This page is http://www.iki.fi/a1bert/Dev/pucrunch/
Other C64-related stuff -> http://www.iki.fi/a1bert/Dev/

Pucrunch found elsewhere in random order:
Maxim's World of Stuff - SMS/Meka stuff
GameBoy Dev'rs - Compression
Headspin's Guide to Compression, Files Systems Screen Effects and MOD Players for the Gameboy Advance
Jeff Frohwein's GameBoy Libs
CrunchyOS - an assembly shell for the TI83+ and TI83+SE calculators

---

First Version 14.3.1997, Rewritten Version 17.12.1997
Another Major Update 14.10.1998
Last Updated 22.11.2008
Pasi Ojala, *a1bert@iki.fi*

An Optimizing Hybrid LZ77 RLE Data Compression Program, aka

Improving Compression Ratio for Low-Resource Decompression

Short:
Pucrunch is a Hybrid LZ77 and RLE compressor, uses an Elias Gamma Code for lengths, mixture of Gamma Code and linear for LZ77 offset, and ranked RLE bytes indexed by the same Gamma Code. Uses no extra memory in decompression.

Programs - Source code and executables

Introduction

Since I started writing demos for the C64 in 1989 I have always wanted to program a compression program. I had a lot of ideas but never had the time, urge or the necessary knowledge or patience to create one. In retrospect, most of the ideas I had then were simply bogus ("magic function theory" as Mark Nelson nicely puts it). But years passed, I gathered more knowledge and finally got an irresistible urge to finally realize my dream.

The nice thing about the delay is that I don't need to write the actual compression program to run on a C64 anymore. I can write it in portable ANSI-C code and just program it to create files that would uncompress themselves when run on a C64. Running the compression program outside of the target system provides at least the following advantages.

- I can use portable ANSI-C code. The compression program can be compiled to run on a Unix box, Amiga, PC etc. And I have all the tools to debug the program and gather profiling information to see why it is so slow :-)
- The program runs much faster than on C64. If it is still slow, there is always multitasking to allow me to do something else while I'm compressing a file.
- There is 'enough' memory available. I can use all the memory I possibly need and use every trick possible to increase the compression ratio as long as the decompression remains possible and feasible on a C64.
- Large files can be compressed as easily as shorter files. Most C64 compressors can't handle files larger than around 52-54 kilobytes (210-220 disk blocks).
- Cross-development is easier because you don't have to transfer a file into C64 just to compress it.
- It is now possible to use the same program to handle VIC20 and C16/+4 files also. It hasn't been possible to compress VIC20 files at all. At least I don't know any other VIC20 compressor around.

Memory Refresh and Terms for Compression

Statistical compression
: Uses the uneven probability distribution of the source symbols to shorten the average code length. Huffman code and arithmetic code belong to this group. By giving a short code to symbols occurring most often, the number ot bits needed to represent the symbols decreases. Think of the Morse code for example: the characters you need more often have shorter codes and it takes less time to send the message.

Dictionary compression
: Replaces repeating strings in the source with shorter representations. These may be indices to an actual dictionary (Lempel-Ziv 78) or pointers to previous occurrances (Lempel-Ziv 77). As long as it takes fewer bits to represent the reference than the string itself, we get compression. LZ78 is a lot like the way BASIC substitutes tokens for keywords: one-byte tokens expand to whole words like PRINT#. LZ77 replaces repeated strings with (length,offset) pairs, thus the string `VICIICI` can be encoded as `VICI(3,3)` -- the repeated occurrance of the string `ICI` is replaced by a reference.

Run-length encoding
: Replaces repeating symbols with a single occurrance of the symbol and a repeat count. For example assembly compilers have a `.zero` keyword or equivalent to fill a number of bytes with zero without needing to list them all in the source code.

Variable-length code
: Any code where the length of the code is not explicitly known but changes depending on the bit values. A some kind of end marker or length count must be provided to make a code a prefix code (uniquely decodable). For ASCII (or Latin-1) text you know you get the next letter by reading a full byte from the input. A variable-length code requires you to read part of the data to know how many bits to read next.

Universal codes
: Universal codes are used to encode integer numbers without the need to know the maximum value. Smaller integer values usually get shorter codes. Different universal codes are optimal for different distributions of the values. Universal codes include Elias Gamma and Delta codes, Fibonacci code, and Golomb and Rice codes.

Lossless compression
: Lossless compression algorithms are able to exactly reproduce the original contents unlike lossy compression, which omits details that are not important or perceivable by human sensory system. This article only talks about lossless compression.

My goal in the pucrunch project was to create a compression system in which the decompressor would use minimal resources (both memory and processing power) and still have the best possible compression ratio. A nice bonus would be if it outperformed every other compression program available. These understandingly opposite requirements (minimal resources and good compression ratio) rule out most of the state-of-the-art compression algorithms and in effect only leave RLE and LZ77 to be considered. Another goal was to learn something about data compression and that goal at least has been satisfied.

I started by developing a byte-aligned LZ77+RLE compressor/decompressor and then added a Huffman backend to it. The Huffman tree takes 384 bytes and the code that decodes the tree into an internal representation takes 100 bytes. I found out that while the Huffman code gained about 8% in my 40-kilobyte test files, the gain was reduced to only about 3% after accounting the extra code and the Huffman tree.

Then I started a more detailed analysis of the LZ77 offset and length values and the RLE values and concluded that I would get better compression by using a variable-length code. I used a simple variable-length code and scratched the Huffman backend code, as it didn't increase the compression ratio anymore. This version became pucrunch.

Pucrunch does not use byte-aligned data, and is a bit slower than the byte-aligned version because of this, but is much faster than the original version with the Huffman backend attached. And pucrunch still does very well compression-wise. In fact, it does very well indeed, beating even LhA, Zip, and GZip in some cases. But let's not get too much ahead of ourselves.

To get an improvement to the compression ratio for LZ77, we have only some options left. We can improve on the encoding of literal bytes (bytes that are not compressed), we can reduce the

number of literal bytes we need to encode, and shorten the encoding of RLE and LZ77. In the algorithm presented here all these improvement areas are addressed both collectively (one change affects more than one area) and one at a time.

1. By using a variable-length code we can gain compression for even 2-byte LZ77 matches, which in turn reduces the number of literal bytes we need to encode. Most LZ77-variants require 3-byte matches to get any compression because they use so many bits to identify the length and offset values, thus making the code longer than the original bytes would've taken.

2. By using a new literal byte tagging system which distinguishes uncompressed and compressed data efficiently we can reduce number of extra bits needed to make this distinction (the encoding overhead for literal bytes). This is especially important for files that do not compress well.

3. By using RLE in addition to LZ77 we can shorten the encoding for long byte run sequences and at the same time set a convenient upper limit to LZ77 match length. The upper limit performs two functions:
   - we only need to encode integers in a specific range
   - we only need to search strings shorter than this limit (if we find a string long enough, we can stop there)

   Short byte runs are compressed either using RLE or LZ77, whichever gets the best results.

4. By doing statistical compression (more frequent symbols get shorter representations) on the RLE bytes (in this case symbol ranking) we can gain compression for even 2-byte run lengths, which in turn reduces the number of literal bytes we need to encode.

5. By carefully selecting which string matches and/or run lengths to use we can take advantage of the variable-length code. It may be advantageous to compress a string as two shorter matches instead of one long match and a bunch of literal bytes, and it can be better to compress a string as a literal byte and a long match instead of two shorter matches.

This document consists of several parts, which are:

Commodore 64 Considerations

Our target environment (Commodore 64) imposes some restrictions which we have to take into consideration when designing the ideal compression system. A system with a 1-MHz 3-register 8-bit processor and 64 kilobytes of memory certainly imposes a great challenge, and thus also a great sense of achievement for good results.

First, we would like it to be able to decompress as big a program as possible. This in turn requires that the decompression code is located in low memory (most programs that we want to compress start at address 2049) and is as short as possible. Also, the decompression code must not use any extra memory or only very small amounts of it. Extra care must be taken to make certain that the compressed data is not overwritten during the decompression before it has been read.

Secondly, my number one personal requirement is that the basic end address must be correctly set by the decompressor so that the program can be optionally saved in uncompressed form after decompression (although the current decompression code requires that you say "clr" before saving). This also requires that the decompression code is system-friendly, i.e. does not change KERNAL or BASIC variables or other structures. Also, the decompressor shouldn't rely on file size or load end address pointers, because these may be corrupted by e.g. X-modem file transfer protocol (padding bytes may be added).

When these requirements are combined, there is not much selection in where in the memory we can put the decompression code. There are some locations among the first 256 addresses (zeropage) that can be used, the (currently) unused part of the processor stack (0x100..0x1ff), the system input buffer (0x200..0x258) and the tape I/O buffer plus some unused bytes (0x334-0x3ff). The screen memory (0x400..0x7ff) can also be used if necessary. If we can do without the screen memory and the tape buffer, we can potentially decompress files that are located from 0x258 to 0xffff.

The third major requirement is that the decompression should be relatively fast. After 10 seconds the user begins to wonder if the program crashed or is it doing anything, even if there is some feedback like border color flashing. This means that the arithmetic used should be mostly 8- or 9-bit (instead of full 16 bits) and there should be very little of it per each decompressed byte. Processor- and memory-intensive algorithms like arithmetic coding and prediction by partial matching (PPM) are pretty much out of the question, and it is saying it mildly. LZ77 seems the only practical alternative. Still, run-length encoding handles long byte runs better than LZ77 and can have a bigger length limit. If we can easily incorporate RLE and LZ77 into the same algorithm, we should get the best features from both.

A part of the decompressor efficiency depends on the format of the compressed data. Byte-aligned codes, where everything is aligned into byte boundaries can be accessed very quickly, non-byte-aligned variable length codes are much slower to handle, but provide better compression. Note that byte-aligned codes can still have other data sizes than 8. For example you can use 4 bits for LZ77 length and 12 bits for LZ77 offset, which preserves the byte alignment.

The New Tagging System

I call the different types of information my compression algorithm outputs `primary units`. The primary units in this compression algorithm are:

- literal (uncompressed) bytes and escape sequences,
- LZ77 (length,offset)-pairs,
- RLE (length,byte)-pairs, and
- EOF (end of file marker).

Literal bytes are those bytes that could not be represented by shorter codes, unlike a part of previously seen data (LZ77), or a part of a longer sequence of the same byte (RLE).

Most compression programs handle the selection between compressed data and literal bytes in a straightforward way by using a prefix bit. If the bit is 0, the following data is a literal byte (uncompressed). If the bit is 1, the following data is compressed. However, this presents the problem that non-compressible data will be expanded from the original 8 bits to 9 bits per byte, i.e. by 12.5 %. If the data isn't very compressible, this overhead consumes all the little savings you may have had using LZ77 or RLE.

