JBIG2 Compression of Monochrome Images with OCR

Diploma thesis

Bc. Radim Hatlapatka

Brno, 2012
Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Radim Hatlapatka

Advisor: doc. RNDr. Petr Sojka, Ph.D.
Acknowledgement

I would like to thank my supervisor doc. RNDr. Petr Sojka, Ph.D. for his guidance and help by providing references to publications relevant or similar to topics discussed in the diploma thesis. I also would like to thank my friends Ján Vorčák and Lukáš Lacko for reading this thesis and helping me with their opinions. Many thanks belong to Tomáš Márton for giving me quick guidance about parallelization in C++. 
Abstract

The aim of the diploma thesis is to design and implement a solution for improving compression ratio of an open-source jbig2enc encoder. The improvement is achieved with created support for using an OCR engine. In order to present created solution, relevant tools working with JBIG2 standard and OCR tools are introduced. Jbig2enc encoder enhanced by using Tesseract OCR engine, in order to get text recognition results, is introduced. The new version of jbig2enc is evaluated on data from digital libraries together with description of its integration into two such libraries: DML-CZ and EuDML.
Keywords

OCR, image preprocessing, JBIG2, compression, compression ratio, scanned image, DML, speed improvement, bitonal images, Tesseract, Leptonica, jbig2enc, DML-CZ, EuDML.
# Contents

1 Introduction ......................................................... 3

2 JBIG2 and Known Tools ........................................... 5
   2.1 JBIG2 Principles .............................................. 5
   2.2 Jbig2dec .......................................................... 6
   2.3 Jbig2enc ............................................................ 7
      2.3.1 Jbig2enc Improvement .................................... 7
   2.4 Pdfl2Im ............................................................ 9
   2.5 JPedal JBIG2 Image Decoder .................................. 9
   2.6 Jbig2-imageio ..................................................... 10
   2.7 PdfCompressor ................................................... 11
   2.8 Summary of Tools .............................................. 11

3 OCR Tools .............................................................. 14
   3.1 ABBYY FineReader .............................................. 14
   3.2 InftyReader ....................................................... 15
   3.3 PrimeOCR ........................................................ 16
   3.4 Tesseract OCR ................................................... 16
   3.5 GOCR ............................................................... 17
   3.6 OCRopus .......................................................... 17
   3.7 Summary of OCR Tools ........................................ 18

4 Enhancement of Jbig2enc Using an OCR .......................... 20
   4.1 Performance Issues and Its Solutions ...................... 20
      4.1.1 Hash Functions ........................................... 21
      4.1.2 OCR Recognition Run in Parallel and Its Limi-
      tations ............................................................ 23
      4.1.3 Summary of Achieved Speed Improvement .......... 24
   4.2 Interface for Calling OCR ..................................... 26
   4.3 Similarity Function ............................................ 28
   4.4 Using Tesseract as OCR Engine .............................. 30
   4.5 Jbig2enc Workflow Using Created OCR API .............. 31

5 Evaluation ............................................................. 33
   5.1 Comparison with Previous Version of Improved Jbig2enc 33
   5.2 Evaluation on Data from EuDML .............................. 36

6 Usage in DML ......................................................... 39
   6.1 DML-CZ ............................................................ 39
   6.2 EuDML .............................................................. 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conclusion</td>
<td>44</td>
</tr>
<tr>
<td>A CD Content</td>
<td>51</td>
</tr>
<tr>
<td>B Manual for Running Jbig2enc Improvement</td>
<td>52</td>
</tr>
<tr>
<td>B.1 Jbig2enc Command-line Arguments Enabling Created Improvements</td>
<td>52</td>
</tr>
</tbody>
</table>
1 Introduction

In the world, more and more information is available in an electronic version and they need to be stored. There are two possibilities for solving the problem of acquiring enough storage space in order to store all the data: either by acquiring additional storage space or by compressing the data.

Compression does not have to be used only for storing purposes. It can be used for decreasing a bandwidth needed to transmit the data or to decrease time to access and load the data from a disk to a memory. Operations on a processor are processed in a matter of nanoseconds, on the other hand, time to access the data on the disk takes milliseconds. Thereby, if a decompression process is faster than a difference between access time to the compressed and the original document, time to access and load the document is reduced.

Digital libraries are good example of collections providing large volumes of data. A well designed digital library (DL) needs to tackle of scalability, persistence, transfer size and speed, linking related data, format migration, etc. Transfer size and speed can be greatly improved using good compression mechanisms.

There exists a vast amount of compression methods where each of them has advantages and disadvantages. Most of them are better for usage at a specific type of data than others. Different compression methods are used for compressing images, videos, text, etc.

JBIG2 [1,2] is a standard for compressing bitonal images. These are images consisting of two colors only. Most of scanned documents are composed of such images. JBIG2 compression method achieves great compression results in both lossless and lossy modes. We work with special kind of lossy compression called perceptually lossless compression. It is a lossy compression method which creates output without any visible loss.

It uses several basic principles in order to improve compression ratio. Specialized compression methods are used for different types of regions. A different method is used for images than for a text. By recognizing the concrete type of data and using a specific compression mechanism, greater compression ratio is achieved.

We focus on a method for compressing text regions. This method
is based on recognizing components which are repeated throughout the image or even throughout more images (pages), and stores them only once. For each different component, a representant is created in a dictionary and all occurrences are just pointers to the dictionary with some additional metadata.

The image often contains flyspecks and dirts. This makes two visually equivalent components (symbols) look different for a computer. We try to detect such symbols and recognize them as equivalent even if they are not the same in each pixel, and thus improving image quality and decreasing storage size of the image. In an ideal case the number of different symbols would be equivalent to a number of different symbols in a born digital document.

In a bachelor thesis JBIG2 Compression [3], a method partially solving the issue was created. In this thesis, a new method is created. The method uses OCR to further improve compression ratio. It takes benefit of OCR text recognition in order to find equivalent symbols between already recognized representative symbols. In order to improve processing speed in comparison with older version created as part of [3], a hash function is created together with other performance improvements.

First, a JBIG2 standard is introduced together with tools working with the standard (see Section 2 on the following page).

In the next part, OCR tools relevant to a process of improving JBIG2 compression are described (see Section 3 on page 14).

Description of a created hash function and other speed improvements are introduced together with their evaluation in Section 4.1 on page 20.

Finally, a created API of the jbig2enc encoder for using an OCR engine is described. It is followed with a Tesseract OCR implementation of the API (see Section 4.2 on page 26).

In the last part, the created tool is evaluated on data from two digital mathematical libraries (DML-CZ and EuDML) together with description of the workflow for their integration (see Sections 5 on page 33 and 6 on page 39).
2 JBIG2 and Known Tools

JBIG2 is a standard for compressing bitonal images developed by Joint Bi-level Image Experts Group. Bitonal images are images consisting only of two colors (usually black and white). The typical area where such images occurs is a scanned text. JBIG2 was published in 2000 as an international standard ITU T.88 [2] and one year later as ISO/IEC 14492 [1].

In Section 2.1, a standard JBIG2 and its basic principles are introduced. Different tools working with a standard JBIG2 are described in the following sections. There are described both open-source and commercial tools.

2.1 JBIG2 Principles

JBIG2 typically generates files three to five times smaller than Fax Group 4 and two to four times smaller than JBIG1 (the previous standard released by Joint Bi-level Image Experts Group).

Standard JBIG2 also supports lossy compression. It increases compression ratio several times without a noticeable difference compared with lossless mode. Lossy compression without visible loss of data is called a perceptually lossless coding. A scanned text often contains flyspecks (tiny pieces of dirt) – perceptually lossless coding helps to get rid of the flyspecks and thus increase the quality of the output image.

The content of each page is segmented into several regions. There are basically three types: text regions, halftone regions and generic regions. The text regions contain text, halftone regions contain halftone images[1] and generic regions contain the rest. In some situations, better results can be obtained if text regions are classified as generic ones and vice versa.

While compressing a text region, a representative is chosen for each new symbol. If the same symbol appears more than once, a representant is chosen instead of storing each symbol occurrence with

all the data. Each occurrence of the symbol points to its representant with memorizing information about its position in the document.

JBIG2 uses modified versions of adaptive Arithmetic and Huffman coding. Huffman coding is used mostly by faxes because of its lower computation demands. But Arithmetic coding gives slightly better results.

JBIG2 supports a multi-page compression used by symbol coding (coding of text regions). Any symbol that is frequently used on more than one page is stored in a global dictionary. Such symbols are stored only once per several pages and thus reducing space needed for storing the document even further. For more information, see thesis JBIG2 Compression [4].

