bytes correspond to chains of characters (compressed) and which bytes correspond to single characters (uncompressed).

With the characters, you also find "beacons" which can allow for data fields of variable size. In ge neral, there is one byte per uncompressed character and two per coded chain.

Well I keep bringin up the word chain but how is the chain composed? There could be a number o f ways but it usually is coded in 2 bytes. Those would be the posistion of the chain of characters a nd its length. It's better to see it on a diagram. Getting this stuff yet?

Let's suppose we have compressed the beginning of the text already up until the end of "YOSHI". Let's start at "TAKA AMANO".

1) We would probably look first at the TAK that we have seen before. It would be pointless to look for less the 3 characters considering it takes 2 bytes to code the chain in return. Also, you may lo ok at previous data now, but at the time of decompression you won't even know what it is. Underline represents already compressed data.

<DIRECTOR<HIRONOBU SAKAGUCHI><IMAGE DESIGN<YOSHITAKA AMAN O>

2) Since we had no luck with TAK, that means we consider T as a normal character and move on.

<DIRECTOR<HIRONOBU SAKAGUCHI><IMAGE DESIGN<YOSHITAKA AMAN O>

3) As you see AKA will work. We must move on to test the forth character. Searching for AKAG bri ng no luck. Three characters will have to do.

4) If you had found a continuation, it could go up to 5,6, or even more characters. Then you could even have Mr ORIAKA AMANA which could be inserted HIRONUBU and YOSHITAKA. (If there i s already a clone earlier then why not?) Then, if you stopped on SAKAGUCHI you wouldn't lose c haracters badly.

5) This isn't the case here, but the only way of testing is to go through all the text up to that posisti on and keep in memory the best found matches.

6) Finally, if you took a close look you should have found in position 21, 3 identical characters. Yo u then replace the new AKA found by the couple (21,3) coded out of 2 bytes. After that, you move on to "AMANO..."

In a less complex way, you break up and check in the following manner:

To be compressed	Compressed text
ABBABBBABABBB	
BBABBBABABBB	A
BABBBABABBB	AB
ABBBABABBB	ABB
BABABBB	ABB(1,3)
ABBB	ABB(1,3)(3,3)
	ABB(1,3)(3,3)(4,4)
13 octets	9 octets

Beware! Text like the chain BBBB can be coded efficiently by B (1,3). It is risky for the program like this but it actually works well when used. In that case you have to retrieve the characters one by one. It grabs each character starting from the one before as it decompresses. This case is seldom taken advantage of however.

That's it for the general information. That concludes the FAQ ;). Now we will look at how the texts are coded with computers and some alternative methods and characteristics.

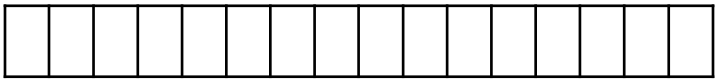
Details on it's coding :

1. Binary Coding :

First, let's look at where the bit combinations are in the coding of the couple (posistion, length). As you may already suspect, you are far more likely to find short similar chains everywhere in the tex t instead of huge blocks of text all in the same place. This is why you need to look at the 16 bits a vailable (2 bytes). Common LZSS uses 4 bits for length of chain and 12 bits for its posistion. 4 bit s for length + 12 bits position = 16 bits or 2 bytes. Some other common methods include 5/11 or 3 /13.

Pay attention to the way to bits are arranged. Some times they can be pretty weird and variable a s you see in the examples. In the example, blue is the bits for length and red is the bits for posisti on.

Wing Commander (SNES) :



Final Fantasy V (SNES) :

In these cases, its best to have a look at the assembly while the game is running. A little common sense and some testing should help too. Don't continue working until you get it right. ;). Let's go on about the coding of the position and length.

2. Coding of the length :

We should start with the easiest, the length of the coded chains. This is usually limited. You can te ll with what we looked at above that its best to have a length greater then 3 to achieve maximum compression. However, some LZSS compression compress 2 bytes. I can't understand why and I think this would cause some slow downs.

Taking into account that we are limiting are chain length to at least 2. You then can remove that m inimum from the coded value saving some bits. I pointed out before that the number of bits is the l imiting factor here so why not start with 0 being 3 instead.

Another reason for limiting the length is for cost of searching for a larger number of allocated bits. LZSS can be just as slow compressing as it is fast while decompressing. This is why the less com plex the chains are that you are looking for the quicker the compression is. Although, if you are go ing to encode something permanently (video game) then the slow compression time is a good tra de off for a better compression ratio. There is no limit to the length which can be used but no one usually rewrites the same exact sentence twice in text.

Thus for FF5, you go from 3 to 35 characters (2^5+3) coded between 00000 and 11111. And for Wing, it's from 2 to 17 characters (2^4+2).

