1 Overview

PNG, the Portable Network Graphics format, is a robust, extensible, general-purpose and patent-free image format. In this chapter we briefly describe the events that led to its creation in early 1995 and how the resulting design decisions affect its compression. Then we examine PNG's compression engine, the deflate algorithm, quickly touch on the zlib stream format, and discuss some of the tunable settings of its most popular implementation. Since PNG's compression filters are not only critical to its efficiency but also one of its more unusual features, we spend some time describing how they work, including several examples. Then, because PNG is a practical kind of a format, we give some practical rules of thumb on how to optimize its compression, followed by some "real-world" comparisons with GIF, TIFF and `gzip`. Finally we cover some of the compression-related aspects of MNG, PNG's animated cousin, and wrap up with pointers to other sources for further reading.

2 Historical Background

Image formats are all about design decisions, and such decisions often can be understood only in an historical context. This is particularly true of PNG, which was created during—and as part of—the "Wild West" days of the Web.

Though PNG's genesis was at the beginning of 1995, its roots go back much further, to the seminal compression papers by Abraham Lempel and Jacob Ziv in 1977 and 1978.[**?**, **?**] The algorithms described in those papers, commonly referred to as "LZ77" and "LZ78" for obvious reasons,[**?**] spawned a whole host of refinements. Our immediate concern, however, is with just one: Terry Welch's, published in 1984,[**?**] now known as Lempel-Ziv-Welch or simply LZW.

LZW initially was used mainly in modems and networking equipment, where its speed and efficiency were matched by the ease of its implementation in silicon. It was also implemented in software in the form of the `compress` command on Unix systems, but the most interesting development from our perspective was its incorporation into CompuServe's GIF image format in 1987. Originally used only within CompuServe, GIF's design soon led it to the wide open spaces of Usenet, where its open and portable design gave it an advantage over the proprietary and platform-specific formats in use at the time. By 1991, GIF—true to its name—had indeed become a format for graphics interchange and was the dominant one on the early Internet. Little wonder that it also became one of the first to be tapped for the budding World Wide Web.

Alas, one important detail got overlooked in all the excitement: the LZW algorithm on which GIF was based had actually been patented by Sperry (which later merged with Burroughs to become Unisys). In fact, the LZW patent was well known, but prevailing wisdom at the time was that algorithms, *per se*, could not be patented—only their instantiations in hardware could. Thus, while modem manufacturers were required to pay licensing fees, most believed that software implementations such as `compress` and GIF applications were exempt.

That belief came to an abrupt and painful end in December 1994. After more than a year of private discussions with Unisys, CompuServe announced on 28 December that developers of GIF-supporting software would henceforth be required to pay licensing fees. As one might imagine, the Internet community grew quite excited over the issue, and one result was the creation of PNG, the Portable Network Graphics format, during the first two months of 1995.

3 Design Decisions

We'll touch on some of the subsequent events surrounding PNG later in this chapter, but for now we have enough background to understand the things that shaped PNG's design—and how, in turn, that design affects its compression.

First and foremost, PNG was designed to be a complete and superior replacement for GIF. At the most basic level, that meant PNG should satisfy the following requirements:

- fully lossless compression method
- fully patent-free compression method (!)
- support for lossless conversion from GIF to PNG and back again
- support for colormapped images (also known as "indexed-color" or "palette-based")
- support for transparency

But PNG's designers were not satisfied with a minimal replacement. Since essentially *any* change would break compatibility with existing software, the group felt that there was little harm in (and considerable benefit to) doing a more extensive redesign. In particular, there was a strong desire for a format that could be used to archive truecolor images in addition to colormapped ones, yet remain considerably simpler and more portable than the heavyweight TIFF format. "Real" transparency support was also a priority. To this end, the following additional features were included in the list:

- better compression than GIF
- better interlacing than GIF
- support for grayscale and truecolor (RGB) modes
- support for partial transparency (i.e., an alpha channel)
- support for extensions without versioning
- support for samples up to 16 bits (e.g., 48-bit RGB)

Finally, a few group members' experience with compression and archiving utilities—particularly with the sorts of problems commonly encountered in transferring such utilities' archives between systems—led to two further requirements:

- support for testing file integrity without viewing
- simple detection of corruption caused by text-mode transfers

We have glossed over one major feature of any proposed GIF replacement: multi-image capability, especially animation. There are two reasons for this. The first is that, historically, multi-image capability *wasn't* a major feature of GIF; in fact, it was hardly ever used. Indeed, not until September 1995, six months after PNG was finalized, did Netscape introduce animated GIFs to the world. The second reason is that the PNG developers decided quite early on to restrict PNG to single-image capability—both for simplicity and for compatibility with Internet standards, which distinguish between still images and animations or video—and later to create a second, related format for animations and other multi-image purposes. Thus was born MNG, the Multiple-image Network Graphics format, which we discuss at the end of this chapter.

The result of the PNG requirements, hammered out over a period of two months, was a clean, simple, component-based format. The fundamental building block of PNG images is the *chunk*, which consists of a type-identifier (loosely speaking, the chunk "name"), a length, some data (usually), and a *cyclic redundancy code* value (CRC). Chunks are divided into two main categories, critical and ancillary, and there are roughly two dozen types defined. But the simplest possible PNG consists of only three critical chunks: an image-header chunk, IHDR, which contains basic info such as the image type and dimensions; an image-data chunk, IDAT, which contains the compressed pixel data; and an end-of-image chunk, IEND, which is the only officially defined chunk that carries no payload. Colormapped images require one more chunk: PLTE, which contains a palette of up to 256 RGB triplets. (Unlike GIF, the number of palette entries is not limited to a power of two.)

In addition to the minimal subset of chunks, all PNG images have one other component at the very beginning: an 8-byte signature that both identifies the file (or stream) as a PNG image and also cleverly encodes the two most common text-file end-of-line characters. Because text-mode transfers of binary files are characterized by the irreversible conversion of such characters, which is guaranteed to destroy virtually any compressed data, the first eight or nine bytes of a PNG image can be used to check for the results of such a transfer and even to identify the most likely kind—e.g., from a Unix system to a DOS/Windows system.

Of course, our concern here is compression, not the gory details of inter-platform file transfers. But we already know enough about PNG's design to note one effect on compression efficiency. Insofar as a chunk's type, size and CRC each require four bytes of storage, every PNG image contains a minimum of 57 bytes of overhead: the 8-byte signature, 12 bytes per chunk in at least three chunks, and 13 bytes of header information within the IHDR chunk. And as we'll see in the sections below on PNG's compression engine, there are at least 9 bytes of additional overhead within the IDAT chunk. Yet even given a minimum of 66 bytes of overhead, PNG's superior compression method more than makes up the difference in most cases, with the obvious exception of tiny images—most commonly the 1×1 spacers and "web bugs" used on many web pages.

Another key principle of PNG's design was the *orthogonality* of its feature set. For example, the core compression algorithm is treated as logically distinct from the preconditioning filters (which we discuss in detail below), and both are independent of the image type and pixel depth.[1] This has clear advantages in terms of creating a clean architecture and simpler, less error-prone implementations, but it does come at a cost. For example, one could imagine that a particular compression scheme might be particularly well suited to palette images but not to grayscale or RGB. For black-and-white images (i.e., 1-bit grayscale), there is no question that both Group 4 facsimile compression and the JBIG algorithm are much better than PNG, but they can be used on deeper images only through the awkward expedient of applying them to individual bit planes—i.e., 24 times per image in the case of truecolor.[2] On the other hand, the simple fact of supporting multiple pixel depths and image types provides some benefits, as well. For example, whereas an 8-bit grayscale GIF must store 768 bytes of extraneous palette data, PNG can avoid this and store the image as native grayscale. For web images on a busy site, a savings of three quarters of a kilobyte, taken many times per second, may have a discernible effect on bandwidth usage.

