shoco: a fast compressor for short strings

shoco is a C library to compress and decompress short strings. It is very fast and easy to use. The default compression model is optimized for english words, but you can generate your own compression model based on *your specific* input data.
shoco is free software, distributed under the MIT license.

Quick Start

Copy shoco.c, shoco.h and shoco_model.h from github/shoco over to your project.
Include `shoco.h` and you are ready to use the API!

API

Here is all of it:

```
size_t shoco_compress(const char * in, size_t len, char * out, size_t bufsize);
size_t shoco_decompress(const char * in, size_t len, char * out, size_t bufsize);
```

If the `len` argument for `shoco_compress` is 0, the input char is assumed to be null-terminated. If it's a positive integer, parsing the input will stop after this length, or at a null-char, whatever comes first. `shoco_decompress` however will need a positive integer for `len` (most likely you should pass the return value of `shoco_compress`).
The return value is the number of bytes written. If it is less than `bufsize`, all is well. In case of decompression, a null-terminator is written. If the return value is exactly `bufsize`, the output is all there, but *not* null-terminated. It is up to you to decide if that's an error or not. If the buffer is not large enough for the output, the return value will be `bufsize` + 1. You might want to allocate a bigger output buffer. The compressed string will never be null-terminated.
If you are sure that the input data is plain ASCII, your `out` buffer for `shoco_compress` only needs to be as large as the input string. Otherwise, the output buffer may need to be up to 2x as

large as the input, if it's a 1-byte encoding, or even larger for multi-byte or variable-width encodings like UTF-8.
For the standard values of *shoco*, maximum compression is 50%, so the `out` buffer for `shoco_decompress` needs to be a maximum of twice the size of the compressed string.

How It Works

Have you ever tried compressing the string "hello world" with gzip? Let's do it now:

```
$ echo "hello world" | gzip -c | wc -c
32
```

So the output is actually *larger* than the input string. And gzip is quite good with short input: xz produces an output size of 68 bytes. Of course, compressing short strings is not what they are made for, because you rarely need to make small strings even smaller – except when you do. That's why shoco was written.
shoco works best if your input is ASCII. In fact, the most remarkable property of shoco is that the compressed size will *never* exceed the size of your input string, provided it is plain ASCII. What is more: An ASCII string is suitable input for the decompressor (which will return the exact same string, of course). That property comes at a cost, however: If your input string is not entirely (or mostly) ASCII, the output may grow. For some inputs, it can grow quite a lot. That is especially true for multibyte encodings such as UTF-8. Latin-1 and comparable encodings fare better, but will still increase your output size, if you don't happen to hit a common character.
Why is that so?
In every language, some characters are used more often than others. English is no exception to this rule. So if one simply makes a list of the, say, sixteen most common characters, four bits would be sufficient to refer to them (as opposed to eight bits – one byte – used by ASCII). But what if the input string includes an uncommon character, that is not in this list? Here's the trick: We use the first bit of a `char` to indicate if the following bits refer to a short common character index, or a normal ASCII byte. Since the first bit in plain ASCII is always 0, setting the first bit to 1 says "the next bits represent short indices for common chars". But what if our character is not ASCII (meaning the first bit of the input `char` is not 0)? Then we insert a marker that says "copy the next byte over as-is", and we're done. That explains the growth for non-ASCII characters: This marker takes up a byte, doubling the effective size of the character.
How shoco actually marks these packed representations is a bit more complicated than that (e.g., we also need to specify *how many* packed characters follow, so a single leading bit won't be sufficient), but the principle still holds.
But shoco is a bit smarter than just to abbreviate characters based on absolute frequency – languages have more regularities than that. Some characters are more likely to be encountered next to others; the canonical example would be q, that's *almost always* followed by a u. In english, *the*, *she*, *he*, *then* are all very common words – and all have a h followed by a e. So if we'd assemble a list of common characters *following common characters*, we can do with even less bits to represent these *successor* characters, and still have a good hit rate. That's the idea of shoco: Provide short representations of characters based on the previous character.

