Unicode data file compression: achieving 40-70% reduction over gzip alone

Jul 3, 2021 · Stephen Gutekanst · unicode,, compression

A little story about how writing a domain-specific compression algorithm in a few days can sometimes yield big benefits, why it's sometimes worth giving it a shot, and how to tell when you should try. Note: this is about Unicode spec data files, not general purpose text compression.

Background

Two weeks ago, I began using Ziglyph ("Unicode processing with Zig, and a UTF-8 string type: Zigstr.") - an awesome library by @jecolon, for grapheme cluster sorting in Zorex, an omnipotent regexp engine.

I don't personally have any prior experience working with the lower level details of Unicode, or compression algorithms for that matter.

Problem

As I stumbled into the wondrous world that is Unicode text sorting (see also my article: Unicode sorting is hard & why browsers added special emoji matching to regexp) and began using Ziglyph, I came across an issue: the standard Unicode collation algorithm, which Ziglyph implements, depends on some large Unicode data tables for normalization and sort keys - even gzipped these were fairly large:

```
hexops-mac:zorex slimsag$ du -sh asset/*
308K    asset/uca-allkeys.txt.gz
```

```
260K    asset/ucd-UnicodeData.txt.gz
```

These file sizes may seem small, but one of my goals is to make Zorex a real competitor to e.g. a browser's native regexp engine. That's challenging because WebAssembly bundle sizes matter *a lot* in that context, and using the browser's regexp implementation is virtually free.

Investigation

I set out to try and reduce the size of these data files. First I opened an issue and asked if anyone else had thoughts around reducing the size of this data. The author of Ziglyph @jecolon is awesome and readily had some ideas and was able to reduce the two files substantially by removing unnecessary data (such as comments, etc.)

Curious how much further we could go, I kept squinting at the data files (warning: large):

- http://www.unicode.org/Public/UCA/latest/allkeys.txt
- http://www.unicode.org/Public/UNIDATA/UnicodeData.txt

Binary encoding?

My first thoughts were that a binary encoding would likely reduce the size a lot. I pulled in some help from Hobbyist reverse engineer @Andoryuuta and he got started on a binary encoding for UnicodeData.txt based on the spec. With that, he was able to reduce the original 1.9M allkeys.txt file down to 250K (125K gzipped) - quite a win.

Differential encoding/compression?

My secondary thought was that, scrolling through these data files it was obvious most entries were derived from prior entries. Many entries were long runs of data where the next entry had the same value, plus a small increment. For example, at the start of the `allkeys.txt` file:

```
0000  ; [.0000.0000.0000] # NULL (in ISO 6429)
0001  ; [.0000.0000.0000] # START OF HEADING (in ISO 6429)
0002  ; [.0000.0000.0000] # START OF TEXT (in ISO 6429)
0003  ; [.0000.0000.0000] # END OF TEXT (in ISO 6429)
0004  ; [.0000.0000.0000] # END OF TRANSMISSION (in ISO 6429)
0005  ; [.0000.0000.0000] # ENQUIRY (in ISO 6429)
0006  ; [.0000.0000.0000] # ACKNOWLEDGE (in ISO 6429)
0007  ; [.0000.0000.0000] # BELL (in ISO 6429)
0008  ; [.0000.0000.0000] # BACKSPACE (in ISO 6429)
000E  ; [.0000.0000.0000] # SHIFT OUT (in ISO 6429)
000F  ; [.0000.0000.0000] # SHIFT IN (in ISO 6429)
```

Of course, not all sections are so sequential. Many sections are a bit more arbitrary:

```
FF9A  ; [.4304.0020.0012] # HALFWIDTH KATAKANA LETTER RE
32F9  ; [.4304.0020.0013] # CIRCLED KATAKANA RE
3355  ; [.4304.0020.001C][.42FB.0020.001C] # SQUARE REMU
3356  ; [.4304.0020.001C][.430A.0020.001C][.42EE.0020.001C][.42E3.0020.001C][.0000.0037.001C][.4
308D  ; [.4305.0020.000E] # HIRAGANA LETTER RO
31FF  ; [.4305.0020.000F] # KATAKANA LETTER SMALL RO
30ED  ; [.4305.0020.0011] # KATAKANA LETTER RO
```

Still, there are obvious patterns one can see in the way these values change.

Go implementation

I did a quick hacky Go implementation of differential encoding on these files to see how well that would work. The results were pretty good, and already beat just `gzip -9` compression of the files:

File	Original	Original + `gzip -9`	My compression	My compression + `gzip -9`
Decompositions.txt	72K	28K	48K	12K
allkeys-minimal.txt	500K	148K	204K	36K

However, because I chose to do these experiments in Go I found a number of inefficiencies:

- There were a lot of locations where I encoded things as 8-bit unsigned integers (Go's smallest value type) instead of a more optimal 4-bit unsigned integer. I could've done bit shifting, but it would've been annoying.
- There were also many places where I encoded Unicode codepoints as 32-bit unsigned integers, rather than a more optimal 21-bit unsigned integer (because valid Unicode codepoints do not exceed that range.)