But these are all relatively minor issues, at least as far as compression is concerned. Aside from losslessness, the design choice with the biggest impact on compression was, unquestionably, the one that triggered the entire effort: the requirement to be free of patents. Together with the implicit requirement that the format be usable—i.e., that it could be encoded and decoded "fast enough," and that doing so would require "little enough" memory—this dramatically limited the possible compression algorithms. Obviously LZW was ruled out immediately, as were many of the other Lempel-Ziv derivatives. Arithmetic compression was known to be an efficient approach, but the fastest variants were all patented by IBM and others, and even those were quite slow compared to the LZ methods. Both Huffman and run-length encoding were fast,

[1] There are a few exceptions, however. For example, even though an alpha (transparency) channel is logically separate from the image type, the PNG spec disallows alpha channels in colormapped images; to compensate, the palette itself may be extended to a table of RGBA values via the optional tRNS chunk. Also, even though an RGB palette is supported in truecolor (RGB and RGBA) images as a means of providing a suggested palette for color-reduction on limited displays, the extended RGBA palette is *not* allowed as a corresponding feature in RGBA images. And, of course, not every bit depth is supported with every image type; there is no such thing as a 6-bit (2-bit-per-sample) RGB PNG, for example.

[2] Presumably the compression ratio is none too good in this case, either; the correlations (i.e., redundancies) between different bit planes would be ignored.

memory-efficient, and patent-free, but their compression efficiency was unacceptably low. And while there were any number of other compression algorithms being researched in universities around the world, most of them were lossy, and none of them were free of the possibility of “submarine patents,” i.e., the kind that are applied for in secret and only become public knowledge years later, when they’re finally granted.

In the end, there were only two serious candidates for PNG’s compression engine: a particular LZ77 variant that was already in wide use and had survived at least one patent review, and a relative newcomer known as Burrows-Wheeler block transform coding (BWT for short, covered by Peter Fenwick in Chapter 6). While there were a few concerns about BWT’s possible coverage by future patents, it did appear to be clean (which still seems to be the case in 2001). But the main problem was that the early implementations available at the time were quite memory-intensive and horrendously slow—roughly four to eight times the memory and two orders of magnitude slower than the LZ77 alternative. Insofar as BWT’s compression was no more than 30% better than the alternative (and often merely equivalent), and given its relative immaturity at the time, it was ultimately rejected.

4 Compression Engine

Thus the core of PNG’s compression scheme is a descendant of LZ77 known as *deflate*. The specifics of the format were defined by PKWARE for their PKZIP 1.93a archiver in October 1991, and Jean-loup Gailly and Mark Adler wrote an open-source implementation that first appeared in Info-ZIP’s Zip and UnZip, and subsequently in GNU `gzip` and the zlib library (`http://www.zlib.org/`). Deflate is comparable to or faster than LZW in both encoding and decoding speed, generally compresses between 5% and 25% better for “typical computer files” (though 100–400% better is not uncommon in truecolor images), and never expands incompressible data by more than a fraction of a percent,[3] but it has a larger memory footprint. Deflate’s compression efficiency tends to be worse than both Burrows-Wheeler and arithmetic schemes by around 30%, but the most common implementations of both approaches also require an order of magnitude more time and memory.

In the simplest terms, deflate uses a sliding window of up to 32 kilobytes (32,768 bytes), with a Huffman encoder on the back end.[4] Recall from Chapter 5[**?**] that a “sliding window” is a very literal description of the key idea of LZ77. Encoding involves finding the longest matching string (or at least *a* long string) in the 32 KB window immediately prior to the “current position,” storing that as a pointer (distance backward) and a length (in bytes), and advancing the current position—and therefore the window—accordingly.

Of course, one should never trust a one-line description of anything, particularly a compression algorithm. The devil, as they say, is in the details. We won’t attempt to cover all of the gruesomeness of the deflate spec—the specification itself is the best reference for that—but there are a number of features that are worth pointing out. We’ll make a note of those that separate deflate from more standard LZ77 and LZSS implementations.

One of the more amusing features is the fact that, although a matching string must *begin* inside the sliding window, *it need not be fully contained within the window.* (At least, not at the point at which the decoder reads the codes for that string.) That is, from the decoder’s perspective, a length/distance pair at a given position may describe bytes that have not yet been output. Consider this example:

```
something something something something something something else
...
```

[3] Actually, a one-byte file could expand by eleven bytes. But the eleven bytes are a fixed overhead, so the claim holds for anything larger than a kilobyte or so. LZW, on the other hand, can expand a 200 KB file to 600 KB in some cases.

[4] PKWARE recently introduced a 64 KB variant, but its compression is typically only a fraction of a percent better than standard (32 KB) deflate.

The first copy of the word (“`something `”) is followed by a distance code of 10 bytes (pointing back at the ‘s’) and a length code of 50 (five additional copies of 10 bytes each). Clearly the 50-byte length extends 40 bytes beyond the end of the (current) sliding window, which at first glance may seem paradoxical. But the decoder need not attempt to emit all 50 bytes at once; the first 10 bytes lie within its sliding window, and by the time it has copied those 10 to its output buffer—and moved the sliding window accordingly—it has 10 more bytes available to copy, and so on until it has reconstructed the full 50 bytes. Indeed, the string could begin at the last byte of the window, and the decoder would still be able to recreate it, one byte at a time.

Deflate limits match-lengths to between 3 and 258 bytes, which has two important consequences. The first is that the maximum conceivable compression ratio, using one bit to encode the distance pointer and one bit for the (258-byte) length, is 1032:1. There are some assumptions built into this limit, of course; among them is that the ratio of actual compressed data to overhead, such as header and trailer bits, is infinite. In practice, however, better than 1030:1 is achievable, even with the overhead.

The other consequence of the length limits is that there must be some alternate mechanism to encode sequences of less than three bytes—particularly single bytes. In order to prime the sliding window and to accommodate bytes in the input stream that don’t appear anywhere in the sliding window, the algorithm must must be able to encode plain characters, or “literals.” There is nothing particularly difficult or interesting about that, but it does mean that there are three kinds of symbols rather than two: lengths, distances, and literals. These three alphabets are the grist for the Huffman stage of the deflate engine.

Deflate actually merges the length and literal codes into a single alphabet of 286 symbols. Values 0–255 are the literals, 256 is a special end-of-block code, and 257–285 encode the lengths. Astute readers will note that there’s something fishy about an algorithm in which 29 values are used to encode 256 possible lengths. Deflate works around this little problem by using some of the 29 values to encode *ranges* of lengths and then appending between one and five extra bits to specify the precise value. A similar approach is used for the distance alphabet; 30 values encode 32,768 possible distances, with all but four of the codes requiring between one and thirteen extra bits.

The two alphabets, lengths/literals and distances, are fed to the Huffman encoder and compressed with either fixed (predefined) or dynamic Huffman codes. A subtlety of the latter case is that the codes are allowed to be no more than 15 bits long; in the understated wording of the deflate specification, “this constraint complicates the algorithm for computing code lengths from symbol frequencies.” (For that matter, finding the longest match in the sliding window is more than a little complex. Fortunately, a decoder need not worry about either issue.)

Since dynamic coding can tailor itself to the input data, it tends to be more efficient for larger data sets. However, unlike the case with fixed codes, dynamic Huffman trees must be explicitly included in the output stream (literal/length tree first, then distance tree). Thus, for smaller input streams, where the overhead of the trees outweighs their improved compression efficiency, fixed codes are better. Because deflate organizes its output into blocks—something at which we hinted earlier, with the end-of-block code—it can alternate between fixed and dynamic Huffman codes as necessary in order to optimize compression.

