Two Diet Plans for Fat PDF

Thomas A. Phelps and Robert Wilensky
University of California, Berkeley
phelps@cs.berkeley.edu, wilensky@cs.berkeley.edu

ABSTRACT
As Adobe's Portable Document Format has exploded in popularity so too has the number PDF generators, and predictably the quality of generated PDF varies considerably. This paper surveys a range of PDF optimizations for space, and reports the results of a tool that can postprocess existing PDFs to reduce file sizes by 20 to 70% for large classes of PDFs. (Further reduction can often be obtained by recoding images to lower resolutions or with newer compression methods such as JBIG2 or JPEG2000, but those operations are independent of PDF per se and not a component of the results reported here.) A new PDF storage format called "Compact PDF" is introduced that achieves for many classes of PDF an additional reduction of 30 to 60% beyond what is possible in the latest PDF specification (version 1.5, corresponding to Acrobat 6); for example, the PDF 1.5 Reference manual shrinks from 12.2MB down to 4.2MB. The changes required by Compact PDF to the PDF specification and to PDF readers are easily understood and straightforward to implement.

Categories and Subject Descriptors
E.3 [**Coding and Information Theory**]: *Data compaction and compression*

General Terms
Algorithms, Measurement, Documentation, Languages

Keywords
PDF, Compression, Multivalent, Compact PDF

MOTIVATION
It is uncontroversial to state that Adobe's Portable Document Format (PDF) is the de facto way final form digital documents are distributed today. There are many reasons for this, including high technical quality and the free Acrobat viewer available on all major platforms. However, as our results will show, PDFs are often 50% larger than they need to be and in some cases 1000% times larger. There are several reasons for this.

In the first place, there are now innumerable PDF generators, including Adobe Distiller, Adobe PDFWriter, Adobe PDF Library, Aladdin Ghostscript, Corel PDF Engine, CL-PDF, DaVince C++ Class Library, Apache FOP, HPA image bureau, Oracle PDF driver, Panda, PDFlib, ClibPDF Library, dvipdfm, dvips + GNU Ghostscript, htmldoc, iSEDQuickPDF iText, pdfTeX, and various OCR engines. Predictably not every one generates the absolutely most space efficient PDF file. Initially Adobe software was the primary way to generate PDF. First the user "printed" to a PostScript file, which was the universal way of communicating with printers and therefore nearly every application could produce PostScript, and then "distilled" the PostScript to PDF with Adobe Distiller. Distiller is engineered by the company that invented PostScript and has a long history of expertise with graphics- and font-related applications, and thus the user could depend on a certain level of quality. Rather than distilling, it is better for an application to directly write PDF in order to better capture the source document's semantics and in order to take advantage of technical features in PDF that are not in PostScript, such as gradients. However, if a PDF generator is just one of many features of a large application, then as a shipping deadline approaches refinements of a basically working subsystem are not high priority.

DocEng'03, November 20-22, 2003, Grenoble, France.

Second, even for those PDF generators and libraries primarily concerned with PDF, the amount of work to track the PDF specification is enormous and ongoing. Adobe regularly improves PDF by adopting new technology, such as JBIG2 over CCITT Fax, JPEG2000 over JPEG, and Flate over LZW, and compressible object streams over individual uncompressed top-level objects. The increasing sophistication of PDF is reflected in PDF Reference — which as of version 1.5 stands at 1,100 pages, and incorporates by reference several other large, complex specifications such as JPEG2000. Moreover, some PDF features interact with one another and multiply complexity. For example, on top of page building command streams, there is compression, optional encryption, and optional painstaking "linearization", which orders content so that the first page can be viewed quickly over a slow network.

Third, regardless of however well PDF generators track the PDF specification, there remain billions of legacy PDFs. While all PDFs are forward compatible with later specifications (another primary reason for the popularity of PDF), they use older, less efficient technology (which of course was all that was available at the time of document generation). In almost all cases, these new PDFs cannot be regenerated from source, since one usually receives many more PDFs than one generates (just like email) and the sources are not available.

We have developed a tool that optimizes PDF space requirements. It postprocesses existing PDFs, working with all PDF generators, inefficient and efficient, old and modern. It centralizes expertise in the back end so that general applications can concentrate on translating their visuals to clean PDF. Or, since an integrated system is often preferable, applications can compare their file sizes and see if significant improvements are possible, and if so applications can examine the tool's output to identify optimization opportunities. Furthermore, since the tool postprocesses PDF, it operates on legacy PDF, bringing the benefits of modern various compression algorithms as well as other new techniques.

This paper surveys a range of PDF optimizations for space, and utilizes the tool to measure their effectiveness. PDF was designed more than 10 years ago, or almost seven Moore's Law doublings ago, and we consider optimizations that are newly technically practical.