Some other data compression algorithms use a value (using variable-length code) that indicates the number of literal bytes that follow, but this is really analogous to a prefix bit, because 1-byte uncompressed data is very common for modestly compressible files. So, using a prefix bit may seem like a good idea, but we may be able to do even better. Let's see what we can come up with. My idea was to somehow use the data itself to mark compressed and uncompressed data and thus I don't need any prefix bits.

Let's assume that 75% of the symbols generated are literal bytes. In this case it seems viable to allocate shorter codes for literal bytes, because they are more common than compressed data. This distribution (75% are literal bytes) suggests that we should use 2 bits to determine whether the data is compressed or a literal byte. One of the combinations indicates compressed data, and three of the combinations indicate a literal byte. At the same time those three values divide the literal bytes into three distinct groups. But how do we make the connection between which of the three bit patters we have and what are the literal byte values?

The simplest way is to use a direct mapping. We use two bits (let them be the two most-significant bits) from the literal bytes themselves to indicate compressed data. This way no actual prefix bits are needed. We maintain an escape code (which doesn't need to be static), which is compared to the bits, and if they match, compressed data follows. If the bits do not match, the rest of the literal

byte follows. In this way the literal bytes do not expand at all if their most significant bits do not match the escape code, thus less bits are needed to represent the literal bytes.

Whenever those bits in a literal byte would match the escape code, an escape sequence is generated. Otherwise we could not represent those literal bytes which actually start like the escape code (the top bits match). This escape sequence contains the offending data and a new escape code. This escape sequence looks like

```
# of escape bits    (escape code)
3                   (escape mode select)
# of escape bits    (new escape bits)
8-# of escape bits  (rest of the byte)
= 8 + 3 + # of escape bits
= 13 for 2-bit escapes, i.e. expands the literal byte by 5 bits.
```

Read further to see how we can take advantage of the changing escape code.

You may also remember that in the run-length encoding presented in the previous article two successive equal bytes are used to indicate compressed data (escape condition) and all other bytes are literal bytes. A similar technique is used in some C64 packers (RLE) and crunchers (LZ77), the only difference is that the escape condition is indicated by a fixed byte value. My tag system is in fact an extension to this. Instead of a full byte, I use only a few bits.

We assumed an even distribution of the values and two escape bits, so 1/4 of the values have the same two most significant bits as the escape code. I call this probability that a literal byte has to be escaped the `hit rate`. Thus, literal bytes expand in average 25% of the time by 5 bits, making the average length 25% * 13 + 75% * 8 = 9.25. Not suprising, this is longer than using one bit to tag the literal bytes. However, there is one thing we haven't considered yet. The escape sequence has the possibility to change the escape code. Using this feature to its optimum (escape optimization), the average 25% `hit rate` becomes the **maximum** `hit rate`.

Also, because the distribution of the (values of the) literal bytes is seldom flat (some values are more common than others) and there is locality (different parts of the file only contain some of the possible values), from which we can also benefit, the actual hit rate is always much smaller than that. Empirical studies on some test files show that for 2-bit escape codes the actual realized hit rate is only 1.8-6.4%, while the theoretical maximum is the already mentioned 25%.

Previously we assumed the distribution of 75% of literal bytes and 25% of compressed data (other primary units). This prompted us to select 2 escape bits. For other distributions (differently compressible files, not necessarily better or worse) some other number of escape bits may be more suitable. The compressor tries different number of escape bits and select the value which gives the best overall results. The following table summarizes the hit rates on the test files for different number of escape bits.

1-bit	**2-bit**	**3-bit**	**4-bit**	**File**
50.0%	25.0%	12.5%	6.25%	Maximum
25.3%	2.5%	0.3002%	0.0903%	ivanova.bin
26.5%	2.4%	0.7800%	0.0631%	sheridan.bin
20.7%	1.8%	0.1954%	0.0409%	delenn.bin
26.5%	6.4%	2.4909%	0.7118%	bs.bin
9.06	**8.32**	**8.15**	**8.050**	**bits/Byte for bs.bin**

As can be seen from the table, the realized hit rates are dramatically smaller than the theoretical maximum values. A thought might occur that we should always select 4-bit (or longer) escapes, because it reduces the hit rate and presents the minimum overhead for literal bytes. Unfortunately increasing the number of escape bits also increases the code length of the compressed data. So, it is a matter of finding the optimum setting.

If there are very few literal bytes compared to other primary units, 1-bit escape or no escape at all gives very short codes to compressed data, but causes more literal bytes to be escaped, which means 4 bits extra for each escaped byte (with 1-bit escapes). If the majority of primary units are literal bytes, for example a 6-bit escape code causes most of the literal bytes to be output as 8-bit codes (no expansion), but makes the other primary units 6 bits longer. Currently the compressor automatically selects the best number of escape bits, but this can be overriden by the user with

the -e option.

The cases in the example with 1-bit escape code validates the original suggestion: use a prefix bit. A simple prefix bit would produce better results on three of the previous test files (although only slightly). For delenn.bin (1 vs 0.828) the escape system works better. On the other hand, 1-bit escape code is not selected for any of the files, because 2-bit escape gives better overall results.

Note: for 7-bit ASCII text files, where the top bit is always 0 (like most of the Calgary Corpus files), the hit rate is 0% for even 1-bit escapes. Thus, literal bytes do not expand at all. This is equivalent to using a prefix bit and 7-bit literals, but does not need seperate algorithm to detect and handle 7-bit literals.

For Calgary Corpus files the number of tag bits per primary unit (counting the escape sequences and other overhead) ranges from as low as 0.46 (book1) to 1.07 (geo) and 1.09 (pic). These two files (geo and pic) are the only ones in the suite where a simple prefix bit would be better than the escape system. The average is 0.74 tag bits per primary unit.

In Canterbury Corpus the tag bits per primary unit ranges from 0.44 (plrabn12.txt) to 1.09 (ptt5), which is the only one above 0.85 (sum). The average is 0.61 tag bits per primary.

Primary Units Used for Compression

The compressor uses the previously described escape-bit system while generating its output. I call the different groups of bits that are generated primary units. The primary units in this compression algorithm are: literal byte (and escape sequence), LZ77 (length,offset)-pair, RLE (length, byte)-pair, and EOF (end of file marker).

If the top bits of a literal byte do not match the escape code, the byte is output as-is. If the bits match, an escape sequence is generated, with the new escape code. Other primary units start with the escape code.

The Elias Gamma Code is used extensively. This code consists of two parts: a unary code (a one-bit preceded by zero-bits) and a binary code part. The first part tells the decoder how many bits are used for the binary code part. Being a universal code, it produces shorter codes for small integers and longer codes for larger integers. Because we expect we need to encode a lot of small integers (there are more short string matches and shorter equal byte runs than long ones), this reduces the total number of bits needed. See the previous article for a more in-depth delve into statistical compression and universal codes. To understand this article, you only need to keep in mind that small integer value equals short code. The following discusses the encoding of the primary units.

The most frequent compressed data is LZ77. The length of the match is output in Elias Gamma code, with "0" meaning the length of 2, "100" length of 3, "101" length of 4 and so on. If the length is not 2, a LZ77 offset value follows. This offset takes 9 to 22 bits. If the length is 2, the next bit defines whether this is LZ77 or RLE/Escape. If the bit is 0, an 8-bit LZ77 offset value follows. (Note that this restricts the offset for 2-byte matches to 1..256.) If the bit is 1, the next bit decides between escape (0) and RLE (1).

The code for an escape sequence is thus `e..e010n..ne....e`, where `E` is the byte, and `N` is the new escape code. Example:

- We are using 2-bit escapes
- The current escape code is `"11"`
- We need to encode a byte `0xca == 0b11001010`
- The escape code and the byte high bits match (both are `"11"`)
- We output the current escape code `"11"`
- We output the escaped identification `"010"`
- We output the new escape bits, for example `"10"` (depends on escape optimization)
- We output the rest of the escaped byte `"001010"`
- So, we have output the string `"1101010001010"`

When the decompressor receives this string of bits, it finds that the first two bits match with the escape code, it finds the escape identification (`"010"`) and then gets the new escape, the rest of the original byte and combines it with the old escape code to get a whole byte.

The end of file condition is encoded to the LZ77 offset and the RLE is subdivided into long and short versions. Read further, and you get a better idea about why this kind of encoding is selected.

When I studied the distribution of the length values (LZ77 and short RLE lengths), I noticed that the smaller the value, the more occurrances. The following table shows an example of length value distribution.

```
        LZLEN   S-RLE
2        1975     477
3-4      1480     330
5-8       492     166
9-16      125      57
17-32      31      33
33-64       8      15
```

The first column gives a range of values. The first entry has a single value (2), the second two values (3 and 4), and so on. The second column shows how many times the different LZ77 match lengths are used, the last column shows how many times short RLE lengths are used. The distribution of the values gives a hint of how to most efficiently encode the values. We can see from the table for example that values 2-4 are used 3455 times, while values 5-64 are used only 656 times. The more common values need to get shorter codes, while the less-used ones can be longer.

Because in each "magnitude" there are approximately half as many values than in the preciding one, it almost immediately occurred to me that the optimal way to encode the length values (decremented by one) is:

```
Value      Encoding       Range     Gained
0000000    not possible
0000001    0              1         -6 bits
000001x    10x            2-3       -4 bits
00001xx    110xx          4-7       -2 bits
0001xxx    1110xxx        8-15      +0 bits
001xxxx    11110xxxx      16-31     +2 bits
01xxxxx    111110xxxxx    32-63     +4 bits
1xxxxxx    111111xxxxxx   64-127    +5 bits
```

The first column gives the binary code of the original value (with `x` denoting 0 or 1, `xx` 0..3, `xxx` 0..7 and so on), the second column gives the encoding of the value(s). The third column lists the original value range in decimal notation.

The last column summarises the difference between this code and a 7-bit binary code. Using the previous encoding for the length distribution presented reduces the number of bits used compared to a direct binary representation considerably. Later I found out that this encoding in fact is Elias Gamma Code, only the assignment of 0- and 1-bits in the prefix is reversed, and in this version the length is limited. Currently the maximum value is selectable between 64 and 256.

So, to recap, this version of the Gamma code can encode numbers from 1 to 255 (1 to 127 in the example). LZ77 and RLE lengths that are used start from 2, because that is the shortest length that gives us any compression. These length values are first decremented by one, thus length 2 becomes `"0"`, and for example length 64 becomes `"11111011111"`.

The distribution of the LZ77 offset values (pointer to a previous occurrance of a string) is not at all similar to the length distribution. Admittedly, the distribution isn't exactly flat, but it also isn't as radical as the length value distribution either. I decided to encode the lower 8 bits (automatically selected or user-selectable between 8 and 12 bits in the current version) of the offset as-is (i.e. binary code) and the upper part with my version of the Elias Gamma Code. However, 2-byte matches always have an 8-bit offset value. The reason for this is discussed shortly.