What about incompressible data, such as data that have already been compressed with deflate or another algorithm? Deflate includes a third option for such cases: store the data as is. Stored blocks can be up to 64 KB in length, although implementations may further limit their size.[5]

[5] Note that the abstract to the deflate specification, in discussing worst-case expansion, implies a limit of 32 KB. This is incorrect for the format, but it is true of the reference implementation.

5 zlib Format

PNG currently specifies a single compression method,[6] deflate. It further specifies that the deflated data must be formatted as a *zlib* stream, which has two implications. One is that the deflate stream immediately acquires six additional bytes: two header bytes and four trailer bytes. The header bytes encode basic information about the compression method; in the current zlib specification, only deflate is defined, and only with window sizes equal to a power of two between 256 and 32,768 bytes, inclusive. The four bytes at the end of the stream are the *Adler-32 checksum* of the (uncompressed) input data. The Adler-32 checksum, defined in Section 2.2 of the zlib specification,[**?**] is a simple extension of the 16-bit Fletcher checksum used in telecommunications, and it is faster to compute than a standard 32-bit CRC is.

The other implication of zlib formatting is that it supports the concept of *preset dictionaries*. Instead of starting with an empty sliding window, as is the usual case, the encoder can prime the sliding window with a predetermined dictionary of strings likely to be found in the data. This is signalled by a flag bit in the zlib header and an additional four-byte identifier before the compressed stream. (The identifier is actually the Adler-32 checksum of the preset dictionary.) Since only the identifier is transmitted, not the entire dictionary, the benefits for smaller files are potentially great—for example, a 512-byte data stream could be encoded in just a few bits if the dictionary contained a good match for the stream. On the other hand, both encoder and decoder must know ahead of time what the dictionary is; this tends to be a problem for an image format like PNG, in which there are many decoders in existence, none of which have *a priori* knowledge of any such dictionary. However, as we'll see in the MNG section later, preset dictionaries *might* play a useful role in a multi-image format, particularly when it is used to store a collection of small icons or an animated cartoon with a limited number of scene changes.

6 zlib Library

Arguably the most common implementation of the zlib format—definitely the most popular for PNG applications—is Jean-loup Gailly and Mark Adler's zlib library, which originally defined the format. zlib was a reimplementation of Info-ZIP's deflate codec and was created in early 1995 specifically for use in PNG implementations. (Since then it has found a home in numerous other arenas, of course, perhaps most notably as part of Java's `java.util.zip` classes and the related JAR archive format.) Let's take a quick look at a few of the implementation-specific details in zlib.

As was discussed at length earlier, PNG was designed in response to the patent issues surrounding LZW in GIF. What about patent issues with deflate? It should come as no surprise to anyone that it is *possible* to write a spec-compliant deflate encoder that infringes on various patents. In fact, PKWARE, which created the deflate format, has a relevant patent (involving sorted hash tables), and there are several others that also could apply to a deflate encoder. But the key reason deflate was chosen for PNG was that it *can* be implemented *without* infringing on any patents—and without any significant impact on speed and compression efficiency. Indeed, Section 4 of the zlib specification outlines such an approach and says, "Since many variations of LZ77 are patented, it is strongly recommended that the implementor of a compressor follow the general algorithm presented here, which is known not to be patented *per se*."

On a much more practical level, the zlib library supports ten compression settings. The lowest, level 0, is straightforward: it indicates no compression, i.e., that the data are written using deflate's stored blocks. (Note that the library limits the size of these blocks to 32 KB rather than the 64 KB that the deflate spec allows.) But the other nine levels encode a number of parameters that collectively trade off between encoding speed and compression efficiency. These are summarized in Table 1.

[6] This is something that, for compatibility and standardization reasons, is unlikely to change for a number of years or until a significantly better—but still unencumbered—compression algorithm is discovered, whichever is longer.

Level	lazy matches	good match length	nice match length	max insert length	max chain length
1	no	4	8	4	4
2	no	4	16	5	8
3	no	4	32	6	32
				max lazy match	
4	yes	4	16	4	16
5	yes	8	32	16	32
6	yes	8	128	16	128
7	yes	8	128	32	256
8	yes	32	258	128	1024
9	yes	32	258	258	4096

Table 1: zlib Library Compression Levels and Parameters

Cenk Sahinalp described in Chapter 5 the concept of lazy evaluation of LZ77 string-matches; such an approach can compress better than a simple "greedy" algorithm, but at a cost in complexity and encoding time. Only the six highest zlib levels use lazy matching.

The "good match length" and "nice match length" parameters are thresholds for the lengths of already-matched strings beyond which zlib respectively either reduces its search effort for longer ones or quits altogether. (Note that the longest possible match length, 258 bytes, is supported only for zlib levels 8 and 9.) For levels 4–9, the "max lazy match" parameter is similar to the "nice" match length, except that it applies to the search for a lazy (secondary) match rather than the primary one.

Since levels 1–3 do not use lazy matching, zlib uses another parameter, "max insert length," in place of the maximum lazy match value; this determines whether a matched string will be added to zlib's internal dictionary (hash table). Only very short strings will be added, which improves speed but degrades compression.

Finally, the "max chain length" parameter sets the limit for how much of its internal dictionary zlib will search for a (better) match. For compression level 9, zlib will check hash chains with as many as 4096 entries before moving to the next window position; as one might imagine, this has a significant impact on encoding speed.

In addition to the various implicit parameters hidden within zlib's nine compression levels, the library also supports a trio of explicit parameters that affect compression. The simplest determines the size of the sliding window, which is usually 32 KB but (for encoders only) can be any smaller power-of-two down to 512 bytes.[7] Smaller windows adversely affect compression efficiency, of course, except in the case that the entire uncompressed stream is smaller than the window size, give or take a couple hundred bytes. And compliant decoders are required to support all window sizes.

The second explicit parameter, `memLevel`, determines the size of zlib's hash table and its buffer for literal strings. It defaults to 8, but the maximum value is 9. Bigger values are faster (assuming sufficient memory is available) and improve compression efficiency slightly.

The third and final parameter is slightly more interesting, if only because it involves the one part of zlib that was specifically designed for PNG. The *strategy* parameter tunes zlib according to the expected class of input data. The default value (Z_DEFAULT_STRATEGY) emphasizes string-matching, which tends to produce the best results for text files and program objects. The Z_HUFFMAN_ONLY value disables string-matching entirely and depends only on Huffman coding, which also makes it relatively fast. A third value,

[7] In principle the library also supports 256-byte windows, but a long-standing bug renders this setting unusable in zlib releases through version 1.1.3.

Z_FILTERED, is designed for "data produced by a filter (or predictor)"—i.e., PNG-like data—and adopts an intermediate approach of less string-matching and more Huffman coding than the default.

7 Filters

Filters, in the context of PNG, are a means by which pixel data are *preconditioned* before being fed to the core compression engine, resulting in improved compression performance. This fact, which may seem counterintuitive at first, is best illustrated with an example.

Consider the following string of characters, which in this case represents a line of 26 pixel values:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

(For simplicity, assume it's a grayscale image, and a = 1, b = 2, . . . , z = 26.) Clearly there's a regular pattern here, but to a byte-oriented compressor, the string is completely incompressible—there are no repeated patterns at all. If, however, we make a simple and reversible transformation, replacing each character with the (numerical) difference between it and the character to its left,[8] the story is quite different:

```
a a a a a a a a a a a a a a a a a a a a a a a a a a
```

With this transformation, even a simple run-length encoder can compress the sequence to two bytes (i.e., 26, a). As long as the decoder knows both the compression method (RLE) and filtering algorithm (subtraction in the horizontal direction, with appropriate boundary conditions), it can reconstruct the original data exactly.