Support for JBIG2Decode filter has been embedded into a PDF since PDF version 1.4 (2001, Acrobat 5, see 3rd edition of the PDF Reference book [5, pages 80–84]). This allows storing compressed images inside PDF according to the standard JBIG2. This allows to spread far and wide the JBIG2 standard without placing any burden on the end users. Users are not forced to install any specific decoder to read PDFs containing JBIG2 encoded images. In the worst case, user would need just to upgrade its PDF reader to a version fully supporting PDF version 1.4 or newer.

When JBIG2 images are stored in the PDF, headers and some other data are discarded. Discarded information is instead stored in a PDF dictionary associated with the image object stream. PDF dictionary is a specific PDF object for holding metadata. It is in format of an associative table containing pairs of objects (key and value). For more information see [5].

2.2 Jbig2dec

Jbig2dec [6] is an open-source decoder for a JBIG2 image format which is developed by ghostscript developers. It can be redistributed or modified under the terms of GNU General Public License version 2 or newer[2]. In spite of this not being a complete implementation, it is maintained to work with available encoders and thus it is able to decode most of the documents that are widely available.

2.3 Jbig2enc

Jbig2enc [7, 8] is an open-source encoder written in C/C++ by Adam Langley with support of Google. It is developed under Apache License, Version 2.0.

Jbig2enc encoder uses an open-source library Leptonica [9] which is being developed by Dan Bloomberg and it is published under a Creative Commons Attribution 3.0 United States License [4]. The Leptonica library is used for manipulating images. For example, it handles page segmentation to regions containing text, images, and other data, segmentation to separate symbols (connected components), logical operations at binary level, skewing or rotating an image. The Leptonica library is used by other programs such as Tesseract OCR engine (for more information see Section 3.4) or jbig2enc encoder.

Halftone coding is not supported in jbig2enc encoder. Instead, jbig2enc encoder uses a generic coding for the halftone images. It supports creating an output in a format suitable for putting into a PDF document. This feature (support) is very useful for tools working with optimized PDF documents that uses the standard JBIG2 for achieving better results.

According to a JBIG2 standard either Huffman coding or Arithmetic coding can be used for symbol coding, but the jbig2enc encoder supports only the Arithmetic coding.

Jbig2enc encoder is able to create an output which can be easily put into a PDF document. It creates one file per each image plus one file corresponding to a global symbol dictionary. They correspond exactly to the PDF image objects and the global dictionary object that are directly put into the PDF document. Thereby, when putting an image into the PDF encoded with the jbig2enc encoder, it is only necessary to correctly fill a PDF dictionary with metadata about the image, mainly its dimensions.

2.3.1 Jbig2enc Improvement

As part of the thesis [3], an additional method improving jbig2enc encoder compression ratio was created. Image is segmented into
components where every component mostly corresponds exactly to one symbol. While identifying symbols, a representative symbol is chosen for each new symbol. Every occurrence of the symbol is then identified just using a pointer to its representant with its relative position to a previous symbol.

Since images contain noise, not all visually equivalent symbols are recognized as equivalent, and they are stored as different symbols. It is shown in Figure 2.1. The ideal case would be to have the number of representants as close as possible to the number of different symbols in a born-digital text. The purpose of jbig2enc improvement is to make us closer to this goal.

To reduce the number of representative symbols and thus to improve compression ratio, an additional comparison process was created. It improves compression ratio by additional 10% [3].

**Figure 2.1: Example of two originally different symbols recognized as equivalent**

The improvement was achieved using an additional comparison method between representants with the same size. The method looks for differences in shapes of lines (horizontal, vertical, diagonal) and points. If such a difference is not found, the representants are considered equivalent. This means all instances pointing to these representants are transferred to point just to one of them, and thus reducing storage requirements.

Quality of output image is influenced by a quality of the chosen symbol representants. The representants quality influences how each of its occurrences will look like. If a representant with a worse quality is chosen, a quality of an output image is decreased. In opposite, if a representant with a better quality is chosen, the output is improved. Jbig2enc improvement chooses first of two compared representative
symbols (which were found equivalent). It has more or less the same effect as choosing a random one (image quality remains mostly the same).

2.4 PdfJbIm

PdfJbIm [10] is a PDF enhancer which optimizes size of bitonal images inside PDF documents. To optimize size of PDF documents, it uses benefits of a standard JBIG2 and of an open-source encoder jbig2enc (see Section 2.3).

PdfJbIm expects a PDF document containing images with a scanned text on an input. These images are rendered from the PDF document. They are encoded using an jbig2enc encoder with enable symbol coding for the text areas. It uses perceptually lossless (visually lossless) compression of the jbig2enc encoder. This is the most suitable coding for this kind of data. Flyspecks appear in all scanned texts and thereby two visually equivalent symbols are not the same in each pixel. If perceptually lossless compression is used, a great improvement in a quality of a compressed image and in a compression ratio can be achieved.

Figure 2.2 shows basic steps of pdfJbIm during a process of optimizing PDF documents with the jbig2enc encoder.

In bachelor thesis [3], the tool pdfJbIm is called PDF re-compressor. Since then, several improvements and bug fixes were added. The main improvement is an added support for multi-layer PDF documents. There are also added options to make the workflow more customizable. It is enhanced with a support for running a new version of the jbig2enc encoder. The new version of the encoder is developed as part of this thesis and adds a support for using an OCR in the process of image compression (see Section 4).

2.5 JPedal JBIG2 Image Decoder

JPedal [11] is a Java PDF library which offers a JBIG2 decoder library. Version 1 is released under a BSD License. On the other hand, version 2 is available under commercial license. The newer version is
improved and mainly its speed is enhanced in comparison with the older version.

The JPedal JBIG2 Image Decoder can be used as a plugin for the ImageIO framework which is a standard API for working with images in Java. Thus for developers used to work with ImageIO framework, it is very easy to decode JBIG2 images using this decoder.

### 2.6 Jbig2-imageio

The Jbig2-imageio [12] is a plugin enabling access to images encoded according to JBIG2 standard. It is a pure Java implementation which does not require use of JNI (Java Native Interface) and is being developed under GNU GPLv3 with support of Levigo.

This plugin can be used by Java ImageIO API, which is part of

---

5. [GNU General Public License V3](https://www.gnu.org/licenses/gpl-3.0.en.html)
a standard Java API for manipulating with images. Since it uses Java ImageIO API, user does not need to make any changes to the code. User needs only to add this plugin as dependency. If he tries to decode an JBIG2 image, its type is automatically recognized and decoder is automatically used.

2.7 PdfCompressor

PdfCompressor \cite{13} is a commercial tool developed by CVISION Technologies, Inc. It makes PDF documents fully searchable using very fast and accurate OCR and highly optimized using modern compression methods. It allows user to choose between accuracy and speed. Higher accuracy is achieved mostly by running additional methods for further recognition of problematic parts of the document.

For compression of images in a PDF document, it uses benefits of JBIG2 and JPEG2000 image compression formats. For bitonal images, it uses JBIG2. For coloured images it uses JPEG2000. Black and white scans are usually compressed by a factor $5–10 \times$ compared to TIFF G4 and coloured scans by a factor $10–100 \times$ compared to JPEG \cite{13}.

High speed for compression and OCR of PDF is achieved because of high optimization for multi-threading. Thereby, it is able to process several documents in matter of a few seconds.

PdfCompressor does not only support compression and OCRing, it also supports encryption of PDFs, web-optimization bates-stamping and other features.

Demo version of this tool is available for download at the project’s home page. Online version of PdfCompressor is available at the same page for users to try it out.

2.8 Summary of Tools

Table \ref{tab:2.1} shows main differences between tools working with standard JBIG2. Last column shows their main features and issues. Second column shows a tool license.