Because the upper part can contain the value 0 (so that we can represent offsets from 0 to 255 with a 8-bit lower part), and the code can't directly represent zero, the upper part of the LZ77 offset is incremented by one before encoding (unlike the length values which are decremented by one). Also, one code is reserved for an end of file (EOF) symbol. This restricts the offset value somewhat, but the loss in compression is negligible.

With the previous encoding 2-byte LZ77 matches would only gain 4 bits (with 2-bit escapes) for each offset from 1 to 256, and 2 bits for each offset from 257 to 768. In the first case 9 bits would be used to represent the offset (one bit for gamma code representing the high part 0, and 8 bits for the low part of the offset), in the latter case 11 bits are used, because each "magnitude" of values in the Gamma code consumes two more bits than the previous one.

The first case (offset 1..256) is much more frequent than the second case, because it saves more bits, and also because the symbol source statistics (whatever they are) guarantee 2-byte matches in recent history (much better chance than for 3-byte matches, for example). If we restrict the offset for a 2-byte LZ77 match to 8 bits (1..256), we don't lose so much compression at all, but instead we could shorten the code by one bit. This one bit comes from the fact that before we had to use one bit to make the selection "8-bit or longer". Because we only have "8-bit" now, we don't need that select bit anymore.

Or, we can use that select bit to a new purpose to select whether this code really is LZ77 or something else. Compared to the older encoding (which I'm not detailing here, for clarity's sake. This is already much too complicated to follow, and only slightly easier to describe) the codes for escape sequence, RLE and End of File are still the same length, but the code for LZ77 has been shortened by one bit. Because LZ77 is the most frequently used primary unit, this presents a saving that more than compensates for the loss of 2-byte LZ77 matches with offsets 257..768 (which we can no longer represent, because we fixed the offset for 2-byte matches to use exactly 8 bits).

Run length encoding is also a bit revised. I found out that a lot of bits can be gained by using the same length encoding for RLE as for LZ77. On the other hand, we still should be able to represent long repeat counts as that's where RLE is most efficient. I decided to split RLE into two modes:

- short RLE for short (e.g. 2..128) equal byte strings
- long RLE for long equal byte strings

The Long RLE selection is encoded into the Short RLE code. Short RLE only uses half of its coding space, i.e. if the maximum value for the gamma code is 127, short RLE uses only values 1..63. Larger values switches the decoder into Long RLE mode and more bits are read to complete the run length value.

For further compression in RLE we rank all the used RLE bytes (the values that are repeated in RLE) in the decreasing probability order. The values are put into a table, and only the table indices are output. The indices are also encoded using a variable length code (the same gamma code, surprise..), which uses less bits for smaller integer values. As there are more RLE's with smaller indices, the average code length decreases. In decompression we simply get the gamma code value and then use the value as an index into the table to get the value to repeat.

Instead of reserving full 256 bytes for the table we only put the top 31 RLE bytes into the table. Normally this is enough. If there happens to be a byte run with a value not in the table we use a similar technique as for the short/long RLE selection. If the table index is larger than 31, it means we don't have the value in the table. We use the values 32..63 to select the 'escaped' mode and simultaneously send the 5 most significant bits of the value (there are 32 distinct values in the range 32..63). The rest 3 bits of the byte are sent separately.

If you are more than confused, forget everything I said in this chapter and look at the decompression pseudo-code later in this article.

Graph Search - Selecting Primary Units

In free-parse methods there are several ways to divide the file into parts, each of which is equally valid but not necessary equally efficient in terms of compression ratio.

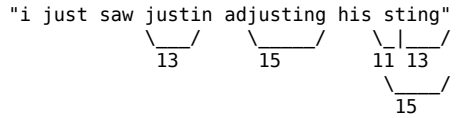
```
"i just saw justin adjusting his sting"

"i just saw", (-9,4), "in ad", (-9,6), "g his", (-25,2), (-10,4)
"i just saw", (-9,4), "in ad", (-9,6), "g his ", (-10,5)
```

The latter two lines show how the sentence could be encoded using literal bytes and (offset, length) pairs. As you can see, we have two different encodings for a single string and they are both valid, i.e. they will produce the same string after decompression. This is what free-parse is: there are several possible ways to divide the input into parts. If we are clever, we will of course select the encoding that produces the shortest compressed version. But how do we find this shortest version? How does the data compressor decide which primary unit to generate in each step?

The most efficient way the file can be divided is determined by a sort of a graph-search algorithm, which finds the shortest possible route from the start of the file to the end of the file. Well, actually

the algorithm proceeds from the end of the file to the beginning for efficiency reasons, but the result is the same anyway: the path that minimizes the bits emitted is determined and remembered. If the parameters (number of escape bits or the variable length codes or their parameters) are changed, the graph search must be re-executed.



Think of the string as separate characters. You can jump to the next character by paying 8 bits to do so (not shown in the figure), unless the top bits of the character match with the escape code (in which case you need more bits to send the character "escaped"). If the history buffer contains a string that matches a string starting at the current character you can jump over the string by paying as many bits as representing the LZ77 (offset,length)-pair takes (including escape bits), in this example from 11 to 15 bits. And the same applies if you have RLE starting at the character. Then you just find the least-expensive way to get from the start of the file to the end and you have found the optimal encoding. In this case the last characters " sting" would be optimally encoded with 8(literal " ") + 15("sting") = 23 instead of 11(" s") + 13("ting") = 24 bits.

The algorithm can be written either cleverly or not-so. We can take a real short-cut compared to a full-blown graph search because we can/need to only go forwards in the file: we can simply start from the end! Our accounting information which is updated when we pass each location in the data consists of three values:

1. the minimum bits from this location to the end of file.
2. the mode (literal, LZ77 or RLE) to use to get that minimum
3. the "jump" length for LZ77 and RLE

For each location we try to jump forward (to a location we already processed) one location, LZ77 match length locations (if a match exists), or RLE length locations (if equal bytes follow) and select the shortest route, update the tables accordingly. In addition, if we have a LZ77 or RLE length of for example 18, we also check jumps 17, 16, 15, ... This gives a little extra compression. Because we are doing the "tree traverse" starting from the "leaves", we only need to visit/process each location once. Nothing located after the current location can't change, so there is never any need to update a location.

To be able to find the minimal path, the algorithm needs the length of the RLE (the number of the identical bytes following) and the maximum LZ77 length/offset (an identical string appearing earlier in the file) for each byte/location in the file. This is the most time-consuming **and** memory-consuming part of the compression. I have used several methods to make the search faster. See String Match Speedup later in this article. Fortunately these searches can be done first, and the actual optimization can use the cached values.

Then what is the rationale behind this optimization? It works because you are not forced to take every compression opportunity, but select the best ones. The compression community calls this "lazy coding" or "non-greedy" selection. You may want to emit a literal byte even if there is a 2-byte LZ77 match, because in the next position in the file you may have a longer match. This is actually more complicated than that, but just take my word for it that there is a difference. Not a very big difference, and only significant for variable-length code, but it is there and I was after every last bit of compression, remember.

Note that the decision-making between primary units is quite simple if a fixed-length code is used. A one-step lookahead is enough to guarantee optimal parsing. If there is a more advantageous match in the next location, we output a literal byte and that longer match instead of the shorter match. I don't have time or space here to go very deeply on that, but the main reason is that in fixed-length code it doesn't matter whether you represent a part of data as two matches of lengths 2 and 8 or as matches of lengths 3 and 7 or as any other possible combination (if matches of those lengths exist). This is not true for a variable-length code and/or a statistical compression backend. Different match lengths and offsets no longer generate equal-length codes.

Note also that most LZ77 compression algorithms need at least 3-byte match to break even, i.e. not expanding the data. This is not surprising when you stop to think about it. To gain something from 2-byte matches you need to encode the LZ77 match into 15 bits. This is very little. A generic LZ77 compressor would use one bit to select between a literal and LZ77, 12 bits for moderate offset, and you have 2 bits left for match length. I imagine the rationale to exclude 2-byte matches

also include "the potential savings percentage for 2-byte matches is insignificant". Pucrunch gets around this by using the tag system and Elias Gamma Code, and does indeed gain bits from even 2-byte matches.

After we have decided on what primary units to output, we still have to make sure we get the best results from the literal tag system. Escape optimization handles this. In this stage we know which parts of the data are emitted as literal bytes and we can select the minimal path from the first literal byte to the last in the same way we optimized the primary units. Literal bytes that match the escape code generate an escape sequence, thus using more bits than unescaped literal bytes and we need to minimize these occurrances.

For each literal byte there is a corresponding new escape code which minimizes the path to the end of the file. If the literal byte's high bits match the current escape code, this new escape code is used next. The escape optimization routine proceeds from the end of the file to the beginning like the graph search, but it proceeds linearly and is thus much faster.

I already noted that the new literal byte tagging system exploits the locality in the literal byte values. If there is no correlation between the bytes, the tagging system does not do well at all. Most of the time, however, the system works very well, performing 50% better than the prefix-bit approach.

The escape optimization routine is currently very fast. A little algorithmic magic removed a lot of code from the original version. Fast escape optimization routine is quite advantageous, because the number of escape bits can now vary from 0 (uncompressed bytes always escaped) to 8 and we need to run the routine again if we change the number of escape bits used to select the optimal escape code changes.

Because escaped literal bytes actually expand the data, we need a safety area, or otherwise the compressed data may get overwritten by the decompressed data before we have used it. Some extra bytes need to be reserved for the end of file marker. The compression routine finds out how many bytes we need for safety buffer by keeping track of the difference between input and output sizes while creating the compressed file.

```
            $1000 .. $2000
            |OOOOOOOO|           O=original file

        $801 ..
        |D|CCCCC|                C=compressed data (D=decompressor)

 $f7..      $1000     $2010
 |D|            |CCCCC|          Before decompression starts
            ^    ^
            W    R               W=write pointer, R=read pointer
```

If the original file is located at $1000-$1fff, and the calculated safety area is 16 bytes, the compressed version will be copied by the decompression routine higher in memory so that the last byte is at $200f. In this way, the minimum amount of other memory is overwritten by the decompression. If the safety area would exceed the top of memory, we need a wrap buffer. This is handled automatically by the compressor. The read pointer wraps from the end of memory to the wrap buffer, allowing the original file to extend upto the end of the memory, all the way to $ffff. You can get the compression program to tell you which memory areas it uses by specifying the "-s" option. Normally the safety buffer needed is less than a dozen bytes.

To sum things up, Pucrunch operates in several steps:

1. Find RLE and LZ77 data, pre-select RLE byte table
2. Graph search, i.e. which primary units to use
3. Primary Units/Literal bytes ratio decides how many escape bits to use
4. Escape optimization, which escape codes to use
5. Update RLE ranks and the RLE byte table
6. Determine the safety area size and output the file.

