Optimizing PDF output size of TEX documents

Abstract
There are several tools for generating PDF output from a TEX document. By choosing the appropriate tools and configuring them properly, it is possible to reduce the PDF output size by a factor of 3 or even more, thus reducing document download times, hosting and archiving costs. We enumerate the most common tools, and show how to configure them to reduce the size of text, fonts, images and cross-reference information embedded into the final PDF. We also analyze image compression in detail.
We present a new tool called pdfsizeopt.py which optimizes the size of embedded images and Type 1 fonts, and removes object duplicates. We also propose a workflow for PDF size optimization, which involves configuration of TEX tools, running pdfsizeopt.py and the Multivalent PDF compressor as well.

1 Introduction

1.1 What does a PDF document contain

PDF is a popular document file format designed for printing and on-screen viewing. PDF faithfully preserves the design elements of the document, such as fonts, line breaks, page breaks, exact spacing, text layout, vector graphics and image resolution. Thus the author of a PDF document has precise control over the document's appearance—no matter what operating system or renderer software is used for viewing or printing the PDF. From the viewer's perspective, a PDF document is a sequence of rectangular pages containing text, vector graphics and pixel-based images. In addition, some rectangular page regions can be marked as hyperlinks, and Unicode annotations can also be added to the regions, so text may be copy-pasted from the documents. (Usually the copy-paste yields only a sequence of characters, with all formatting and positioning lost. Depending on the software and the annotation, the bold and italics properties can be preserved.) A tree-structured table of contents can be added as well, each node consisting of an unformatted caption and a hyperlink within the document.

Additional features of PDF include forms (the user fills some fields with data, clicks on the submit button, and the data is sent to a server in an HTTP request), event handlers in JavaScript, embedded multimedia files, encryption and access protection.

PDF has almost the same 2D graphics model (text, fonts, colors, vector graphics) as PostScript, one of the most widespread page description and printer control language. So it is possible to convert between PDF and PostScript without loss of information, except for a few constructs, e.g. transparency and color gradients are not supported by PostScript. Conversion from PDF to PostScript may blow up the file size if there are many repetitions in the PDF (e.g. a logo drawn to each page). Some of the interactive features of PDF (such as forms, annotations and bookmarks) have no PostScript equivalent either; other nonprintable elements (such as hyperlinks and the document outline) are supported in PostScript using `pdfmark`, but many PDF-to-PostScript converters just ignore them.

1.2 How to create PDF

Since PDF contains little or no structural and semantic information (such as in which order the document should be read, which regions are titles, how the tables are built and how the charts generated), word processors, drawing programs and typesetting systems usually can export to PDF, but for loading and saving they keep using their own file format which preserves semantics. PDF is usually not involved while the author is composing (or typesetting) the document, but once a version of a document is ready, a PDF can be exported and distributed. Should the author distribute the document in the native file format of the word processor, he might risk that the document doesn't get rendered as he intended, due to software version differences or because slightly different fonts are installed on the rendering computer, or the page layout settings in the word processor are different.

Most word processors and drawing programs and image editors support exporting as PDF. It is also possible to generate a PDF even if the software doesn't have a PDF export feature. For example, it may be possible to install a printer driver, which generates PDF instead of sending the document to a real printer. (For example, on Windows, PDFCreator [22] is such an open-source driver.) Some old programs can emit PostScript, but not PDF. The ps2pdf [28] tool (part of Ghostscript) can be used to convert the PostScript to PDF.

There are several options for PDF generation from TEX documents, including pdfTEX, dvipdfmx and dvips + ps2pdf. Depending on how the document uses hyperlinks and PostScript programming in graphics, some of these would not work. See the details in Subsection 2.1. See [13] for some more information about PDF and generating it with LATEX.

1.3 Motivation for making PDF files smaller

Our goal is to reduce the size of PDF files, focusing on those created from TEX documents. Having smaller PDF files reduces download times, web hosting costs and storage costs as well. Although there is no urgent need for reducing PDF storage costs for personal use (since hard drives in modern PCs are large enough), storage costs are significant for publishing houses, print shops, e-book stores and hosting services, libraries and archives [26]. Usually lots of copies and backups are made of PDF files originating from such places; saving 20% of the file size right after generating the PDF would save 20% of all future costs associated with the file.

Although e-book readers can store lots of documents (e.g. a 4 GB e-book reader can store 800 PDF books of 5 MB average reasonable file size), they get full quickly if we don't pay attention to optimized PDF generation. One can easily get a PDF file 5 times larger than reasonable by generating it with software which doesn't pay attention to size, or not setting the export settings properly. Upgrading or changing the generator software is not always feasible. A PDF recompressor becomes useful in these cases.

It is not our goal to propose or use alternative file formats, which support a more compact document representation or more aggressive compression than PDF. An example for such an approach is the Multivalent *compact PDF* file format [25], see Section 5 for more details. There is no technical reason against using a compact format for storage, and converting it on the fly to regular PDF before processing if needed. The disadvantage of a nonstandard compact format is that most PDF viewers and tools don't support it by default, so the user has to install and run the conversion tool, which some users can't or won't do just for viewing a PDF. When archiving compact PDF files for a long term, we have to make sure that we'll have a working converter at restore time. With Multivalent, this is possible by archiving the `.jar` file containing the code of the converter. But this may not suit all needs, because Multivalent is not open source, there are no alternative implementations, and there is no detailed open specification for its compact PDF file format.

A pixel-based (fixed resolution) alternative of PDF is DjVu (see Section 5).

It is possible to save space in a PDF by removing non-printed information such as hyperlinks, document outline elements, forms, text-to-Unicode mapping or user annotations. Removing these does not affect the output when the PDF is printed, but it degrades the user experience when the PDF is viewed on a computer, and it may also degrade navigation and searchability. Another option is to remove embedded fonts. In such a case, the PDF viewer will pick a font with similar metrics if the font is not installed on the viewer machine. Please note that unembedding the font doesn't change the horizontal distance between glyphs, so the page layout will remain the same, but maybe glyphs will look funny or hard-to-read. Yet another option to save space is to reduce the resolution of the embedded images. We will not use any of the techniques mentioned in this paragraph, because our goal is to reduce redundancy and make the byte representation more effective, while preserving visual and semantic information in the document.

1.4 PDF file structure

It is possible to save space in the PDF by serializing the same information more effectively and/or using better compression. This section gives a high-level introduction to the data structures and their serialization in the PDF file, focusing on size optimization. For a full description of the PDF file format, see [3].

PDF supports integer, real number, boolean, null, string and name as simple data types. A string is a sequence of 8-bit bytes. A name is also a sequence of 8-bit bytes, usually a concatenation of a few English words in CamelCase, often used as a dictionary key (e.g. `/MediaBox`) or an enumeration value (e.g. `/DeviceGray`). Composite data types are the list and the dictionary. A dictionary is an unordered sequence of key–value pairs, where keys must be names. Values in dictionaries and list items can be primitive or composite. There is a simple serialization of values to 8-bit strings, compatible with PostScript LanguageLevel 2. For example,

```
<</Integer 5 /Real -6.7 /Null null
  /StringInHex <Face> /String ((C)2009\\))
  /Boolean true /Name /Foo /List [3 4 5]>>
```

defines a dictionary with values of various types. All data types are immutable.

It is possible to define a value for future use by defining an *object*. For example, `12 0 obj [/PDF /Text] endobj` defines object number 12 to be an array of two items (`/PDF` and `/Text`). The number 0 in the definition is the so-called generation number, signifying that the object has not been modified since the PDF was generated. PDF makes it possible to store old versions of an object with different generation numbers, the one with the highest number being the most recent. Since most of the tools just create a new PDF instead of updating parts of an existing one, we can assume for simplicity that the generation number is always zero. Once an object is defined it is possible to refer to it (e.g. `12 0 R`) instead of typing its value. It is possible to define self-referential lists and dictionaries using object definitions. The PDF specification requires that some PDF structure elements (such as the `/FontDescriptor` value) be an indirect reference, i.e. defined as an object. Such elements cannot be inlined into other object, but they must be referred to.

A PDF file contains a header, a list of objects, a *trailer* dictionary, cross-reference information (offsets of object definitions, sorted by object number), and the end-of-file marker. The header contains the PDF version (PDF-1.7 being the latest). All of the file elements above except for the PDF version, the list of objects and the trailer are redundant, and can be regenerated if lost. The parsing of the PDF starts at the trailer dictionary. Its `/Root` value refers to the catalog dictionary object, whose `/Pages` value refers to a dictionary object containing the list of pages. The interpretation of each object depends on the reference path which leads to that object from the trailer. In addition, dictionary objects may have the `/Type` and/or `/Subtype` value indicating the interpretation. For example, `<</Subtype/Image ...>>` defines a pixel-based image.

In addition to the data types above, PDF supports streams as well. A *stream* object is a dictionary augmented by the stream data, which is a byte sequence. The syntax is `X Y obj << dict-items >> stream stream-data endstream endobj`. The stream data can be compressed or otherwise encoded (such as in hex). The `/Filter` and `/DecodeParms` values in the dictionary specify how to uncompress/decode the stream data. It is possible to specify multiple such filters, e.g. `/Filter [/ASCIIHexDecode /FlateDecode]` says that the bytes after `stream` should be decoded as a hex string, and then uncompressed using PDF's ZIP implementation. (Please note that the use of `/ASCIIHexDecode` is just a waste of space unless one wants to create an ASCII PDF file.) The three most common uses for streams are: image pixel data, embedded font files and content streams. A content stream contains the instructions to draw the contents of the page. The stream data is ASCII, with a syntax similar to PostScript, but with different operators. For example, `BT/F 20 Tf 1 0 0 1 8 9 Tm(Hello world)Tj ET` draws the text "Hello World" with the font `/F` at size 20 units, shifted up by 8 units, and shifted right by 9 units (according to the transformation matrix `1 0 0 1 8 9`).