THE STRUCTURE OF PDF

In order to understand the ways PDF can be optimized, a high-level familiarity with the PDF file format is needed. PDF is relatively simple. A brief header of the form `%PDF-`*`m.n`* marks the file as PDF of version *m.n*, a number at the very end of the file points to a cross-reference table, the cross-reference table holds the exact byte offsets of PDF objects, and everything else is one of those objects. Objects can be of the usual types found in programming languages, including strings, integers and real numbers, and arrays. A core data type is the *dictionary*, which is in effect a hash table. Dictionaries and arrays can nest objects, including other dictionaries and arrays. Objects are identified by number, and objects can refer to other objects by number by *indirect references*. Arbitrary byte sequences can be embedded in *streams*, which are dictionaries with metadata (length, compression type, data type) followed by the data bytes. Streams are used for image data, embedded fonts, and arbitrary embedded files, among others uses. Page contents are a sequence of PostScript-like textual commands that are stored in streams and that are executed to build the page as a series of graphical operations. Only streams can be compressed. PDF 1.5 [2] also introduced cross-reference streams, which are more flexible and compressible than previous cross-reference tables.

So as not to overwhelm the reader, we introduce refinements to this basic description as they become relevant.

OPTIMIZATIONS

Techniques

PDF is a rich format and few PDFs take advantage of every aspect: many PDFs have JPEG images, some have JPEG2000 images, some are scanned paper CCITT FAX images lightly wrapped in PDF data structure, some have no images; some have embedded Type 1 fonts, some have embedded TrueType, some rely on the "core 14" set of fonts guaranteed by Acrobat; some have an additional SGML-like structure tree, but most do not; a few have embedded video; HTML conversions have many hyperlinks, but many have no links; and so on. Thus, optimizations specific to images or fonts or annotations can have a great effect on PDFs that use those features, but zero effect on the rest.

We consider only optimizations guaranteed to be safe, with no loss of quality or information. With lossy image compression such as JPEG, one can achieve very high compression by sacrificing quality. Macintosh OS X uses PDF as its imaging model, but the generated PDF files do not use JPEG compression. One PDF compression product achieves most of its effect by compressing image raw samples into JPEG, but JPEG compression loses information and a program must at best rely on heuristics or manual intervention to decide whether the loss is significant or not. While PDF structure information, which is not related to the visual appearance, is relatively new and as yet seldom used, a program cannot automatically determine whether the structure is meaningful or unintended bloat. PDFs can have many named destinations, which are similar to HTML anchors; if not all of them are referenced within the document, the unused may be referenced from other PDFs or instead be due to overzealous labeling (as by FrameMaker). Such optimizations can be enabled with explicit switches to our tool and other PDF optimization tools have target "profiles" that specify the combinations that are appropriate, but none is used in the results reported in this paper.

From among the many possible PDF space optimizations, the following are those that are most effective on most document instances.

Use a modern compression algorithm

PDF is fundamentally a text-based format, writing objects as human-readable text, as opposed to a binary format with carefully defined bit fields. However, compression is essential for reasonable file sizes. Originally the general-purpose compression algorithm was LZW, but this has been superceded by the superior performance of Flate [7]. Only PDF streams can be compressed; the new PDF 1.5 of May 2003 introduces *object streams*, which collect one or more non-streams into streams, which can then be compressed. This is especially useful for hyperlinks and annotations, of which there can be many and which share much of the same content such as dictionary entries (`/Subtype /Link`, `/Border [0 0 0]`).

Modern compression of images can also result in large space savings. Images can be compressed with a variety of formats, and PDF 1.4 and PDF 1.5, respectively, introduced JBIG2 for bitonal images (such as black and white scanned paper) and JPEG2000 for continuous-tone images (such as color photographs). However, image compression is independent of PDF per se: armed with an image compressor, applying it to PDF is a simple matter of rewriting the PDF image's data stream; the other objects in the PDF are unaffected, except their file offsets in the cross reference. For that reason and due to the lack of an available JBIG2 compressor, image recompression is not considered in the results below (which is to say, further compression is possible). Also, as mentioned above, recompressing images can be lossy and therefore problematic for automatic postprocessing concerned about information fidelity.

Remove useless or archaic data

Often slides for a talk repeat a logo image from slide to slide. Inefficient PDF generators produce a separate copy of the logo for each page, rather than using indirect references to share a single copy. PDFs can have tens or hundreds of thousands of objects of sometimes deeply nested structure, and for a PDF generator to catch potential duplicate objects can involve considerable bookkeeping.

PDFs can be incrementally updated, with new objects such as annotations added cheaply to the end of the file. Existing objects are superceded by giving a new object the same number as the object it replaces. While it can be useful in some occasions to retain old versions of objects so as to trace the updates to the PDF, revisions to a document are generally done to some other source such as a Microsoft Word document, and old objects are usually dead weight.

Adobe has carefully tended PDF and Acrobat so that PDFs are always upwardly compatible. However, some constructs used in PDF 1.0 of 10 years ago are archaic in PDF 1.5. PDFs can contain page thumbnail images, but current processors can compute thumbnails rapidly on the fly. Older versions of Acrobat used `ProcSets` summarizing the kind of the content of each page (painting and graphics state, text, color image) in order to know what PostScript preambles to send to the printer, but `ProcSet` are now obsolete. For early versions of PDF it was important to deliver raw PDFs over 7-bit ASCII channels such as e-mail, and PDF included ASCII filters to wrap binary streams, although if the communications program translated line endings the cross reference table could be corrupted anyway (though in a way that could be repaired). Today ASCII transmission is ensured externally to the PDF (e.g., uuencode wrapping for email attachments), making ASCII encoding within PDFs obsolete.