Of course, this particular example is a special case; most images are less amenable. On the other hand, most images *do* contain regions of relatively slowly varying colors, and the sequence above represents a specific kind of variation that's quite common in computer graphics: it's a horizontal gradient. So it shouldn't require a great stretch of the imagination to see how pixel-filtering can enhance compression.

PNG uses five such filters, collectively referred to as "filter method 0." (At the time of this writing, there are no other filter methods either defined or planned for PNG, although a refinement added to the MNG specification in late 2000 might find its way into a future major revision of the PNG spec, if and when such a revision occurs.) Each row of an image may have a different filter; an extra byte at the beginning of the transformed row indicates which was selected. We describe the filters in detail below, but first there are some ground rules that apply to all five.

The first rule is that the final result of all filtering arithmetic is output as unsigned 8-bit integers, even though intermediate steps may require larger values and/or signed ones. Modulo-256 arithmetic is used to generate the final values; thus -1 and 511 both map to a final value of 255.

The second rule is that PNG filtering algorithms always operate on "like values." In particular, in an RGB image, the differencing scheme described above would subtract each pixel's red value from its neighbor's red value, and similarly green from green and blue from blue. The rationale behind this approach is simply that the alternative of *intra*pixel differencing does not take advantage of "slowly varying colors"; it sees only the largely uncorrelated differences between the component red, green and blue values. For example, consider a solid row of orange-yellow pixels, each with RGB values of 255, 192, and 0, respectively. While PNG's like-from-like differencing produces a stream of highly compressible zeroes, intrapixel differencing would result in the sequence 255, 193, 64, 255, 193, 64, . . . , which to the deflate engine looks very much like the untransformed row.

The third rule is that filters always apply to *bytes*, not pixels. For low-bit-depth images—for example, 1-bit grayscale, also known as black-and-white—this means that up to eight pixel values are transformed

[8] The issue of boundary conditions—i.e., what lies to the left of the leftmost character—is discussed later.

in each atomic filter operation. For images with 16-bit samples, on the other hand, each sample (gray, red, green, blue, or alpha) is subjected to two independent filtering operations, one on the low-order byte and one on the high byte. While this approach was chosen for simplicity (and orthogonality) of both encoding and decoding implementations, it does have a price: PNG's compression of 16-bps images is decidedly poorer than that of other schemes that handle 16-bit samples "natively." And for the low-bit-depth case, it turns out that filtering is rarely beneficial—although, as always, there are exceptions to the rule.

The fourth rule is that in an interlaced image, filtering applies to each interlace pass as if it were a separate image with its own height and width. To understand this issue, one needs to know a little more about PNG's interlacing scheme and how it differs from GIF. In the GIF case, there are four passes. In the first pass, every eighth row is transmitted; in the second, the interval is again every eighth row, but offset by four; in the third pass, every fourth row is sent, evenly centered between the previous rows; and in the fourth pass, every other row is transmitted, completing the image. (Simple addition should convince you even if the algorithm isn't quite obvious: $1/8 + 1/8 + 1/4 + 1/2 = 1$.) The idea is to present the user with an overall impression of the image very quickly, then fill in the gaps as quickly as possible. While there is a small cost in compression efficiency due to the mixing of different rows and reduction in inter-row correlation, the overall effect is that the image is *perceived* to download faster, or at least to be more quickly recognizable—and perhaps usable.

But the GIF approach has an unnecessary limitation: it interlaces in only one dimension. Horizontally, the image is at full resolution from the very beginning, and the result in most browsers is that early passes appear extremely stretched in the vertical direction (by up to a factor of eight). PNG's approach is to interlace both horizontally and vertically, in a total of seven passes, with a stretch-factor of two or less at all times. But since different passes will then have different widths, and since some filter types apply to pairs of rows, such operations are conceptually much simpler if each pass is treated as a completely separate image for the purposes of filtering.

Now let's have a closer look at the filters themselves.

Of the five filters defined in the current PNG spec, the simplest is the *None* filter, which implements the trivial case of no filtering. As we noted above, the None filter is usually the best choice for grayscale and palette images that are less than 8 bits deep, but for palette images it turns out to be the best *at* 8 bits, also.

The next simplest filters, *Sub* and *Up*, are conceptually similar. Both are differencing filters, like the alphabetic example given earlier; Sub operates on the previous "corresponding byte" (i.e., of the pixel or pixels to the left), while Up operates on the corresponding byte in the previous row of the image. The region outside the image is considered to be filled with zero bytes, so there is never a problem using Up on pixels in the top row or Sub on the first pixel of any row. The PNG specification uses the following notation:

$$\texttt{Sub(x)} = \texttt{Raw(x)} - \texttt{Raw(x-bpp)}$$

$$\texttt{Up(x)} = \texttt{Raw(x)} - \texttt{Prior(x)}$$

Here `bpp` is the number of bytes per pixel, rounded up to one in the case of fractional values. Thus for a standard 24-bit RGB image, $\texttt{bpp} = 3$, while for a 2-bit (four-color) colormapped image, $\texttt{bpp} = 1$. (The maximum value of `bpp` is 8, for a 64-bit RGBA image.) `Raw(x)` is the unfiltered byte at position `x`, while `Raw(x-bpp)` is the corresponding byte to the left (Filtering Rule 3) and `Prior(x)` is the corresponding byte in the previous row. `Sub(x)` and `Up(x)` are the filtered bytes, of course, and the subtraction is performed modulo 256 (i.e., as unsigned bytes). The decoder can easily reverse the filtering (again using modulo-256 arithmetic), since at each point in the output stream it knows the filtered byte at the current position and the raw (decoded) bytes at all previous positions:

$$\texttt{Raw(x)} = \texttt{Sub(x)} + \texttt{Raw(x-bpp)}$$

$$\texttt{Raw(x)} = \texttt{Up(x)} + \texttt{Prior(x)}$$

The fourth filter type, *Average*, is basically a combination of Sub and Up. That is, instead of subtracting *either* the value to the left *or* the one above, it subtracts the average of the two:

$$\texttt{Avg(x)} = \texttt{Raw(x)} - (\texttt{Raw(x-bpp)} + \texttt{Prior(x)})/2$$

The only subtlety is that the sum of `Raw(x-bpp)` and `Prior(x)` is performed without overflow (that is, *not* modulo 256), and the division is the integer variety, wherein any fractional parts are truncated (or, alternatively, any remainder is dropped). Again, decoding is straightforward:

$$\texttt{Raw(x)} = \texttt{Avg(x)} + (\texttt{Raw(x-bpp)} + \texttt{Prior(x)})/2$$

The fifth and final PNG filter type is the most complex and least intuitive of the set, although in absolute terms it is not particularly tricky. It is called the *Paeth* filter, after Alan W. Paeth, and it not only involves the pixels to the left of and above the current position, but also the pixel to the upper left. The basic idea is to make an “ideal” prediction (also known as the *initial estimate*) of the current byte value based on a simple linear function of the other three; approximate the ideal prediction by selecting the closest of the other three byte values; and then, as with the other filters, subtract this approximate value (the actual Paeth predictor) from the byte value of the current position. The following pseudocode, adapted from that in version 1.2 of the PNG specification, is a more precise description of the predictor algorithm:

```
function PaethPredictor (a, b, c)
begin
     ; all calculations use full-precision, signed arithmetic
     ; a = left, b = above, c = upper left
     est := a + b - c      ; initial estimate
     da := abs(est - a)    ; distances to a, b, c
     db := abs(est - b)
     dc := abs(est - c)
     ; return nearest of a,b,c, breaking ties in order a,b,c.
     if da $\le$ db AND da $\le$ dc then return a
     else if db $\le$ dc then return b
     else return c
end
```

The filtered value is then given by:

$$\texttt{Paeth(x)} = \texttt{Raw(x)} - \texttt{PaethPredictor(Raw(x-bpp), Prior(x), Prior(x-bpp))}$$

And, as with the other filters, it may be easily inverted by the decoder to produce the original value:

$$\texttt{Raw(x)} = \texttt{Paeth(x)} + \texttt{PaethPredictor(Raw(x-bpp), Prior(x), Prior(x-bpp))}$$

But the Paeth predictor is perhaps best understood through examples. Consider the following pixel values (which may be actual pixels in a grayscale image, just the green values in an RGB image, or some subset of the high bytes in a 64-bit RGBA image):

```
20  20             c   b
20  20             a   d
ideal estimate = 20 + 20 - 20 = 20
PaethPredictor = 20
filtered value = 20 - 20 = 0
```

This is the most trivial case: all values are identical, so the prediction should be flawless. The lower right value, d (in boldface), is the one we're currently filtering. The initial (or ideal) estimate is 20, and its distance from each of the three pixel values is the same (zero). Thus the approximate estimate is the same as the ideal one, and the final filtered byte is zero.