Streams can use the following generic compression methods: ZIP (also called flate), LZW and RLE (run-length encoding). ZIP is almost always superior. In addition to those, PDF supports some image-specific compression methods as well: JPEG and JPEG2000 for true-color images and JBIG2 and G3 fax (also called CCITT fax) for bilevel (two-color) images. JPEG and JPEG2000 are lossy methods, they usually yield the same size at the same quality settings—but JPEG2000 is more flexible. JBIG2 is superior to G3 fax and ZIP for bilevel images. Any number of compression filters can be applied to a stream, but usually applying more than one yields a larger compressed stream size than just applying one. ZIP and LZW support predictors as well. A predictor is an easy-to-compute, invertible filter which is applied to the stream data before compression, to make the data more compressible. One possible predictor subtracts the previous data value from the current one, and sends the difference to the compressor. This helps reduce the file size if the difference between adjacent data values is mostly small, which is true for some images with a small number of colors.

There is cross-reference information near the end of the PDF file, which contains the start byte offset of all object definitions. Using this information it is possible to render parts of the file, without reading the whole file. The most common format for cross-reference information is the *cross-reference table* (starting with the keyword `xref`). Each item in the table consumes 20 bytes, and contains an object byte offset. The object number is encoded by the position of the item. For PDFs with several thousand objects, the space occupied by the cross-reference table is not negligible. PDF 1.5 introduces *cross-reference streams,* which store the cross-reference information in compact form in a stream. Such streams are usually compressed as well, using ZIP and a predictor. The benefit of the predictor is that adjacent offsets are close to each other, so their difference will contain lots of zeros, which can be compressed better.

Compression cannot be applied to the PDF file as a whole, only individual parts (such as stream data and cross-reference information) can be compressed. However, there can be lots of small object definitions in the file which are not streams. To compress those, PDF 1.5 introduces *object streams.* The data in an object stream contains a concatenation of any number of non-stream object definitions. Object streams can be compressed just as regular stream data. This makes it possible to squeeze repetitions spanning over multiple object definitions. Thus, with PDF 1.5, most of the PDF file can be stored in compressed streams. Only a few dozen header bytes and end-of-file markers and the stream dictionaries remain uncompressed.

Table 1: Output file sizes of PDF generation from *The TEXbook*, with various methods. The PDF was optimized with pdfsizeopt.py, then with Multivalent.

method	*PDF bytes*	*optimized PDF bytes*
pdfTEX	2283510	1806887
dvipdfm	2269821	1787039
dvipdfmx	2007012	1800270
dvips+ps2pdf	3485081	3181869

2 Making PDF files smaller

2.1 How to prepare a small, optimizable PDF with TEX

When aiming for a small PDF, it is possible to get it by using the best tools with the proper settings to create the smallest possible PDF from the start. Another approach is to create a PDF without paying attention to the tools and their settings, and then optimize PDF with a PDF size optimizer tool. The approach we suggest in this paper is a mixture of the two: pay attention to the PDF generator tools and their fundamental settings, so generating a PDF which is small enough for temporary use and also easy to optimize further; and use an optimizer to create the final, even smaller PDF.

This section enumerates the most common tools which can generate the temporary PDF from a `.tex` source. As part of this, it explains how to enforce the proper compression and font settings, and how to prepare vector and pixel-based images so they don't become unnecessarily large.

Pick the best PDF generation method. Table 2 lists features of the 3 most common methods (also called *drivers*) which produce a PDF from a TEX document, and Table 1 compares the file size they produce when compiling *The TEXbook*. There is no single best driver because of the different feature sets, but looking at how large the output of dvips is, the preliminary conclusion would be to use pdfTEX or dvipdfm(x) except if advanced PostScript features are needed (such as for psfrag and pstricks).

We continue with presenting and analyzing the methods mentioned.

dvips This approach converts TEX source → DVI → PostScript → PDF, using dvips [29] for creating the PostScript file, and ps2pdf [28] (part of Ghostscript) for creating the PDF file. Example command-lines for compiling `doc.tex` to `doc.pdf`:

```
$ latex doc
$ dvips doc
$ ps2pdf14 -d{\PDF}SETTINGS=/prepress doc.ps
```

Table 2. Features supported by various PDF output methods.

Feature	pdfTEX	dvipdfm(x)	dvips
hyperref	+	+	+
Ti*k*Z	+	+	+
beamer.cls	+	$+^{o}$	$+^{u}$
include PDF	+	$+^{b}$	+
embed bitmap font	+	+	+
embed Type 1 font	+	+	+
embed TrueType font	+	+	−
include EPS	−	+	+
include JPEG	+	$+^{x}$	−
include PNG	+	$+^{x}$	−
include MetaPost	$+^{m}$	$+^{m}$	$+^{r}$
psfrag	$-^{f}$	$-^{f}$	+
pstricks	$-^{f}$	$-^{f}$	+
pdfpages	+	−	−
line break in link	+	+	−

b: bounding box detection with ebb or pts-graphics-helper
f: see [21] for workarounds
m: convenient with `\includegraphicsmps` defined in pts-graphics-helper
r: rename file to `.eps` manually
o: with `\documentclass[dvipdfm]{beamer}`
u: use `dvips -t unknown doc.dvi` to get the paper size right.
x: with `\usepackage[dvipdfmx]{graphics}` and shell escape running extractbb

dvipdfmx The tool dvipdfmx [7] converts from DVI to PDF, producing a very small output file. dvipdfmx is part of TEX Live 2008, but since it's quite new, it may be missing from other TEX distributions. Its predecessor, dvipdfm has not been updated since March 2007. Notable new features in dvipdfmx are: support for non-latin scripts and fonts; emitting the Type 1 fonts in CFF (that's the main reason for the size difference in Table 2); parsing pdfTEX-style font `.map` files. Example command-lines:

```
$ latex doc
$ dvipdfmx doc
```

pdfTEX The commands pdftex or pdflatex [41] generate PDF directly from the `.tex` source, without any intermediate files. An important advantage of pdfTEX over the other methods is that it integrates nicely with the editors TEXShop and TEXworks. The single-step approach ensures that there would be no glitches (e.g. images misaligned or not properly sized) because the tools are not integrated properly. Example command-line:

```
$ pdflatex doc
```

The command `latex doc` is run for both dvips and dvipdfm(x). Since these two drivers expect a bit different \specials in the DVI file, the driver name has to be communicated to the TEX macros generating the \specials. For LATEX, dvips is the default. To get dvipdfm(x) right, pass dvipdfm (or dvipdfmx) as an option to \documentclass or to both \usepackage{graphicx} and \usepackage{hyperref}. The package pts-graphics-helper [34] sets up dvipdfm as default unless the document is compiled with pdflatex.

Unfortunately, some graphics packages (such as psfrag and pstricks) require a PostScript backend such as dvips, and pdfTEX or dvipdfmx don't provide that. See [21] for a list of workarounds. They rely on running dvips on the graphics, possibly converting its output to PDF, and then including those files in the main compilation. Most of the extra work can be avoided if graphics are created as external PDF files (without text replacements), Ti*k*Z [8] figures or MetaPost figures. Ti*k*Z and MetaPost support text captions typeset by TEX. Inkscape users can use textext [46] within Inkscape to make TEX typeset the captions.

The \includegraphics command of the standard graphicx LATEX-package accepts a PDF as the image file. In this case, the first page of the specified PDF will be used as a rectangular image. With dvipdfm(x), one also needs a .bb (or .bbx) file containing the bounding box. This can be generated with the ebb tool (or the extractbb tool shipping with dvipdfm(x). Or, it is possible to use the pts-graphics-helper package [34], which can find the PDF bounding box directly (most of the time).

dvipdfm(x) contains special support for embedding figures created by MetaPost. For pdfTEX, the graphicx package loads supp-pdf.tex, which can parse the output of MetaPost, and embed it to the document. Unfortunately, the graphicx package is not smart enough to recognize MetaPost output files (jobname.1, jobname.2 etc.) by extension. The pts-graphics-helper package overcomes this limitation by defining \includegraphicsmps, which can be used in place of \includegraphics for including figures created by MetaPost. The package works consistently with dvipdfm(x) and pdfTEX.

With pdfTEX, it is possible to embed page regions from an external PDF file, using the pdfpages LATEX-package. Please note that due to a limitation in pdfTEX, hyperlinks and outlines (table of contents) in the embedded PDF will be lost.

Although dvipdfm(x) supports PNG and JPEG image inclusion, calculating the bounding box may be cumbersome. It is recommended that all external images should be converted to PDF first. The recommended software for that conversion is sam2p [38, 39], which creates a small PDF (or EPS) quickly.

Considering all of the above, we recommend using pdfTEX for compiling TEX documents to PDF. If, for some reason, using pdfTEX is not feasible, we recommend dvipdfmx from TEX Live 2008 or later. If a 1% decrease in file size is worth the trouble of getting fonts right, we recommend dvipdfm. In all these cases, the final PDF should be optimized with pdfsizeopt.py (see later).

Get rid of complex graphics. Some computer algebra programs and vector modeling tools emit very large PDF (or similar vector graphics) files. This can be because they draw the graphics using too many little parts (e.g. they draw a sphere using several thousand triangles), or they draw too many parts which would be invisible anyway since other parts cover them. Converting or optimizing such PDF files usually doesn't help, because the optimizers are not smart enough to rearrange the drawing instructions, and then skip some of them. A good rule of thumb is that if a figure in an optimized PDF file is larger than the corresponding PNG file rendered in 600 DPI, then the figure is too complex. To reduce the file size, it is recommended to export the figure as a PNG (or JPEG) image from the program, and embed that bitmap image.

Downsample high-resolution images. For most printers it doesn't make a visible difference to print in a resolution higher than 600 DPI. Sometimes even the difference between 300 DPI and 600 DPI is negligible. So converting the embedded images down to 300 DPI may save significant space without too much quality degradation. Downsampling before the image is included is a bit of manual work for each image, but there are a lot of free software tools to do it (such as GIMP [10] and the convert tool of ImageMagick). It is possible to downsample after the PDF has been created, for example with the commercial software PDF Enhancer [20] or Adobe Acrobat. ps2pdf (using Ghostscript's -dDEVICE=pdfwrite, and setdistillerparams to customize, see parameters in [28]) can read PDF files, and downsample images within as well, but it usually grows other parts of the file too much (15% increase in file size for *The TEXbook*), and it may lose some information (it does keep hyperlinks and the document outline, though).