Most of tools described are decoders. Developing an encoder is relatively easy and there exists a variety of them. In contradiction, there exists only a few encoders. PdfCompressor is developed by CVISION
Table 2.1: Summary of tools working with standard JBIG2

<table>
<thead>
<tr>
<th>Tool name</th>
<th>License</th>
<th>Global dictionary support</th>
<th>Additional information (advantages/disadvantages)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jbig2dec</td>
<td>GNU General Public License</td>
<td>Yes</td>
<td>Development progress very slow, but works for commonly available JBIG2 image formats</td>
</tr>
<tr>
<td>Jbig2enc</td>
<td>Apache License, Version 2.0</td>
<td>Yes</td>
<td>PDF output support; uses Leptonica library</td>
</tr>
<tr>
<td>PdfjbIm</td>
<td>Apache License, Version 2.0</td>
<td>Yes</td>
<td>Optimizes PDF documents; uses jbig2enc</td>
</tr>
<tr>
<td>JPedal JBIG2 Image Decoder</td>
<td>Commercial (since version 2); BSD License (older versions)</td>
<td>No</td>
<td>Since version 2 commercial; pluggable for ImageIO framework</td>
</tr>
<tr>
<td>Jbig2-imageio</td>
<td>GNU GPLv3</td>
<td>No</td>
<td>Used as plugin by Java ImageIO API</td>
</tr>
<tr>
<td>PdfCompressor</td>
<td>Commercial</td>
<td>Yes</td>
<td>Very powerful and fast; only for MS Windows</td>
</tr>
</tbody>
</table>

Technologies which stood at the beginning of the JBIG2 standard release. It provided a great advantage and made PdfCompressor the dominant JBIG2 encoder providing best compression results done using a JBIG2. On the other hand, several years later an open-source encoder jbig2enc was developed. It created an opportunity for other developers working with the JBIG2 standard and to spread images compressed according the standard more rapidly.

In contradiction there exist a large variety. In all PDF readers supporting PDF version 1.4 or newer, some kind of a JBIG2 decoder needs to be implemented.

There is several decoders developed in Java. They implement Java ImageIO API. It provides a uniform access to images for the Java developer. This tools mostly does not support a global dictionary. They are able to process only JBIG2 images stored in one file. This tools mainly are created for a Java developers.

In opposite, jbig2dec decoder is a command line utility support-
2. JBIG2 AND KNOWN TOOLS

Handling large variety of JBIG2 formats. It handles JBIG2 encoded images containing a global dictionary and also JBIG2 images stored inside a PDF document.

In the next chapter (see Chapter 3) we present several OCR tools and engines. We focus on tools that are specific and could be considered for integrating to JBIG2 encoder as a possibility to improve its compression ratio.
3 OCR Tools

As we already know, JBIG2 compression of a text region is based on segmenting an image into tokens (symbols). For each different token a representant is chosen. The main problem is that scanned text contains noise and flyspecks. This makes visually equivalent symbols to seem different for the computer even though they look the same for a human eye. The problem is similar to the problem of OCR (optical character recognition) where each symbol is recognized to be readable by a computer.

The process of recognizing symbols by OCR engine is very similar to the process of identifying representative symbols occurring repeatedly on the image. OCR engine has disadvantage to JBIG2 encoder – for each symbol on the image, it needs to decide what is its computer representation even if it is uncertain. In contradiction, JBIG2 encoder can choose if the symbol points to an existing representant or a new representant should be created.

OCR (Optical Character Recognition) is a technology that enables machine to translate images into a text format which is easily searchable and editable [14].

There exists extreme amount of different OCR engines and OCR software but in this chapter we introduce only several of them. We focus mainly on OCR engines, which are interesting or special in some way or are widely used. There exist a lot of so called converters. They contain the OCR engine as part of themselves. As an example, we mention PdfCompressor introduced in Section 2.7. It just converts PDF document into a searchable one. Description of such documents is out of scope of this thesis and they are not described here any further.

3.1 ABBYY FineReader

ABBYY FineReader [15] is a commercial OCR software for creating editable and searchable documents. ABBYY offers two main editions. Professional, which is suitable for individual users, and Corporate, which is more suitable for usage in enterprise environment. The main difference is an extra support for batch processing in Corporate Edi-
tion. ABBYY FineReader is developed only for MS Windows and Mac.

ABBYY also offers an SDK version of FineReader which allows to integrate ABBYY’s state-of-the-art of software technologies for document recognition and conversion. An SDK version of FineReader is used as part of tools used for digitalization of mathematical texts inside DML-CZ [16, 17].

There is a very interesting ABBYY’s Mobile OCR Engine SDK [18]. It allows developers of mobile applications to integrate highly accurate OCR technologies to their mobile applications. As an example, we mention applications which capture words and translate them on-the-go.

Current version ABBYY FineReader 11 supports more than 189 languages for OCR and 113 languages for ICR. It contains special features for pattern training and creation of user specific languages. It greatly improves processing speed in Comparison with older version FineReader 10. According to [15] FineReader converts documents to their searchable and editable version with up to 99.8% accuracy.

FineReader does not only recognize texts, but also recognizes document layout which it is able to retain. This means the output document not only contains recognized text but also in the same or similar font and formatting including images, tables, charts and more.

FineReader supports wide range of output formats. As an example we can mention plain text, rtf, html, doc/docx, odt, xls/xlsx, pptx, DjVu, PDF or EPUB.

3.2 InftyReader

InftyReader [19] is a commercial OCR software for recognizing scientific documents including math formulae. It creates output in various file formats including \LaTeX, MathML, XHTML, IML, HRTeX. It is developed in the laboratory of M. Suzuki, Faculty of Mathematics,

---

1. Intelligent character recognition is an advanced OCR used for recognition of a handwritten text
2. HRTeX (Human Readable \LaTeX) – a simplified LaTeX-like notation which is easier “to read” specially designed for the blinds. See [http://www.access2science.com/braille/chezdom.net.htm](http://www.access2science.com/braille/chezdom.net.htm)
Kyushu University, in collaboration with several cooperation partners.

They offer a trial version which can be used freely for 15 days. In InftyReader version 2.9 is now available plugin for using FineReader OCR engine which can be used for recognition of ordinary texts of the documents. In order to use it inside InftyReader, user needs to purchase a special license of FineReader.

InftyReader detects blocks containing mathematical formulae and uses structural analysis for math formulas. While recognizing mathematical formulas, network representation is used to represent mathematical expression. In order to choose suitable OCR result from this network representation, modification of minimal spanning tree is used. For more information see [20].

3.3 PrimeOCR

PrimeOCR [21] is a commercial Windows OCR engine which reduces OCR error rates by implementing “Voting” OCR technology. They claim reduction of errors made by standard OCR engine by up to 65–80%. PrimeOCR allows to use more OCR engines. If a low image quality is detected, more than one OCR engine is used in order to improve results of the OCR.

It is not necessary to identify language of the document in advance. PrimeOCR is able to identify it automatically. Processing is significantly faster if the language is specified in advance. PrimeOCR recognizes one dominant language per page but it allows to recognize secondary language for which English is usually used.

PrimeOCR is created mainly to be used in production systems than by individual users. Their licensing prices correspond to it. For developers they offer SDK with an orthogonal API accessible via a DLL library. On a product page a detailed usage manual [22] is available together with several usage examples.

3.4 Tesseract OCR

Tesseract is an open-source OCR engine written in C/C++, originally developed as proprietary software at Hewlett-Packard between
In the following decade, most of its development was stopped. In 2005 it was released as open-source by Hewlett-Packard and UNLV\(^3\). Since 2006 Tesseract is sponsored by Google and is released under the Apache License, Version 2.0. In 2005 Tesseract was evaluated by UNLV as one of top three OCR engines in terms of character accuracy [23].

It uses an open-source library Leptonica for manipulating with images. Images are internally stored using Leptonica structures. Tesseract by itself does not come with any graphical interface and is run from a command-line. But there was created several projects which provide GUI for Tesseract. Here we show several of them.

- **FreeOCR** – a Windows Tesseract GUI.
- **GImageReader** – GTK GUI frontend for Tesseract that supports selection columns and parts of document. It allows to process PDF documents and even spell check the output.
- **OcrGUI** – a Linux GUI written in C using GLib and GTK+, supports Tesseract and GOCR. It includes spell checking using an open source spell checker Hunspell [24, 25].

### 3.5 GOCR

GOCR [26] is an OCR program developed under GNU Public License. GOCR is also known under JOCR name which was created while registering GOCR at Sourceforge, where the name gocr was already taken and new name needed to be created.

It can be used with different front-ends and thus it makes it very easy to port to different OSes and architectures. It can open many image formats and its quality is being improved on a daily basis [26].

### 3.6 OCRopus

OCRopus [27] is a free document analysis and optical character recognition system being developed in Python and released under Apache
3. OCR TOOLS

License, Version 2.0. Design of this project is made to allow usage of plugins. OCRopus is currently developed under the lead Thomas Breuel from the German Research Centre for Artificial Intelligence and is sponsored by Google.

It is based on two research projects: a high-performance handwriting recognition developed in the mid-90’s and novel high-performance layout analysis methods.