String Match Speedup

To be able to select the most efficient combination of primary units we of course first need to find out what kind of primary units are available for selection. If the file doesn't have repeated bytes,

we can't use RLE. If the file doesn't have repeating byte strings, we can't use LZ77. This string matching is the most time-consuming operation in LZ77 compression simply because of the amount of the comparison operations needed. Any improvement in the match algorithm can decrease the compression time considerably. Pucrunch is a living proof on that.

The RLE search is straightforward and fast: loop from the current position (`P`) forwards in the file counting each step until a different-valued byte is found or the end of the file is reached. This count can then be used as the RLE byte count (if the graph search decides to use RLE). The code can also be optimized to initialize counts for all locations that belonged to the RLE, because by definition there are only one-valued bytes in each one. Let us mark the current file position by `P`.

```
unsigned char *a = indata + P, val = *a++;
int top = inlen - P;
int rlelen = 1;

/* Loop for the whole RLE */
while (rlelen<top && *a++ == val)
    rlelen++;

for (i=0; i<rlelen-1; i++)
    rle[P+i] = rlelen-i;
```

With LZ77 we can't use the same technique as for RLE (i.e. using the information about current match to skip subsequent file locations to speed up the search). For LZ77 we need to find the longest possible, and **nearest** possible, string that matches the bytes starting from the current location. The nearer the match, the less bits are needed to represent the offset from the current position.

Naively, we could start comparing the strings starting from `P-1` and `P`, remembering the length of the matching part and then doing the same at `P-2` and `P`, `P-3` and `P`, .. `P-j` and `P` (`j` is the maximum search offset). The longest match and its location (offset from the current position) are then remembered and initialized. If we find a match longer or equal than the maximum length we can actually use, we can stop the search there. (The code used to represent the length values may have an upper limit.)

This may be the first implementation that comes to your (and my) mind, and might not seem so bad at first. In reality, it is a very slow way to do the search, the **Brute Force** method. It could take somewhere about (`n^3`) byte compares to process a file of the length `n` (a mathematically inclined person would probably give a better estimate). However, using the already determined RLE value to our advantage permits us to rule out the worst-case projection, which happens when all bytes are the same value. We only search LZ77 matches if the current file position has shorter RLE sequence than the maximum LZ77 copy length.

The first thing I did to improve the speed is to remember the position where each byte has last been seen. A simple 256-entry table handles that. Using this table, the search can directly start from the first potential match, and we don't need to search for it byte-by-byte anymore. The table is continually updated when we move toward to the end of the file.

That didn't give much of an improvement, but then I increased the table to 256*256 entries, making it possible to locate the latest occurrance of any byte **pair** instead. The table indexed with the byte values and the table contents directly gives the position in file where these two bytes were last seen. Because the shortest possible string that would offer any compression (for my encoding of LZ77) is two bytes long, this byte-pair history is very suitable indeed. Also, the first (shortest possible, i.e. 2-byte) match is found directly from the byte-pair history. This gave a moderate 30% decrease in compression time for one of my test files (from 28 minutes to 17 minutes on a 25 MHz 68030).

The second idea was to quickly discard the strings that had no chance of being longer matches than the one already found. A one-byte hash value (sort of a checksum here, it is never used to index a hash table in this algorithm, but I rather use "hash value" than "checksum") is calculated from each three bytes of data. The values are calculated once and put into a table, so we only need two memory fetches to know if two 3-byte strings are different. If the hash values are different, at least one of the data bytes differ. If the hash values are equal, we have to compare the original bytes. The hash values of the strategic positions of the strings to compare are then .. compared. This strategic position is the location two bytes earlier than the longest match so far. If the hash values differ, there is no chance that the match is longer than the current one. It may be not even be as long, because one of the two earlier bytes may be different. If the hash values are equal, the

brute-force byte-by-byte compare has to be done. However, the hash value check already discards a huge number of candidates and more than generously pays back its own memory references. Using the hash values the compression time shortens by 50% (from 17 minutes to 8 minutes).

Okay, the byte-pair table tells us where the latest occurrance of any byte pair is located. Still, for the latest occurrance before **that** one we have to do a brute force search. The next improvement was to use the byte-pair table to generate a linked list of the byte pairs with the same value. In fact, this linked list can be trivially represented as a table, using the same indexing as the file positions. To locate the previous occurrance of a 2-byte string starting at location P, look at `backSkip[P]`.

```
    /* Update the two-byte history & backSkip */
    if (P+1<inlen) {
        int index = (indata[P]<<8) | indata[P+1];

        backSkip[P] = lastPair[index];
        lastPair[index] = P+1;
    }
```

Actually the values in the table are one bigger than the real table indices. This is because the values are of type unsigned short (can only represent non-negative values), and I wanted zero to mean "not occurred".

This table makes the search of the next (previous) location to consider much faster, because it is a single table reference. The compression time was reduced from 6 minutes to 1 minute 10 seconds. Quite an improvement from the original 28 minutes!

```
        backSkip[]   lastPair[]
    ___ _______ ____
       \/      \/    \
    ...JOVE.....JOKA..JOKER
       ^        ^     ^
       |        |     |
       next     |     position
                current match (3)
       C        B     A
```

In this example we are looking at the string "JOKER" at location `A`. Using the `lastPair[]` table (with the index "JO", the byte values at the current location `A`) we can jump directly to the latest match at `B`, which is "JO", 2 bytes long. The hash values for the string at `B` ("JOK") and at `A` ("JOK") are compared. Because they are equal, we have a potential longer match (3 bytes), and the strings "JOKE.." and "JOKA.." are compared. A match of length 3 is found (the 4th byte differs). The `backSkip[]` table with the index `B` gives the previous location where the 2-byte string "JO" can be found, i.e. `C`. The hash value for the strategic position of the string in the current position `A` ("OKE") is then compared to the hash value of the corresponding position in the next potential match starting at `C` ("OVE"). They don't match, so the string starting at `C` ("JOVE..") can't include a longer match than the current longest match at `B`.

There is also another trick that takes advantage of the already determined RLE lengths. If the RLE lengths for the positions to compare don't match, we can directly skip to the next potential match. Note that the RLE bytes (the data bytes) are the same, and need not be compared, because the first byte (two bytes) are always equal on both positions (our `backSkip[]` table guarantees that). The RLE length value can also be used to skip the start of the strings when comparing them.

Another improvement to the search code made it dramatically faster than before on highly redundant files (such as `pic` from the Calgary Corpus Suite, which was the Achilles' heel until then). Basically the new seach method just skips over the RLE part (if any) in the seach position and then checks if the located position has equal number (and value) of RLE bytes before it.

```
backSkip[]      lastPair[]
    _____ ________
         \/        \
      ...AB.....A..ABCD    rle[p] # of A's, B is something else
         ^      ^  ^
         |      |  |
         i      p  p+rle[p]-1
```

The algorithm searches for a two-byte string which starts at `p + rle[p]-1`, i.e. the last rle byte ('A') and the non-matching one ('B'). When it finds such location (simple `lastPair[]` or `backSkip[]` lookup), it checks if the rle in the compare position (`i-(rle[`p]-1)) is long enough (i.e. the same number of A's before the B in both places). If there are, the normal hash value check is performed on the strings

and if it succeeds, the brute-force byte-compare is done.

The rationale behind this idea is quite simple. There are dramatically less matches for "AB" than for "AA", so we get a huge speedup with this approach. We are still guaranteed to find the most recent longest match there is.

Note that a compression method similar to RLE can be realized using just LZ77. You just emit the first byte as a literal byte, and output a LZ77 code with offset 1 and the original RLE length minus 1. You can thus consider RLE as a special case, which offers tighter encoding of the necessary information. Also, as my LZ77 limits the copy size to 64/128/256 bytes, a RLE version providing lengths upto 32 kilobytes is a big improvement, even if the code for it is somewhat longer.

The Decompression Routine

Any lossless compression program is totally useless unless there exists a decompression program which takes in the compressed file and -- using only that information -- generates the original file exactly. In this case the decompression program must run on C64's 6510 microprocessor, which had its impact on the algorithm development also. Regardless of the algorithm, there are several requirements that the decompression code must satisfy:

1. Correctness - the decompression must behave accurately to guarantee lossless decompression
2. Memory usage - the less memory is used the better
3. Speed - fast decompression is preferred to slower one

The latter two requirements can be and are complementary. A somewhat faster decompression for the same algorithm is possible if more memory can be used (although in this case the difference is quite small). In any case the correctness of the result is the most important thing.

A short pseudo-code of the decompression algorithm follows before I go to the actual C64 decompression code.

```
copy the decompression code to low memory
copy the compressed data forward in memory so that it isn't
  overwritten before we have read it (setup safety & wrap buffers)
setup source and destination pointers
initialize RLE byte code table, the number of escape bits etc.
set initial escape code
do forever
    get the number of escape bits "bits"
    if "bits" do not match with the escape code
        read more bits to complete a byte and output it
    else
        get Elias Gamma Code "value" and add 1 to it
        if "value" is 2
            get 1 bit
            if bit is 0
                it is 2-byte LZ77
                get 8 bits for "offset"
                copy 2 bytes from "offset" bytes before current
                    output position into current output position
            else
                get 1 bit
                if bit is 0
                    it is an escaped literal byte
                    get new escape code
                    get more bits to complete a byte with the
                        current escape code and output it
                    use the new escape code
                else
                    it is RLE
                    get Elias Gamma Code "length"
                    if "length" larger or equal than half the maximum
                        it is long RLE
                        get more bits to complete a byte "lo"
                        get Elias Gamma Code "hi", subtract 1
                        combine "lo" and "hi" into "length"
                    endif
                    get Elias Gamma Code "index"
                    if "index" is larger than 31
                        get 3 more bits to complete "byte"
                    else
```

```
                            get "byte" from RLE byte code table from
                                index "index"
                        endif
                        copy "byte" to the output "length" times
                    endif
                endif
            else
                it is LZ77
                get Elias Gamma Code "hi" and subtract 1 from it
                if "hi" is the maximum value - 1
                    end decompression and start program
                endif
                get 8..12 bits "lo" (depending on settings)
                combine "hi" and "lo" into "offset"
                copy "value" number of bytes from "offset" bytes before
                    current output position into current output position
            endif
        endif
    end do
```

The following routine is the pucrunch decompression code. The code runs on the C64 or C128's C64-mode and a modified version is used for Vic20 and C16/Plus4. It can be compiled by at least DASM V2.12.04. Note that the compressor automatically attaches this code to the packet and sets the different parameters to match the compressed data. I will insert additional commentary between strategic code points in addition to the comments that are already in the code.