Now consider the slightly more interesting case in which there is a horizontal linear gradient:

```
18  20            c   b
18  20            a   d
ideal estimate = 18 + 20 - 18 = 20
PaethPredictor = 20
filtered value = 20 - 20 = 0
```

Once again, both the ideal estimate and the Paeth predictor give the same value, 20, which results in a filtered value of zero. The vertical case is similar.

45-degree diagonal gradients are the next step up in complexity:

```
20  22            c   b
18  20            a   d
ideal estimate = 18 + 22 - 20 = 20
PaethPredictor = 20
filtered value = 20 - 20 = 0

22  20            c   b
20  18            a   d
ideal estimate = 20 + 20 - 22 = 18
PaethPredictor = 20
filtered value = 18 - 20 = -2 = 254 (mod 256)
```

In the first case the gradient is along the a–b diagonal, and as before, the prediction is exact. But in the second case, the gradient is along the c–d diagonal, and while the initial estimate is an exact match, the Paeth predictor is not—resulting in a filtered value of either 2 or 254, depending on the direction of the gradient. We'll comment further on this below.

Our final gradient examples are still linear, but they are oriented (more) arbitrarily with respect to the natural axes of the bitmapped image:

```
18  19            c   b
32  33            a   d
ideal estimate = 32 + 19 - 18 = 33
PaethPredictor = 32
filtered value = 33 - 32 = 1

18  20            c   b
32  34            a   d
ideal estimate = 32 + 20 - 18 = 34
PaethPredictor = 32
filtered value = 34 - 32 = 2
```

As in the previous examples, the ideal estimate turns out to be exact—it is, after all, a linear estimate—while the Paeth predictor is off by one or two.

Finally, consider a pair of "worst case" examples, at least from the perspective of the initial estimate's range:

```
 0  255           c   b
255  0            a   d
ideal estimate = 255 + 255 - 0 = 510
PaethPredictor = 255
filtered value = 0 - 255 = -255 = 1 (mod 256)
```

```
255  0            c   b
 0  255           a   d
ideal estimate = 0 + 0 - 255 = -255
PaethPredictor = 0
filtered value = 255 - 0 = 255
```

These represent the two possible cases for a checkerboard arrangement of pixels, and neither the "ideal" estimate nor the Paeth predictor is a good prediction. Indeed, the resulting byte stream alternates between 1 and 255, which, from the compression engine's perspective, is essentially identical to the unfiltered byte stream.

All of these examples prompt an obvious question: given that the ideal estimate seems to be a better predictor than the Paeth approximation, why not simply use the ideal one instead? As the last examples showed, the ideal estimate can vary between -255 and $+510$; but the filtering operation subtracts using modulo-256 arithmetic anyway, so there's no harm in that.

In fact, there are two reasons. The first is intrinsic to the design of the filter; it was created to predict smoothly varying colors. But if the byte values are near their maximum, for example, it is possible for the ideal estimate to wrap around to the minimum value instead:

```
253 254           c   b
255 254           a   d
ideal estimate = 255 + 254 - 253 = 256 = 0 (mod 256)
PaethPredictor = 255
```

Of course, since the filtering operation is performed with modular arithmetic, there is little difference in the final byte stream (254 vs. 255 in this example), so this is a weak reason, at best. The second reason is more substantial: there will be many more possible filtered values with the ideal estimate—which need not match any of the input bytes—than with the Paeth estimate. For example, if one has an RGB image in which only four colors are used, there are, at most, 37 possible values for the filtered bytes: zero (when the predicted value equals the target value), plus 4×3 nonzero possibilities per color channel. Using the ideal estimate would produce up to 163 possible values for the filtered bytes, almost four and a half times as many. Paeth's claim was that the larger alphabet of byte values would reduce the compressibility of the filtered stream, which is likely. On the other hand, the greater propensity for exact matches would produce an output stream with more zeroes, which should improve compression—at least for some images. To our knowledge, this has never actually been tested in the context of PNG, so it might be an interesting experiment to try.

We'll close our discussion of the Paeth filter with a quick look at one of the more famous examples of its use, a $512 \times 32,768$ RGB image containing one pixel of every possible 24-bit color. The name of the image is `16million`—logically enough—and its uncompressed size is 48 MB. Standard deflate (`gzip`, in this case) reduces it to just under 36 MB. But the Paeth-filtered PNG (with the exception of the top row, which uses the Sub filter) rings in at 115,989 bytes, an overall compression factor of 434:1 and more than 300 times better than the unfiltered approach.[9]

[9] A 4096×4096 variant compresses to 59,852 bytes, thanks to the longer runs of uninterrupted pixels. (Recall that the filter type is encoded as a byte at the beginning of each row; its value is 4 for Paeth.) Obviously these images are highly unusual, but the point remains: filtering can have a dramatic effect on compression efficiency.

8 Practical Compression Tips

Knowing the mechanics of PNG's five compression filters and seeing their effects on one (fairly contrived) example is only the beginning, however. In fact, arguably the most interesting aspect of filtering is determining which filter types to use. A simple counting argument shows that an exhaustive "brute force" approach is untenable for all but the smallest images: with a choice of five filters for each row, the total number of possibilities is $5^{\texttt{image\_height}}$, which comes to nearly 10 million for a 10-row image and almost 10^{70} for a 100-row image. To get a handle on this, consider that at a rate of 3000 possibilities per second—which, keep in mind, means that the *entire image* is being test-compressed 3000 times a second—a computer would be occupied for the entire current age of the universe (ten billion years) checking all possible filter combinations for a single, 30-row image. And that doesn't even cover the different zlib compression levels and strategies, which would extend the duration by a relatively paltry additional factor of 27.

So brute force is Right Out. Instead, the PNG developers experimented on different image types and came up with some rules of thumb that are simple, fast, and (apparently) close to optimal, at least in most cases. The first rule of thumb is simple: use filter type None for all palette images and all sub-8-bit grayscale images. Byte-level filtering simply doesn't work very well when each byte contains multiple pixels; and in the case of 8-bit colormapped images, the palette indices (which constitute the actual pixel data) are usually quite uncorrelated spatially, even though the colors they represent may vary smoothly. Colormapped images are also often subject to dithering, which amounts to a compression-destroying partial randomization of the pixels.

The second rule of thumb is also fairly simple: use adaptive filtering for all other PNG image types. More specifically, for each row choose the filter that *minimizes the sum of absolute differences*. What this means is that, rather than using modulo-256 arithmetic when subtracting the predicted value from the actual value at each pixel position, instead do so using standard (signed) arithmetic. Then take the absolute value of each result, add all of them together for the given row, and compare to the sums for the other filter types. Select the filter produces the smallest sum of absolute differences, and proceed to the next row.