OCRopus is now divided to several well-defined native code modules and high-level Python code called ocropy. The low-level modules are mostly written in C++. To use OCRopus in another application user only needs to include ocropy, iulib [28] for manipulating with images and its main component colib [28] for basic data structures and algorithms.

OCRopus supports recognition of ligatures and contains tools for clustering and correcting character shapes. It supports training at very large datasets which consists even of millions of samples.

It supports not only a plain-text as the output format, but the hOCR output format as well. HOCR is a (X)HTML compatible OCR output format.

3.7 Summary of OCR Tools

There exists many OCR recognition tools. Many of them are specific and trained only on specific type of data. We have introduced tools with more general usage.

In Table 3.1 is shown summary of tools described in Chapter 3 with focus on their specific behaviour and properties. There is also shown the License under which they are published and being distributed.

A lot of OCR tools are developed under a commercial license and only few of them are developed under an open-source license. InftyReader is the only OCR engine supporting structural analysis in order to recognize math formulas. The FineReader belongs to one of most accurate OCR engines used.

Accuracy of the OCR engine is greatly influenced by a type of the data being processed. In order to achieve maximum performance and to minimize error rate, the OCR engine needs to be trained on them.

In order to maintain a usability of a jbig2enc encoder, we focus
3. OCR TOOLS

Table 3.1: Summary of OCR tools

<table>
<thead>
<tr>
<th>Tool name</th>
<th>License</th>
<th>Math support</th>
<th>Additional informations (advantages/disadvantages)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABBYY FineReader</td>
<td>Commercial</td>
<td>No</td>
<td>Retains formatting including tables and charts</td>
</tr>
<tr>
<td>InftyReader</td>
<td>Commercial</td>
<td>Yes</td>
<td>Support for additional OCR engines put as plugins (e.g. plugin for FineReader)</td>
</tr>
<tr>
<td>PrimeOCR</td>
<td>Commercial</td>
<td>No</td>
<td>Implements &quot;Voting&quot; technology which greatly reduces error rate</td>
</tr>
<tr>
<td>Tesseract</td>
<td>Apache License, Version 2.0</td>
<td>No</td>
<td>One of best character recognition rates among free OCR engines</td>
</tr>
<tr>
<td>GOCR</td>
<td>GNU Public License</td>
<td>No</td>
<td>Portable to different front-ends; Usable on different architectures and OSes</td>
</tr>
<tr>
<td>OCRopus</td>
<td>Apache License, Version 2.0</td>
<td>No</td>
<td>Document analysis and OCR written in Python</td>
</tr>
</tbody>
</table>

On an open-source solutions. Other OCR tools are mentioned in this chapter as a possible OCR engines used on specific collections. For example, for a collection containing mathematical texts such as digital mathematical libraries, an InftyReader would enable to achieve greater compression results than other OCR engines.

Creating module for an commercial OCR engine is out of scope of this thesis, but OCR API designed in Section 4.2 on page 26 should be prepared even for this case.
4 Enhancement of Jbig2enc Using an OCR

Main idea behind JBIG2 and its processing of text regions is segmentation of text regions to components where each component mostly corresponds exactly to one symbol. This is very similar to OCR processing which segments image into words and symbols and then recognizes them.

Our goal is to improve the jbig2enc encoder using a good OCR engine. It should not put any constraints on users and developers using an actual version of the jbig2enc encoder. Thereby, as default OCR engine we require an open-source product licensed under a license compatible with jbig2enc encoder and its API for using OCR. It should be straightforward for implementation of the designed API.

Tesseract is developed under an Apache License 2.0 compatible for purposes of enhancing jbig2enc encoder. It supports wide range of languages. As a bonus, it uses the same library for holding image data structures as the jbig2enc encoder. This makes it an ideal candidate for integrating it with the encoder.

We create an API for using OCR engine as much as possible independent on a concrete OCR engine. It is relatively easy replaceable but little modifications of code are still required.

Before description of the created API, we tackle performance issues and provide its solutions.

4.1 Performance Issues and Its Solutions

The version of jbig2enc improvement developed as part of [3] does not contain any hash function. All representants are compared with each other in order to find out if they are visually equivalent, or not.

In order to prevent running most computationally expensive part of a comparison method, at start of the comparison method, dimensions of representants and total amount of different pixels are checked and compared. If sizes of the representant bitmaps are different or total amount of different pixels is extremely high, method automatically marks the representants as different without necessity of further processing. This checking at least partly prevents unnecessary computations in a matter of distinguishing most different representants.
There is another limitation that increases the computation time. When two images are compared and they are found equivalent, they are immediately unified. It means if there is found another equivalent representant to already unified representant it needs to be reindexed again causing multiple reindexing instead running it only once per all equivalent symbols.

With the reindexing issue, another problem is connected. If two representants are being unified, one of them is removed. To maintain array compactness it is necessary to move a representant to fill the place of removed one (used last one to prevent unnecessary reindexing computations). Moving representant in array requires changing pointers of all its instances to its new position.

If OCR recognition is added, which is an expensive operation, the process of finding equivalent symbols would become extremely slow.

In order to prevent this high increase in compression time, we have designed and implemented new hash function and decreased computation times needed for reindexing and unification of equivalent symbols. We have added features for running OCR recognition in parallel if it is allowed during compilation. In Section 4.1.1 we describe details about hash functions used and how it influences speed performance. In Section 4.1.2 we show how OCR text recognition can be done in parallel and what limitations it brings.

### 4.1.1 Hash Functions

There are significant differences between using or not using the OCR. Thus we have designed and implemented two different hash functions where basically hash function using OCR is an extension of the standard hash function. To be more precise, hash function using OCR is not only a hash function but it also maintains results retrieved from symbol recognition of the OCR engine used. It is to prevent multiple runs of OCR recognition per one symbol (representant).

Hash function is counted from sizes of the representant (symbol) and number of holes found in the symbol. For example symbol 'B' has three holes (two inside the symbol and one represented by outer

---

1. Reindexing means transferring pointers from one representant to another one
zone around the symbol). Number of holes is retrieved using method for counting connected components.

\[
hash = \text{holes} + 10 \times \text{height} + 10000 \times \text{width}
\]

Figure 4.1: Formula for counting hash without OCR results

With the formula represented in Figure 4.1 information about number of holes, height and width of the representant are easily retrieved back if needed. The hash value is stored by allocating for every provided information several digits and thereby allowing to retrieve the information directly back from the hash value.

The digits in decimal representation are allocated as follows: last digit holds a number of holes, next three digits ,from the end, holds a height of the symbol representant and the rest is allocated for storing a its width. Main benefit of this approach is that each of attributes for counting hash is remembered in its result and it is possible to use this information to check even symbols with similar sizes.

Representants are stored in linked lists where in each such list only representants with the same hash value are stored. In C/C++ code, they are stored in map<hash,list<representantIdentificator> where hash is a counted hash value as shown in Figure 4.1 and list of representantIndicators are indexes of array holding all representants.

Hash function, which is used if OCR is enabled, uses two layer hash representation. The outer key is represented as integer number counted from recognized text as sum of integer values of individual characters (for standard letters these are ASCII values of chars). The inner key is represented by simplified function in compare to hash function that does not include OCR results. This hash does not contain information about holes. It becomes irrelevant because this information is already contained in the outer key as part of recognized text by OCR. More often, comparisons of symbols with slightly different sizes are expected. Thereby, the inner hash key value is counted by multiplying width and hight of the representative symbol.

Using hash function does not significantly improve speed performance. It is because older version of jbig2enc improvement already was checking if further more expensive computation is needed or not.
At least, number of holes recognized in a symbol decreases amount of comparisons made. It is because, this attribute was not checked in older version of a jbig2enc encoder improvement, but it is now part of the counted hash value.

Hash function allows another approach of counting whether two symbols are equivalent or not. Hash function puts together symbols that have a chance to be equivalent. Thereby, it prevents comparing irrelevant combinations of symbols.

Methods are modified to benefit from this approach. Only symbols with the same hash value are being compared and if two symbols are found equivalent, it is remembered instead of immediate unification. In the process, symbols are only compared with one of the symbols marked as equivalent. It limits unnecessary comparisons too. When all symbols with same hash value are processed, reindexing and unification is done in one go.

As part of a method counting hash that uses OCR, a method for text recognition is called. The text recognition is an expensive operation. In order to prevent multiple runs of the text recognition per each symbol, results are stored and provided for further usage.

The initialization of the OCR engine also is an expensive operation. It can be limited to a number of OCR API used. This is usually one, but if text recognition is run in parallel, new instance of the API is created for each thread in order to prevent collisions inside OCR engine.