This does not allow for optimal compression – by far. But if one carefully aligns the representation packs to byte boundaries, and uses the ASCII-first-bit-trick above to encode the indices, it works well enough. Moreover, it is blazingly fast. You wouldn't want to use shoco for strings larger than, say, a hundred bytes, because then the overhead of a full-blown compressor like gzip begins to be dwarfed by the advantages of the much more efficient algorithms it uses.
If one would want to classify shoco, it would be an entropy encoder, because the length of the representation of a character is determined by the probability of encountering it in a given input string. That's opposed to dictionary coders that maintain a dictionary of common substrings. An optimal compression for short strings could probably be achieved using an arithmetic coder (also a type of entropy encoder), but most likely one could not achieve the same kind of performance that shoco delivers.
How does shoco get the information about character frequencies? They are not pulled out of thin air, but instead generated by analyzing text with a relatively simple script. It counts all *bigrams* – two successive characters – in the text and orders them by frequency. If wished for, it also tests for best encodings (like: Is it better to spend more bits on the leading character or on the successor character?), and then outputs its findings as a header file for `shoco.c` to include. That means the statistical model is compiled in; we simply can't add it to the compressed string without blowing it out of proportions (and defeating the whole purpose of this exercise). This script is shipped with shoco, and the next section is about how *you* can use it to generate a model that's optimized for *your* kind of data. Just remember that, with shoco, you need to control both ends of the chain (compression and decompression), because you can't decompress data correctly if you're not sure that the compressor has used the same model.

Generating Compression Models

Maybe your typical input isn't english words. Maybe it's german or french – or whole sentences. Or file system paths. Or URLs. While the standard compression model of shoco should work for all of these, it might be worthwile to train shoco for this specific type of input data.
Fortunately, that's really easy: shoco includes a python script called `generate_compression_model.py` that takes one or more text files and ouputs a header file ready for shoco to use. Here's an example that trains shoco with a dictionary (btw., not the best kind of training data, because it's dominated by uncommon words):

```
$ ./generate_successor_table.py /usr/share/dict/words -o
shoco_model.h
```

There are options on how to chunk and strip the input data – for example, if we want to train shoco with the words in a readme file, but without punctuation and whitespace, we could do

```
$ ./generate_successor_table.py --split=whitespace --
strip=punctuation README.md
```

Since we haven't specified an output file, the resulting table file is printed on stdout.

This is most likely all you'll need to generate a good model, but if you are adventurous, you might want to play around with all the options of the script:
Type `generate_compression_model.py --help` to get a friendly help message. We won't dive into the details here, though – just one word of warning: Generating tables can be slow if your input data is large, and *especially* so if you use the `--optimize-encoding` option.
Using pypy can significantly speed up the process.

Comparisons With Other Compressors

smaz

There's another good small string compressor out there: smaz. smaz seems to be dictionary based, while shoco is an entropy encoder. As a result, smaz will often do better than shoco when compressing common english terms. However, shoco typically beats smaz for more obscure input, as long as it's ASCII. smaz may enlarge your string for uncommon words (like numbers), shoco will never do that for ASCII strings.
Performance-wise, shoco is typically faster by at least a factor of 2. As an example, compressing and decompressing all words in `/usr/dict/share/words` with smaz takes around 0.325s on my computer and compresses on average by 28%, while shoco has a compression average of 33% (with the standard model; an optimized model will be even better) and takes around 0.145s. shoco is *especially* fast at decompression.
shoco can be trained with user data, while smaz's dictionary is built-in. That said, the maximum compression rate of smaz is hard to reach for shoco, so depending on your input type, you might fare better with smaz (there's no way around it: You have to measure it yourself).

gzip, xz

As mentioned, shoco's compression ratio can't (and doesn't want to) compete with gzip et al. for strings larger than a few bytes. But for very small strings, it will always be better than standard compressors.
The performance of shoco should always be several times faster than about any standard compression tool. For testing purposes, there's a binary inlcuded (unsurprisingly called `shoco`) that compresses and decompresses single files. The following timings were made with this command line tool. The data is `/usr/share/dict/words` (size: 4,953,680), compressing it as a whole (not a strong point of shoco):

compressor	**compression time**	**decompression time**	**compressed size**
shoco	0.070s	0.010s	3,393,975
gzip	0.470s	0.048s	1,476,083
xz	3.300s	0.148s	1,229,980

This demonstates quite clearly that shoco's compression rate sucks, but also that it's *very* fast.