Crop large images. If only parts of a large image contain useful and relevant information, one can save space by cropping the image.

Choose the JPEG quality. When using JPEG (or JPEG2000) compression, there is a tradeoff between quality and file size. Most JPEG encoders based on libjpeg accept an integer quality value between 1 and 100. For true color photos, a quality below 40 produces a severely degraded, hard-to-recognize image, with 75 we get some harmless glitches, and with 85 the degradation is hard to notice. If the document contains lots of large JPEG images, it is worth reencoding those with a lower quality setting to get a smaller PDF file. PDF Enhancer can reencode JPEG images in an existing PDF, but sometimes not all the images have to be reencoded. With GIMP it is possible to get a real-time preview of the quality degradation before saving, by moving the quality slider.

Please note that some cameras don't encode JPEG files effectively when saving to the memory card, and it is possible to save a lot of space by reencoding on the computer, even with high quality settings.

Optimize poorly exported images. Not all image processing programs pay attention to size of the image file they save or export. They might not use compression by default; or they compress with suboptimal settings; or (for EPS files) they try to save the file in some compatibility mode, encoding and compressing the data poorly; or they add lots of unneeded metadata. These poorly exported images make TEX and the drivers run slowly, and they waste disk space (both on the local machine and in the revision control repository). A good rule of thumb to detect a poorly exported image is to use sam2p to convert the exported image to JPEG and PNG (`sam2p -c ijg:85 exported.img test.jpg; sam2p exported.img test.png`), and if any of these files is a lot smaller than the exported image, then the image was exported poorly.

Converting the exported image with sam2p (to any of EPS, PDF, JPEG and PNG) is a fast and effective way to reduce the exported image size. Although sam2p, with its default settings, doesn't create the smallest possible file, it runs very quickly, and it creates an image file which is small enough to be embedded in the temporary PDF.

Embed vector fonts instead of bitmap fonts. Most fonts used with TEX nowadays are available in Type 1 vector format. (These fonts include the Computer Modern families, the Latin Modern families, the URW versions of the base 14 and some other Adobe fonts, the TEX Gyre families, the Vera families, the Palatino family, the corresponding math fonts, and some symbol and drawing fonts.) This is a significant shift from the original TEX (+ dvips) concept, which used bitmap fonts generated by MetaFont. While drivers still support embedding bitmap fonts to the PDF, this is not recommended, because bitmaps (at 600 DPI) are larger than their vector equivalent, they render more slowly and they look uglier in some PDF viewers.

If a font is missing from the font `.map` file, drivers tend to generate a bitmap font automatically, and embed that. To make sure this didn't happen, it is possible to detect the presence of bitmap fonts in a PDF by running `grep -a "/Subtype */Type3" doc.pdf`. Here is how to instruct pdfTEX to use bitmap fonts only (for debugging purposes): `pdflatex "\pdfmapfile\input" doc`. The most common reason for the driver not finding a corresponding vector font is that the `.map` file is wrong or the wrong map file is used. With TEX Live, the updmap tool can be used to regenerate the `.map` files for the user, and the updmap-sys command regenerates the system-level `.map` files. Table 3 shows which driver reads which `.map` file. Copying over pdftex_dl14.map to the current directory as the driver-specific `.map` file usually makes the driver find the font. Old TEX distributions had quite a lot of problems finding fonts, upgrading to TEX Live 2008 or newer is strongly recommended.

Table 3: Font `.map` files used by various drivers and their symlink targets (default first) in TEX Live 2008.

Driver	Font `.map` file
xdvi	ps2pk.map
dvips	psfonts.map →
	psfonts_t1.map | (psfonts_pk.map)
pdfTEX	pdftex.map →
	pdftex_dl14.map | (pdftex_ndl14.map)
dvipdfm(x)	dvipdfm.map →
	dvipdfm_dl14.map | (dvipdfm_ndl14.map)

Some other popular fonts (such as the Microsoft web fonts) are available in TrueType, another vector format. dvipdfm(x) and pdfTEX can embed TrueType fonts, but dvips cannot (it just dumps the `.ttf` file to the `.ps` file, rendering it unparsable).

OpenType fonts with advanced tables for script and feature selection and glyph substitution are supported by Unicode-aware TEX-derivatives such as XeTEX, and also by dvipdfmx.

Omit the base 14 fonts. The base 14 fonts are Times (in 4 styles, Helvetica (in 4 styles), Courier (in 4 styles), Symbol and Zapf Dingbats. To reduce the size of the PDF, it is possible to omit them from the PDF file, because PDF viewers tend to have them. However, omitting the base 14 fonts is deprecated since PDF 1.5. Adobe Reader 6.0 or newer, and other PDF viewers (such as xpdf and evince) don't contain those fonts either, but they can find them as system fonts. On Debian-based Linux systems, those fonts are in the gsfonts package.

In TEX Live, directives *pdftexDownloadBase14* and *dvipdfmDownloadBase14* etc. in the configuration file `texmf-config/web2c/updmap.cfg` specify whether to embed the base 14 fonts. After modifying this file (either the system-wide one or the one in `$HOME/.texlive2008`) and running the updmap command, the following font map files would be created:

pdftex_dl14.map Font map file for pdfTEX with the base 14 fonts embedded. This is the default.
pdftex_ndl14.map Font map file for pdfTEX with the base 14 fonts omitted.
pdftex.map Font map file used by pdfTEX by default. Identical to one of the two above, based on the *pdftexDownloadBase14* setting.
dvipdfm_dl14.map Font map file for dvipdfm(x) with the base 14 fonts embedded. This is the default.
dvipdfm_ndl14.map Font map file for dvipdfm(x) with the base 14 fonts omitted.
dvipdfm.map Font map file used by dvipdfm(x) by default. Identical to one of the two above, based on the *dvipdfmDownloadBase14* setting.

It is possible to specify the base 14 embedding settings without modifying configuration files or generating .map files. Example command-line for pdfTEX (type it without line breaks):

```
pdflatex "\pdfmapfile{pdftex_ndl14.map}
         \input" doc.tex
```

However, this will display a warning *No flags specified for non-embedded font.* To get rid of this, use

```
pdflatex "\pdfmapfile{=
         pdftex_ndl14_extraflag.map}
         \input" doc.tex
```

instead. Get the .map file from [34].

The .map file syntax for dvipdfm is different, but dvipdfmx can use a .map file of pdfTEX syntax, like this:

```
dvipdfmx -f pdftex_dl14.map doc.dvi
```

Please note that dvipdfm loads the *.map* files specified in dvipdfmx.cfg first, and the .map files loaded with the -f flag override entries loaded previously, from the configuration file. To have the base 14 fonts omitted, run (without a line break):

```
dvipdfmx -f pdftex_ndl14.map
    -f dvipdfmx_ndl14_extra.map doc.tex
```

Again, you can get the last .map file from [34]. Without dvipdfmx_ndl14_extra.map, a bug in dvipdfm prevents it from writing a PDF file without the font—it would embed a rendered bitmap font instead.

Subset fonts. Font subsetting is the process when the driver selects and embeds only the glyphs of a font which are actually used in the document. Font subsetting is turned on by default for dvips, dvipdfm(x) and pdfTEX when emitting glyphs produced by TEX.

2.2 Extra manual tweaks on TEX-to-PDF compilation

This sections shows a couple of methods to reduce the size of the PDF created by a TEX compilation manually. It is not necessary to implement these methods if the temporary PDF gets optimized by pdfsizeopy.py + Multivalent, because this combination implements the methods discussed here.

Set the ZIP compression level to maximum. For pdfTEX, the assignment \pdfcompresslevel9 selects maximum PDF compression. With TEX Live 2008, this is the default. Here is how to specify it on the command-line (without line breaks):

```
pdflatex "\pdfcompresslevel9
         \input" doc.tex
```

For dvipdfm(x), the command-line flag -z9 can be used to maximize compression. This is also the default. PDF itself supports redundancy elimination in many different places (see in Subsection 2.3) in addition to setting the ZIP compression level.

There is no need to pay attention to this tweak, because Multivalent recompresses all ZIP streams with maximum effort.

Generate object streams and cross-reference streams. pdfTEX can generate object streams and cross-reference streams to save about 10% of the PDF file size, or even more if the file contains lots of hyperlinks. (The actual saving depends on the file structure.) Example command-line for enabling it (without line breaks):

```
pdflatex "\pdfminorversion5
         \pdfobjcompresslevel3
         \input" doc.tex
```

According to [27], if ZIP compression is used to compress the object streams, in some rare cases it is possible to save space by starting a new block within the ZIP stream just at the right points.

There is no need to pay attention to this tweak, because Multivalent generates object streams and cross-reference streams by default.

Encode Type 1 fonts as CFF. CFF [2] (Type 2 or /Subtype /Type1C) is an alternative, compact, highly compressible binary font format that can represent Type 1 font data without loss. By embedding vector fonts in CFF instead of Type 1, one can save significant portion of the PDF file, especially if the document is 10 pages or less (e.g. reducing the PDF file size from 200 kB to 50 kB). dvipdfmx does this by default, but the other drivers (pdfTEX, dvipdfm, ps2pdf with dvips) don't support CFF embedding so far.

There is no need to pay attention to this tweak, because pdfsizeopt.py converts Type 1 fonts in the PDF to CFF.

Create graphics with font subsetting in mind. For glyphs coming from external sources such as the included

PostScript and PDF graphics, the driver is usually not smart enough to recognize the fonts already embedded, and unify them with the fonts in the main document. Let's suppose that the document contains included graphics with text captions, each graphics source PostScript or PDF having the font subsets embedded. No matter whether dvips, dvipdfm(x) or pdfTEX is the driver, it will not be smart enough to unify these subsets to a single font. Thus space would be wasted in the final PDF file containing multiple subsets of the same font, possibly storing duplicate versions of some glyphs.