On the face of it, the absolute-differences heuristic may not seem particularly good. Among other things, it is biased *against* the None filter for relatively light images (i.e., with color values near 255) but *in favor* of it for dark images. The PNG specification even contains the caveat that "it is likely that much better heuristics will be found as more experience is gained with PNG." Yet in the six years since PNG was created, no better approach has been found, and the test results in the next section suggest that there may not *be* a significantly better heuristic in the case of most larger images. Still, one never knows.

Both of these rules of thumb are implemented in `libpng`, the free reference library used by most PNG-supporting applications, so neither application developers nor users *should* need to worry about them in most cases. Sadly, there are a few well-known apps whose programmers overlooked the *Recommendations for Encoders* section of the PNG spec, with the unfortunate result that they expose their users to the complexity and unpredictability of PNG filter selection but fail to provide any way to adjust the zlib compression level—the one parameter that is simultaneously the most intuitive and the most effective means of trading off compression efficiency for encoding speed. Users of such applications will need to keep the first two rules in mind or else rely on a third-party utility to improve the compression of their PNG images.

If the adaptive approach of Rule Two is too computationally expensive (for example, if implemented in an embedded device), then the Paeth filter is recommended instead. As we've seen, it performs well when linear gradients are involved.

The third rule of thumb has to do with packing pixels and may seem slightly counterintuitive, given the first rule's stricture about not filtering low-bit-depth images. If filtering is ineffective for such images, why not first expand the 1-, 2-, or 4-bit data to 8 bits (i.e., one byte per pixel), then filter and compress the result for a net gain? As it turns out, such an approach is rarely effective—although there are always exceptions to the rule. In general, though, the initial expansion by a factor of two, four or eight is almost never recovered by the deflate engine, much less exceeded. Packing as many pixels as possible into each byte remains the best approach in most cases.

There are other rules of thumb for PNG compression, but for the most part they amount to no more than common sense. For example, as in many endeavors, *use the best tool for the job*. If your image is photographic and absolute losslessness is not required, then by all means use JPEG or an equivalent lossy method! It will be far more efficient than any lossless approach. On the other hand, if the image contains only a handful of colors, standard JPEG is almost never appropriate; the implicit initial conversion from, say, two-bit pixels to 24-bit ones will overwhelm even JPEG's lossy compression algorithm—and on top of that, the results most likely will look horrible. Also, avoid converting JPEG images to PNG whenever possible. The bit-level effects of JPEG's lossy algorithm are quite detrimental to PNG's row filters and deflate engine. Even worse, a low-bit-depth image improperly stored in JPEG format will acquire many more colors than it started with, requiring deeper pixels in PNG format and further degrading compression.

Within the realm of PNG image types, be aware of the differences and, again, use the best one for the job. Never store an image as 24-bit RGB if an 8-bit palette will do, and don't use a palette if grayscale will suffice. Avoid interlacing if it's not useful (such as in small images or those not intended for Web use); PNG's two-dimensional scheme is even more harmful to compression than GIF's one-dimensional approach is. And it should go without saying that an alpha channel is a complete waste of space in an opaque image—yet there are more than a few images floating around the Internet with precisely that problem.

Finally, be wary of misleading comparisons. A 24-bit image saved in PNG or TIFF formats will be saved losslessly at 24 bits 99% of the time; the same image stored in GIF format will, of necessity, be reduced to an 8-bit file. It should be no surprise to anyone that the GIF will be smaller in this case—after all, it was saved using what amounts to a lossy compression method.

9 Compression Tests and Comparisons

Speaking of comparisons, we've discussed compression theory and the details of its implementation at moderate length but have yet to show much in the way of numbers and real-life results. Since we began the chapter with a discussion of PNG's origins in the GIF/LZW controversy, we'll first compare the two formats directly. We'll use the *Waterloo BragZone Repertoire*[10] for all of the comparisons; it's a set of 32 grayscale and color images with a good combination of natural, synthetic or "artistic," and mixed subject matter. In the case of GIF, however, we'll necessarily limit the comparison to just the 24 grayscale images, since GIF realistically cannot support more than 256 colors (or shades of gray) without loss.[11]

Table 2 shows the results of GIF and PNG compression on the Repertoire's GreySet1, which is composed entirely of 256×256 images. Raw values are the byte sizes of the image files, rather than just the LZW or filtered-deflate sizes—this chapter, after all, is devoted to an image format, with all of its features and overhead, and not merely a compression method. One of the first things to notice is that GIF/LZW actually *expands* three of the images. A 256×256, 8-bit, uncompressed grayscale image requires precisely 65,536 bytes (plus perhaps a few hundred bytes of overhead for the file format), but *bridge*, *goldhill*, and *lena* are larger than that by 10975, 2914, and 5579 bytes, respectively. This is characteristic of standard LZW implementations, which have been observed to expand extreme cases by a factor of three or more. An optimal LZW encoder need never expand its input by more than a factor of $9/8$ or so (12.5%), however.

[10] `http://links.uwaterloo.ca/bragzone.base.html`

[11] In principle, GIF *can* support more than 256 colors, but only by using multiple images (either by segmenting an image into rectangular tiles or by defining an appropriate set of transparent overlays), each of which carries its own palette and is therefore limited to no more than 256 colors. In the worst case, this would require more than 1024 bytes per 256 pixels (768 bytes for the palette, 256 bytes for the pixel indices, and a few bytes of overhead for the image block), resulting in an overall expansion factor of 33% over uncompressed RGB data. In addition, it is unclear whether more than a handful of GIF decoders can display such images. As always, use the right tool for the job.

image	GIF	standard PNG		“optimal” PNG				comp.	
name	(bytes)	(bytes)	vs. GIF	(bytes)	vs. GIF	vs. PNG	filters	level	strategy
bird	47516	32474	−31.7%	31448	−33.8%	−3.2%	all	2	Huffman
bridge	76511	48511	−36.6%	48451	−36.7%	−0.1%	all	9	filtered
camera	55441	38248	−31.0%	38087	−31.3%	−0.4%	all	2	Huffman
circles	1576	1829	+16.1%	1117	−29.1%	−38.9%	0	9	filtered
crosses	1665	2000	+20.1%	1365	−18.0%	−31.8%	2	9	default
goldhill	68450	44941	−34.3%	44850	−34.5%	−0.2%	all	2	Huffman
horiz	683	693	+1.5%	297	−56.5%	−57.1%	0	9	filtered
lena	71115	41204	−42.1%	41142	−42.1%	−0.2%	all	4	filtered
montage	42449	24096	−43.2%	23848	−43.8%	−1.0%	all	9	filtered
slope	29465	11683	−60.3%	11204	−62.0%	−4.1%	2	9	filtered
squares	931	640	−31.3%	181	−80.6%	−71.7%	0	9	default
text	4205	2339	−44.4%	1126	−73.2%	−51.9%	all	9	filtered
total size	400007	248658	−37.8%	243116	−39.2%	−2.2%			

Table 2: Waterloo GreySet1 (256×256)

The *standard PNG* column is the result of a straight conversion to PNG with standard filter heuristics, in this case using the `gif2png` utility. PNG significantly outperforms GIF/LZW overall, but it is larger on three of the synthetic images (*circles*, *crosses*, *horiz*). This is because the PNGs were created as 8-bit grayscale images, while the corresponding GIFs all have depths of 3 bits or less. The *“optimal” PNG* column is the result of the `pnmtopng` utility—which is smart enough to use a gray palette for low-bit-depth images—and of the `pngcrush` utility, which explores (in this case) many combinations of PNG row-filters, zlib compression levels, and zlib stategies. The word “many” is important and is why “optimal” is in quotes; as we noted earlier, a complete search of the parameter space would take eons, at least with anything short of a full-blown quantum computer. In the case of `pngcrush`, six filter combinations are checked: five in which one PNG filter type is used for the entire image, and one adaptive strategy using the “absolute differences” heuristic on each row. Together with the various zlib compression settings, it tests up to 120 different possibilities.[12]