Javascript Version

For showing off, shoco ships with a Javascript version (`shoco.js`) that's generated with emscripten. If you change the default compression model, you need to re-generate it by typing `make js`. You do need to have emscripten installed. The output is asm.js with a small shim to provide a convenient API:

```
compressed = shoco.compress(input_string);
output_string = shoco.decompress(compressed);
```

The compressed string is really a Uint8Array, since that resembles a C string more closely. The Javascript version is not as furiously fast as the C version because there's dynamic (heap) memory allocation involved, but I guess there's no way around it.
`shoco.js` should be usable as a node.js module.

Tools And Other Included Extras

Most of them have been mentioned already, but for the sake of completeness – let's have a quick overview over what you'll find in the repo:

`shoco.c`, `shoco.h`, `shoco_model.h`
The heart of the project. If you don't want to bother with nitty-gritty details, and the compression works for you, it's all you'll ever need.

`models/*`
As examples, there are more models included. Feel free to use one of them instead of the default model: Just copy it over `shoco_model.h` and you're all set. Re-build them with `make models`.

`training_data/*`
Some books from Project Gutenberg used for generating the default model.

`shoco.js`
Javascript library, generated by emscripten. Also usable as a node.js module (put it in `node_modules` and `require` it). Re-build with `make js`.

`shoco.html`
A example of how to use `shoco.js` in a website.

`shoco`
A testing tool for compressing and decompressing files. Build it with `make shoco` or just `make`. Use it like this:

```
$ shoco compress file-to-compress.txt compressed-file.shoco
$ shoco decompress compressed-file.shoco decompressed-file.txt
```

It's not meant for production use, because I can't image why one would want to use shoco on entire files.

`test_input`
Another testing tool for compressing and decompressing every line in the input file. Build it with `make test_input`. Usage example:

```
$ time ./test_input < /usr/share/dict/words
Number of compressed strings: 479828, average compression ratio: 33%

real    0m0.158s
user    0m0.145s
sys     0m0.013s
```

Adding the command line switch `-v` gives line-by-line information about the compression ratios.

`Makefile`
It's not the cleanest or l33test Makefile ever, but it should give you hints for integrating shoco into your project.

`tests`
Invoke them with `make check`. They should pass.

Things Still To Do

shoco is stable, and it works well – but I'd have only tested it with gcc/clang on x86_64 Linux. Feedback on how it runs on other OSes, compilers and architectures would be highly appreciated! If it fails, it's a bug (and given the size of the project, it should be easy to fix). Other than that, there's a few issues that could stand some improvements:

- There should be more tests, because there's *never* enough tests. Ever. Patches are very welcome!
- Tests should include model generation. As that involves re-compilation, these should probably written as a Makefile, or in bash or Python (maybe using `ctypes` to call the shoco-functions directly).
- The Python script for model generation should see some clean-up, as well as documentation. Also it should utilize all cpu cores (presumably via the `multiprocess`-module). This is a good task for new contributers!

- Again for model generation: Investigate why pypy isn't as fast as should be expected (jitviewer might be of help here).
- Make a real node.js module.
- The current SSE2 optimization is probably not optimal. Anyone who loves to tinker with these kinds of micro-optimizations is invited to try his or her hand here.
- Publishing/packaging it as a real library probably doesn't make much sense, as the model is compiled-in, but maybe we should be making it easier to use shoco as a git submodule (even if it's just about adding documentation), or finding other ways to avoid the copy&paste installation.

Feedback

If you use shoco, or like it for whatever reason, I'd really love to hear from you! If wished for, I can provide integration with shoco for your commercial services (at a price, of course), or for your totally awesome free and open source software (for free, if I find the time). Also, a nice way of saying thanks is to support me financially via git tip or flattr.
If you find a bug, or have a feature request, file it! If you have a question about usage or internals of shoco, ask it on stackoverflow for good exposure – and write me a mail, so that I don't miss it.

Authors

shoco is written by Christian Schramm.