It is possible to avoid this waste by using a graphics package implemented in pure TEX (such as Ti*k*Z) or using MetaPost (for which there is special support in dvips, dvipdfm(x) and pdfTEX to avoid font and glyph duplication). The package psfrag doesn't suffer from this problem either if the EPS files don't contain any embedded fonts.

There is no need to pay attention to this tweak, because pdfsizeopt.py unifies font subsets.

Disable font subsetting before concatenation. If a PDF document is a concatenation of several smaller PDF files (such as in journal volumes and conference proceeding), and each PDF file contains its own, subsetted fonts, then it depends on the concatenator tool whether those subsets are unified or not. Most concatenator tools (pdftk, Multivalent, pdfpages, ps2pdf; see [32] for more) don't unify these font subsets.

However, if you use ps2pdf for PDF concatenation, you can get font subsetting and subset unification by *disabling* font subsetting when generating the small PDF files. In this case, Ghostscript (run by ps2pdf) will notice that the document contains the exact same font many times, and it will subset only one copy of the font.

There is no need to pay attention to this tweak, because pdfsizeopt.py unifies font subsets.

Embed each graphics file once. When the same graphics file (such as the company logo on presentation slides) is included multiple times, it depends on the driver whether the graphics data is duplicated in the final PDF. pdfTEX doesn't duplicate, dvipdfm(x) duplicates only MetaPost graphics, and dvips always duplicates.

There is no need to pay attention to this tweak, because both pdfsizeopt.py and Multivalent eliminate duplicates of identical objects.

2.3 How PDF optimizers save space

This subsection describes some methods PDF optimizers use to reduce the file size. We focus on ideas and methods relevant to TEX documents.

Use cross-reference streams compressed with the y-predictor. Each offset entry in an (uncompressed) cross-reference table consumes 20 bytes. It can be reduced by using compressed cross-reference streams, and enabling the y-predictor. As shown in column *xref* of Table 4, a reduction factor of 180 is possible if the PDF file contains many objects (e.g. more than 10^5 objects in *pdfref*, with less than 12000 bytes in the cross-reference stream).

The reason why the y-predictor can make a difference of a factor of 2 or even more is the following. The y-predictor encodes each byte in a rectangular array of bytes by subtracting the original byte above the current byte from the current byte. So if each row of the rectangular array contains an object offset, and the offsets are increasing, then most of the bytes in the output of the y-predictor would have a small absolute value, mostly zero. Thus the output of the y-predictor can be compressed better with ZIP than the original byte array.

Some tools such as Multivalent implement the y-predictor with PNG predictor 12, but using TIFF predictor 2 avoids stuffing in the extra byte per each row—pdfsizeopt.py does that.

Use object streams. It is possible to save space in the PDF by concatenating small (non-stream) objects to an object stream, and compressing the stream as a whole. One can even sort objects by type first, so similar objects will be placed next to each other, and they will fit to the 32 kB long ZIP compression window.

Please note that both object streams and cross-reference streams are PDF 1.5 features, and cross-reference streams must be also used when object streams are used.

Use better stream compression. In PDF any stream can be compressed with any compression filter (or a combination of filters). ZIP is the most effective general-purpose compression, which is recommended for compressing content streams, object streams, cross-reference streams and font data (such as CFF). For images, however, there are specialized filters (see later in this section).

Most PDF generators (such as dvipdfm(x) and pdfTEX) and optimization tools (such as Multivalent) use the zlib code for general-purpose ZIP compression. zlib lets the user specify the *effort* parameter between 0 (no compression) and 9 (slowest compression, smallest output) to balance compression speed versus compressed data size. There are, however alternative ZIP compressor implementations (such as the one in KZIP [30] and PNGOUT [31, 9]), which provide an even higher effort—but the author doesn't know of any PDF optimizers using those algorithms.

Recompress pixel-based images. PDF supports more than 6 compression methods (and any combination of them) and more than 6 predictors, so there are lots of possibilities to make images smaller. Here we focus on lossless compression (thus excluding JPEG and JPEG2000 used for compressing photos). An image is rectangular array of pixels. Each pixel is encoded as a vector of one or more components in the color space of the image. Typical color spaces are RGB (`/DeviceRGB`), grayscale (`/Device▷Gray`), CMYK (`/DeviceCMYK`), color spaces where colors are device-independent, and the palette (indexed) versions of those. Each color component of each pixel is encoded as a nonnegative integer with a fixed number of bits (bits-per-component, BPC; can be 1, 2, 4, 8, 12 or 16). The image data can be compressed with any combination of the PDF compression methods.

Before recompressing the image, usually it is worth extracting the raw RGB or CMYK (or device-independent) image data, and then compressing the image the best we can. Partial approaches such as optimizing the palette only are usually suboptimal, because they may be incapable of converting an indexed image to grayscale to save the storage space needed by the palette.

To pick the best encoding for the image, we have to decide which color space, bits-per-component, compression method(s) and predictor to use. We have to choose a color space which can represent all the colors in the image. We may convert a grayscale image to an RGB image (and back if all pixels are grayscale). We may also convert a grayscale image to a CMYK image (and maybe back). If the image doesn't have more than 256 different colors, we can use an indexed version of the color space. A good rule of thumb (no matter the compression) is to pick the color space + bits-per-component combination which needs the least number of bits per pixel. On a draw, pick the one which doesn't need a palette. These ideas can also be applied if the image contains an alpha channel (which allows for transparent or semi-transparent pixels).

It is possible to further optimize some corner cases, for example if the image has only a single color, then it is worth encoding it as vector graphics filling a rectangle of that color. Or, when the image is a grid of rectangles, where each rectangle contains a single color, then it is worth encoding a lower resolution image, and increase the scale factor in the image transformation matrix to draw the larger image.

High-effort ZIP is the best compression method supported by PDF, except for bilevel (two-color) images, where JBIG2 can yield a smaller result for some inputs. JBIG2 is most effective on images with lots of 2D repetitions, e.g. images containing lots of text (because the letters are repeating). Other lossless compression methods supported by PDF (such as RLE, LZW and G3 fax) are inferior to ZIP and/or JBIG2. Sometimes the image is so small (like 10×10 pixels) that compressing would increase its size. Most of the images don't benefit from a predictor (used together with ZIP compression), but some of them do. PDF supports the PNG predictor image data format, which makes it possible to choose a different predictor for scanline (image row). The heuristic default algorithm in pnmtopng calculates all 5 scanline variations, and picks the one having the smallest sum of absolute values. This facilitates bytes with small absolute values in the uncompressed image data, so the Huffman coding in ZIP can compress it effectively.

Most of the time it is not possible to tell in advance if ZIP or JBIG2 should be used, or whether a predictor should be used with ZIP or not. To get the smallest possible output, it is recommended to run all 3 variations and pick the one yielding the smallest image object. For very small images, the uncompressed version should be considered as well. If the image is huge and it has lots repetitive regions, it may be worth to apply ZIP more than once. Please note that metadata (such as specifying the decompression filter(s) to use) also contributes to the image size.

Most PDF optimizers use the zlib code for ZIP compression in images. The output of some other image compressors (most notably PNGOUT [31], see also OptiPNG [43] and [42] for a list of 11 other PNG optimization tools, and more tools in [15]) is smaller than what zlib produces with its highest effort, but those other compressors usually run a 100 times or even slower than zlib.

How much a document size decreases because of image recompression depends on the structure of the document (how many images are there, how large the images are, how large part of the file size is occupied by images) and how effectively the PDF was generated. The percentage savings in the *image* column of Table 4 suggests that only a little saving is possible (about 5%) if the user pays attention to embed the images effectively, according to the image-related guidelines presented in Section 2.1. It is possible to save lots of space by decreasing the image resolution, or decreasing the image quality by using some lossy compression method (such as JPEG or JPEG2000) with lower quality settings. These kinds of optimizations are supported by Adobe Acrobat Pro and PDF Enhancer, but they are out of scope of our goals to decrease the file size while not changing its rendered appearance.

JPEG files could benefit from a lossless transformation, such as removing EXIF tags and other metadata. Compressing JPEG data further with ZIP wouldn't save space. The program packJPG [33] applies custom lossless compression to JPEG files, saving about 20%. Unfortunately, PDF doesn't have a decompression filter for that.

Convert some inline images to objects. It is possible to inline images into content streams. This PDF feature saves about 30 bytes per image as compared to having the image as a standalone image object. However, inline images cannot be shared. So in order to save the most space, inline images which are used more than once should be converted to objects, and image objects used only once should be converted to inline images. Images having palette duplication with other images should be image objects, so the palette can be shared.

Unify duplicate objects. If two or more PDF objects share the same serialized value, it is natural to save space by keeping only the first one, and modifying references to the rest so that they refer to the first one. It is possible to optimize even more by constructing equivalence classes, and keeping only one object per class. For example, if the PDF contains

```
5 0 obj << /Next 6 0 R /Prev 5 0 R >> endobj
6 0 obj << /Next 5 0 R /Prev 6 0 R >> endobj
7 0 obj << /First 6 0 R >> endobj
```

then objects 5 and 6 are equivalent, so we can rewrite the PDF to

```
5 0 obj << /Next 5 0 R /Prev 5 0 R >> endobj
7 0 obj << /First 5 0 R >> endobj
```

PDF generators usually don't emit duplicate objects on purpose, but it just happens by chance that some object values are equal. If the document contains the same page content, font, font encoding, image or graphics more than once, and the PDF generator fails to notice that, then these would most probably become duplicate objects, which can be optimized away. The method dvips + ps2pdf usually produces lots of duplicated objects if the document contains lots of duplicate content such as `\includegraphics` loading same graphics many times.

Remove image duplicates, based on visible pixel value. Different color space, bits-per-pixel and compression settings can cause many different representations of the same image (rectangular pixel array) to be present in the document. This can indeed happen if different parts of the PDF were created with different (e.g. one with pdfTEX, another with dvips), and the results were concatenated. To save space, the optimizer can keep only the smallest image object, and update references.