Low-level Writing

The process of transcribing PDF data structures to disk in PDF syntax is simple, but without attention to seemingly insignificant matters much space can be wasted. As the PDF Reference Manual 1.2 says, "omit unnecessary spaces". Many generators insert a space where a syntax metacharacter alone would delimit parse tokens, and write linefeed-newline pairs where one would do. For example, the PDF Reference 1.4 has 30979 objects, and writing space only where necessary saved 747K out of an 8.95MB file. The inefficiency is only an average of tens of bytes per object, but over possibly tens of thousands of objects, the result is bloating by a thousand cuts.

PDFs as a whole can be written linearly, so documents of any length can be written in a single pass with limited memory usage. Stream data can be written as it is generated, with its length given as an indirect forward reference to a number object that is written after the data. This was important when microcomputer memories were measured in kilobytes, but today US$650 buys a PC with 256MB memory, as compared to a very large single compressed object which may be 1MB. The overhead for writing the length as an indirect object is the cost of an indirect reference (e.g., `31699 0 R`) plus the object wrapper for the number itself (`31699 0 obj 24947 endobj`), plus 20 bytes for that object in the cross-reference table, or a total of 40-45 bytes per stream. The PDF 1.5 Reference does not write streams lengths as separate objects, and by doing so it saves this amount 1357 times, for about 55 KB. Some PDF generators apparently think this old convention is mandatory and continue writing stream lengths as separate objects — sometimes before the stream data, negating the original reason.

Other examples of small inefficiencies that add up are writing explicit values that are identical to their default values, and repeating identical settings such as bounding boxes across pages, rather than pushing them higher in the page tree where they can be inherited by individual pages and shared across pages. Also, it is well known that the Flate compression algorithm can be set to run fast and produce sub-optimal compression ratios or run slower for best compression. Moreover, even at the best compression setting it can produce different results. Sometimes compressing all the data in a single Flate "block" works best, but sometimes not: according to a co-author of ZLIB and gzip, "more frequent blocks cost more overhead for the code descriptions, but may improve compression by adapting more rapidly to changing data" [1].

PDF 1.5's object streams can compress away a lot of the inefficiency as a space-slash costs very slightly more than a single slash, but only if the object stream groups many objects with similar inefficiencies. At this writing the one PDF 1.5 document found in the wild, produced by Adobe InDesign 2.0.2 using Adobe PDF Library 5.0, had many streams with 100 component objects but also many with only a single object.

Tool

The tool used to compute the compression results below performs the following optimizations:

- detects and eliminates duplicate objects
- recodes LZW to Flate
- strips off ASCII encoding
- collects objects into PDF 1.5 object streams in groups of 200, which are then compressed with Flate
- writes cross-reference table as a compressed cross-reference stream
- writes objects in compact syntax
- removes old versions of objects
- removes obsolete objects such as thumbnails and `ProcSet`
- inlines small objects such as stream lengths
- reference counts objects and eliminates unused objects, such as single-use objects that were inlined
- omits default values
- shrinks gaps in cross-reference table due to duplicate, inlined or deleted objects. Objects and indirect references overall are renumbered accordingly.

However:

- A document's linearization dictionary, if any, which enables fast viewing of the first page over a network, is lost. This information must be recomputed when a PDF is rewritten, and it is a limitation of the tool that it does not do this. Thus, for those documents that had linearization, compression savings is overstated by a couple thousand bytes.
- The tool is written in Java, but Java's built-in Flate library does not provide control over flushing Flate "blocks"; in all cases exactly one block is produced. While in the great majority of cases the compression produced is identical to that with multiple blocks, in rare cases it is considerably worse.

Results

We ran our compression tool on 1,054 PDF files. Compression ratios ranged from 0% to 99%. By contrast, all HTML is basically text sprinkled with a fixed set of tags and attributes, so one would expect a relatively constant compression ratio, of something somewhat better than plain text as the tags and attributes increase the incidence common strings. The compression ratio depends heavily on the PDF features used, the age of the PDF generator, and the quality of the PDF generator. It would be of little use to report one number for the average compression ratio since that number is so heavily dependent on the individual characteristics of the given PDF tested. For example, on the papers from the Document Engineering 2002 symposium as retrieved from the ACM Digital Library, we observe the following compression ratios: 12%, 37%, 16%, 23%, 22%, 14%, 22%, 18%, 15%, 38%, 51%, 35%, 39%, 53%, 7%, 44%, 12%, 5%. However, if we group by PDF creator code,

rough trends emerge: the generator dvips is associated with 37%, 16%, 14%, 22%, 18%, 38%, 51%, 35%, 39%, 44%, 5%; while Microsoft Word has 12%, 23%, 14%, 22%, 18%, 7%, 12%. (These compression ratios depend on technology introduced after the creators and is not an evaluation of these PDF generators.) Also, for PDF compression there is no common benchmark data set like the common text corpus collections in Information Retrieval.

Thus, for the PDF results below report *representative* ratios (not the best observed) for *classes* of similar documents. Compression obtained by a straight gzip on the full PDF is reported as a baseline. Documents given with a six-digit number are taken from the PDF Database [16], a common collection of about 500 PDFs used to test PDF parsers, and repurposed here.