For a real implementation, I switched over to Zig.

Zig implementation

Actually, two things made working on this in Zig much easier than in Go:

1. Zig has variable bit-width integers: I could just write `u4` and `u21` values instead of needing to handle bit packing within larger size integers myself. That was *nice*.
2. In the Zig standard library it provides:

- `std.io.BitWriter`
- `std.io.BitReader`

With these two features, it became incredibly easy to write the most optimal bit-packed encoding of the data.

In fact, the basic uncompressed binary format was only a few lines to encode:

```
pub fn compressTo(self: *DecompFile, writer: anytype) !void {
    var buf_writer = std.io.bufferedWriter(writer);
    var out = std.io.bitWriter(.Little, buf_writer.writer());
    try out.writeBits(@intCast(u16, self.entries.items.len), 16);
    while (self.next()) |entry| {
        try out.writeBits(entry.key_len, 3);
        _ = try out.write(entry.key[0..entry.key_len]);
        try out.writeBits(@enumToInt(entry.value.form), @bitSizeOf(Form));
        try out.writeBits(entry.value.len, 5);
        for (entry.value.seq[0..entry.value.len]) |s| try out.writeBits(s, 21);
    }
    try out.flushBits();
    try buf_writer.flush();
}
```

Differential encoding state machine

To handle the compression, I started out *really* simple. First I encoded just a binary version of the data with no compression. The most important thing was to get to a point where I could start testing some theories about what would compress the data really well, and validate that it was in fact being losslessly compressed/decompressed without issues via tests:

```
test "compression_is_lossless" {
    const allocator = testing.allocator;

    // Compress UnicodeData.txt -> Decompositions.bin
    var file = try parseFile(allocator, "src/data/ucd/UnicodeData.txt");
    defer file.deinit();
    try file.compressToFile("src/data/ucd/Decompositions.bin");

    // Reset the raw file iterator.
    file.iter = 0;

    // Decompress the file.
    var decompressed = try decompressFile(allocator, "src/data/ucd/Decompositions.bin");
    defer decompressed.deinit();
    while (file.next()) |expected| {
        var actual = decompressed.next().?;
        try testing.expectEqual(expected, actual);
    }
}
```

A stream of op codes

I settled on a really simple idea: these data files all have basically just a variable number of integers per line. And if I kept "registers" representing the current value for each integer, I could determine the difference between the past line and the subsequent one to produce a difference. If I encoded that difference as a stream of opcodes with associative data, then to decompress the file I could simply "replay" those operations based on the opcodes and then iteratively come up with more finely-specified, specific opcodes to handle specific types of data.

I started out simple, really just with two opcodes:

```
// A UDDC opcode for a decomposition file.
const Opcode = enum(u4) {
    // Sets all the register values with no compression.
    set,

    // Denotes the end of the opcode stream. This is so that we don't need to encode the total
    // number of opcodes in the stream up front (note also the file is bit packed: there may be
    // a few remaining zero bits at the end as padding so we need an EOF opcode rather than say
    // catching the actual file read EOF.)
    eof,
};
```

Using these two opcodes, I was able to effectively encode the entire file. The `set` opcode had some associative data which effectively expressed an entire raw, uncompressed entry in the file (one line.) This increased the file size since it was effectively just adding 4 bits (the opcode) as additional overhead.

Iteratively finding the most lucrative opcodes

To find the most lucrative (i.e. compressed) opcodes, I printed the data I would associate with an opcode (like `set`) and then looked for repetitions. Sometimes manually, and sometimes by e.g. piping data to a combination of `sort|uniq -c|sort -r` to find common patterns.

Since I was printing *differences* between e.g. the current value and previous value, it was really easy to find common patterns that appeared in the file very frequently, such as specific fields incrementing by specific amounts with one field being arbitrary:

```
    // increments key[3] += 1; sets value.seq[0]; emits an entry.
    // 1685 instances
    increment_key_3_and_set_value_seq_0_and_emit,
```

Once I had narrowed down to a larger group of opcodes that more specifically represented the data, I was able to print the number of bits required to store the change in specific fields (like `value.seq[0]`) and add even more specific opcodes to use variable bit widths:

```
    // increments key[3] += 1; sets value.seq[0]; emits an entry.
    // 1685 instances
    increment_key_3_and_set_value_seq_0_2bit_and_emit, // 978 instances, 2323 byte reduction
    increment_key_3_and_set_value_seq_0_8bit_and_emit, // 269 instances, 437 byte reduction
    increment_key_3_and_set_value_seq_0_21bit_and_emit, // 438 instances
```

It being a stream of opcodes was quite nice, because it allowed me to determine how much space was being consumed by a given opcode in sum and target further reducing the size of opcodes that took up the most space. It also made it really easy to find opcodes that I though *might* help, but in practice turned out to not be that frequent. Just print them, pipe to `sort|uniq -c|sort -r` to count them - and remove the lowest hanging fruit.