Remove unused objects. Some PDF files contain objects which are not reachable from the `/Root` or trailer objects. These may be present because of incremental updates, concatenations or conversion, or because the file is a linearized PDF. It is safe to save space by removing those unused objects. A linearized PDF provides a better web experience to the user, because it makes the first page of the PDF appear earlier. Since a linearized PDF can be automatically generated from a non-linearized one any time, there is no point keeping a linearized PDF when optimizing for size.

Extract large parts of objects. Unifying duplicate objects can save space only if a whole object is duplicated. If a paragraph is repeated on a page, it will most probably remain duplicated, because the duplication is within a single object (the content stream). So the optimizer can save space by detecting content duplication in the sub-object level (outside stream data and inside content stream data), and extracting the duplicated parts to individual objects, which can now be unified. Although this extraction would usually be too slow if applied to all data structures in the PDF, it may be worth applying it to some large structures such as image palettes (whose maximum size is 768 bytes for RGB images).

Reorganize content streams and form XObjects. Instructions for drawing a single page can span over multiple content streams and form XObjects. To save space, it is possible to concatenate those to a single content stream, and compress the stream at once. After all those concatenations, large common instruction sequences can be extracted to form XObjects to make code reuse possible.

Remove unnecessary indirect references. The PDF specification defines whether a value within a compound PDF value must be an indirect reference. If a particular value in the PDF file is an indirect reference, but it doesn't have to be, and other objects are not referring to that object, then inlining the value of the object saves space. Some PDF generators emit lots of unnecessary indirect references, because they generate the PDF file sequentially, and for some objects they don't know the full value when they are generating the object—so they replace parts of the value by indirect references, whose definitions they give later. This strategy can save some RAM during the PDF generation, but it makes the PDF about 40 bytes larger than necessary for each such reference.

Convert Type 1 fonts to CFF. Since drivers embed Type 1 fonts to the PDF as Type 1 (except for dvipdfmx, which emits CFF), and CFF can represent the same font with less bytes (because of the binary format and the smart defaults), and it is also more compressible (because it doesn't have encryption), it is natural to save space by converting Type 1 fonts in the PDF to CFF.

Subset fonts. This can be done by finding unused glyphs in fonts, and getting rid of them. Usually this doesn't save any space for TEX documents, because drivers subset fonts by default.

Unify subsets of the same font. As discussed in Section 2.1, a PDF file may end up containing multiple subsets of the same font when typesetting a collection of articles (such as a journal volume or a conference proceedings) with LaTeX, or embedding graphics containing text captions. Since these subsets are not identical, unifying duplicate objects will not collapse them to a single font. A font-specific optimization can save file size by taking a union of these subsets in each font, thus eliminating glyph duplication and improving compression effectiveness by grouping similar data (font glyphs) next to each other.

Remove data ignored by the PDF specification. For compatibility with future PDF specification versions, a PDF viewer or printer must accept dictionary keys which are not defined in the PDF specification. These keys can be safely removed without affecting the meaning of the PDF. An example for such a key is `/PTEX.Fullbanner` emitted by pdfTeX.

Omit explicitly specified default values. The PDF specification provides default values for many dictionary keys. Some PDF generators, however, emit keys with the default value. It is safe to remove these to save space.

Recompress streams with ZIP. Uncompressing a stream and recompressing it with maximum-effort ZIP makes the stream smaller most of the time. That's because ZIP is more effective than the other general purpose compression algorithms PDF supports (RLE and LZW).

For compatibility with the PostScript language, PDF supports the `/ASCIIHexDecode` and `/ASCII85Decode` filters on streams. Using them just makes the stream in the file longer (by a factor of about 2/1 and 5/4, respectively). These filters make it possible to embed binary stream data in a pure ASCII PDF file. However, there is no significant use case for an ASCII-only PDF nowadays, so it is recommended to get rid of these filters to decrease to file size.

Remove page thumbnails. If the PDF file has page thumbnails, the PDF viewer can show them to the user to make navigation easier and faster. Since page thumbnails are redundant information which can be regenerated any time, it is safe to save space by removing them.

Serialize values more effectively. Whitespace can be omitted between tokens, except between a name token and a token starting with a number or a letter (e.g. `/Ascent 750`). Whitespace in front of `endstream` can be omitted as well. The binary representation of strings should be used instead of the hexadecimal, because it's never longer and it's shorter most of the time if used properly. Only the characters `( \ )` have to be escaped with a backslash within strings, but parentheses which nest can be left unescaped. So, e.g. the string `a(()))(()\b` can be represented as `(a(())\)(\(\\b)`.

Shrink cross-reference data. Renumbering objects (from 1, consecutively) saves space in the cross-reference data, because gaps don't have to be encoded. (Each gap of consecutive missing objects costs about 10 bytes.) Also if an object is referenced many times, then giving it a small object number reduces the file size by a few bytes.

Remove old, unused object versions. PDF can store old object versions in the file. This makes incremental updates (e.g. the *File / Save* action in Adobe Acrobat) faster. Removing the old versions saves space.

Remove content outside the page. `/MediaBox`, `/CropBox` and other bounding box values of the page define a rectangle where drawing takes place. All content (vector graphics or parts of it, images or parts of them, or text) than falls outside this rectangle can be removed to save space. Implementing this removal can be tricky for partially visible content. For example, 8-pixel wide bars can be removed from the edge of a JPEG image without quality loss in the remaining part.

Remove unused named destinations. A named destination maps a name to a document location or view. It can be a target of a hyperlink within the document, or from outside. Some PDF generator software (such as FrameMaker) generates lots of named destinations never referenced. But care has to be taken when removing those, because then hyperlinks from outside the document wouldn't work.

Flatten structures. To facilitate incremental updates, PDF can store some structures (such as the page tree and the content streams within a page) spread to more objects and parts than necessary. Using the simplest, single-level or single-part structure saves space.

3 PDF size optimization tools

3.1 Test PDF files

In order to compare the optimization effectiveness of the tools presented in this section, we have compiled a set of test PDF files, and optimized them with each tool. The *totals* column of Table 4 shows the size of each file (the + and − percentages can be ignored for now), and other columns show the bytes used by different object types. The test files can be downloaded from [36]. Some more details about the test files:

cff 62-page technical documentation about the CFF file format. Font data is a mixture of Type 1, CFF and TrueType. Compiled with FrameMaker 7.0, PDF generated by Distiller 6.0.1.

beamer1 75 slide-steps long presentation created with

Table 4. PDF size reduction by object type, when running pdfsizeopy.py + Multivalent.

document	*contents*	*font*	*image*	*other*	*xref*	*total*
cff	141153 − 02%	25547 − 02%	0	178926 − 91%	174774 − 100%	521909 − 65%
beamer	169789 − 03%	44799 − 54%	115160 − 00%	445732 − 96%	56752 − 98%	832319 − 62%
eu2006	1065864 − 01%	3271206 − 91%	3597779 − 06%	430352 − 80%	45792 − 94%	8411464 − 43%
inkscape	10679156 − 20%	230241 − 00%	6255203 − 20%	943269 − 79%	122274 − 94%	18245172 − 24%
lme2006	1501584 − 14%	314265 − 73%	678549 − 06%	176666 − 91%	31892 − 93%	2703119 − 25%
pdfref	6269878 − 05%	274231 − 04%	1339264 − 00%	17906915 − 79%	6665536 − 100%	32472771 − 65%
pgf2	2184323 − 03%	275768 − 51%	0	1132100 − 84%	190832 − 96%	3783193 − 36%
texbook	1507901 − 01%	519550 − 48%	0	217616 − 84%	35532 − 87%	2280769 − 21%
tuzv	112145 − 03%	201155 − 84%	0	21913 − 77%	2471 − 88%	337764 − 57%

The first number in each cell is the number of bytes used in the original document.
The −. . . % value indicates the percentage saved by optimization.
The data in this table was extracted from the original and optimized PDF files using `pdfsizeopy.py --stats`.

contents: content streams
font: embedded font files
image: pixel-based image objects and inline images, the latter created by sam2p
other: other objects
xref: cross-reference tables or streams
total: size of the PDF file

beamer.cls [40]. Contains hyperlinks, math formulas, some vector graphics and a few pixel-based images. Compiled with pdfTEX. Font data is in Type 1 format.

eu2006 126-page conference proceedings (of EuroTEX 2006) containing some large images. Individual articles were compiled with pdfTEX, and then PDF files were concatenated. Because of the concatenation, many font subsets were embedded multiple times, so a large part of the file is font data. Font data is mostly CFF, but it contains some Type 1 and TrueType fonts as well. Most fonts are compressed with the less effective LZW instead of ZIP.

inkscape 341-page software manual created with codeMantra Universal PDF [5]. Contains lots of screenshots and small images. Font data is a mixture of Type 1, CFF and TrueType.

lme2006 240-page conference proceedings in Hungarian. Contains some black-and-white screenshot images. Individual articles were compiled with LATEX and dvips (without font subsetting), and the PostScript files were concatenated and converted to PDF in a single run of a modified ps2pdf. Since font subsetting was disabled in dvips, later ps2pdf was able to subset fonts without duplication. Font data is in CFF.

pdfref 1310-page reference manual about PDF 1.7 containing quite a lot of duplicate `xref` tables and XML metadata of document parts. Optimization gets rid of both the duplicate `xref` tables and the XML metadata. Font data is in CFF. Compiled with FrameMaker 7.2, PDF generated by Acrobat Distiller 7.0.5.

pgf2 560-page software manual about Ti*k*Z, with lots of vector graphics as examples, with an outline, without hyperlinks. Compiled with pdfTEX. Font data is in Type 1 format.

texbook 494-page user manual about TEX (*The TEXbook*), compiled with pdfTEX. No pixel images, and hardly any vector graphics.

tuzv Mini novel in Hungarian, typeset on 20 A4 pages in a 2-column layout. Generated by dvipdfm. It contains no images or graphics. Font data is in Type 1 format.

None of the test PDF files used object streams or cross-reference streams.