The initialized OCR engine API can either be provided as parameter of a method for counting a hash, or it can be initialized directly in a method counting hash values for whole collection. In the second case, structure holding results of the OCR text recognition needs to be returned together with a structure holding data sorted according to counted hash function. We use the second approach. It makes the process of running OCR recognition in parallel more straightforward. For details see Section 4.1.2.

### 4.1.2 OCR Recognition Run in Parallel and Its Limitations

Running OCR recognition is an expensive operation. It needs to be run at least once per each representant (symbol). Because the recognitions of different representants does not depend on each other, they can be run in parallel.
4. ENHANCEMENT OF JBIG2ENC USING AN OCR

To run OCR recognition in parallel, it is required that each instance of the OCR API is thread safe. Tesseract OCR, the chosen basic OCR engine for testing purposes, has its API thread safe for most of provided methods. In our case, only the thread safe methods are used, thus running Tesseract OCR recognition in parallel is safe.

Not all OCR engines have API thread safe. It needs to be possible to disable the parallel run if required.

To implement support for running OCR recognition in parallel, we use an OpenMP library and its API [29, 30]. It allows to mark parts of a code which shall be run in parallel. It allows to create restrictions and locks in order to prevent an unwanted behaviour. It is defined using a declarative in code. In C++, there is added the declarative 

```c++
#pragma omp
```

which does not directly change the code. The OpenMP declaratives are processed only if switches `-fopenmp` and `-D_REENTRANT` are used during a compilation process. If they are not used or are commented, the code is compiled as one threaded application ignoring all `#pragma omp` declaratives.

Initialization of OCR engine is not a cheap operation. It is suitable to limit number of created OCR APIs. If application creates OCR API per each symbol, it would cause an increase of a computation time instead of its decrease.

OpenMP standardly limits number of created threads to the number of CPUs in the system. It stands that number of created OCR APIs is never greater than maximum number of threads used in parallel. It gives us satisfactory condition for preventing overflow of created OCR APIs.

4.1.3 Summary of Achieved Speed Improvement

The results shown here were achieved on computer with CPU Intel Core i7 (2.7 GHz) with 8 GB of memory. In order to prevent intervention from a system, which can influence computation time, all images are processed hundred times in each case. As a result a median value is taken. It seems to be better option for representing computation time than average value. Median value filters border values better and thus makes results more precise.

Table 4.1 shows computation times for a set of images scanned with resolution 600 dpi stored in TIFF format encoded using an LZW
4. Enhancement of Jbig2enc Using an OCR

compression method. Images used for testing speed improvements are stored on the attached CD, for details see Appendix A.

There is a visible difference in time taken if OCR is used or not. There is visible influence the hash function together with additional improvements on the computation time. In the last column, the processing time reduction is shown when OCR text recognition is run in parallel. Times are written in seconds.

All improvements add an additional comparison process for finding equivalent symbols between already recognized representants in order to reduce their number and the size of the output image. Column showing time taken to process image or sequence of images with the original Jbig2enc encoder is put here only for comparison. It allows to see cost of an improvement made on computation time.

Table 4.1 also shows number of images (pages) processed at once and their impact on the computation time. As is shown with increasing number of images, the speed improvements made has more significant effect. In graph shown in Figure 4.2 differences in computation times are nicely visible, mainly difference if encoder is run with or without hash function and difference between running OCR recognition in parallel and in one thread. There is clearly visible that OCR recognition is an expensive operation and that it greatly slows the process of compressing an image down.

Table 4.1: Comparison of computational times based on speed improvements of improved Jbig2enc (computation time is in seconds)

<table>
<thead>
<tr>
<th>Number of images</th>
<th>Original version</th>
<th>Bachelor thesis’s version</th>
<th>Hash function</th>
<th>Hash function with OCR</th>
<th>OCR run in parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.142</td>
<td>0.208</td>
<td>0.196</td>
<td>1.769</td>
<td>1.188</td>
</tr>
<tr>
<td>2</td>
<td>0.474</td>
<td>0.635</td>
<td>0.581</td>
<td>3.960</td>
<td>2.463</td>
</tr>
<tr>
<td>3</td>
<td>0.598</td>
<td>0.859</td>
<td>0.623</td>
<td>4.955</td>
<td>3.112</td>
</tr>
<tr>
<td>4</td>
<td>0.623</td>
<td>0.983</td>
<td>0.804</td>
<td>6.316</td>
<td>3.954</td>
</tr>
<tr>
<td>5</td>
<td>0.749</td>
<td>1.313</td>
<td>1.026</td>
<td>7.715</td>
<td>4.732</td>
</tr>
<tr>
<td>6</td>
<td>0.828</td>
<td>1.577</td>
<td>1.195</td>
<td>8.784</td>
<td>5.189</td>
</tr>
<tr>
<td>7</td>
<td>0.972</td>
<td>1.976</td>
<td>1.466</td>
<td>10.283</td>
<td>6.165</td>
</tr>
</tbody>
</table>

2. Jbig2enc encoder which does not used any made improvements
4. Enhancement of Jbig2enc Using an OCR

![Graph showing compression times for different versions of improved Jbig2enc and different amounts of images processed at once.](image)

**Figure 4.2**: Comparison of compression times for different versions of improved Jbig2enc and different amount of images processed at once

4.2 Interface for Calling OCR

OCR and image compression according to JBIG2 standard are very similar. In both cases, it is necessary to segment an image into components that are further processed. In OCR, it is necessary to detect text blocks and individual symbols in order to recognize them. In JBIG2, it is also necessary to detect text blocks. In ideal case, it should detect individual symbols in order to achieve the maximum compression ratio. There is one main difference between image compression that follows the JBIG2 standard, and OCR.

In OCR, there is necessary to provide results even when OCR engine is uncertain, and to have the OCR engine trained in order to know the symbols contained in the image. When compressing an image with perceptually lossless compression, encoder cannot afford errors, but it has an advantage. If it is uncertain about a symbol, it can
classify it as a new one, thereby preventing unwanted errors. It also does not need to know font information in advance.

To integrate OCR with the jbig2enc encoder, we designed an API consisting of two parts (modules): one represents the data structure holding the results of OCR, and the other one represents methods for running the OCR engine and retrieving its results.

Our goal is to make the API for holding OCR results and the API for using the OCR engine as adaptable as possible. It allows easy interchangeability of the OCR engines and it prevents unnecessary modifications of existing code.

Because jbig2enc is written in C++, our improvement and API need to be written in C/C++ as well. We decided to use an object hierarchy. It allows the creation of a common class with required methods; for creating new modules, an inheritance is used. A new module using a different OCR engine is easily made by inheriting the relevant class and implementing defined methods. The methods are implemented specifically for the OCR engine actually used.

For holding OCR results, we need to allow the storage of additional data specific to the concrete OCR engine. The additional data can be used to improve the comparison of representants and thus create a specific similarity function that is most suitable for concrete OCR engine.

In this section, we describe the upper layer of module for retrieving data from the used OCR engine. The basic class for holding results of OCR engine is introduced together with examples of possible extensions. The lower layer is specific for a concrete OCR engine. In Section 4.4, we introduce Tesseract implementation of this module.

Figure 4.3 shows classes representing the interface for using an OCR engine. On the left, there are classes representing the module for using an OCR engine and its function. On the right, classes holding results of OCR recognition are described. There is a main class for storing simple structures with representative symbols. For holding OCR results, it is necessary to store additional information, such as a text recognized by the OCR engine and its confidence level.

For this purpose, the class OcrResult is created. It can be extended by creating a new class in order to store additional information provided by the OCR engine. The class needs to be created using an
4.3 Similarity Function

When rendering text from images using OCR, the most important part is to recognize what is written there, not in what format and fonts. This information is also welcome, but it is not the most important one. If you want to compress an image, it is necessary to differentiate between symbols in different fonts because the font information can be of value to the user. When using OCR in detecting equivalent symbols, it is necessary to take this information into account, and if this information is not provided by OCR itself, it needs to be handled by additional methods.

It is also necessary to handle properly occurrences of atypical symbols in the documents or of damaged symbols (Figure 4.4 shows example of such problematic symbols). From the OCR point of view, math symbols can be considered as atypical symbols. It is either possible to use specialized OCR with a math recognition support such as the InftyReader [31], or to detect that it is a math and process it specifically.