A stream of opcodes for a state machine: a natural progression from a binary format?

I chose an opcode stream for a reason: so that I could encode some complex logic in the form of a state machine. This came in handy for the `allkeys.txt` file in specific, as it allowed me to introduce *incrementors* into the mix which would *increment register values by a chosen amount each iteration (value "emission").*

The final opcodes for the allkeys.txt file ended up being:

```
// A UDDC opcode for an allkeys file.
const Opcode = enum(u3) {
    // Sets an incrementor for the key register, incrementing the key by this much on each emiss
    // 10690 instances, 13,480.5 bytes
    inc_key,

    // Sets an incrementor for the value register, incrementing the value by this much on each e
    // 7668 instances, 62,970 bytes
    inc_value,

    // Emits a single value.
    // 31001 instances, 15,500.5 bytes
    emit_1,
    emit_2,
```

```
    emit_4,
    emit_8,
    emit_32,

    // Denotes the end of the opcode stream. This is so that we don't need to encode the total
    // number of opcodes in the stream up front (note also the file is bit packed: there may be
    // a few remaining zero bits at the end as padding so we need an EOF opcode rather than say
    // catching the actual file read EOF.)
    eof,
};
```

This meant I could determine the difference in the `key` and `value` fields (what those actually are isn't important, just that they are all minor incremental differences on the prior entry in the file) - set an *incrementor* to do some work on each emission, such as say increment the `key` array by `[0, 1, 5]` each emission, and then say "now emit_32 values!".

Suddenly, instead of encoding 32 key entries (32 * 3 key values * 21 bits) I am just setting an incrementor (3 key values * 21 bits) and a single opcode to emit 32 values (3 bits).

Overall, this gave me a very nice, natural-feeling progression from a "raw binary format" to something a bit more specific - a bit more *compressed*.

Results? Better than gzip/brotli; and even better *with* them!

For lack of better words, I'll call my compression algorithm here Unicode Data Differential Compression, since it's differential and specifically for the Unicode data table files - or UDDC for short.

The two files went from the original 568K (with gzip) down to just 61K (with UDDC+gzip). With this, we are able to equal or match both `gzip -9` and `brotli -9` on their own, AND when combined with gzip or brotli we are able to reduce by 40-70%:

File	Before (bytes)	After (bytes)	Change
`Decompositions.bin`	48,242	19,072	-60.5% (-29,170 bytes)
`Decompositions.bin.br`	24,411	14,783	-39.4% (-9,628 bytes)
`Decompositions.bin.gz`	30,931	15,670	-49.34% (15,261 bytes)
`allkeys.bin`	373,719	100,907	-73.0% (-272,812 bytes)
`allkeys.bin.br`	108,982	44,860	-58.8% (-64,122 bytes)
`allkeys.bin.gz`	163,237	46,996	-71.2% (-116,241 bytes)

- Before represents binary format without UDDC compression.
- After represents binary format with UDDC compression.
- `.br` represents `brotli -9 <file>` compression
- `.gz` represents `gzip -9 <file>` compression

Why test with gzip/brotli but not others?

I chose to compare against gzip/brotli specifically because you get those effectively for free in WebAssembly: browsers already know how to decompress those and ship with gzip/brotli decompressors - so you can use them for free without shipping any additional code.

How complex is the implementation?

The final implementation for both files is only a few hundred lines (excluding blank lines, comments, and tests):

- `AllKeysFile.zig`: 298 lines
- `DecompFile.zig` 336 lines

I have not measured produced machine code size yet, but suspect it is relatively negligible compared to the gains.

Notable mention

I should mention that the Unicode spec, as @jecolon pointed out to me, does suggest ways to reduce sort key lengths and implement Run-length Compression:

- https://unicode.org/reports/tr10/#Reducing_Sort_Key_Lengths
- https://unicode.org/reports/tr10/#Run-length_Compression

I wasn't able to locate an implementation of this (I'd be curious to compare results!) but suspect that, as the run-length compression does not fit the data as tightly, it would not compress quite as well (although would handle any major changes to the type of data in the files without requiring compression algorithm changes better.)

Also of note is that their algorithm only seems to be mentioned in the context of allkeys.txt / the Unicode Collation Algorithm, not in the context of normalization/decompositions from `UnicodeData.txt`.

Conclusion

Ask questions, stay curious, don't be afraid to experiment even if it's outside of your domain of expertise. You might surprise yourself and find something interesting, challenging, and worthwhile.