Note that the version presented here will not be updated when changes are made to the actual version. This version is only presented to help you to understand the algorithm.

```
        processor 6502

BASEND  EQU $2d         ; start of basic variables (updated at EOF)
LZPOS   EQU $2d         ; temporary, BASEND *MUST* *BE* *UPDATED* at EOF

bitstr  EQU $f7         ; Hint the value beforehand
xstore  EQU $c3         ; tape load temp

WRAPBUF EQU $004b       ; 'wrap' buffer, 22 bytes ($02a7 for 89 bytes)

        ORG $0801
        DC.B $0b,8,$ef,0        ; '239 SYS2061'
        DC.B $9e,$32,$30,$36
        DC.B $31,0,0,0

        sei             ; disable interrupts
        inc $d030       ; or "bit $d030" if 2MHz mode is not enabled
        inc 1           ; Select ALL-RAM configuration

        ldx #0          ;** parameter - # of overlap bytes-1 off $ffff
overlap lda $aaaa,x     ;** parameter start of off-end bytes
        sta WRAPBUF,x   ; Copy to wrap/safety buffer
        dex
        bpl overlap

        ldx #block200_end-block200+1    ; $54   ($59 max)
packlp  lda block200-1,x
        sta block200_-1,x
        dex
        bne packlp

        ldx #block_stack_end-block_stack+1      ; $b3   (stack! ~$e8 max)
packlp2 lda block_stack-1,x
        dc.b $9d                ; sta $nnnn,x
        dc.w block_stack_-1     ; (ZP addressing only addresses ZP!)
        dex
        bne packlp2

        ldy #$aa        ;** parameter SIZE high + 1 (max 255 extra bytes)
cploop  dex             ; ldx #$ff on the first round
        lda $aaaa,x     ;** parameter DATAEND-0x100
        sta $ff00,x     ;** parameter ORIG LEN-0x100+ reserved bytes
        txa             ;cpx #0
        bne cploop
        dec cploop+6
        dec cploop+3
        dey
        bne cploop
        jmp main
```

The first part of the code contains a sys command for the basic interpreter, two loops that copy the decompression code to zeropage/stack ($f7-$1aa) and to the system input buffer ($200-$253). The latter code segment contains byte, bit and Gamma Code input routines and the RLE byte code table, the former code segment contains the rest.

This code also copies the compressed data forward in memory so that it won't be overwritten by the decompressed data before we have had a change to read it. The decompression starts at the beginning and proceeds upwards in both the compressed and decompressed data. A safety area is calculated by the compression routine. It finds out how many bytes we need for temporary data expansion, i.e. for escaped bytes. The wrap buffer is used for files that extend upto the end of memory, and would otherwise overwrite the compressed data with decompressed data before it has been read.

This code fragment is not used during the decompression itself. In fact the code will normally be overwritten when the actual decompression starts.

The very start of the next code block is located inside the zero page and the rest fills the lowest portion of the microprocessor stack. The zero page is used to make the references to different variables shorter and faster. Also, the variables don't take extra code to initialize, because they are copied with the same copy loop as the rest of the code.

```
block_stack
#rorg $f7       ; $f7 - ~$1e0
block_stack_

bitstr  dc.b $80        ; ZP    $80 == Empty
esc     dc.b $00        ; ** parameter (saves a byte when here)

OUTPOS = *+1            ; ZP
putch   sta $aaaa       ; ** parameter
        inc OUTPOS      ; ZP
        bne 0$          ; Note: beq 0$; rts; 0$: inc OUTPOS+1;rts would be
; $0100                 ;       faster, but 1 byte longer
        inc OUTPOS+1    ; ZP
0$      rts
```

`putch` is the subroutine that is used to output the decompressed bytes. In this case the bytes are written to memory. Because the subroutine call itself takes 12 cycles (6 for `jsr` and another 6 for `rts`), and the routine is called a lot of times during the decompression, the routine itself should be as fast as possible. This is achieved by removing the need to save any registers. This is done by using an absolute addressing mode instead of indirect indexed or absolute indexed addressing (`sta $aaaa` instead of `sta ($zz),y` or `sta $aa00,y`). With indexed addressing you would need to save+clear+restore the index register value in the routine.

Further improvement in code size and execution speed is done by storing the instruction that does the absolute addressing to zero page. When the memory address is incremented we can use zero-page addressing for it too. On the other hand, the most time is spent in the bit input routine so further optimization of this routine is not feasible.

```
newesc  ldy esc         ; remember the old code (top bits for escaped byte)
        ldx #2          ; ** PARAMETER
        jsr getchkf     ; get & save the new escape code
        sta esc
        tya             ; pre-set the bits
        ; Fall through and get the rest of the bits.
noesc   ldx #6          ; ** PARAMETER
        jsr getchkf
        jsr putch       ; output the escaped/normal byte
        ; Fall through and check the escape bits again
main    ldy #0          ; Reset to a defined state
        tya             ; A = 0
        ldx #2          ; ** PARAMETER
        jsr getchkf     ; X=2 -> X=0
        cmp esc
        bne noesc       ; Not the escape code -> get the rest of the byte
        ; Fall through to packed code
```

The decompression code is first entered in `main`. It first clears the accumulator and the Y register and then gets the escape bits (if any are used) from the input stream. If they don't match with the

current escape code, we get more bits to complete a byte and then output the result. If the escape bits match, we have to do further checks to see what to do.

```
        jsr getval      ; X = 0
        sta xstore      ; save the length for a later time
        cmp #1          ; LEN == 2 ?
        bne lz77        ; LEN != 2      -> LZ77
        tya             ; A = 0
        jsr get1bit     ; X = 0
        lsr             ; bit -> C, A = 0
        bcc lz77_2      ; A=0 -> LZPOS+1
        ;***FALL THRU***
```

We first get the Elias Gamma Code value (or actually my independently developed version). If it says the LZ77 match length is greater than 2, it means a LZ77 code and we jump to the proper routine. Remember that the lengths are decremented before encoding, so the code value 1 means the length is 2. If the length is two, we get a bit to decide if we have LZ77 or something else. We have to clear the accumulator, because `get1bit` does not do that automatically.

If the bit we got (shifted to carry to clear the accumulator) was zero, it is LZ77 with an 8-bit offset. If the bit was one, we get another bit which decides between RLE and an escaped byte. A zero-bit means an escaped byte and the routine that is called also changes the escape bits to a new value. A one-bit means either a short or long RLE.

```
        ; e..e01
        jsr get1bit     ; X = 0
        lsr             ; bit -> C, A = 0
        bcc newesc      ; e..e010
        ;***FALL THRU***

        ; e..e011
srle    iny             ; Y is 1 bigger than MSB loops
        jsr getval      ; Y is 1, get len, X = 0
        sta xstore      ; Save length LSB
        cmp #64         ; ** PARAMETER 63-64 -> C clear, 64-64 -> C set..
        bcc chrcode     ; short RLE, get bytecode

longrle ldx #2          ; ** PARAMETER  111111xxxxxx
        jsr getbits     ; get 3/2/1 more bits to get a full byte, X = 0
        sta xstore      ; Save length LSB

        jsr getval      ; length MSB, X = 0
        tay             ; Y is 1 bigger than MSB loops
```

The short RLE only uses half (or actually 1 value less than a half) of the gamma code range. Larger values switches us into long RLE mode. Because there are several values, we already know some bits of the length value. Depending on the gamma code maximum value we need to get from one to three bits more to assemble a full byte, which is then used as the less significant part for the run length count. The upper part is encoded using the same gamma code we are using everywhere. This limits the run length to 16 kilobytes for the smallest maximum value (`-m5`) and to the full 64 kilobytes for the largest value (`-m7`).

Additional compression for RLE is gained using a table for the 31 top-ranking RLE bytes. We get an index from the input. If it is from 1 to 31, we use it to index the table. If the value is larger, the lower 5 bits of the value gives us the 5 most significant bits of the byte to repeat. In this case we read 3 additional bits to complete the byte.

```
chrcode jsr getval      ; Byte Code, X = 0
        tax             ; this is executed most of the time anyway
        lda table-1,x   ; Saves one jump if done here (loses one txa)

        cpx #32         ; 31-32 -> C clear, 32-32 -> C set..
        bcc 1$          ; 1..31 -> the byte to repeat is in A

        ; Not ranks 1..31, -> 111110xxxxx (32..64), get byte..
        txa             ; get back the value (5 valid bits)
        jsr get3bit     ; get 3 more bits to get a full byte, X = 0

1$      ldx xstore      ; get length LSB
        inx             ; adjust for cpx#$ff;bne -> bne
dorle   jsr putch
        dex
        bne dorle       ; xstore 0..255 -> 1..256
        deym
        bne dorle       ; Y was 1 bigger than wanted originally
```

```
mainbeq beq main        ; reverse condition -> jump always
```

After deciding the repeat count and decoding the value to repeat we simply have to output the value enough times. The X register holds the lower part and the Y register holds the upper part of the count. The X register value is first incremented by one to change the code sequence `dex ; cpx #$ff ; bne dorle` into simply `dex ; bne dorle`. This may seem strange, but it saves one byte in the decompression code and two clock cycles for each byte that is outputted. It's almost a ten percent improvement. :-)

The next code fragment is the LZ77 decode routine and it is used in the file parts that do not have equal byte runs (and even in some that have). The routine simply gets an offset value and copies a sequence of bytes from the already decompressed portion to the current output position.

```
lz77    jsr getval      ; X=0 -> X=0
        cmp #127        ; ** PARAMETER  Clears carry (is maximum value)
        beq eof         ; EOF

        sbc #0          ; C is clear -> subtract 1  (1..126 -> 0..125)
        ldx #0          ; ** PARAMETER (more bits to get)
        jsr getchkf     ; clears Carry, X=0 -> X=0

lz77_2  sta LZPOS+1     ; offset MSB
        ldx #8
        jsr getbits     ; clears Carry, X=8 -> X=0
                        ; Note: Already eor:ed in the compressor..
        ;eor #255       ; offset LSB 2's complement -1 (i.e. -X = ~X+1)
        adc OUTPOS      ; -offset -1 + curpos (C is clear)
        sta LZPOS

        lda OUTPOS+1
        sbc LZPOS+1     ; takes C into account
        sta LZPOS+1     ; copy X+1 number of chars from LZPOS to OUTPOS
        ;ldy #0         ; Y was 0 originally, we don't change it

        ldx xstore      ; LZLEN
        inx             ; adjust for cpx#$ff;bne -> bne
lzloop  lda (LZPOS),y
        jsr putch
        iny             ; Y does not wrap because X=0..255 and Y initially 0
        dex
        bne lzloop      ; X loops, (256,1..255)
        beq mainbeq     ; jump through another beq (-1 byte, +3 cycles)
```

There are two entry-points to the LZ77 decode routine. The first one (`lz77`) is for copy lengths bigger than 2. The second entry point (`lz77_2`) is for the length of 2 (8-bit offset value).

```
        ; EOF
eof     lda #$37        ; ** could be a PARAMETER
        sta 1
        dec $d030       ; or "bit $d030" if 2MHz mode is not enabled
        lda OUTPOS      ; Set the basic prg end address
        sta BASEND
        lda OUTPOS+1
        sta BASEND+1
        cli             ; ** could be a PARAMETER
        jmp $aaaa       ; ** PARAMETER
#rend
block_stack_end
```

Some kind of a end of file marker is necessary for all variable-length codes. Otherwise we could not be certain when to stop decoding. Sometimes the byte count of the original file is used instead, but here a special EOF condition is more convenient. If the high part of a LZ77 offset is the maximum gamma code value, we have reached the end of file and must stop decoding. The end of file code turns on BASIC and KERNEL, turns off 2 MHz mode (for C128) and updates the basic end addresses before allowing interrupts and jumping to the program start address.