Compression correctness was validated by a tool developed for this purpose that detects structural differences between to PDFs. Two PDFs are *structurally equivalent* if they render identically and have the same auxiliary data, such as outline trees. Non-structural details include object numbering and dictionary key order. The structural equivalence tool operates by reading the original and compressed versions from files into semantic objects, normalizing data streams to remove compression and ASCII, and finally comparing data structure trees object by object.

Class	Representative Document	Original Size (in bytes)	Simple gzip	Compression	Compression with PDF 1.5	% savings
early PDF	Thinking in PostScript (PDF 1.0)	895156	442025	520066	353086	60%
	stpope_siren7 (PDF 1.1)	2750544	1733318	2135095	2128779	22%
	Old PDFs have ASCII wrappers and LZW for general-purpose compression. The more efficient Flate compression was not introduced until PDF 1.2. Older PDFs all have `ProcSet`s, which were required until PDF 1.4.					
image dominated or high quality generator	unit1	899172	677053	879590	870968	3%
	p231-hall	105595	98578	96937	95895	9%
	If a document is dominated by images and a high quality PDF generator is used, little additional compression is possible. "p231-hall" is typical of the ACM Digital Library's older conference proceedings, which has scanned paper as Group 4 FAX and minimally wrapped it in PDF data structures.					
FrameMaker / hyperlinks	Core API Reference	10422916	4536514	7050445	4325589	58%
	Java Language Specification 2.0	4419906	1622296	2120720	1229672	72%
	collection of Tcl 8.4.2 documentation	8135892	3784950	6234650	3697416	54%
	PDF Reference 1.5 draft	12765416	7399695	10735266	7160361	43%
	PDFs with many hyperlinks used to be expensive. With object streams, the size of the PDF, which is directly readable, is approximately the same size as that produced by running general-purpose gzip (Flate) compression, which requires a separate decompression step before reading. FrameMaker generates many links and many named destinations (anchors), most of a name like `G10.1047755`. Names are verbose but inside object streams compress very well as they often share 9 or 10 of their 11 letters. These documents also have many pages, each with a page dictionary with entries for `Parent`, `Type` (of value `Page`), which also compress well. (Adobe distributed the PDF Reference 1.5 in advance of the Acrobat 6.0 required to read the object streams it describes.)					
duplicate objects / PDFWriter	Hong	12256915	3036573	1350493	1203121	90%
	Navigation	234532	49368	50571	40826	82%
	Slideshows with repeated logo images, each instance of which is in the PDF, compress well as these duplicates are eliminated.					
Improving generators	iccv01	1740164	371088	401840	391774	77%
	000344	385149	368779	338686	328004	14%
	Ghostscript 5.10 did not compress images in "iccv01"; Ghostscript 7.05 does in "000344". However, the legacy 5.10 document is still at its bloated size.					
new PDF generators	000503	146841	30119	38099	35573	75%
	000019	851990	689302	446132	447999	47%
	"Creating PDFs from Microsoft Office Documents" / cmccue_pdfmsofice	3786960	3628342	952234	911080	75%

	New software usually has other concerns of higher priority than optimized PDF. The Apache Formatting Object Processor v0.14, which generated "000503", does not compress content streams. The Oracle PDF Driver, of "000019", does not compress content stream, and uses ASCII85 and LZW on bilevel images rather than Group 4 FAX. Even the "dot-oh" software from Adobe used in "Creating PDFs", Adobe PDF Library 5.0 and Adobe InDesign 2.0, is inefficient, arguing for a postprocessor that centralizes optimization expertise.					
book, magazine, newsletter	UNIX Haters	3639172	2803546	2538438	2424777	33%
	Real World Go Live	18530903	15692402	16463032	15930290	14%
	Journal of Mundane Behavior v3 #3	2165348	1167063	1515347	1014721	53%
	Java Developers Journal v7 #3	13280252	11762178	12002568	11702274	11%
	Seybold Report on Internet Publishing v3 #12 / 0899ip0312	1763859	1629102	1593828	1537953	12%
	It is increasingly popular to distribute full books, magazines, and newsletters as PDFs, since full content and appearance are preserved. A new issue can lead to a network storm in which many people try to download the work at the same time. It is very important to distributors to reduce the size as much as possible.					
Image compressors	AnnualReport	393768	351250	371247	362547	7%
	The CVision PDFCompressor 2.0 mainly applies JBIG2 compression. The results of this compressor can further be reduced by 7% with general techniques.					

OPTIMIZING BEYOND ADOBE'S PDF SPECIFICATION: "COMPACT PDF"

It has been more than 10 years since the definition of PDF, when, as Jim King writes [9], the machine of the day had 640KB of memory and a 80286 processor. Unsurprisingly, some PDF design decisions made under those constraints are no longer relevant. New design decisions assuming 256MB of memory and a 1GHz processor can yield an additional 30 to 60% space savings, while retaining the speed and ease of use the user expects. The few changes required to the PDF specification are easily understood, straightforward to implement, and mesh well with other PDF features such as encryption and linearization. We collectively call our proposed features *Compact PDF*.

This section proposes three ways to achieve significant additional compression beyond what is possible in today's PDF 1.5, measures the effectiveness of the techniques, and considers how to integrate the techniques with standard PDF.