3. Coding of the position:

Here is where it starts getting complicated. ; (Prepare to be a little lost. :) Once you find out ho w the bits are arranged for the posistion, it's really not that bad. Once the arrangement is found it should be checked by finding the actual position in the file.

It's necessary to make sure you don't go searching too far. Here is where you may run in to some confusing findings. One small addition may throw it off. In most cases, there is a more or less logi cal offset. (There is always a function. For example, 221+35%256=1 from what I remember) Even when you have the decompressed text to look at, the harder part is the amount of time finding the chains and where to find the peices when decompressing. There is usually a spot where the char acters are in a block of 4096 (2^12) if the compression was coded with 12 bits for the posistion. T his means you will only have to look for the chains in this window. So if you are starting at 0, only search for the chains above it 4096 bytes. This position is relative to the starting location. There are unlimited ways for the compression to be coded but it’s usually standard this way and it’s the way the original Lempel-Ziv-Storer-Szymanski was originally utilized (LZSS).

0	4096	9192	Pseudo absolute	0	4096	9192	Relative

4. This type of coding (easy to understand):

Let me take a paragraph to say that the bits of data are sometimes reversed compared to the ord er we have seen them. In this same manner, a 0 could mean a single character and a 1 could me an chain. But that's easy to find if you using the practice methods.

Practice methods (because practice makes perfect) :

1. To find a LZSS scheme:

There‘s almost nothing easier or more obvious than this :) . Like you have seen above, the LZS S is a byte of information that follows data. However, there are 8 bytes of data if nothing can be c oded. It doesn't have to have it in front. Thus, if you know the characters of the text, you can find t he chain (you can even use a relative search). If you come upon a byte equal to 00 or FF you kno w that the next byte has our information.

If you choose another method, like looking through the file until you find the text, it still should be r ecognized quickly. Either way you will see the FF or a different byte and you should be able to rea d at least some of it. (If you can’t read some of it, it’s probably not LZSS!)

If you got this far you have past the first step, finding the text. There‘s still more to do though ;).

2. Understanding the operations (Where is the "decode" button?) :

First off you got to look at the structure of the byte heading. First determine if it's the 0 or the 1 tha t encodes the characters. Then you got to look at the 8 readable characters (9 bytes) until finding the first strange character. In theory, you should quickly determine the order of the bits (reversed or not). Next you have to look for the number that corresponds to the chain. You have to find the o rder of the bits in the byte.

Thus, when you are writing code that will return the chain you have to know which bits are doing what. To practice, take several and compare them. This is tedious but the only way to find how th e bits are arranged. (this is for when you are not disassembling the code while running and viewin g the code. It is easier to get the arrangement then.) And, at this stage, in my humble opinion, it is better to start with the programming. Trying to decompress an LZSS without writing a program is i nsane! Constant testing is the best way to find the solution. The length is represented by the last bits. For the position, don't get to proud and think that 0 in the beginning points to the position (yo u can have very well 111111011111 with an offset of 33 and the point on the position 0 but when y ou add one you get (1)000000000000).

Once you have figured out the bits, all that is left is finding the offset. (if its coded that simple). To f ind the offset you have to do an upward search to find the chain of characters. Pay attention beca use sometimes you will find it in the first search. Sometimes it will be the last one you find. You m ay not find it at all. In any case, its best to have the location of the text. Write down where its at.

There‘s not really anything more to finding the coding besides that. You can't continue on really if you can't find it in the 2^ (number of bits of the position) window you are searching in. Make sure you search absolute and relative when you are looking.

3. How about compressing ?

Ah, well it's same thing but in reverse. It's pointless to get in to details cause there is only one wa y to do it. It's not very complicated once you know your limits in the preceding paragraph. Of cour se if you want to compress text directly without following the rules of an existing decompression t han the only limits are that of the computer you are using. On a side note, the more complex your program is the quicker you will lose memory and speed. The differences become very minimal aft er a certain point.

4. Finishing the program?

Now that we have everything figured out we should double check to make sure our offsets and ev erything is right. At this point its necessary to right a compression/decompression routine. I don't s uggest a program that does everything. Firstly, I write these types of programs exclusively in C. I' m afraid that if I do that it will be bad for certain purist because my code can be confusing or I forg et to write some comments. Secondly, everyone does not use C and if this FAQ survives at least a few years and is even lost of people's hard drives (including mine) other languages will replace this one when writing these types of programs (which is already happening).

This is why on the other hand, I'm gonna go ahead and write this code in abstracted algorithmic la nguage (that’s my primitive way of saying English ;) Nothing I wrote is unchangeable. I complicat ed some things and didn't optimize others. My purpose was to get the point across in the least co nfusing way. So optimizations that hide certain details will be avoided.