The next code fragment is put into the system input buffer. The routines are for getting bits from the encoded message (`getbits`) and decoding the Elias Gamma Code (`getval`). The table at the end contains the ranked RLE bytes. The compressor automatically decreases the table size if not all of the values are used.

```
block200
#rorg   $200    ; $200-$258
block200_

getnew  pha             ; 1 Byte/3 cycles
INPOS = *+1
        lda $aaaa       ;** parameter
        rol             ; Shift in C=1 (last bit marker)
        sta bitstr      ; bitstr initial value = $80 == empty
        inc INPOS       ; Does not change C!
        bne 0$
        inc INPOS+1     ; Does not change C!
        bne 0$
        ; This code does not change C!
        lda #WRAPBUF    ; Wrap from $ffff->$0000 -> WRAPBUF
        sta INPOS
0$      pla             ; 1 Byte/4 cycles
        rts


; getval : Gets a 'static huffman coded' value
; ** Scratches X, returns the value in A **
getval  inx             ; X must be 0 when called!
        txa             ; set the top bit (value is 1..255)
0$      asl bitstr
        bne 1$
        jsr getnew
1$      bcc getchk      ; got 0-bit
        inx
        cpx #7          ; ** parameter
        bne 0$
        beq getchk      ; inverse condition -> jump always

; getbits: Gets X bits from the stream
; ** Scratches X, returns the value in A **
get1bit inx             ;2
getbits asl bitstr
        bne 1$
        jsr getnew
1$      rol             ;2
getchk  dex             ;2              more bits to get ?
getchkf bne getbits     ;2/3
        clc             ;2              return carry cleared
        rts             ;6+6


table   dc.b 0,0,0,0,0,0,0
        dc.b 0,0,0,0,0,0,0,0
        dc.b 0,0,0,0,0,0,0,0
        dc.b 0,0,0,0,0,0,0,0

#rend
block200_end
```

Target Application Compression Tests

The following data compression tests are made on my four C64 test files:
`bs.bin` is a demo part, about 50% code and 50% graphics data
`delenn.bin` is a BFLI picture with a viewer, a lot of dithering
`sheridan.bin` is a BFLI picture with a viewer, dithering, black areas
`ivanova.bin` is a BFLI picture with a viewer, dithering, larger black areas

Packer	**Size**	**Left**	**Comment**
bs.bin	**41537**		
ByteBonker 1.5	27326	65.8%	Mode 4
Cruelcrunch 2.2	27136	65.3%	Mode 1
The AB Cruncher	27020	65.1%	
ByteBoiler (REU)	26745	64.4%	
RLE + ByteBoiler (REU)	26654	64.2%	
PuCrunch	26300	63.3%	-m5 -fdelta

delenn.bin	**47105**		
The AB Cruncher	N/A	N/A	Crashes
ByteBonker 1.5	21029	44.6%	Mode 3
Cruelcrunch 2.2	20672	43.9%	Mode 1
ByteBoiler (REU)	20371	43.2%	
RLE + ByteBoiler (REU)	19838	42.1%	
PuCrunch	19710	41.8%	-p2 -fshort
sheridan.bin	**47105**		
ByteBonker 1.5	13661	29.0%	Mode 3
Cruelcrunch 2.2	13595	28.9%	Mode H
The AB Cruncher	13534	28.7%	
ByteBoiler (REU)	13308	28.3%	
PuCrunch	12502	26.6%	-p2 -fshort
RLE + ByteBoiler (REU)	12478	26.5%	
ivanova.bin	**47105**		
ByteBonker 1.5	11016	23.4%	Mode 1
Cruelcrunch 2.2	10883	23.1%	Mode H
The AB Cruncher	10743	22.8%	
ByteBoiler (REU)	10550	22.4%	
PuCrunch	9820	20.9%	-p2 -fshort
RLE + ByteBoiler (REU)	9813	20.8%	
LhA	9543	20.3%	Decompressor not included
gzip -9	9474	20.1%	Decompressor not included

Calgary Corpus Suite

The original compressor only allows files upto 63 kB. To be able to compare my algorithm to others I modified the compressor to allow bigger files. I then got some reference results using the Calgary Corpus test suite.

Note that these results do not include the decompression code (`-c0`). Instead a 46-byte header is attached and the file can be decompressed with a standalone decompressor.

To tell you the truth, the results surprised me, because the compression algorithm **IS** developed for a very special case in mind. It only has a fixed code for LZ77/RLE lengths, not even a static one (fixed != static != adaptive)! Also, it does not use arithmetic code (or Huffman) to compress the literal bytes. Because most of the big files are ASCII text, this somewhat handicaps my compressor, although the new tagging system is very happy with 7-bit ASCII input. Also, decompression is relatively fast, and uses no extra memory.

I'm getting relatively near LhA for 4 files, and shorter than LhA for 9 files.

The table contains the file name (file), compression options (options) except `-c0` which specifies standalone decompression, the original file size (in) and the compressed file size (out) in bytes, average number of bits used to encode one byte (b/B), remaining size (ratio) and the reduction (gained), and the time used for compression. For comparison, the last three columns show the compressed sizes for LhA, Zip and GZip (with the -9 option), respectively.

FreeBSD epsilon3.vlsi.fi PentiumPro® 200MHz								**LhA**	**Zip**	**GZip-9**
Estimated decompression on a C64 (1MHz 6510) 6:47										
file	**options**	**in**	**out**	**b/B**	**ratio**	**gained**	**time**	**out**	**out**	**out**

bib	-p4	111261	35180	2.53	31.62%	68.38%	3.8	40740	35041	34900
book1	-p4	768771	318642	3.32	41.45%	58.55%	38.0	339074	313352	312281
book2	-p4	610856	208350	2.73	34.11%	65.89%	23.3	228442	206663	206158
geo	-p2	102400	72535	5.67	70.84%	29.16%	6.0	68574	68471	68414
news	-p3 -fdelta	377109	144246	3.07	38.26%	61.74%	31.2	155084	144817	144400
obj1	-m6 -fdelta	21504	10238	3.81	47.61%	52.39%	1.0	10310	10300	10320
obj2		246814	82769	2.69	33.54%	66.46%	7.5	84981	81608	81087
paper1	-p2	53161	19259	2.90	36.23%	63.77%	0.9	19676	18552	18543
paper2	-p3	82199	30399	2.96	36.99%	63.01%	2.4	32096	29728	29667
paper3	-p2	46526	18957	3.26	40.75%	59.25%	0.8	18949	18072	18074
paper4	-p1 -m5	13286	5818	3.51	43.80%	56.20%	0.1	5558	5511	5534
paper5	-p1 -m5	11954	5217	3.50	43.65%	56.35%	0.1	4990	4970	4995
paper6	-p2	38105	13882	2.92	36.44%	63.56%	0.5	13814	13207	13213
pic	-p1	513216	57558	0.90	11.22%	88.78%	16.4	52221	56420	52381
progc	-p1	39611	13944	2.82	35.21%	64.79%	0.5	13941	13251	13261
progl	-p1 -fdelta	71646	16746	1.87	23.38%	76.62%	6.1	16914	16249	16164
progp		49379	11543	1.88	23.38%	76.62%	0.8	11507	11222	11186
trans	-p2	93695	19234	1.65	20.53%	79.47%	2.2	22578	18961	18862
total		**3251493**	**1084517**	**2.67**	**33.35%**	**66.65%**	**2:21**			

Canterbury Corpus Suite

The following shows the results on the Canterbury corpus. Again, I am quite pleased with the results. For example, pucrunch beats GZip -9 for `lcet10.txt`.

FreeBSD epsilon3.vlsi.fi PentiumPro® 200MHz								**LhA**	**Zip**	**GZip-9**
Estimated decompression on a C64 (1MHz 6510) 6:00										
file	**options**	**in**	**out**	**b/B**	**ratio**	**gained**	**time**	**out**	**out**	**out**
alice29.txt	-p4	152089	54826	2.89	36.05%	63.95%	8.8	59160	54525	54191
ptt5	-p1	513216	57558	0.90	11.22%	88.78%	21.4	52272	56526	52382
fields.c		11150	3228	2.32	28.96%	71.04%	0.1	3180	3230	3136
kennedy.xls		1029744	265610	2.07	25.80%	74.20%	497.3	198354	206869	209733
sum		38240	13057	2.74	34.15%	65.85%	0.5	14016	13006	12772
lcet10.txt	-p4	426754	144308	2.71	33.82%	66.18%	23.9	159689	144974	144429
plrabn12.txt	-p4	481861	198857	3.31	41.27%	58.73%	33.8	210132	195299	194277
cp.html	-p1	24603	8402	2.74	34.16%	65.84%	0.3	8402	8085	7981
grammar.lsp	-m5	3721	1314	2.83	35.32%	64.68%	0.0	1280	1336	1246
xargs.1	-m5	4227	1840	3.49	43.53%	56.47%	0.0	1790	1842	1756
asyoulik.txt	-p4	125179	50317	3.22	40.20%	59.80%	5.9	52377	49042	48829
total		**2810784**	**799317**	**2.28**	**28.44%**	**71.56%**	**9:51**			

Conclusions

In this article I have presented a compression program which creates compressed executable files for C64, VIC20 and Plus4/C16. The compression can be performed on Amiga, MS-DOS/Win machine or any other machine with a C-compiler. A powerful machine allows asymmetric compression: a lot of resources can be used to compress the data while needing minimal resources

for decompression. This was one of the design requirements.

Two original ideas were presented: a new literal byte tagging system and an algorithm using hybrid RLE and LZ77. Also, a detailed explanation of the LZ77 string match routine and the optimal parsing scheme was presented.

The compression ratio and decompression speed is comparable to other compression programs for Commodore 8-bit computers.