Compact Technique 1: Bulk compression of entire PDF

PDF has always had compression, such as general-purpose LZW and image-specific JPEG, and has regularly introduced new compression technology, such as Flate over LZW and JPEG2000 over JPEG. However, one feature of PDF has prevented more effective use of this compression: its page independence. One problem with PostScript for onscreen reading was that to guarantee correct output for a randomly chosen page, one had to generate all the preceding pages, because PostScript was a programming language and settings made early in the program could affect pages arbitrarily far downstream. One important property of PDF is that every page is independent of the others so that arbitrary pages can be read directly and in any order. Related to but separate from programmatic page independence, every page is compressed independently of the others.

Unfortunately, separate compression is terrible for LZW and Flate. These algorithms work by computing a "dictionary" of strings (byte sequences), and when a sequence has been seen before it can be replaced by a short code that points into the dictionary. Separate compression means that the dictionary has to be reconstructed for each page. Instead, we propose compressing all pages together in a single stream for maximum benefit from shared dictionaries. It for this reason that compressed PostScript (`.ps.gz`) is often smaller than the PDF equivalent. This same technique is used in a different context to compress Java class files [14]. For a pure text document, this yields an additional 40% compression over the best possible in PDF 1.5.

The Compact stream is somewhat similar to PDF 1.5 object streams in that numerous objects are written to the same stream. However, object streams cannot embed other streams, which is essential for sharing across pages.

Perhaps surprisingly, compression is generally increased by putting images, which are already compressed, into the single large page stream. Sometimes images will share a similar color palette; for JPEG images this is embedded in each JPEG bitstream and not shared, but if these JPEGs are put into the same compression stream, they in effect are shared and produce additional compression. When images are different from one another and are effectively noise to the general-purpose compressor, compression degrades by usually less than 1%.

Compact Technique 2: Type 1 font compression

Fonts can be embedded in a PDF in order to guarantee that they are available to the recipient. Acrobat guarantees a "core 14" set of common fonts and missing fonts can be approximated, but if exact appearance is important or the font has unusual glyphs (as symbolic fonts and TeX fonts do), then fonts should be embedded. It is a common practice to *subset* fonts, including only those characters that are actually used in the text. Beyond

that, one important class of font, Adobe's Type 1 [4], can be further compressed.

Type 1 fonts are encrypted. Type 1 font encryption was broken long ago, and now Adobe publishes the encryption method. However, Type 1 fonts embedded in a PDF are still encrypted, presumably so that they can be directly transmitted to a PostScript interpreter that expects to find them this way. Inside the Compact stream this acts like random noise and degrades compression. (In fact, part of the encryption scheme inserts literally random bytes into the font.)

Furthermore, an official part of a Type 1 font is a set of 512 zero bytes that trail the glyph definitions. PDF has a means to make this implicit, but incredibly some PDF generators write this out.

Compact PDF rewrites individual objects, and this is especially effective for embedded Type 1 fonts. On writing the Compact format, Type 1 encryption is stripped out (and the random bytes cleared to space characters). At the very least fonts compress by 14% as they use 8 bits per byte over the previous 7, and all fonts make the 512 zero bytes implicit. On top of this, fonts are susceptible to general-purpose compression for the first time.

Compact Technique 3: More effective compression algorithm: BZip2

For general purpose compression, Flate is very popular. It is very fast for compression and uncompression on all types of data and is free of patents. It compresses better than LZW, is the basis for the popular `gzip` utility, and is the most commonly use compression method in the popular `.zip` format.

For text data, however, the BZip2 compression algorithm [15] usually achieves better compression ratios, often much better. BZip2 is well suited to PDF because, underneath its compression and encryption, PDF is a text-based format. PDF data structure objects and page command streams are both written as text, as opposed to some binary format with carefully defined bit fields.

However, during compression BZip2 is slower than Flate, sometimes much slower. In one case, some preprocessing of the data is needed in order to avoid a worst case for BZip2. Raw image samples, with the same long byte sequences found throughout a long data stream, provoke inordinately long compression times. Fortunately, this special case is easily identified, and the data can be compressed by Flate instead. Otherwise compression is often several times slower than for Flate, but since this is a one-time operation, it is worth the cost. Uncompression is slower than Flate as well, but is usually competitive.

Results

As for the previous results, typical compression is reported for classes of documents, with representative documents providing detail. The base measurements are the original size of the PDF, the size obtainable by a simple gzip, and the smallest size possible that remains compliant with PDF 1.5. Compare that to what the Compact format can achieve. The Compact numbers are reported in subcategories for Flate and BZip2 compression applied to the large Compact stream. The final column reports the amount of space wasted by PDF 1.5 over the Compact format; this number is the inverse of compression savings, for instance, if an additional 50% compression is possible by using Compact rather than PDF 1.5, then twice as many Compact PDFs fit into the same space, or in other words PDF 1.5 wastes +100% of the size of Compact.