The “optimal” results are interesting in several respects. For example, the five images showing the most significant improvement (more than 30%) are the smallest; i.e., they’re already the most compressed. They also happen to be the only ones for which sub-8-bit, palette-based encoding is possible, and together with *slope*, they are the only purely synthetic images in the set. Also note that only two of the images, *crosses* and *squares*, use zlib’s default (`gzip`-like) strategy, while three of the photographic images compressed best with the Z_HUFFMAN_ONLY setting. The latter three are also the only ones for which a very low zlib compression level (2) was used, but this is slightly misleading: while the Z_HUFFMAN_ONLY strategy could, in principle, show different behavior depending on whether or not lazy matching is used, empirically its effects are entirely independent of the compression level. Finally, note that adaptive filtering (using the standard PNG heuristic) is preferred in most of the images, followed by no filtering (filter type 0). The “up” filter (type 2) worked best in two of the synthetic images, however.

Table 3 shows the corresponding results on the Repertoire’s GreySet2, which consists of eight 512×512 images and four odd sizes (one smaller, three larger). Unlike the previous set, only one image in this set is completely synthetic (*france*), and one image is a Web-like montage of photographs separated by white

[12] This is for `pngcrush`’s `-brute` (force) option. The utility’s default mode checks only five or six possibilities and almost always comes within half a percent of the brute-force result. But here we were interested in achieving the best possible results without regard for the time required.

image	GIF	standard PNG		“optimal” PNG				comp.	
name	(bytes)	(bytes)	vs. GIF	(bytes)	vs. GIF	vs. PNG	filters	level	strategy
barb	287781	173773	−39.6%	173374	−39.8%	−0.2%	all	2	Huff.
boat	236608	151914	−35.8%	150639	−36.3%	−0.8%	all	2	Huff.
france	33282	17571	−47.2%	14503	−56.4%	−17.5%	0	9	default
frog	160739	231993	+44.3%	148302	−7.7%	−36.1%	0	9	default
goldhill	255937	160151	−37.4%	159458	−37.7%	−0.4%	all	2	Huff.
lena	264422	150926	−42.9%	150596	−43.0%	−0.2%	all	4	filtered
library	113658	104873	−7.7%	96594	−15.0%	−7.9%	0	9	default
mandrill	306690	204063	−33.5%	203744	−33.6%	−0.2%	all	2	Huff.
mountain	233975	253550	+8.4%	202946	−13.3%	−20.0%	0	9	default
peppers	265732	158764	−40.3%	158260	−40.4%	−0.3%	all	2	Huff.
washsat	87800	105784	+20.5%	74701	−14.9%	−29.4%	4	2	Huff.
zelda	251370	139395	−44.5%	138464	−44.9%	−0.7%	all	2	Huff.
total size	2497994	1852757	−25.8%	1671581	−33.1%	−9.8%			

Table 3: Waterloo GreySet2 (various sizes, mostly 512×512)

space and adorned with short captions (*library*). Also unlike the first set, all twelve PNGs are 8-bit grayscale, although two of the GIFs are 7-bit (*frog* and *mountain*) and one is 6-bit (*washsat*). (Recall that the next lowest depth supported by PNG is 4 bits.)

The most interesting feature of this set of results is that the Z_HUFFMAN_ONLY zlib strategy is most effective in seven of the cases, while the Z_FILTERED strategy works best in only one. The other four cases, for which the default zlib strategy proved most effective, are also those for which filtering was least effective (that is, filter type None worked best). It is also worth noting that, although GIF outperformed the standard PNG conversion in three cases, “optimal” PNG compression won in every case, with an overall 33% improvement over GIF/LZW.

The results from the first two sets of images clearly demonstrate that different combinations of zlib compression parameters and PNG filters can have a dramatic effect on overall compression efficiency, but what is not clear is how much of the effect is due to each degree of freedom. In Table 4 we take a somewhat different approach: instead of simply comparing against standard LZW results (in this case TIFF/LZW, since the final set consists of 24-bit RGB images), we also include the results of ordinary, unfiltered deflate compression—specifically, using the raw PPM format of Jef Poskanzer’s PBMPLUS utilities, with standard `gzip` providing the deflate compression.

As before, there are a number of interesting features in these results. For one thing, even ordinary `gzip` (compression level 9) performs much better than TIFF/LZW—three times better in several cases, in fact, and 40% better overall. But when we add PNG filtering on top of that, we lop off another 25% over unfiltered `gzip`, or an additional 15 percentage points relative to TIFF. In the cases with the largest gains, adaptive filtering and the Z_FILTERED zlib strategy usually worked best, while the two cases with the smallest gains used (unsurprisingly) exactly the same parameters as `gzip`: no filtering, maximal compression level, and default zlib strategy. In fact, the three images with the smallest differences between PNG and `gzip`—*clegg*, *frymire*, and *serrano*—are also the three “artistic” images; the other five are photographs.

One could devote considerable effort to statistical analysis of the filters used in the adaptive cases, how those choices interact with the deflate engine (or an alternative, such as the BWT-based `bzip2`), and whether, given the existing set of five PNG filter types, one might derive a better heuristic for adaptive filtering. But that is considerably beyond the scope of this chapter. It is worth noting, however, that in the

image	TIFF	gzip'd PPM		"optimal" PNG				comp.	
name	(bytes)	(bytes)	vs. TIFF	(bytes)	vs. TIFF	vs. gzip	filters	level	strategy
clegg	1736820	518240	−71.2%	484646	−72.1%	−6.5%	all	9	filtered
frymire	758358	252448	−66.7%	251616	−66.8%	−0.3%	0	9	default
lena	950128	733327	−22.8%	475487	−50.0%	−35.2%	all	4	filtered
monarch	1215930	846772	−30.4%	615329	−49.4%	−27.3%	all	9	filtered
peppers	911336	678298	−25.6%	425617	−53.3%	−37.3%	all	2	Huff.
sail	1272498	952235	−25.2%	777176	−38.9%	−18.4%	3	9	default
serrano	315432	107181	−66.0%	106419	−66.3%	−0.7%	0	9	default
tulips	1373006	1004650	−26.8%	680950	−50.4%	−32.2%	all	9	filtered
total size	8533508	5093151	−40.3%	3817240	−55.3%	−25.1%			

Table 4: Waterloo ColorSet (various sizes, 512×512 to 1118×1105)

six years since the PNG specification was created, no better heuristic than the "minimal sum of absolute differences" has ever been found. Indeed, even the computationally intensive "greedy filtering" approach, wherein each row is test-compressed with all five filters in order to find the one producing the smallest output, appears to be better than the standard heuristic only some of the time, and rarely by more than a few percent.[13]

10 MNG

MNG, short for *Multiple-image Network Graphics*, was originally conceived as a minimal meta-format for gluing together sequences of PNG images. But because there was no pressing need for a multi-image or animated PNG extension in early 1995, its development was left on hold for more than a year, allowing developers time instead to write the first PNG implementations and to begin the long process of standardization.