The similarity function, in the designed API represented by the method `getDistance`, is the function counting differences between rep-
4. ENHANCEMENT OF JBIG2ENC USING AN OCR

Figure 4.4: Problematic versions of the same letter ‘e’

resentants (symbols). Based on counted difference it decides about their equivalence and thus if they should be unified. The quality of this function highly influences compression ratio achieved and potential error rate.

It works with a set of information that can be imagined as a vector. For all data provided (value in the vector) a distance metric is counted. These values are then put together using weighing and different arithmetic operations to return a single value. The returned value represents similarity distance of two symbols. Two symbols are considered equivalent if their similarity distance is lower than preset value.

The similarity function made like this does not depend on the jbig2enc encoder itself but only on the OCR engine and provided information. The quality of the used similarity function thereby depends on a developer of the module for using a concrete OCR engine.

As part of a module for holding OCR results (class OcrResult), a method getDistance representing a similarity function is implemented. It uses basic information provided in class OcrResult to count similarity of two symbols. A confidence of an recognized text provided by the OCR engine is used as a parameter of the similarity function. Thereby, used OCR engine influences the result. We have tested the function using Tesseract OCR. If another OCR engine is used, implementation of a specific similarity function needs to be taken into consideration.
4.4 Using Tesseract as OCR Engine

To show how OCR improves jbig2enc compression we implement designed API using a Tesseract as the OCR engine. Tesseract is chosen because it is an open-source OCR and it provides reasonable text recognition error rates.

Both Tesseract and jbig2enc are written in C++ and use Leptonica library for holding image data structures. This makes Tesseract ideal for using as a prototype solution. It also supports wide range of languages even with absolutely different types of symbols.

Figure 4.5 shows Tesseract implementation of a module for running an OCR engine and retrieving text recognition results. The TesseractOcr class needs to implement only two methods. They are defined in the module interface. Attribute api represents API of the Tesseract OCR engine. It is put into a class attribute in order to prevent necessity of its initialization for each call of the method recognizeLetter.

As is shown, the implementation of the module for running an OCR engine is simple. The rest depends on the OCR engine used and what functionalities its API offers.

![Class diagram of Tesseract module implementing jbig2enc API for using OCR](image)

Tesseract module is implemented in two methods. Method init() initializes Tesseract API and sets language which is used for a text and a character recognition. Method recognizeLetter uses methods

---

3. For further information about Tesseract OCR engine see Section 3.4
from provided Tesseract API in order to retrieve results of OCR text recognition.

First, it sets a source image resolution, sets an image segmentation mode and retrieves a recognized text in UTF-8 encoding together with a text recognition confidence from the Tesseract OCR engine. The image segmentation mode defines how the image is treated. It can be treated as a single character, a single word, a single paragraph, a page, etc.

We set image segmentation mode to use automatic detection because it gives the best recognition results. If an image segmentation mode is set to treat image as a single character, Tesseract gives wrong results for images (representants) containing more characters than one character. Tesseract in this image segmentation mode forces text recognition to return exactly one UTF-8 character. If the automatic image segmentation mode is selected and image contains just a single character, it is recognized as such. Tesseract OCR engine returns recognized text as a single character if and only if it recognizes the text as a single character with the highest confidence.

If Tesseract renders from image more characters with a higher confidence than by classifying it as one character, the whole recognized text is returned and average confidence counted from individual letter confidences is assigned to returned value.

The recognized text and the Tesseract confidence is stored in basic OCR API class for holding results of OCR engine named `OcrResult`. To prevent affecting recognition of different representative templates (symbols) adaptive classifier of Tesseract is cleared after each run of a text recognition.

### 4.5 Jbig2enc Workflow Using Created OCR API

We have described specific issues affecting computation time and we have designed and implemented solutions decreasing computation time. We have introduced API for using an OCR engine together with a Tesseract module as an example of implementation of a defined API. We need to define how to connect these parts in order to use them inside jbig2enc encoder.
Procedure for Finding Equivalent Symbols Using OCR

1. Counting hash function together with OCR recognition results.

2. Counting distances between all symbols with same hash value using a similarity function.
   - If the distance (from similarity point of view) between two symbols is lower than a specified value, they are marked as equivalent.

3. Choosing best representative symbol from equivalent ones based on a confidence of an OCR text recognition.

4. Unifying equivalent symbols.

Function `uniteTemplatesInList` unifies equivalent symbols. The symbols are provided in a list and it accepts another argument determining new chosen representative symbol for them. Unification of equivalent symbols is based on reindexing procedure. By reindexing it is meant a procedure of changing pointers of all occurrences of the specified symbol (representant) to another one. When this is done, old representant is removed. In order to maintain a compact array (dictionary) holding representative symbols, position of old (removed) representant needs to be filled. It is done by moving last representant to its place.

   The procedure is more complicated and solves other issues connected with unifying more than two symbols. In order to prevent moving the same representative symbol twice, it checks if the representative symbol is not one of the other representants. These representants are provided in the list of symbols prepared for unification.
5 Evaluation

5.1 Comparison with Previous Version of Improved Jbig2enc

In order to evaluate created tool on real data we use a set of more than 800 PDF documents (provided on an attached CD, for details see Appendix A). Documents are chosen randomly from a Czech digital mathematical library (DML-CZ, see Section 6.1 on page 39). The chosen documents are from different journals and were published in different years, some of them were published even more than fifty years ago. Documents consist of images compressed in FAX G4 and contain scanned text in different languages (English, Czech, Slovak, Russian).

The used dataset contains even old documents with a lower image quality. In order to prevent unwanted errors, a thresholding of jbig2enc encoder needs to be set to a minimal loss (enabling encoder option \(-t \ 0.9\)). By default, encoder counts difference of two symbols using a Hausdorff distance\(^1\). If the similarity according to Hausdorff distance of two symbols is greater than the set threshold value, symbols are considered equivalent. In case of some older articles, if threshold value is left default, some symbols are considered equivalent even if they are not.

For running the jbig2enc encoder and its improved versions on a PDF document, a tool pdfJbIm is used (see Section 2.4 on page 9).

Subset of more than fifty documents chosen randomly was tested for occurrences of visual errors, and there were found none.

A similarity distance function used for results of Tesseract is set rather to safe values. With further testing weights can be modified and accustom better in order to achieve even better results without causing errors.

Instead of presenting concrete values achieved on individual documents, average values are used. Thus there are used average values of recognized symbols and average sizes of PDF documents after

---

1. For further information about Hausdorff-based image comparison see \(\text{http://www.cs.cornell.edu/vision/hausdorff/hausmatch.html}\)
running different versions of jbig2enc encoder. Results are shown for different number of pages processed at once.

Table 5.1 shows average numbers of representative symbols recognized. It contains results for different versions of jbig2enc encoder and for different number of pages (from 1 to 10 pages). There is a visible improvement achieved using different versions of the enhanced jbig2enc encoder. The basic improvement (without OCR) of the encoder decreases number of representative symbols in total by five percent. If Tesseract is used as the OCR engine for further improvement of image quality, number of representative symbols is decreased by additional three percents.

Version of jbig2enc encoder evaluated in [3] was evaluated even on documents containing coloured images. The colored images were thresholded to bitonal version and then processed. Also there were found small bug which did not caused errors on test data. The error was repaired. Those are the reasons why in [3] better results were presented than the results that are shown here for the previous version of jbig2enc encoder.

Table 5.1: Number of different symbols (representative symbols)

<table>
<thead>
<tr>
<th>Number of pages</th>
<th>Original jbig2enc</th>
<th>Improved jbig2enc without OCR</th>
<th>Improved jbig2enc with OCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,562</td>
<td>1,504</td>
<td>1,467</td>
</tr>
<tr>
<td>2</td>
<td>3,088</td>
<td>2,962</td>
<td>2,900</td>
</tr>
<tr>
<td>3</td>
<td>4,556</td>
<td>4,328</td>
<td>4,231</td>
</tr>
<tr>
<td>4</td>
<td>5,961</td>
<td>5,636</td>
<td>5,503</td>
</tr>
<tr>
<td>5</td>
<td>7,489</td>
<td>7,084</td>
<td>6,919</td>
</tr>
<tr>
<td>6</td>
<td>9,045</td>
<td>8,565</td>
<td>8,370</td>
</tr>
<tr>
<td>7</td>
<td>10,415</td>
<td>9,776</td>
<td>9,531</td>
</tr>
<tr>
<td>8</td>
<td>11,817</td>
<td>11,161</td>
<td>10,887</td>
</tr>
<tr>
<td>9</td>
<td>13,276</td>
<td>12,429</td>
<td>12,125</td>
</tr>
<tr>
<td>10</td>
<td>14,428</td>
<td>13,409</td>
<td>13,075</td>
</tr>
</tbody>
</table>

Graph shown in Figure 5.1 shows how the number of processed pages influences amount of recognized representative symbols. The increase or recognized symbols is almost linear, even though the
logarithmic seems more logical. It is mainly influenced by different alignment causing slightly different segmentation for symbols located on different pages. This is an issue planned to solve in the next release of the improved encoder by expanding comparison process even to symbols with slightly different dimensions.