But what are then the real advantages of pucrunch compared to traditional C64 compression programs in addition to that you can now compress VIC20 and Plus4/C16 programs? Because I'm lousy at praising my own work, I let you see some actual user comments. I have edited the correspondence a little, but I hope he doesn't mind. My comments are indented.

A big advantage is that pucrunch does RLE and LZ in one pass. For demos I only used a cruncher and did my own RLE routines as it is somewhat annoying to use an external program for this. These programs require some memory and ZP-addresses like the cruncher does. So it can easily happen that the decruncher or depacker interfere with your demo-part, if you didn't know what memory is used by the depacker. At least you have more restrictions to care about. With pucrunch you can do RLE and LZ without having too much of these restrictions.

> Right, and because pucrunch is designed that way from the start, it can get better results with one-pass RLE and LZ than doing them separately. On the other hand it more or less requires that you _don't_ RLE-pack the file first..

This is true, we also found that out. We did a part for our demo which had some tables using only the low-nybble. Also the bitmap had to be filled with a specific pattern. We did some small routines to shorten the part, but as we tried pucrunch, this became obsolete. From 59xxx bytes to 12xxx or 50 blocks, with our own RLE and a different cruncher we got 60 blks! Not bad at all ;)

Not to mention that you have the complete and commented source-code for the decruncher, so that you can easily change it to your own needs. And it's not only very flexible, it is also very powerful. In general pucrunch does a better job than ByteBoiler+Sledgehammer.

In addition to that pucrunch is of course much faster than crunchers on my C64, this has not only to do with my 486/66 and the use of an HDD. See, I use a cross-assembler-system, and with pucrunch I don't have to transfer the assembled code to my 64, crunch it, and transfer it back to my pc. Now, it's just a simple command-line and here we go... And not only I can do this, my friend who has an amiga uses pucrunch as well. This is the first time we use the same cruncher, since I used to take ByteBoiler, but my friend didn't have a REU so he had to try another cruncher.

So, if I try to make a conclusion: It's fast, powerful and extremly flexible (thanks to the source-code).

Just for your info...

We won the demo-competition at the Interjam'98 and everything that was crunched ran through the hands of pucrunch... Of course, you have been mentioned in the credits. If you want to take a look, search for KNOOPS/DREAMS, which should be on the ftp-servers in some time. So, again, THANKS! :)

Ninja/DREAMS

So, what can I possibly hope to add to that, right?:-)

If you have any comments, questions, article suggestions or just a general hello brewing in your mind, send me mail or visit my homepage.

See you all again in the next issue!

-Pasi

Source Code and Executables

Sources

The current compressor can compress/decompress files for C64 (-c64), VIC20 (-c20), C16/+4 (-c16), or for standalone decompressor (-c0).

As of 21.12.2005 Pucrunch is under GNU LGPL: See http://creativecommons.org/licenses/LGPL/2.1/ or http://www.gnu.org/copyleft/lesser.html.

The decompression code is distributed under the WXWindows Library Licence: See http://www.wxwidgets.org/licence3.txt. (In short: binary version of the decompression code can accompany the compressed data or used in decompression programs.)

The current compressor version
- pucrunch.c pucrunch.h (Makefile for gcc, smakefile for SAS/C)
- uncrunch.asm - the embedded decompressor
- sa_uncrunch.asm - the standalone decompressor
- uncrunch-z80.asm - example decompressor for GameBoy
- http://bree.homeunix.net/~ccfg/pucrunch/ - Fast and short decompressor for Z80 by Jussi Pitkänen

A very simple 'linker' utility
- cbmcombine.c

Executables

AmigaDOS executables
- pucrunch_Amiga.lha (old version)

WIN95/98/NT/WIN2000 executables
- pucrunch_x86.zip (old version 15.7.2003) (the older version pucrunch_x86_old.zip can be used with plain DOS)

  Note: this PC version is compiled with VisualC. The old version should work with plain DOS (with the GO32-V2.EXE extender), W95 and NT but it contains some bugs.

Usage

```
Usage: pucrunch [-«flags»] [«infile» [«outfile»]]
     c«val»    machine:
     a         avoid video matrix (for VIC20)
     d         data (no loading address)
     l«val»    set/override load address
     x«val»    set execution address
     e«val»    force escape bits
     r«val»    restrict lz search range
    +f         disable fast mode
    -flist     lists available decompressor versions
    -fbasic    select the decompressor for basic programs (VIC20 and C64)
    -ffast     select the faster but longer decompressor, if available
    -fshort    select the shorter but slower decompressor, if available
    -fdelta    use waveform matching
     n         no RLE/LZ length optimization
     s         full statistics
     p«val»    force extralzposbits
     m«val»    max len 5..7 (64/128/256)
     i«val»    interrupt enable after decompress (0=disable)
     g«val»    memory configuration after decompress
     u         unpack
```

Pucrunch expects any number of options and upto two filenames. If you only give one filename, the compressed file is written to the stardard output. If you leave out both filenames, the input is in addition read from the stardard input. Options needing no value can be grouped together. All values can be given in decimal (no prefix), octal (prefix 0), or hexadecimal (prefix $ or 0x).

Example: `pucrunch demo.prg demo.pck -m6 -fs -p2 -x0xc010`

Option descriptions:

c«val»
: Selects the machine. Possible values are 128(C128), 64(C64), 20(VIC20), 16(C16/Plus4), 0(standalone). The default is 64, i.e. Commodore 64. If you use -c0, a packet without the embedded decompression code is produced. This can be decompressed with a standalone routine and of course with pucrunch itself. The 128-mode is not fully developed yet. Currently it overwrites memory locations $f7-$f9 (Text mode lockout, Scrolling, and Bell settings) without restoring them later.

a
: Avoids video matrix if possible. Only affects VIC20 mode.

d
: Indicates that the file does not have a load address. A load address can be specified with `-l` option. The default load address if none is specified is 0x258.

l«val»
: Overrides the file load address or sets it for data files.

x«val»
: Sets the execution address or overrides automatically detected execution address. Pucrunch checks whether a SYS-line is present and tries to decode the address. Plain decimal addresses and addresses in parenthesis are read correctly, otherwise you need to override any incorrect value with this option.

e«val»
: Fixes the number of escape bits used. You don't usually need or want to use this option.

r«val»
: Sets the LZ77 search range. By specifying 0 you get only RLE. You don't usually need or want to use this option.

+f
: Disables 2MHz mode for C128 and 2X mode in C16/+4.

-fbasic
: Selects the decompressor for basic programs. This version performs the RUN function and enters the basic interpreter automatically. Currently only C64 and VIC20 are supported.

-ffast
: Selects the faster, but longer decompressor version, if such version is available for the selected machine and selected options. Without this option the medium-speed and medium-size decompressor is used.

-fshort
: Selects the shorter, but slower decompressor version, if such version is available for the selected machine and selected options. Without this option the medium-speed and medium-size decompressor is used.

-fdelta
: Allows delta matching. In this mode only the waveforms in the data matter, any offset is allowed and added in the decompression. Note that the decompressor becomes 22 bytes longer if delta matching is used and the short decompressor can't be used (24 bytes more). This means that delta matching must get more than 46 bytes of total gain to get any net savings. So, always compare the result size to a version compressed without `-fdelta`.

  Also, the compression time increases because delta matching is more complicated. The increase is not 256-fold though, somewhere around 6-7 times is more typical. So, use this option with care and do not be surprised if it doesn't help on your files.

n
: Disables RLE and LZ77 length optimization. You don't usually need or want to use this option.

s
: Display full statistics instead of a compression summary.

p«val»
: Fixes the number of extra LZ77 position bits used for the low part. If pucrunch tells you to to use this option, see if the new setting gives better compression.

m«val»
: Sets the maximum length value. The value should be 5, 6, or 7. The lengths are 64, 128 and 256, respectively. If pucrunch tells you to to use this option, see if the new setting gives better compression. The default value is 7.

i«val»
: Defines the interrupt enable state to be used after decompression. Value 0 disables interrupts, other values enable interrupts. The default is to enable interrupts after decompression.

g«val»
: Defines the memory configuration to be used after decompression. Only used for C64 mode (`-c64`). The default value is $37.

u
: Unpacks/decompresses a file instead of compressing it. The file to decompress must have a decompression header compatible with one of the decompression headers in the current version.

Note: Because pucrunch contains both RLE and LZ77 and they are specifically designed to work together, **DO NOT** RLE-pack your files first, because it will decrease the overall compression ratio.

The Log Book

26.2.1997
: One byte-pair history buffer gives 30% shorter time compared to a single-byte history buffer.

28.2.1997
: Calculate hash values (byte) for each threes of bytes for faster search, and use the 2-byte history to locate the last occurrance of the 2 bytes to get the minimal LZ sequence. 50% shorter time.

  'Reworded' some of the code to help the compiler generate better code, although it still is not quite 'optimal'.. Progress reports halved. Checks the hash value at old maxval before checking the actual bytes. 20% shorter time. 77% shorter time total (28->6.5).

1.3.1997
: Added a table which extends the lastPair functionality. The table backSkip chains the positions with the same char pairs. 80% shorter time.

5.3.1997
: Tried reverse LZ, i.e. mirrored history buffer. Gained some bytes, but its not really worth it, i.e. the compress time increases hugely and the decompressor gets bigger.

6.3.1997
: Tried to have a code to use the last LZ copy position (offset added to the lastly used LZ copy position). On bs.bin I gained 57 bytes, but in fact the net gain was only 2 bytes (decompressor becomes ~25 bytes longer, and the lengthening of the long rle codes takes away the rest 30).

10.3.1997
: Discovered that my representation of integers 1-63 is in fact an Elias Gamma Code. Tried Fibonacci code instead, but it was much worse (~500 bytes on bs.bin, ~300 bytes on delenn.bin) without even counting the expansion of the decompression code.

12.3.1997
: 'huffman' coded RLE byte -> ~70 bytes gain for bs.bin. The RLE bytes used are ranked, and top 15 are put into a table, which is indexed by a Elias Gamma Code. Other RLE bytes get a prefix "1111".

15.3.1997

The number of escape bits used is again selectable. Using only one escape bit for delenn.bin gains ~150 bytes. If -e-option is not selected, automatically selects the number of escape bits (is a bit slow).

16.3.1997
Changed some arrays to short. 17 x inlen + 64kB memory used. opt_escape() only needs two 16-element arrays now and is slightly faster.

31.3.1997
Tried to use BASIC ROM as a codebook, but the results were not so good. For mostly-graphics files there are no long matches -> no net gain, for mostly-code files the file itself gives a better codebook.. Not to mention that using the BASIC ROM as a codebook is not 100% compatible.

1.4.1997
Tried maxlen 128, but it only gained 17 bytes on ivanova.bin, and lost ~15 byte on bs.bin. This also increased the LZPOS maximum value from ~16k to ~32k, but it also had little effect.