In the meantime, as we noted earlier, Netscape introduced animated GIFs in late 1995, and by early 1996 it was abundantly clear that they were a very popular feature and likely to remain so for the foreseeable future. As a result, in May 1996 the PNG group began discussing MNG once more, in hopes of hammering out a PNG-like specification within a PNG-like timeframe—more or less. However, unlike the PNG case, in which there was broad consensus on the main requirements for the format almost from the very beginning, the MNG feature set was the subject of considerable disagreement. Proposals ranged from the minimalist "glue" format envisioned in the beginning to a full-blown, audio/video format approaching MPEG in complexity. In addition, MNG simply was not as interesting as PNG to many of the original developers; for most applications, including print, a still image is far more useful than a moving one, and a moving image that makes noise can be downright annoying.

As a result, MNG development was exceedingly slow, even by international standards. In the end, a middle ground between the minimalist approach and the high end was found; but in truth, this was due more to attrition than to a true consensus of all involved. Nevertheless, the final MNG specification, approved in January 2001, is a good, solid design that should keep both developers and designers busy for some years to come.

In brief, MNG is a proper superset of PNG designed for animation and other multi-image purposes. As such, its chunk structure and compression engine are essentially identical to PNG's (with one major

[13] This has not been tested recently, however. The sole implementation was an SGI-only hack using a modified version of `gzip`, and no one has yet reimplemented the method using zlib.

exception, on which we touch at the end), so we won't cover them in detail. But there at least half a dozen important differences that can improve MNG's compression relative to a collection of individual PNG images, and these are certainly worthy of some discussion.

The first is simply the sharing of image information at the chunk level, typically a global palette in colormapped images. Particularly for collections of small images such as icons, if all or most of the images use the same palette, then eliminating all but one copy of it can be quite beneficial. Just how beneficial depends entirely on the nature of the images, of course, but keep in mind that palettes are uncompressed and can be as large as 780 bytes.

The second improvement over independent PNGs also involves sharing data between sub-images, but at a much more intimate level. MNG supports the concept of image *deltas* or differences, essentially subtracting one image from the next and encoding the result.[14] Insofar as this is one of the most complex features of the MNG specification (as defined by the MNG complexity profile), at the time of this writing there is still very little software available to test MNG delta-coding. However, the MNG version of `16million.png` uses this technique and reduces the file size from 115,989 bytes to 472 bytes, an additional 246:1 improvement over PNG and a fairly impressive 106,635:1 total compression ratio (relative to the uncompressed image size of 50,331,648 bytes).

The third difference is yet another twist on the data-sharing theme: object reuse. Unlike GIF, MNG supports the concept of image objects or "sprites." Rather than repeatedly encoding the same image data in order to move a section of the image, MNG simply encodes it once, assigns it an object number (index), and thereafter references only the index for moves and copies. In other words, this is yet another example of dictionary-based compression, but on a more macroscopic scale. The potential file-size savings over a corresponding GIF animation are enormous, but they are likely to be realized only if the animation is created as a MNG. Automatic conversion of a repeated-subimage GIF into a object-based MNG is certainly possible, but it would require more intelligence than most conversion programs possess.

Loops are another important difference from PNG, and the ability to nest them gives MNG a substantial advantage over GIF—though at a cost in complexity, too, of course. `16million.mng` uses a pair of nested loops in combination with delta-coding in order to collapse 32,768 image rows to just two.

Finally, a late change to the MNG specification was the addition of a new filter type, intrapixel differencing, borrowed from the LOCO-I compression method. Unlike all of the other PNG filter types, which operate on any PNG image type and involve differences between bytes of *different* pixels, the new filter applies only to RGB or RGBA images, affects only the red and blue values by subtracting the green value from them, and feeds *that* result to the normal PNG filtering procedure. The combination of the new filter type and the existing PNG filters is collectively known as "filter method 64." To the surprise of most PNG and MNG designers, this approach produces noticeably better compression than the standard PNG approach—on the order of 10% better for photographic images. If and when the PNG standard is updated incompatibly, filter method 64 will probably be added to it, as well.

These five differences from PNG are immediately available to any MNG implementation. But in discussing deflate and the zlib compression format earlier, we also alluded to the fact that preset LZ77 dictionaries could, in principle, improve compression in a multi-image format. Although there has been no serious investigation of this possibility to date (and it may very well *not* prove useful in more than a tiny handful of situations), the basic idea would be to include a dictionary of commonly seen byte sequences in an as-yet-undesigned global MNG chunk—which may itself be compressed. All of the subsequent individual image streams would refer to this preset dictionary, which a decoder would pass to the zlib library using standard function calls. They would thereby reap immediate benefits in compression efficiency rather than

[14] In this sense, it is an *inter-image* filtering method similar to the Sub and Up filter types; imagine the images stacked on top of each other, and the three filters correspond to the three principal axes or dimensions. One could also view it as a lossless analogue to standard MPEG-1 techniques.

slowly ramping up as the more common dynamic dictionary is generated. Such an approach could be quite beneficial to a collection of small images, all of which might be significantly smaller than the 32 KB deflate sliding window. On the other hand, the images would be completely indecipherable to any decoder that did not understand the hypothetical new preset-dictionary chunk—which is all of them, currently—so this idea is more of a theoretical possibility than a realistic compression approach.[15]

MNG's last major difference from PNG is truly fundamental. Because lossy compression has always been recognized as far more efficient than lossless, at least for many image types, MNG developers decided to support standard JPEG compression as well as lossless PNG compression. For example, a photographic background (stored in JPEG format) can be combined with a lossless, transparent, animated text or graphic overlay (stored in PNG format), thus allowing the author to combine the best features of both formats within a single file. JPEG is also supported for MNG alpha channels, which is not necessarily useful for the sharp-edged kind that are typical of antialiased images or text, but it can be highly effective for softer-edged alpha layers such as the feathered ovals often used in portraiture.

11 Further Reading

- *PNG: The Definitive Guide*, Greg Roelofs (O'Reilly and Associates, 1999)
- *The Data Compression Book, Second Edition*, Mark Nelson and Jean-loup Gailly (M&T Books, 1996)
- *Compressed Image File Formats*, John Miano (ACM Press/Addison Wesley Longman, 1999)
- *Encyclopedia of Graphics File Formats, Second Edition*, James D. Murray and William vanRyper (O'Reilly and Associates, 1996)
- Portable Network Graphics (PNG) home site, Greg Roelofs, `http://www.libpng.org/pub/png/`
- Multiple-image Network Graphics (MNG) home site, Greg Roelofs, `http://www.libpng.org/pub/mng/`
- zlib home site, Jean-loup Gailly, `http://www.zlib.org/`
- "An Explanation of the Deflate Algorithm," Antaeus Feldspar, `http://www.zlib.org/feldspar.html`
- "The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS," M. Weinberger, G. Seroussi, and G. Sapiro (Hewlett-Packard Laboratories Technical Report No. HPL-98-193R1, November 1998, revised October 1999), `http://www.hpl.hp.com/loco/`
- "comp.compression Newsgroup FAQs," Jean-loup Gailly, `http://www.faqs.org/faqs/by-newsgroup/comp/comp.compression.html`
- "PNG (Portable Network Graphics) Specification, version 1.2," Thomas Boutell, Glenn Randers-Pehrson, et al., `http://www.libpng.org/pub/png/spec/`
- "MNG (Multiple-image Network Graphics) Format, version 1.0," Glenn Randers-Pehrson et al., `http://www.libpng.org/pub/mng/spec/`
- "DEFLATE Compressed Data Format Specification, version 1.3," L. Peter Deutsch, `http://www.zlib.org/rfc-deflate.html`

[15] A similar but fully backward-compatible approach would be to concatenate all of the images into a single, large image and then use an appropriate MNG clipping region to select appropriate subframes. Of course, the applicability of either approach would depend entirely on the nature of the application(s) used to encode and decode the MNG file.

- “ZLIB Compressed Data Format Specification, version 3.3,” L. Peter Deutsch and Jean-loup Gailly, `http://www.zlib.org/rfc-zlib.html`