Figure 5.1: Number of different symbols (representative symbols)

Table 5.2 shows results achieved when compressing PDF documents with different amount of pages. It shows results achieved using original version of jbig2enc encoder\(^2\), version from bachelor thesis \(^3\) and a new version of the encoder enhanced by using Tesseract as the OCR engine.

Results show the improvements achieved by each encoder version. The version using OCR gives an additional two percent improvement in comparison with a previous version \(^3\).

Figure 5.3 shows two images. First image is the original in a TIFF format FAX G4. The second image is compressed using an improved

\(^2\) Version without the improvement made in \(^3\) and without the improvement using OCR.
Table 5.2: Results of an enhanced jbig2enc encoder

<table>
<thead>
<tr>
<th>Number of pages</th>
<th>Original PDF</th>
<th>Original jbig2enc</th>
<th>Improved jbig2enc without OCR</th>
<th>Improved jbig2enc with OCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>107.11</td>
<td>88.64 (82.8%)</td>
<td>86.72 (81%)</td>
<td>84.63 (79%)</td>
</tr>
<tr>
<td>2</td>
<td>240.76</td>
<td>203.19 (84.3%)</td>
<td>198.47 (82.4%)</td>
<td>193.83 (80.5%)</td>
</tr>
<tr>
<td>3</td>
<td>353.87</td>
<td>296.73 (83.9%)</td>
<td>288.11 (81.4%)</td>
<td>281.21 (79.5%)</td>
</tr>
<tr>
<td>4</td>
<td>476.82</td>
<td>401.13 (84.1%)</td>
<td>388.85 (81.6%)</td>
<td>379.38 (79.6%)</td>
</tr>
<tr>
<td>5</td>
<td>592.42</td>
<td>499.82 (84.4%)</td>
<td>484.31 (81.7%)</td>
<td>472.61 (79.8%)</td>
</tr>
<tr>
<td>6</td>
<td>722.71</td>
<td>609.02 (84.3%)</td>
<td>590.66 (81.7%)</td>
<td>576.42 (79.8%)</td>
</tr>
<tr>
<td>7</td>
<td>822.41</td>
<td>691.49 (84.1%)</td>
<td>667.13 (81.1%)</td>
<td>650.51 (79.1%)</td>
</tr>
<tr>
<td>8</td>
<td>949.18</td>
<td>800.55 (84.3%)</td>
<td>775.36 (81.7%)</td>
<td>756.16 (79.7%)</td>
</tr>
<tr>
<td>9</td>
<td>1,080.05</td>
<td>913.35 (84.6%)</td>
<td>880.55 (81.5%)</td>
<td>858.71 (79.5%)</td>
</tr>
<tr>
<td>10</td>
<td>1,161.09</td>
<td>975.56 (84%)</td>
<td>936.53 (80.6%)</td>
<td>913.19 (78.6%)</td>
</tr>
</tbody>
</table>

jbig2enc encoder with using Tesseract as the OCR engine. Size of the second image is one third of the first one and there are no visible differences.

But if we look more closely, they are not the same in each pixel. The difference between those two images is shown in Figure 5.4. We can see that the improved version of jbig2enc greatly reduces flyspecks and dirts, and at the same time, a quality of the output image is not decreased.

### 5.2 Evaluation on Data from EuDML

In EuDML system a problem with retrieving PDF documents from certain content providers occurred. The PDF documents are not correctly available from content providers. It is caused that the EuDML system is still in the development process and not all issues have been solved yet.

In the current state, the EuDML system does not support processing of a collection subset loaded in the system (It always processes the whole collection). We use a relatively small collection of data
5. Evaluation

Figure 5.2: Compression results of jbig2enc and its improved versions consisting only of around 2,000 PDF documents. Used documents were gathered from CEDRAM collection where CEDRAM is a French provider. Data from DML-CZ are in EuDML already optimized using JBIG2 compression. PdfJbIm in current version does not support recompression of JBIG2 images that use a global dictionary. Thereby, optimizing the already optimized PDFs from DML-CZ would not provide any improvements.

In original, CEDRAM collection requires 1.4 GB in order to be stored. After compression is done using pdfJbIm with improved jbig2enc encoder supporting OCR, only 1.3 GB of storage space is required. Not all PDF documents contain images with a scanned text. A lot of PDFs contain images that are not bitonal, but rather grayscale. These images are not processed in default and they are left as they were.

3. The PDF documents we do not provide on attached CD because the data are provided only for EuDML system internal use.
According to Feynman, to calculate a scattering amplitude, one sums over all possible arrangements of particles branching and rejoining. Moreover, for a particle traveling between two spacetime events $x$ and $y$, one must in quantum mechanics allow for all possible classical trajectories, as in figure 3. To evaluate the propagator of a particle from $x$ to $y$, one integrates over all possible paths between $x$ and $y$, using a weight factor derived from the classical action for the path.

According to Feynman, to calculate a scattering amplitude, one sums over all possible arrangements of particles branching and rejoining. Moreover, for a particle traveling between two spacetime events $x$ and $y$, one must in quantum mechanics allow for all possible classical trajectories, as in figure 3. To evaluate the propagator of a particle from $x$ to $y$, one integrates over all possible paths between $x$ and $y$, using a weight factor derived from the classical action for the path.

Figure 5.3: Image before and after compression according to JBIG2 standard

Figure 5.4: Difference between original image and image compressed with a jbig2enc encoder using OCR
6 Usage in DML

Digital mathematical libraries (DMLs) contain a large volume of documents with scanned text (more than 80% of EuDML is scanned), which are mostly created by scanning older papers which were written and published earlier and their digital versions are already lost. Documents created this way are referred to as retro-born digital documents.

Research in math is influenced greatly by older articles and papers. When something new in math is discovered or researched, it is often based on older papers and discoveries. To make research more comfortable users require easy access to these kinds of documents. Thus DMLs need to provide documents that are easy to both find and access.

6.1 DML-CZ

Project DML-CZ (Czech digital mathematical library) funded from 2005 to 2009 was developed in order to digitize mathematical texts published in the Czech Republic. It comprises periodicals, selected monographs and conference proceedings since nineteenth century up to most recently produced mathematical publications. It is available at [dml.cz](http://dml.cz) serving almost 30,000 articles on 300,000 pages to the public.

PdfJbIm and Improved Jbig2enc Integration into Project DML-CZ

The data are stored in a specific directory structure. All articles are stored in directories which corresponds to a path consisting of a paper type, including a journal name, a volume, a year of a release and an id identifying it uniquely inside the journal.

The versions of documents served at [dml.cz](http://dml.cz) are created running a set of scripts and tools. Perl scripts are mostly used to manage the workflow. Specific script generates article in a format of a PDF document. First step is creating a PDF document. It is either created from \LaTeX{} sources, if they are provided, or from image files containing scanned pages. Further metadata information are gathered and extracted from the created document. A cover page is added as first
page of the PDF document. This way a basic PDF document is created. The PDF document is then optimized and as a final step the PDF document is digitally signed.

The Perl script directly engages tools from a command-line and thus the integration is simple. The PdfJbIm, together with improved jbig2enc using an OCR engine to improve compression ratio, is integrated as one of tools used for optimizing the PDF.

6.2 EuDML

Project of European Digital Mathematical Library (EuDML) \[32\] creates an infrastructure system aimed to integrate the mathematical contents available online throughout Europe to allow both extensive and specialized mathematical discovery. It is a three years project funded by the European Commission, started in February 2010.

The primary aim of EuDML project is to create an infrastructure and an uniform interface in order to provide mathematical content available throughout Europe. Documents are gathered from content providers with their metadata. The metadata needs to fulfill only a minimal criteria. Documents and metadata are then internally processed to provide enhanced information for browsing and searching.

The EuDML system maps metadata provided from content providers to its internal structure. OAI-PMH\[41\] is a primary mean for content harvesting but other methods will be considered throughout the systems’ lifetime.

The primary goal of EuDML system is not to provide innovative tools, but rather to integrate existing tools and maximize the accessibility of mathematical content for the users.