Class	Representative Document	Original Size	Simple gzip	Compressed PDF 1.5 compliant	Compact / Flate	Compact / BZip2	savings Compact over Original	inefficiency PDF 1.5 over Compact
pure text	EyesWideShut	138495	99334	80119	46881	35291	74%	+127%
	Pure text PDFs, such as this movie script, benefit greatly from sharing compression dictionaries across pages. From a best case PDF 1.5 compliant size of 80K, an additional compression down to 35K is possible, meaning that PDF 1.5 is 127% larger than necessary to transmit the same information. BZip2 gives better compression than Flate, 35K vs 46K. As expected this technique dramatically outperforms a simple gzip across pages, which although it uses Flate compression, finds individual page streams already compressed and cannot share compression dictionaries across pages.							
template / reference manual / catalog	Acrobat Core API Reference	10422916	4536514	4325589	1675981	1176719	88%	+267%
	DPS.refmanuals.TK	893511	835930	621789	129662	108398	87%	+473%
	Effective Java Chapter 6 / blockch6	189279	173854	125229	43384	36432	80%	+243%
	OpenDoc_Cookbook	2895271	2648687	1953324	428929	384460	86%	+408%
	PostScript Language Reference Manual Level 3	7769823	3687298	3126790	2208720	1670895	78%	+87%
	PDF Reference 1.5 draft	12765416	7399695	7160361	5499771	4420156	65%	+61%

	collection of Tcl 8.4.2 documentation	8135892	3784950	3697416	2016017	1420939	82%	+160%
	Reference manuals and catalogs often have a strong design template repeated from page to page. PDF generators should extract this repetition into a PDF `Form` XObject (as opposed to an interactive fill-in form), which is similar to a program subroutine. Most do not, or rather most applications do not cooperate with PDF generators in a way that makes determination and separation of the template efficient. Instead, the template is repeated on every page. By compressing all these pages together, the template has effectively zero cost after the first copy. The felicitous result is that enormous compression of often 80% is achieved on the largest documents.							
embedded Type 1	brookings	198200	135778	144179	93114	86184	56%	+67%
	gentlesgml	486807	207922	173811	88395	68139	59%	+91%
	riggs	252283	199579	221601	168102	138022	45%	+60%
	These three documents were written in TeX. TeX fonts are non-standard and, in contrast to other outline fonts, different point sizes are different fonts. This can result in quite a few embedded fonts: 16 embedded fonts for brookings, 3 for gentlesgml, 21 for riggs. The comparative compression sizes leaving the fonts encrypted are: brookings 124182 bytes encrypted vs 86184 unencrypted, gentlesgml 82175 vs 68139, riggs 252283 vs 138022. Brookings and gentlesgml were generated by pdfTeX, which at least as of version 13.d needlessly included 512 zero bytes for each font, whereas riggs was generated by Ghostscript, which is not wasteful.							
book, magazine, newsletter	UNIX Haters	3639172	2803546	2424777	2240801	2013136	44%	+20%
	Real World Go Live	18530903	15692402	15930290	13153002	12689475	31%	+25%
	Journal of Mundane Behavior v3 #3	2165348	1167063	1014721	939968	738558	65%	+37%
	Java Developers Journal v7 #3	13280252	11762178	11702274	10727017	10140968	23%	+15%
	Seybold Report on Internet Publishing v3 #12 / 0899ip0312	1763859	1629102	1537953	1331265	1338916	24%	+14%
	Mass distributed documents with mixed content can achieve a double digit reduction in size.							
images with similar color maps	Unit1	899172	677053	870968	842570	331741	63%	+163%
	ManningJDK14	10168352	8871949	8498854	8410687	2913370	71%	+191%
	A surprising result is that — on occasion — BZip2 finds compression in images that eludes Flate.							

Practicality

Undoubtedly the technique of compressing an entire document in one large stream was considered by Adobe's PDF architects. After all, before PDF a common distribution format was compressed PostScript, which in essence is the same. However, with today's hardware, it is newly practical. The key is that a Compact PDF can rapidly be transformed into a standard PDF. Decompression and Type 1 font re-encryption, and recompression of individual independent objects can be done very quickly. For PDFs of up to an original size of, roughly, 1 MB — which is the majority of PDFs — Compact-to-standard rewriting for a Flate-compressed Compact can be done in less than a second (on a 500MHz Pentium III). Once rewritten, PDF viewers and tools can operate normally, without modification. Depending on the PDF library, the standard version can be held in memory (since it is still small) or written to disk.

The largest PDF in our tests, at 15MB, took 30 seconds to rewrite. Even as a rare worst case, that is too long to wait if the PDF is heavily referenced, but PDF viewers can imitate web browsers which cache expensive fetches over the network, and simply cache expensive Compact PDF rewritings. Cacheing can automatically adjust to improving memory and processor, and eliminate that step: if the rewriting takes less than a second, do not write to disk. This technique relies on the two different ways PDF pages are independent: it still relies on programmatic page independence, but random access is sacrificed for compression, with the insight that random access can be rapidly reconstructed on demand.

Integrating with Standard PDF

How much work would authors of PDF viewers, generators, and manipulation libraries have to undertake to support Compact PDF? Not much, we claim. How well does Compact PDF integrate with the various features of standard PDF such as encryption and linearization? Very well, we claim. Individual users can interoperate between Compact PDF and standard PDF already, by rewriting Compact to standard, working with the viewer or tool, and converting back. Of course this is awkward and PDF software should be *Compact aware*.