2.4.1997
Changed to coding so that LZ77 has the priority. 2-byte LZ matches are coded in a special way without big loss in efficiency, and codes also RLE/Escape.

5.4.1997
Tried histogram normalization on LZLEN, but it really did not gain much of anything, not even counting the mapping table from index to value that is needed.

11.4.1997
8..14 bit LZPOS base part. Automatic selection. Some more bytes are gained if the proper selection is done before the LZ/RLELEN optimization. However, it can't really be done automatically before that, because it is a recursive process and the original LZ/RLE lengths are lost in the first optimization..

22.4.1997
Found a way to speed up the almost pathological cases by using the RLE table to skip the matching beginnings.

2.5.1997
Switched to maximum length of 128 to get better results on the Calgary Corpus test suite.

25.5.1997
Made the maximum length adjustable. -m5, -m6, and -m7 select 64, 128 and 256 respectively. The decompression code now allows escape bits from 0 to 8.

1.6.1997
Optimized the escape optimization routine. It now takes almost no time at all. It used a whole lot of time on large escape bit values before. The speedup came from a couple of generic data structure optimizations and loop removals by informal deductions.

3.6.1997
Figured out another, better way to speed up the pathological cases. Reduced the run time to a fraction of the original time. All 64k files are compressed under one minute on my 25 MHz 68030. `pic` from the Calgary Corpus Suite is now compressed in 19 seconds instead of 7 minutes (200 MHz Pentium w/ FreeBSD). Compression of ivanova.bin (one of my problem cases) was reduced from about 15 minutes to 47 seconds. The compression of bs.bin has been reduced from 28 minutes (the first version) to 24 seconds. An excellent example of how the changes in the algorithm level gives the most impressive speedups.

6.6.1997
Changed the command line switches to use the standard approach.

11.6.1997
Now determines the number of bytes needed for temporary data expansion (i.e. escaped bytes). Warns if there is not enough memory to allow successful decompression on a C64.

Also, now it's possible to decompress the files compressed with the program (must be the same version). (-u)

17.6.1997
Only checks the lengths that are power of two's in OptimizeLength(), because it does not seem to be any (much) worse than checking every length. (Smaller than found maximum lengths are checked because they may result in a shorter file.) This version (compiled with optimizations on) only spends 27 seconds on ivanova.bin.

19.6.1997
Removed 4 bytes from the decrunch code (begins to be quite tight now unless some features are removed) and simultaneously removed a not-yet-occurred hidden bug.

23.6.1997
Checked the theoretical gain from using the lastly outputted byte (conditional probabilities) to set the probabilities for normal/LZ77/RLE selection. The number of bits needed to code the selection is from 0.0 to 1.58, but even using arithmetic code to encode it, the original escape system is only 82 bits worse (ivanova.bin), 7881/7963 bits total. The former figure is calculated from the entropy, the latter includes LZ77/RLE/escape select bits and actual escapes.

18.7.1997
In LZ77 match we now check if a longer match (further away) really gains more bits. Increase in match length can make the code 2 bits longer. Increase in match offset can make the code even longer (2 bits for each magnitude). Also, if LZPOS low part is longer than 8, the extra bits make the code longer if the length becomes longer than two.

ivanova -5 bytes, sheridan -14, delenn -26, bs -29

When generating the output rescans the LZ77 matches. This is because the optimization can shorten the matches and a shorter match may be found much nearer than the original longer match. Because longer offsets usually use more bits than shorter ones, we get some bits off for each match of this kind. Actually, the rescan should be done in OptimizeLength() to get the most out of it, but it is too much work right now (and would make the optimize even slower).

29.8.1997
4 bytes removed from the decrunch code. I have to thank Tim Rogers (timr@eurodltd co uk) for helping with 2 of them.

12.9.1997
Because SuperCPU doesn't work correctly with inc/dec $d030, I made the 2 MHz user-selectable and off by default. (-f)

13.9.1997
Today I found out that most of my fast string matching algorithm matches the one developed by [Fenwick and Gutmann, 1994]*. It's quite frustrating to see that you are not a genius after all and someone else has had the same idea. :-) However, using the RLE table to help still seems to be an original idea, which helps immensely on the worst cases. I still haven't read their paper on this, so I'll just have to get it and see..

* [Fenwick and Gutmann, 1994]. P.M. Fenwick and P.C. Gutmann, "Fast LZ77 String Matching", Dept of Computer Science, The University of Auckland, Tech Report 102, Sep 1994

14.9.1997
The new decompression code can decompress files from $258 to $ffff (or actually all the way upto $1002d :-). The drawback is: the decompression code became 17 bytes longer. However, the old decompression code is used if the wrap option is not needed.

16.9.1997
The backSkip table can now be fixed size (64 kWord) instead of growing enormous for "BIG" files. Unfortunately, if the fixed-size table is used, the LZ77 rescan is impractical (well, just a little slow, as we would need to recreate the backSkip table again). On the other hand the rescan did not gain so many bytes in the first place (percentage). The define BACKSKIP_FULL enables the old behavior (default). Note also, that for smaller files than 64kB (the primary target files) the default consumes less memory.

The hash value compare that is used to discard impossible matches does not help much. Although it halves the number of strings to consider (compared to a direct one-byte compare),

speedwise the difference is negligible. I suppose a mismatch is found very quickly when the strings are compared starting from the third charater (the two first characters are equal, because we have a full hash table). According to one test file, on average 3.8 byte-compares are done for each potential match. A define HASH_COMPARE enables (default) the hash version of the compare, in which case "inlen" bytes more memory is used.

After removing the hash compare my algorithm quite closely follows the [Fenwick and Gutmann, 1994] fast string matching algorithm (except the RLE trick). (Although I *still* haven't read it.)

14 x inlen + 256 kB of memory is used (with no HASH_COMPARE and without BACKSKIP_FULL).

18.9.1997
One byte removed from the decompression code (both versions).

30.12.1997
Only records longer matches if they compress better than shorter ones. I.e. a match of length N at offset L can be better than a match of length N+1 at 4*L. The old comparison was "better or equal" (">="). The new comparison "better" (">") gives better results on all Calgary Corpus files except "geo", which loses 101 bytes (0.14% of the compressed size).

An extra check/rescan for 2-byte matches in OptimizeLength() increased the compression ratio for "geo" considerably, back to the original and better. It seems to help for the other files also. Unfortunately this only works with the full backskip table (BACKSKIP_FULL defined).

21.2.1998
Compression/Decompression for VIC20 and C16/+4 incorporated into the same program.

16.3.1998
Removed two bytes from the decompression codes.

17.8.1998
There was a small bug in pucrunch which caused the location $2c30 to be incremented (inc $2c30 instead of bit $d030) when run without the -f option. The source is fixed and executables are now updated.

21.10.1998
Added interrupt state (`-i«val»`) and memory configuration (`-g«val»`) settings. These settings define which memory configuration is used and whether interrupts will be enabled or disabled after decompression. The executables have been recompiled.

13.2.1999
Verified the standalone decompressor. With -c0 you can now generate a packet without the attached decompressor. The decompressor allows the data anywhere in memory provided the decompressed data doesn't overwrite it.

25.8.1999
Automatised the decompressor relocation information generation and merged all decompressor versions into the same `uncrunch.asm` file. All C64, VIC20 and C16/+4 decompressor variations are now generated from this file. In addition to the default version, a fast but longer, and a short but slower decompressor is provided for C64. You can select these with `-ffast` and `-fshort`, respectively. VIC20 and C16/+4 now also have a non-wrap versions. These save 24 bytes compared to the wrap versions.

3.12.1999
Delta LZ77 matching was added to help some weird demo parts on the suggestion of Wanja Gayk (Brix/Plush). In this mode (`-fdelta`) the DC offset in the data is discarded when making matches: only the 'waveform' must be identical. The offset is restored when decompressing. As a bonus this match mode can generate descending and ascending strings with any step size. Sounds great, but there are some problems:

- these matches are relatively rare -- usually basic LZ77 can match the same strings
- the code for these matches is quite long (at least currently) -- 4-byte matches are needed to gain anything
- the decompressor becomes 22 bytes longer if delta matching is used and the short decompressor version can't be used (24 bytes more). This means that delta matching

must get more than 46 bytes of total gain to get any net savings.
- the compression time increases because matching is more complicated -- not 256-fold though, somewhere around 6-7 times is more typical
- more memory is also used -- 4*insize bytes extra

However, if no delta matches get used for a file, pucrunch falls back to non-delta decompressor, thus reducing the size of the decompressor automatically by 22 (normal mode) or 46 bytes (`-fshort`).

I also updated the C64 test file figures. `-fshort` of course shortened the files by 24 bytes and `-fdelta` helped on bs.bin. Beating RLE+ByteBoiler by amazing 350 bytes feels absolutely great!

4.12.1999
See 17.8.1998. This time "dec $d02c" was executed instead of "bit $d030" after decompression, if run without the `-f` option. Being sprite#5 color register, this bug has gone unnoticed since 25.8.1999.

7.12.1999
Added a short version of the VIC20 decompressor. Removed a byte from the decrunch code, only the "-ffast" version remained the same size.

14.12.1999
Added minimal C128 support. $f7-$f9 (Text mode lockout, Scrolling, and Bell settings) are overwritten and not reinitialized.

30.1.2000
For some reason -fshort doesn't seem to work with VIC20. I'll have to look into it. It looks like a jsr in the decompression code partly overwrites the RLE byte code table. The decompressor needs 5 bytes of stack. The stack pointer it now set to $ff at startup for non-C64 short decompressor versions, so you can't just change the final jmp into rts to get a decompressed version of a program.

Also, the "-f" operation for C16/+4 was changed to blank the screen instead of using the "freeze" bit.

2MHz or other speedups are now automatically selected. You can turn these off with the "+f" option.

8.3.2002
The wrong code was used for EOF and with -fdelta the EOF code and the shortest DELTA LZ length code overlapped, thus decompression ended prematurely.

24.3.2004
-fbasic added to allow compressing BASIC programs with ease. Calls BASIC run entrypoint ($a871 with Z=1), then BASIC Warm Start ($a7ae). Currently only VIC20 and C64 decompressors are available. What are the corresponding entrypoints for C16/+4 and C128?

Some minor tweaks. Reduced the RLE byte code table to 15 entries.

21.12.2005
Pucrunch is now under GNU LGPL. And the decompression code is distributed under the WXWindows Library Licence. (In short: binary version of the decompression code can accompany the compressed data or used in decompression programs.)

28.12.2007
Ported the development directory to FreeBSD. Noticed that the -fbasic version has not been released. -flist option added.

13.1.2008
Added -fbasic support for C128.

18.1.2008
Added -fbasic support for C16/+4.

22.11.2008
Fix for sys line detection routine. Thanks for Zed Yago.

To the homepage of *a1bert@iki.fi*