PdfJbIm and Improved Jbig2enc Integration into EuDML System

It is required for the EuDML system to be platform independent, modular and easily extensible. Scalability is also very important in order to be able to manage a vast amount of users and data.

Core of EuDML system is created in Java bringing platform independence to the system. It provides a modular API allowing to

---

1. The Open Archives Initiative Protocol for Metadata Harvesting
easily change existing services or add a new one. Many tools in EuDML system are so-called processing nodes. They can be chained to so-called processes. The initial node in a process typically generates or otherwise obtains chunks of data which are consecutively processed by following nodes. A processing node basically enhances chunks of data it receives on its input and sends them further with possible side effects such as indexing the contents of the chunk. The final node in a process typically stores the enhanced chunks of data in a storage or discards it.

There was developed a framework written in Java that orchestrates a flow between processing nodes. Author of an individual tool, thereby just needs to implement a processing node with well-defined inputs and outputs. In a configuration file (in XML format), the desired flow between processing nodes is defined.

Several tools used in EuDML system are not written in Java and their reimplementation would be too expensive. Therefore they are either provided with remote interface, and communication is handled either using REST or SOAP, or they are provided as binaries. They are then processed using a Java runtime environment. Java runtime environment allows to execute a code outside JVM (Java Virtual Machine) directly from Java code. This brings additional computation requirements but in compare to opening and processing a PDF document it is neglectable.

In order to integrate PdfJbIm into the EuDML system and therefore to integrate the improved jbig2enc, it is necessary to create a processing node. It is necessary to configure a process workflow that serves a PDF document at start and returns its compressed version in the end. PdfJbIm is written in Java and therefore its integration is relatively straightforward and mostly corresponds to its main method that is used if PdfJbIm is run from a command-line.

A processing node which handles integration of PdfJbIm is represented by the class PdfCompressorProcessingNode. At start, it takes an EnhancementProcessMessage, where information about stored PDF document is provided. It retrieves the PDF document as a byte array using a provided class EnhancementUtils.ImageExtractor class of pdfJbIm

2. For information about PdfJbIm see Section 2.4
requires an input stream or a file on input, the byte array is thus transferred to *ByteArrayInputStream*.

The rest is the same as shown on diagram in Figure 2.2 with exception that input file is provided as input stream and output is returned in a *ByteArrayOutputStream*. It is then transfered to a byte array. In the end, a content containing compressed version of a PDF document is added into the *EnhancerProcessMessage* and provided for the next processing node.

To prevent collisions of filenames from multiple simultaneous runs, all temporary files such as extracted images are stored in a temporary directory with a unique filename. The unique filename consists of document ID concatenated with a string “pdfJbIm”. In order to distinguish it from other processes and improve its debugging, it is concatenated with an actual date and time expressed in milliseconds elapsed since 1.1.1970.

In order to use an improved version of a jbig2enc encoder, which uses an OCR for further improvement of image quality, pdfJbIm was modified to allow setting of additional parameters when engaging the jbig2enc encoder. One of the new parameters is a language that OCR engine uses to improve recognition quality. Input *EnhancerProcessMessage* does not only contain the PDF document but also a metadata information about the document being processed. The document language information is provided mostly using standardized language codes (ISO 639-1 and ISO 639-2 language codes). Thereby, created class (enum) *TesseractLanguage* handles translating from provided code from standardized form to a Tesseract specific language code. If the language is not supported by the Tesseract OCR engine, a default language code is used: currently set to English (“eng”).

Jbig2enc is written in a C/C++ and is provided as a compiled binary. The OCR engine and other libraries are provided as dynamic (shared) libraries that need to be located at specified directory to which environment variable points. In order to make this part independent to the Java implementation of the method engaging the jbig2enc encoder, a shell script is created. The shell script sets system environment property *LD_LIBRARY_PATH* to point to a direc-

---

3. LD_LIBRARY_PATH contains paths to directories with shared libraries separated by a colon
tory with shared libraries needed by jbig2enc encoder. It also sets TESSDATA variable to point to a directory containing Tesseract language data. And finally, jbig2enc encoder is engaged together with provided arguments.
7 Conclusion

The main goal of this thesis was to create a uniform interface for using an OCR engine in order to improve compression ratio of the jbig2enc encoder. The API was successfully created and a solution using Tesseract as the OCR engine was implemented. We have shown results achieved on the data in digital mathematical libraries.

As part of jbig2enc encoder improvement, solutions decreasing computation time were created. Hash function and methods benefiting from new approach for holding data were presented. Speed improvement, computation requirements of OCR and an influence of running OCR in parallel were presented.

By implementing support for OCR, better possibilities for choosing a more adequate representative symbol were created. This allows not only to improve the compression ratio, but also to improve the quality of the output image.

The created similarity function requires further testing before putting this tool into the real working environment. When using an OCR, there are acceptable certain errors, but the compression should be resistant to errors. This requires more complex testing in order to achieve both maximum compression ratio and zero error rate.

In order to improve compression ratio of the Tesseract module, new global dictionary containing all standardly used symbols should be created.

For further improvement of computation performance, other parts of the jbig2enc encoder could be parallelized.

The achieved results imply that we came closer to the ideal compression ratio, but there is still a long way to go. The way can be shortened by implementing a module for an OCR engine with a greater accuracy. It also can be shortened by training the OCR engine on the data which are later being processed by the encoder.

We have shown how to integrate a new version of the jbig2enc encoder into two digital mathematical libraries. Czech digital mathematical library already uses an older version of the jbig2enc encoder (developed as part of bachelor thesis [3]) for more than a year. This shows that such tool is useful and beneficial in a real environment.
Bibliography


7. CONCLUSION


7. Conclusion


7. Conclusion


# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of two originally different symbols recognized as equivalent</td>
</tr>
<tr>
<td>2.2</td>
<td>PdfJbIm workflow</td>
</tr>
<tr>
<td>4.1</td>
<td>Formula for counting hash without OCR results</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison of compression times for different versions of improved jbig2enc and different amount of images processed at once</td>
</tr>
<tr>
<td>4.3</td>
<td>Class diagram of jbig2enc API for using OCR</td>
</tr>
<tr>
<td>4.4</td>
<td>Problematic versions of the same letter ‘e’</td>
</tr>
<tr>
<td>4.5</td>
<td>Class diagram of Tesseract module implementing jbig2enc API for using OCR</td>
</tr>
<tr>
<td>5.1</td>
<td>Number of different symbols (representative symbols)</td>
</tr>
<tr>
<td>5.2</td>
<td>Compression results of jbig2enc and its improved versions</td>
</tr>
<tr>
<td>5.3</td>
<td>Image before and after compression according to JBIG2 standard</td>
</tr>
<tr>
<td>5.4</td>
<td>Difference between original image and image compressed with a jbig2enc encoder using OCR</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summary of tools working with standard JBIG2</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Summary of OCR tools</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Comparison of computational times based on speed improvements of improved jbig2enc (computation time is in seconds)</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Number of different symbols (representative symbols)</td>
<td>34</td>
</tr>
<tr>
<td>5.2</td>
<td>Results of an enhanced jbig2enc encoder</td>
<td>36</td>
</tr>
</tbody>
</table>
A CD Content

- Jbig2enc and its improved version
  - jbig2enc_modified – Jbig2enc encoder source codes including binaries and libraries.

- Test data
  - speedImprovementTesting – Images used for testing created speed improvements of a jbig2enc encoder.
  - testData – PDF documents used for comparison with previous version of jbig2enc encoder.

- Text of diploma thesis
  - text – Text of diploma thesis in PDF and \TeX\ sources.
B Manual for Running Jbig2enc Improvement

B.1 Jbig2enc Command-line Arguments Enabling Created Improvements

- Enabled symbol coding (option \texttt{-s}).
- Enabled the option \texttt{-autoThresh} which defines usage of additional improvements.

In order to use an OCR engine for further improvement of compression ratio options mentioned earlier are required. There are additional requirements which needs to be met in order to use an OCR engine:

- OCR engine needs to have implemented a module which implements defined API and have it linked correctly during compilation process.
- An option \texttt{-useOcr} needs to be enabled in order to allow a jbig2enc encoder API for using the OCR engine.
- Input image resolution needs to be either greater or equal to 200 dpi or unknown with an option \texttt{-ff} being enabled.
- An optional argument \texttt{-lang <lang>} in order to set language of text at the input image and therefore improve OCR engine recognition quality. The format of a language setting needs to be provided in a format specific for a used OCR engine. If not set default language settings is used.