The Program! Give use the Program!

1. Decompression :

To find out certain constants of the routine:
- Number of bits of the position (NBPOS)
- Which bits do what when linking to the chain (with the limit in a table)
- the offset for the position (OFFP) and the cut (OFFC)

To prepare the matching variables:
- a zone of memory to store the text in, at least 2^NBPOS (BUFFER)
- a table or a byte for the bits indicating the type of coding (TYPES)
- a position of reading and writing to be initialized (POSR, POSW)
- size and position of the chain that will be returned (POS,SIZE)

Make a function that will convert a byte to bits in your desired way (CONVERT)
(I wont deal with that as you should be able to do that yourself)
DO WHILE (As long as there is compressed text. That is for you to determine.) MAKE
Look at the coding of the byte in POSL and apply CONVERT to get it to the table in TYPE
S
FOR I from 1 to 8 (for the eight elements of data) MAKE
IF (the type of coding is that of a character) THEN
Copy the character (byte)
Increment POSR and POSW
IF NOT
Recover the size and position of the 2 bytes in POSW
Add OFFP to POS and make the window 2^NBPOS (the number of bits
must agree)
Add OFFC to SIZE
Copy SIZE characters starting from POS in POSR (and in BUFFER)
Add 2 to POSW and SIZE with INSTALLATION
END IF
IF (If it comes to end of BUFFER, POSL%(2^NBPOS)=0) THEN
Then go back to the beginning of BUFFER
END IF
END FOR
END DO WHILE

2. Compression :

(Very similar at the beginning except for slight differences)
Constants :
- number of bits length (NBLONG)

New variables :
- Area of memory to store compressed data (you have to wait for all 8 bits to write anything because the byte heading is continually obtained) (COMP)
- Variables for research: long length gotten, position of this one, position of the current rese arch, … (MLONG, MPOS, LONG, POS)

This is a function that will convert 8 bits to a byte in the way you specify (RECONVERT)
DO WHILE (There is text to be compressed) MAKE
FOR I FROM 1 to 8 MAKE
IF (End of text found)
You fill TYPES with the code for the characters and COMP with a null ch
aracter
IF NOT (POSL exceeded OFFT) THEN
You make research by initializing POS at a good spot in the 2^NBPOS
Create MLONG and set to 0

```
                        DO WHILE (POS<POSE, Like I stated at the beginning of the FAQ, at th
e end of the first part
          ET and MLONG should be less than NBLONG ET MLONG<(2^NBLONG+OFFT)) MAKE
                                Look for the longest chain possible that matches the current posi
tion
                                In order to do that, increment LONG and POs where neccessary
                                IF (LONG>MLONG) THEN
                                Store the information of POS and LONG in MPOS and MLONG
                                END IF
                        END DO WHILE
                        IF (MLONG>2, you found a useful chain)
                        Remove OFFP with MPOS and OFFT with MLONG
                                Copy data from MLONG and MPOS into COMP (one byte)
                                Put TYPES[i] for a chain
                        IF NOT
                                Put in the corresponding info for the characters
                                Put MLONG at 1 and all the cases
                        END IF
                        Increment POSL of MLONG
                IF NOT
                        Put in TYPES[i] the code for a character
                        Recopy the character into COMP
                        Increment POSL
                END IF
        END FOR
        Recopy TYPES after converting it into a byte
        Recopy the bytes of COMP at the end
        Increment POSE continually : 1+2*(Number of coded chains)+(Number of coded charact
ers)
END DO WHILE
```

Final Thanks (they deserve it) :

Thanks to Copernic who has accepted me at Terminus and pushed me a little with hacking (finally at a decent level), NeoMithrandil and Manz who often gave me information, and i can't forget Ryu san who straight up demanded that I make this FAQ ^^ (who is not even considering the questions I put aside)
I also thank many authors who have written documents on various compressions, lzss especially, and finally the people to spend their lives playing video games because without them I would never have wanted to do this.
Considering that this is the first FAQ that I have finished, I would like to greet everyone who wrote the hacking documents that I read before writing this. These people include Jay, Copernic, Skeud, Rysley, Moogle and many others.

And of course I have to thank you since you have read up to this far ;).

If you are looking for other information, then I can't really help you with that. Documents dealing with LZSS and compression are very abundant but those that are speciafically for the rom hacking aspect of that are less common. Nevertheless, if you want information on comrpessions, then Jay's "Compression Tutorial" may be helpful for you.

You can always reach me at tt_nemesis@yahoo.fr <mailto:tt_nemesis@yahoo.fr> (send before the address is useless considering what i've just done. Perhaps there will be others of them too :)

Nemesis 26/03/03 (after about four hours of drafting and work ;)