Supporting Compact PDF requires reading objects from a stream and writing them in standard format, including the byte offsets for the cross-reference table. Any software engineer that

understands PDF reading simply has to reverse the process to write. We have adapted our PDF viewer [11] to recognize Compact PDF. It required about 100 lines of code, in addition to about 300 lines from a PDF writing library. The viewer transparently rewrites to standard format upon reading a PDF. As well, we have written a number of PDF manipulation tools, and because all of these use the same parsing engine as the viewer, they are in fact unaware that a PDF may be in Compact format as the parser completely masks this fact.

Compact PDF is compatible with PDF encryption, incremental writing, and linearization. Syntactically, Compact PDF is valid PDF. Existing viewers cannot find the page streams and other objects to display, but PDF manipulation libraries see only a few unfamiliar dictionary keys, which they ignore, a very large stream in one object, and an unusually sparse cross-reference table. From the point of view of encryption, the Compact stream and other objects written outside the stream in standard format are ordinary objects available for encryption.

PDF can incrementally add content by writing the new objects at the end and writing a new cross-reference table for the new objects and a hook that points to the previous cross-reference table — which is to say, in the standard way incremental content is added. This is sufficient for PDF manipulation libraries. Viewers operating on Compact-aware parser engines could fetch objects through the engine unaware of whether the PDF was rewritten, and write annotations to the original Compact PDF.

Putting the entire document into a single stream defeats the purpose of linearized PDF, which organizes PDF content so that the objects relevant to the first page appear first and that objects are otherwise clustered so that random access to pages requires a minimum and contiguous additional fetch over a slow network connection. However, one could leave the objects relating to the first page out of the Compact stream; this suffers some loss of compression, but regains the fast viewing of the first page over the network. If one wants random access to every page, the Compact format is not suitable, but if the Compact version is 80% smaller, perhaps the cost of transmitting the remaining pages is acceptable.

FUTURE WORK

Other compression algorithms besides Flate and BZip2, of which there are multitudes, could be used. Adobe has chosen open standards for important reasons and maintaining this eliminates many compression algorithms. In practical terms, algorithm implementations should run fast and produce smaller output than what is already achieved. The tension is always between new technology and universal readability of the result and costs of maintaining support in the future.

Based on the Compact format, a few other techniques could deliver significant space savings for some classes of PDF. The Compact format compresses the commonality across objects in the same PDF, and one could consider identifying commonality across PDF documents. For example, when converting to Compact format, any embedded fonts could be stripped out and placed in a shared collection. When converting back to standard format, the fonts could be simply referred to if they are available through the OS, or the fonts could be re-embedded if the PDF is to be redistributed. Font subsetting and duplicate font names of what are in fact different fonts make this nontrivial, but it is likely to be practical for TeX documents, which have a canonical set of fonts and which often embed them.

One Compact technique decrypts and compresses embedded Type 1 fonts. Adobe has a "Compact Font Format" (CFF), a binary format, which may or may not be significantly more compact than compressed decrypted Type 1. CFF defines a default set of character encodings, a savings that could be applied to embedded Type 1 as well.

Some information in a PDF is redundant. In the page tree, parents point to children and children point to parents. In the outline graph, siblings point forward and backward to one another. Object types given by an explicit attribute are often implicit from their position in the structure and their other attributes. All this redundant information could be stripped out before compression in the Compact stream and reconstituted upon rewriting to standard format. A surprising result of a preliminary investigation shows that this can degrade BZip2 compression. That is, less data compresses to a larger size than does this data with additional data. It is surmised the reason is that the additional data, such as type attributes, help BZip2 sort the data into larger homogeneous regions, which then compress better overall.

The Compact format sweeps up gains from several more sophisticated compression techniques — from one perspective the simple single stream compression is dishearteningly effective. For example, if duplicate top-level objects were not already eliminated, Compact would have achieved the same space savings. One could consider identifying page templates and separating them into shared XObjects, but Compact already compresses them across pages. Such duplicate identification and separation techniques remain useful for PDF 1.5 compatibility and for possible non-compression-centric document analysis.

In addition to page templates, another source of repetition in page streams is embedded vector clip art (not bitmap images). Clip art should be separated out in a `Form` XObject, and multiple instances of the art scaled and positioned with different affine transforms. However, in practice clip art seems to be embedded in the page stream for every instance. Moreover the coordinates of the line art are "flattened" to the final positions, rather than keeping identical coordinates and relying on affine transforms, and therefore is resistant even to compression across pages. It would be taxing to find clip art as a program would have to look for streams of, say, 100 commands that are congruent to another stream of 100 commands through some affine transform, from among the millions of commands in the entire PDF.

Compact PDF ‘s large Compact stream with almost all document content is fundamentally opposed to Linearized format which serves pieces of the document over the network. As mentioned, one compromise is to place the first few pages in Linearized format and the rest in Compact. Another possible compromise is to cluster small groups of pages together for better compression while still limiting the data size for incremental serving.

Our compression tool could integrate other research. For instance, other researchers have developed a technique to replace TeX bitmap Type 3 fonts with better-looking outline Type 1 versions [13], and it would be convenient for users to integrate such useful technology in one place.