3.2 ps2pdf

The ps2pdf [28] script (and its counterparts for specific PDF versions, e.g. ps2pdf14) runs Ghostscript with the flag `-sDEVICE=pdfwrite`, which converts its input to PDF. Contrary to what the name suggests, ps2pdf accepts not only PostScript, but also PDF files as input.

ps2pdf works by converting its input to low-level PostScript drawing primitives, and then emitting them as a PDF document. ps2pdf wasn't written to be a PDF size optimizer, but it can be used as such. Table 5 shows that ps2pdf increases the file size many times. For the documents *cff* and *pdfref*, we got a file size decrease because ps2pdf got rid of some metadata, and for *pdfref*, it optimized the cross-reference table. For *eu2006* it saved space by recompressing fonts with ZIP. The document *tuzv* became smaller because ps2pdf converted Type 1 fonts to CFF. The reason for the extremely large growth in *beamer1* is that ps2pdf blew up images, and it also embedded multiple instances of the same image as separate images. (It doesn't always do so: if the two instances of the image are close to each other, then ps2pdf reuses the same object in the PDF for representing the image.)

ps2pdf keeps all printable features of the original PDF, and hyperlinks and the document outline as well. However, it recompresses JPEG images (back to a different JPEG, sometimes larger than the original), thus losing quality. The only way to disable this is specifying the flags `-dEncodeColorImages=false -dEncodeGrayImages=false`, but it would blow up the file size even more, because it will keep photos uncompressed. Specifying `-dColorImageFilter=/FlateEncode` would convert JPEG images to use ZIP compression without quality loss, but this still blows up the file size. Thus, it is not possible to set up pdf2ps to leave JPEG images as is: it will either blow up the image size (by uncompressing the image or recompressing it with ZIP), or it will do a transformation with quality loss. The Distiller option `/PassThroughJPEGImages` would solve this problem, but Ghostscript doesn't support it yet.

ps2pdf doesn't remove duplicate content (although it removes image duplicates if they are close by), and it also doesn't minimize the use of indirect references (e.g. it emits the `/Length` of content streams as an indirect reference). The only aspects ps2pdf seems to optimize effectively is converting Type 1 fonts to CFF and removing content outside the page. Since this conversion is also done by pdfsizeopt.py, it is not recommended to use ps2pdf to optimize PDF files.

Table 5. PDF optimization effectiveness of ps2pdf.

document	*input*	*ps2pdf*	*psom*
cff	521909	264861	180987
beamer1	832319	*3027368*	317351
eu2006	8411464	6322867	4812306
inkscape	18245172	failed	13944481
lme2006	2703119	*3091842*	2033582
pdfref	32472771	15949169	11237663
pgf2	3783193	*4023581*	2438261
texbook	2280769	*2539424*	1806887
tuzv	337764	199279	146414

All numeric values are in bytes. Italic values indicate that the optimizer increased the file size.
ps2pdf: Ghostscript 8.61 run as
`ps2pdf14 -dPDFSETTINGS=/prepress`
psom: pdfsizeopt.py + Multivalent

3.3 PDF Enhancer

PDF Enhancer [20] is commercial software which can concatenate, split, convert and optimize PDF documents, and remove selected PDF parts as well. It has lots of conversion and optimization features (see the table in [4]), and it is highly configurable. With its default settings, it optimizes the PDF without removing information. It is a feature-extended version of the PDF Shrink software from the same company. The use of the GUI version of PDF Enhancer is analyzed in [12]. A single license for the *server edition*, needed for batch processing, costs about $1000, and the *advanced server edition* (with JBIG2 support) costs about twice as much. The *standard edition* with the GUI costs only $200.

Columns *input* and *pdfe* of Table 6 show how effectively PDF Enhancer optimizes. The *server edition* was used in our automated tests, but the *standard edition* generates PDF files of the same size. Looking at columns *pdfe* and *a9p4* we can compare PDF Enhancer to Adobe Acrobat Pro. Please note that PDF Enhancer doesn't generate object streams or cross-reference streams, that's why we compare it to *a9p4* instead of *a9p5* in the table. Feeding the output of PDF Enhancer to Multivalent decreases the file size even further, because Multivalent generates those streams. The column *epsom* of Table 6 shows the PDF output file sizes of the PDF Enhancer + pdfsizeopt.py + Multivalent combination, which seems to be the most effective for TeX documents.

According to the messages it prints, PDF Enchancer optimizes content streams within the page. Most other optimizers (except for Adobe Acrobat Pro) don't do this. Text-only content streams generated from TeX don't benefit from such an optimization, but for the *pgf2* document, which contains lots of graphics, this optimization saved about 10% of the content streams.

It is worth noting that PDF Enhancer failed when optimizing one of the test documents (see in Table 6). The developers of PDF Enhancer reply quickly to bug reports, and they are willing to track and fix bugs in the software.

3.4 Adobe Acrobat Pro

Adobe's WYSIWYG PDF manipulation program, Adobe Acrobat Pro [1] also contains a PDF optimizer (menu item *Advanced / PDF Optimizer*). A single license of the whole software costs $450; it is not possible to buy only the optimizer. There seems to be no direct way to run the optimizer on multiple files in batch mode. Columns *a9p4* and *a9p5* of Table 6 shows the effectiveness of the optimizer: values in the column *a9p4* are for PDF 1.4 output, and column *a9p5* belongs to PDF 1.5 output. The PDF 1.5 files are much smaller because they make use of object streams and cross-reference streams. The optimizer lets

Table 6. PDF optimization effectiveness of PDF Enhancer and Adobe Acrobat Pro.

document	*input*	*pdfe*	*epsom*	*psom*	*apsom*	*a9p4*	*a9p5*
cff	521909	229953	174182	180987	158395	548181	329315
beamer1	832319	756971	296816	317351	317326	765785	363963
eu2006	8411464	failed	n/a	4812306	3666315	8115676	7991997
inkscape	18245172	14613044	12289136	13944481	11807680	14283567	13962583
lme2006	2703119	2263227	1781574	2033582	1830936	2410603	2279985
pdfref	32472771	23794114	11009960	11237663	9360794	23217668	20208419
pgf2	3783193	3498756	2245797	2438261	n/a	failed	failed
texbook	2280769	2273410	1803166	1806887	1804565	2314025	2150899
tuzv	337764	*338316*	*147453*	146414	*150813*	*344215*	328843

All numeric values are in bytes. Italic values indicate that the optimizer increased the file size.
pdfe: PDF Enhancer 3.2.5 (1122r) server edition
epsom: PDF Enhancer + pdfsizeopt.py + Multivalent
psom: pdfsizeopt.py + Multivalent
apsom: Adobe Acrobat Pro 9 creating PDF 1.4 + pdfsizeopt.py + Multivalent
a9p4: Adobe Acrobat Pro 9 creating PDF 1.4
a9p5: Adobe Acrobat Pro 9 creating PDF 1.5

the user specify quite a few settings. For the tests we have enabled all optimizations except those which lose information (such as image resampling). It turned out that we had to disable *Discard User Data / Discard all comments, forms and multimedia*, otherwise the optimizer removed hyperlinks from the document *beamer1*.

It is worth noting that Adobe Acrobat Pro 9 failed with an image-related error when optimizing document *pgf2*. Oddly enough, that PDF file doesn't contain any images.

3.5 pdfcompress

pdfcompress [45] is the command-line version of the PDF optimizer in Advanced PDF Tools. It is commercial software, a single-computer license costs less than $80. It can resample and recompress images based on a few set of settings for monochrome, gray and color images. It can also recompress streams, and it can remove some PDF features (such metadata, JavaScript, page thumbnails, comments, embedded files, outlines, private data and forms). We haven't analyzed it, because PDF Enhancer seems to have all the features of pdfcompress.

3.6 Multivalent tool.pdf.Compress

Multivalent [17] is a collection of programs for document viewing, annotation, organization, conversion, validation, inspection, encryption and text extraction (etc.). It supports multiple file formats such as HTML, PDF, DVI and man pages. It is implemented in Java; the 2006 January version is available for download [18] as a single `.jar` file, and it needs Java 1.4 or later. It contains a PDF optimizer [24, 27], which can be invoked like this at the command-line (without line breaks):

```
java -cp Multivalent20060102.jar
```

Table 7: PDF optimization effectiveness of Multivalent and pdfsizeopt.py.

document	*input*	*multi*	*psom*	*pso*
cff	521909	181178	180987	230675
beamer1	832319	341732	317351	443253
eu2006	8411464	7198149	4812306	4993913
inkscape	18245172	13976597	13944481	17183194
lme2006	2703119	2285956	2033582	2349035
pdfref	32472771	11235006	*11237663*	23413875
pgf2	3783193	2584180	2438261	3449386
texbook	2280769	2057755	1806887	1992958
tuzv	337764	314508	146414	166863

All numeric values are in bytes. The Italic value indicates that Multivalent alone was better than with pdfsizeopt.py.
multi: Multivalent 20060102 *tool.pdf.Compress*
psom: pdfsizeopt.py + Multivalent
pso: pdfsizeopt.py without Multivalent

```
    tool.pdf.Compress doc.pdf
```

This creates the optimized PDF in file `doc-o.pdf`. If we don't indicate otherwise, by the term Multivalent we mean its PDF optimizer. Although the 2006 January version of Multivalent with full functionality is available for download, Multivalent is not free software or open source. For example, its license allows running the PDF optimizer from the command-line. For other uses of the optimizer, a commercial license has to be acquired. The web site doesn't show details about commercial licenses.

According to [27], the Mutivalent did the following optimizations in 2003: remove object duplicates; recompress LZW to ZIP; generate object streams; generate a cross-reference stream; serialize values more effectively; remove old object versions; remove page thumbnails;

remove some obsolete values such as `/ProcSet`; inline small objects such as stream lengths; remove unused objects; omit default values; shrink cross-reference data. In addition to those above, Multivalent recompresses all streams with maximum-effort ZIP, and it also moves up `/MediaBox` etc. in the page tree.

Column *multi* of Table 7 shows how effectively Multivalent optimizes. The column *psom* indicates that running pdfsizeopt.py before Multivalent usually decreases the file size even more. That's because pdfsizeopt.py can convert Type 1 fonts to CFF, unify CFF font subsets, and it also has a more effective image optimizer than Multivalent.

3.7 pdfsizeopt.py

pdfsizeopt.py [37] was written as part of this work. Its purpose is to implement the most common optimizations typical TEX documents benefit from, but only those which are not already done by Multivalent. As described in Section 4, to get the smallest PDF, the optimizations done by pdfsizeopt.py should be applied first, and the result should be processed by Multivalent. The 20060102 version of Multivalent optimizes images, and it replaces the image even if the optimized version is larger than the original, so pdfsizeopt.py implements a final step to put those original images back which are smaller.

pdfsizeopt.py can be used as a stand-alone PDF optimizer (without Multivalent), but the final PDF will be much smaller if Multivalent is run as well.

pdfsizeopt.py is free software licensed under the GPL. It is written in Python. It needs Python 2.4 (or 2.5 or 2.6). It uses only the standard Python modules, but it invokes several external programs to help with the optimizations. These are: Ghostscript (8.61 or newer is recommended), sam2p [38] (0.46 is needed), pngtopnm, `tool.pdf.Compress` of Multivalent [24] (which needs Sun's JDK or OpenJDK), optionally jbig2 [14], optionally PNGOUT [31]. Installation instructions are given in [35]. Most of these are free software, except for the Multivalent tools, which are not free software or open source, but they can be downloaded and used on the command line free of charge; for other uses they have to be licensed commercially. PNGOUT is not free software or open source either, but the binaries available free of charge can be used without restriction.

pdfsizeopt.py implements these PDF size optimization methods:

Convert Type 1 fonts to CFF It is done by generating a PostScript document with all fonts, converting it to PDF with Ghostscript (just like ps2pdf), and extracting the CFF fonts from the PDF. Another option would be to use dvipdfmx, which can read Type 1 fonts, and emit them as CFF fonts. Please note that Ghostscript inlines subroutines (`/Subrs`) in the Type 1 font, so the CFF becomes larger—but we are compressing the font with ZIP anyway, which eliminates most of the repetitions.

Unify subsets of the same CFF font Ghostscript is used for parsing CFF to a font dictionary, and also for serializing the modified dictionary as CFF. Again, the latter is done by generating a PostScript file with all the fonts, then converting it to a PDF using Ghostscript. Limitations: it only works for CFF (and former Type 1) fonts; it doesn't unify fonts with different names; it won't unify some fonts if one of them has slightly different metrics.

Convert inline images to objects We need this because most tools (including pdfsizeopy.py) do not optimize inline images. Limitations: it only detects inline images generated by sam2p; it only detects inline images within a form XObject (not in a content stream).

Optimize individual images First the data gets decompressed (with Ghostscript if the image data is compressed with anything other than simple ZIP), then it is recompressed with high-effort ZIP, then it is converted to PNG, then several external PNG compressors are run to get the optimized PNG, and finally the smallest representation (among the optimized PNG files, intermediate images and the original image) is picked, i.e. the one with the smallest PDF image object representation, counting the stream dictionary and the compressed stream as well. The following PNG optimizers are used: sam2p without predictor, sam2p with PNG predictor, PNGOUT (very slow, but generates a few percent smaller PNG files) and jbig2 (only for bilevel images). Limitations: no CMYK support; no device-independent color space support (only RGB with or without palette and grayscale is supported); no images with an alpha channel; only some types of transparency; images with lossy compression (JPEG or JPEG2000) are not optimized.

Remove object duplicates Equivalence classes are used, so duplicate subtrees referring to objects between themselves or each other are also removed. (Multivalent also has this feature.)

Remove image duplicates Images are compared based on RGB pixel data, so duplicates using a different compression or color space or bits-per-component are also detected and removed. This is useful if the PDF is a concatenation of PDF files in the same collection, each PDF compiled with a different method, and then concatenated. The newest version of sam2p (0.46) produces exactly the same output file for two images with identical RGB pixel data, so image duplicates are identified by comparing the files created by sam2p. There are also several early checks in the optimization algorithm to detect the duplicate before wasting time on running the many different optimizers.

Remove unused objects All objects unreachable from the trailer object are removed.

Serialize values more effectively Extra spaces are removed; hex strings are converted to binary; strings are serialized without extra backslashes; comments are removed; garbage between object definitions is removed; gaps in the cross-reference table are removed; objects with high reference counts are given low numbers.

The column *pso* of Table 7 shows how effectively pdfsizeopt.py optimizes. The column *psom* shows the combined effectiveness of pdfsizeopt.py + Multivalent. Please note that it is not with running pdfsizeopt.py alone, because pdfsizeopt.py was designed to do only those optimizations which Multivalent does not provide (or, such as image compression, does suboptimally). On the other hand, it is almost always worth running pdfsizeopt.py before Multivalent, rather than running Multivalent alone. The only exception we could find was the document *pdfref*, where the combined approach yielded a 0.02% larger file size.

pdfsizeopt.py can count the total byte size of various object types in a PDF. Table 4 shows the results on our test PDF files. The percentages in the table cells are savings by running pdfsizeopt.py + Multivalent. Although it is not visible in the table, most of the savings come from Multivalent, except in the *font* and *image* columns, where the contributions of pdfsizeopt.py are important. The large font savings for the document *tuzv* are because the document is short and it contains many Type 1 fonts. For the document *eu2006* we get an even larger saving, because there was lots of glyph duplication across the articles in the collection, and also because LZW was used instead of ZIP to compress the fonts. Only a few of our test documents benefit from image optimization, and even there the contribution of pdfsizeopt.py is small because the original PDF contains the images emitted effectively, and also Multivalent does a decent (though suboptimal) job at image optimization. So for the document *eu2006* Multivalent alone saves about 1.55%, and *pdfsizeopt.py* alone saves 6.14%. (There is no data on the extra size reduction by combining the two tools, because pdfsizeopt.py disables Multivalent's image optimizations since most images won't benefit.) For the document *lme2006* Multivalent alone saves 3.41%, and pdfsizeopy.py alone saves 6.39%. The document *inkscape* benefits most from image recompression: Multivalent alone saves 19.87%, and pdfsizeopy.py alone saves 20.35%.

Columns *psom*, *apsom* and *epsom* of Table 6 show that optimizing with PDF Enhancer or Adobe Acrobat Pro before running the pdfsizeopt.py + Multivalent combination almost always decreases the file size, sometimes by a few percent, but in the case of document *beamer1* the extra gain of running PDF Enhancer first was 6.46%. It seems that for TEX documents PDF Enhancer (with the combination) is the more effective, and Adobe Acrobat Pro is more effective for other documents.

See ideas for improving pdfsizeopt.py in Section 6.

4 Suggested PDF optimization workflow

Based on the optimization tests in Section 3 we suggest the following PDF generation and optimization workflow:

1. Upgrade Ghostscript to at least 8.61, and upgrade to TEX Live 2008.
2. For TEX documents, create the PDF using pdfTEX or dvipdfmx, with the settings discussed in Subsection 2.1. Use dvips + ps2pdf only if absolutely necessary, because of the large PDF files it produces.
3. Use pdftk or Multivalent's PDF merge tool (as shown in [32]) to concatenate PDF files if necessary. Pay attention to the hyperlinks and the document outline after concatenation. Don't concatenate with Ghostscript, because that it would blow up the file size.
4. If you have access to PDF Enhancer, optimize the PDF with it. Otherwise, if you have access to Adobe Acrobat Pro, optimize the PDF with it.
5. Optimize the PDF with pdfsizeopt.py, including the last step of running Multivalent as well.

Most of the optimization steps above can be fully automated and run in batch, except if Adobe Acrobat Pro is involved.

5 Related work

There are several documents discussing PDF optimization. [23] gives a list of ideas how to generate small PDF files. Most of those are present in this work as well. PDF Enhancer and Adobe Acrobat Pro are analyzed in [12], but that article focuses on reducing image resolution and unembedding fonts, which are not information-preserving optimizations. [44] gives a simple introduction to (possibly lossy) PDF image compression and content stream compression.

Since web browsers can display PNG images, several PNG optimization tools [15, 43, 31] have been developed to reduce web page loading times. These tools can be used for optimizing (mainly non-photo) images in PDF documents as well. But since PDF has a more generic image and compression model than PNG, it would be possible to save a little bit more by developing PDF-specific tools, which take advantage of e.g. using the TIFF predictor and ZIP compression together.

An alternative document file format is DjVu [6], whose most important limitation compared to PDF is that it doesn't support vector graphics. Due to the sophisticated image layer separation and compression, the size of a 600 DPI DjVu file is comparable to the corresponding optimized PDF document: if the PDF contains text with embedded vector fonts and vector graphics, the DjVu file can be about 3 times larger than the PDF. If the PDF contains mainly images (such as a sequence of scanned sheets), the DjVu file will become slightly smaller than the PDF. Of course these ratios depend on the software used for encoding as well. There are only a few DjVu encoders available: pdf2djvu and djvudigital are free, and Document Express is a commercial application. PDF is more complex than DjVu: the PDF 1.7 reference [3] itself is 1310 pages long, and it relies on external specifications such as ZIP, JBIG2, G3 fax, JPEG, JPEG2000, Type 1, CFF, TrueType, OpenType, CMap, CID font, XML, OPI, DSA, AES, MD5, SHA-1, PKCS, PANOSE, ICC color profiles, JavaScript and more. PDF 1.7 became an ISO standard [11] in 2008, which adds additional long documents. Having to understand many of these makes PDF viewers hard to implement and complex. This problem can become more severe for long-term archiving if we want to view a PDF 20 or 50 years from now; maybe today's PDF viewers won't work on future architectures, so we have to implement our own viewer. In contrast, the DjVu specification [16] is only 71 pages long, and more self-contained. Since the DjVu file format uses very different technologies than PDF, one can archive both the PDF and the DjVu version of the same document, in case a decent renderer won't be available for one of the formats decades later.

The PDF Database [19] contains more than 500 PDF documents by various producers, with different sizes and versions. These PDF files can be used can be used for testing PDF parsers and optimizers.