RELATED WORK

Several people have assembled lists of ways to reduce PDF size. Adobe's PDF Reference Version 1.2 [3] of 1996 devotes 30 pages to "Optimizing PDF Files" (while some recommendations are no longer as relevant, unfortunately this section has been removed from more recent editions). Adobe's Dov Isaacs gives a popular talk [8] that recommends settings in Acrobat for different goals (screen vs print, PDF 1.4 compatiblity vs new PDF 1.5 features) and to work around bugs in other software. Shlomo Perets presents 11 ways to "reduc[e] the size of your PDFs" [12].

Acrobat 6.0 has an "Optimize PDF" function. It collects objects into object streams (and can resample and recompress images). However it seems not to eliminate duplicate objects and not to perform well on large PDFs (larger than a few megabytes).

Apago's PDFshrink [5] was originally designed for Macintosh OS X, which uses PDF as its imaging model but, as of version 10.2 “Jaguar”, does not apply JPEG compression. PDFshrink applies JPEG compression and eliminates duplicate objects. Presumably it will add PDF 1.5 object streams in a future version. Close inspection of PDFs compressed by PDFshrink suggests that they their duplicate object algorithm has a bug. In a test of 24 randomly chosen PDFs, our compression tool (restricted to PDF 1.4) produced smaller PDFs in every case and usually ran twice as fast.

metaobject's PdfCompress [10] is a Mac OS X application that compresses color images with JPEG and black and white images with CCITT Fax Group 4. CVision's CVista PdfCompressor [6] compresses black and white images with JBIG2.

AVAILABILITY

The tool that generates PDF 1.5-compatible compressed PDFs and Compact PDFs is available at `http://www.cs.berkeley.edu/~phelps/Multivalent`. One can use it to archive documents in Compact format, and then use the tool again to convert back to standard PDF for non-Compact-aware PDF tools. Compression ratios for Compact format slightly lower than those reported here because of the space devoted to a new first page that is shown in non-Compact-aware viewers to point to more information. Other Compact-aware PDF tools and a Compact-aware PDF viewer are available there as well. All tools are free. All are implemented in Java and therefore run on Solaris, Macintosh OS X, Linux, Windows, and elsewhere.

The general PDF manipulation library used by the compression tool, the other PDF tools and the viewer is available at the same web site. It is free and open source.

The “Compact PDF Specification” details the changes to PDF 1.5 in the form of the PDF Reference and is posted there as well.

CONCLUSION

Adobe judiciously adopts new technology for PDF, such as JPEG2000 and JBIG2. But old PDFs or those generated with inefficient PDF generators are much larger than they should be. A tool that postprocesses PDFs can centralize optimization expertise for all PDF generators, and update legacy PDFs to current compression technology. Furthermore, now that PDF is more than 10 years old, it makes sense to reexamine the design decisions made in the days of 640KB main memories and 80286 processors. Experiments with our tool show that substantial additional space savings are practical for modern computer hardware.

ACKNOWLEDGEMENTS

This research was supported by the Digital Libraries Initiative under grant NSF CA98-17353. Andy McFadden investigated the two cases where BZip2 wildly outperformed Flate. Derek B. Noonburg commented on how to make the technology transfer. Jim Meehan of Adobe emphasized the importance of Fast Web View for slow network connections.

REFERENCES

[1] Mark Adler. Personal communication.
[2] Adobe Systems Incorporated. "PDF Reference, Third Edition".
[3] Adobe Systems Incorporated. "Portable Document Format Reference Manual, Version 1.2", Addison-Wesley.
[4] Adobe Systems Incorporated. "Adobe Type 1 Font Format", 1990. Third printing 1993, version 1.1.
[5] Apago. PDFshrink. http://www.apago.com/
[6] CVision. CVista PdfCompressor. http://www.cvisiontech.com/
[7] L. Peter Deutsch. "RFC 1951: DEFLATE Compressed Data Format Specification version 1.3", 1996.
[8] Dov Isaacs. "Installing and Configuring Acrobat for Fun and Profit", PDF Conference, Bethesda, MD, June 2-4, 2003. http://www.planetpdf.com/planetpdf/pdfs/pdf2k/03e/isaacs_reliablepdf.pdf
[9] James C. King. "PDF Has It Been 10 Years?", Seybold PDF Summit, Amsterdam, June, 2003.
[10] metaobject. PdfCompress. http://www.metaobject.com/Products.html#PdfCompress
[11] Thomas A. Phelps and Robert Wilensky. "The Multivalent Browser: A Platform for New Ideas", Proceedings of Document Engineering 2001, November 2001, Atlanta, Georgia.
[12] Shlomo Perets. "Reducing the size of your PDFs", PlanetPDF. http://www.planetpdf.com/mainpage.asp?webpageid=1519
[13] Steve Probets and David Brailsford. "Substituting outline fonts for bitmap fonts in archived PDF files", Software--Practice and Experience, Volume 33, Number 9, July 2003.
[14] William Pugh. "Compressing Java Class Files", ACM SIGPLAN Conference on Programming Language Design and Implementation, May 2–4, 1999, pages 247-258.
[15] Julian Seward. "The bzip2 and libbzip2 official home page". http://sources.redhat.com/bzip2/
[16] Michael Still, editor. PDF Database. http://www.stillhq.com/pdfdb/db.html