Multivalent introduced the custom file format *compact PDF* [25, 27], which is about 30% to 60% smaller than a regular PDF. The disadvantage is that only Multivalent can read or write this format so far (but it supports fast and lossless conversion to regular PDF). Compact PDF achieves the size reduction by grouping similar objects next to each other, and compressing the whole document as one big stream with bzip2, which is superior to ZIP. Another improvement is that compact PDF stores Type 1 fonts unencrypted, with boilerplate such as the 512-byte font tailer and random bytes for encryption stripped out.

6 Conclusion and future work

Since it is not the primary goal for most PDF generators to emit the smallest possible PDF, simple techniques done by Multivalent and pdfsizeopt.py can yield significant size reduction (up to a factor of 3) depending on the generator and the PDF features used. Rearranging the drawing instructions (contents streams and form XObjects, as done by Adobe Acrobat Pro and PDF Enhancer) is a more complicated optimization, and saves some more space in addition to the simple techniques. It also matters how the PDF was generated (e.g. pdfTEX generates a smaller and more optimizable PDF than dvips + ps2pdf).

The workflow proposed in this article has too many dependencies. Python (for pdfsizeopt.py) and Java (for Multivalent) runtimes, and Ghostscript (needed by pdfsizeopt.py for Type 1 and CFF font parsing, CFF generation and arbitrary stream filtering) are the heaviest ones. It is possible to get rid of these by reimplementing pdfsizeopt.py from scratch. To get rid of Python, we could use Lua, and build a statically linked C binary with the Lua interpreter, zlib and all the Lua bytecode linked in. We could reimplement the optimizations done by Multivalent in Lua. (This would include reading and writing object streams and cross-reference streams.) Gradually we could move some functionality to C or C++ code to speed up the optimizer. We could reuse the xpdf codebase to be able to use all PDF filters without invoking Ghostscript. We would have to implement Type 1 and CFF parsing and CFF generation, possibly relying on the dvipdfmx codebase. Other dependencies such as jbig2, sam2p, pngtopnm, PNGOUT and PDF Enhancer are not so problematic, because they can be compiled to small, statically linked, stand-alone executables.

Some optimizations of pdfsizeopt.py could be generalized to cover more cases. Examples are: add CMYK image optimization; make CFF matching more permissive (before unification); recognize more inline images (not only those created by sam2p, and not only in form XObjects). pdfsizeopt.py would also benefit from compiling a test set of PDF files (possibly based on the PDF Database [19]), and adding a framework which automatically checks that pdfsizeopt.py detected the opportunity to optimize, and did the optimization properly in each case.

When preparing a collection (such as a journal volume or a conference proceedings) with TEX, in a typical workflow individual articles are compiled to PDF, and the PDF files are then concatenated. See [32] for tools which can do PDF concatenation. The concatenated document can be optimized using pdfsizeopt.py + Multivalent to get rid of redundancy (such as duplicate glyphs in fonts and duplicate images) across articles. Not all concatenators can preserve hyperlinks and the document outline for TEX documents. Adding concatenation support to pdfsizeopt.py would make creating small and interactive collections more straightforward.

References

[1] Adobe. Adobe Acrobat Pro 9 (project page). http://www.adobe.com/products/acrobatpro/.

[2] Adobe. *The Compact Font Format Specification*, 1.0 edition, 4 December 2003. http://www.adobe.com/devnet/font/pdfs/5176.CFF.pdf.

[3] Adobe. *PDF Reference, Adobe Portable Document Format Version 1.7*. Adobe, 6th edition, November 2006. http://www.adobe.com/devnet/acrobat/pdfs/pdf_reference_1-7.pdf.

[4] Apago. Which features are in what PDF Enhancer edition?, 29 July 2009. http://www.apagoinc.com/prod_feat.php?feat_id=30&feat_disp_order=7&prod_id=2.

[5] codeMantra Universal PDF, a PDF generator. http://codemantra.com/universalpdf.htm.

[6] DjVu: A tutorial. http://www.djvuzone.org/support/tutorial/chapter-intro.html.

[7] DVIPDFMx, an extended DVI-to-PDF translator. http://project.ktug.or.kr/dvipdfmx/.

[8] Till Tantau ed. *The TikZ and PGF Packages*. Institute für Theoretische Informatik, Universität zu Lübeck, 2.00 edition, 20 February 2008. http://www.ctan.org/tex-archive/graphics/pgf/base/doc/generic/pgf/pgfmanual.pdf.

[9] Jonathon Fowler. PNGOUT port for Unix systems, 2007. http://www.jonof.id.au/kenutils.

[10] Gimp, the GNU Image Manipulation Program. http://www.gimp.org/.

[11] ISO 32000-1:2008 Document management—Portable document format—part 1: PDF 1.7, 2008. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51502.

[12] Andy King. Optimize PDF files. http://websiteoptimization.com/speed/tweak/pdf/, 25 September 2006.

[13] Ralf Koening. Creative use of PDF files in LATEX environments. http://www.tu-chemnitz.de/urz/anwendungen/tex/stammtisch/chronik/pdf_interna.pdf, 18 June 2004.

[14] Adam Langley. jbig2enc, a JBIG2 encoder (project page). http://github.com/agl/jbig2enc/tree/master.

[15] List of PNG recompressors. http://en.wikipedia.org/wiki/OptiPNG#See_also.

[16] Lizardtech. *DjVu Reference*, djVu v3 edition, November 2005. http://djvu.org/docs/DjVu3Spec.djvu.

[17] Multivalent, digital documents research and development. http://multivalent.sourceforge.net/.

[18] Multivalent, download location. http://sourceforge.net/projects/multivalent/files/.

[19] PDF database, 20 April 2005. http://www.stillhq.com/pdfdb/db.html.

[20] PDF Enhancer, a PDF converter, concatenator and optimizer. http://www.apagoinc.com/prod_home.php?prod_id=2.

[21] Workarounds for PDF output with the PSTricks LATEX package. http://tug.org/PSTricks/main.cgi?file=pdf/pdfoutput.

[22] PDFCreator, a free tool to create PDF files from nearly any Windows application. http://www.pdfforge.org/products/pdfcreator.

[23] Shlomo Perets. Best practices #1: Reducing the size of your PDFs, 7 August 2001. http://www.planetpdf.com/creative/article.asp?ContentID=6568.

[24] Thomas A. Phelps. Compress, the Multivalent PDF compression tool. http://multivalent.sourceforge.net/Tools/pdf/Compress.html.

[25] Thomas A. Phelps. Compact PDF specification, March 2004. http://multivalent.sourceforge.net/Research/CompactPDF.html.

[26] Thomas A. Phelps and P.B. Watry. A no-compromises architecture for digital document preservation. In *Proceedings of European Conference on Digital Libraries*, September 2005. http://multivalent.sourceforge.net/Research/Live.pdf.

[27] Thomas A. Phelps and Robert Wilensky. Two diet plans for fat PDF. In *Proceedings of ACM Symposium on Document Engineering*, November 2003. http://multivalent.sourceforge.net/Research/TwoDietPlans.pdf.

[28] ps2pdf, a PostScript-to-PDF converter. http://pages.cs.wisc.edu/~ghost/doc/svn/Ps2pdf.htm.

[29] Tomas Rokicki. Dvips: A DVI-to-PostScript translator, January 2007. http://mirror.ctan.

org/info/doc-k/dvips.pdf.

[30] Ken Silverman. KZIP, a PKZIP-compatible compressor focusing on space over speed. http://advsys.net/ken/utils.htm#kzip.

[31] Ken Silverman. PNGOUT, a lossless PNG size optimizer, 2009. http://advsys.net/ken/utils.htm#pngout.

[32] Matthew Skala. How to concatenate PDFs without pain, 13 May 2008. http://ansuz.sooke.bc.ca/software/pdf-append.php.

[33] Matthias Stirner and Gerhard Seelmann. packJPG, a lossless compressor for JPEG images (project page), 21 November 2007. http://www.elektronik.htw-aalen.de/packjpg/.

[34] Péter Szabó. Extra files related to PDF generation and PDF size optimization. http://code.google.com/p/pdfsizeopt/source/browse/#svn/trunk/extra.

[35] Péter Szabó. Installation instructions for pdfsizeopt.py. http://code.google.com/p/pdfsizeopt/wiki/InstallationInstructions.

[36] Péter Szabó. PDF test files for pdfsizeopt.py. http://code.google.com/p/pdfsizeopt/wiki/ExamplePDFsToOptimize.

[37] Péter Szabó. pdfsizeopt.py, a PDF file size optimizer (project page). http://code.google.com/p/pdfsizeopt.

[38] Péter Szabó. sam2p, a pixel image converter which can generate small PostScript and PDF. http://www.inf.bme.hu/~pts/sam2p/.

[39] Péter Szabó. Inserting figures into TEX documents. In *EuroBachoTEX*, 2002. http://www.inf.bme.hu/~pts/sam2p/sam2p_article.pdf.

[40] Till Tantau. *The beamer class*, 3.07 edition, 11 March 2007. http://www.ctan.org/tex-archive/macros/latex/contrib/beamer/doc/beameruserguide.pdf.

[41] Hàn Thế Thành, Sebastian Rahtz, Hans Hagen, et al. *The pdfTEX manual*, 1.671 edition, 25 January 2007. http://www.ctan.org/get/systems/pdftex/pdftex-a.pdf.

[42] Cosmin Truţa. A guide to PNG optimization. http://optipng.sourceforge.net/pngtech/optipng.html, 2008.

[43] Cosmin Truţa. OptiPNG, advanced PNG optimizer (project page), 9 June 2009. http://optipng.sourceforge.net/.

[44] VeryPDF.com. Compressing your PDF files, 13 July 2006. http://www.verypdf.com/pdfinfoeditor/compression.htm.

[45] VeryPDF.com. *PDF Compress Command Line User Manual*, 13 July 2006. http://www.verypdf.com/pdfinfoeditor/pdfcompress.htm.

[46] Pauli Virtanen. TexText, an Inkscape extension for adding LATEX markup, 6 February 2009. http://www.elisanet.fi/ptvirtan/software/textext/.

Péter Szabó
Google
Brandschenkestrasse 110
CH-8002, Zürich, Switzerland
pts (at) google dot com
http://www.inf.bme.hu/~pts/