Project Thesis
Token-Based Compression

van Beusekom Joost

May 18, 2005

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Token-Based Compression

Token-Based Compression (TBC) was first introduced by Ascher and Nagy [AN74]. The basic idea was to speed up digital transmission of scanned documents, by sending only the “new” tokens (here a token can be seen as a scanned character), where “new” means that this kind of token has not been sent before. If a new, unknown token is to be sent, it is saved in a dictionary and then transmitted as image. The receiver gets the image and saves it in the dictionary and in the output image. If the token to be sent has already been transmitted then only the number of the token in the dictionary will be transmitted. The other side then knows that is has to take the token saved previously in the dictionary.

The word ”Otto” for example would be transmitted in the following way (suppose we are reading from left to right): first the token “O” is read and transmitted as image and saved into both dictionaries. The dictionaries now contain “1:image(O)”. Then for the first “t” the procedure is the same. Now the dictionaries contain “1:image(O), 2:image(t)”. For the second “t”, the program recognises that is has already a similar token in the dictionary of the sender, and thus only transmits the number of the token. In this case it would be 2. The receiver now knows that he has to put the token 2 of his dictionary to the specified location. The last character “o” is a new character, and it is processed as the first one.

The basic concept of this method can be found in an image format called ”DjVu” [BHH+98]. This format uses a lot of techniques in order to achieve better compression rates for black-and-white and colour documents. For bitone scanned documents (documents holding only black or white pixels) “DjVu” utilises a method called JB2. JB2 is a TBC-implementation for black-and-white scanned documents. It is a variation of a the JBIG2 standard invented by AT&T’s for fax compression.

1.2 Use of Token-Based Compression

There are a lot of huge libraries and archives containing many documents, books, magazines, etc. In order to share these over the World Wide Web (or any other digital network), these books need to be digitised. After digitising, one image of every single page is obtained. These images can be compressed by standard compression techniques, but even then they are far too big to be downloaded in a reasonable amount of time. One way to decrease the space used by the image, is to use a lower resolution, which decreases the readability of the text. The other way is to use Optical Character Recognition for converting the image into text.

Despite all the progress that has been done in the domain of Optical Character Recognition (OCR), OCR still does not work accurate enough for digitising in an unsupervised way, whole archives of scanned documents. The other problem is that commercial OCR systems are very expensive.

So another way has to be gone in order to share whole libraries of documents over the Internet. This can be done by Token-Based Compression, a compression technique that has been developed especially for compressing scanned documents. With this method it is possible (as shown by ”DjVu”) to share large amounts of scanned documents, because of their very small file size, compared with other image formats.

1.3 Overview

1.3.1 Goal

The aim of this project thesis was to implement a TBC method to test how good a simple approach to this technique would work. The program should also allow to compress lossless and lossy. Lossless compression means that the original image has to be reconstructed identically. For lossy compression, the image should contain the same information, but slight differences are allowed, as long as they don't change the information it contains, e.g. changing a “c” to an “e”. Furthermore it should be possible to use one dictionary for more than one page (e.g. to use a dictionary for 30 pages scanned out of the same book). Here, in this document, the word “dictionary” refers to a special image file containing the prototypes of the tokens. A prototype is one token that has been chosen as “representative” of this class of tokens. To keep the introduction understandable, a token will here be seen as an image of a character. Different tokens may all contain the same character. These tokens should then be regarded as the same class of tokens, or, in other words, they are considered “similar”.

1.3.2 Processing Steps

Binarization The first step is to binarize a greyscale image. Binarization is the process of converting a greyscale image to a black-and-white image. This can be done by a global threshold that sets a pixel to black if the value of the pixel is below the threshold, otherwise it is set to white. This method works fast, but quite badly for images where the illumination is not uniform.

The method used here, is a local adaptive thresholding method. For a neighbouring region of the pixel, a local threshold is calculated, and the pixel is set to black or white, according to this local threshold.

An other method that has been implemented, first calculates the background and foreground image by histogram manipulations. Then it compares the background image with the foreground image to get the binarized image which consists ideally only of the characters. We called this method the "difference method".

Connected Components The second step consists of extracting the connected components. A connected component is a set of black pixels that are connected. "Connected" means, in this case, that the black pixels are "touching" each other.

This algorithm uses a wide-spread labelling method using a "union-find" data structure. Region Growing has also been implemented.

In this part of the program, the residual image is created. This image contains all connected components that are larger than a certain amount of the total size of the page (e.g. 5%). These connected components are simply saved in the residual image. This image is later used for reconstruction of the original image.

Noise Reduction This part simply scans for connected components that contain only a very small number of black pixels (e.g. 4 or fewer). These connected components are simply considered as noise and are removed from further processing.

Token Recognition & Similarities This part of the process scans the connected components for tokens that are similar and then saves a dictionary containing only prototypes and a text file containing information about the positions of the tokens, used for reconstruction. In the case of lossless compression, it also saves the pixels of the original token, that differ from the prototype in order to allow a lossless reconstruction. This data is saved in the residual image.

The similarity between two tokens is calculated in two steps: the idea was to try different methods of similarity measurement. Finally, it has been tried to do a first step that can be done quickly. Then afterwards pixelwise comparison is done. One method that has been used for the first step used central moments of the tokens.

The second method used Fast Fourier Transform (FFT): the idea is to consider the positions of the pixels of the contour of a token as a complex function. Then this function is Fourier transformed and the resulting Fourier coefficients are used to compare two tokens. The Fourier coefficients contain information about the contour of the token. The first Fourier coefficients represent the low frequencies of the contour, that means the rough contour. The last coefficients represent the high frequencies (and the noise). So it is enough to compare only the first few coefficients and only if these are somehow similar, the second step, namely pixelwise comparison, is done.

If pixelwise comparison gives us two similar tokens, then only the position of the new token, and the number of the replacing prototype is saved. Otherwise a the new token is saved as prototype.

In order to allow the use of one dictionary for more than one page, the program allows to read a dictionary before proceeding with the image to be compressed.

Is also allows to update the residual image, created during the extraction of the connected components, in order to allow lossless compression.

Reconstruction This part of the project work does the reconstruction of the original image out of the dictionary (an image containing the prototypes), the positions data (information about where in the original image to put what prototype) and the substitutions. The uses of the substitutions has become necessary because of the saving method for the prototypes: as these are sorted before saving them to the image file, their original unique number is lost. So additional information has to be saved to allow us to reconstruct the original prototype number of each prototype. This original number is the number that is saved in the positions data to give us the number of the prototype that is needed in a certain location of the image.

Chapter 2

Main Part

2.1 Binarization

2.1.1 General Description

Binarization is the process that converts an greyscale image to a binary image (black-and-white image). In our case the greyscale values are 8bit values. “8bit greyscale” means that each pixel is coded by 8 bits. This 8 bits give the darkness respectively the lightness of the pixel. Value 0 is ”black”, value 128 is grey, value 255 is white. See also figure 2.1. Ideally, binarization separates background (paper) from foreground (characters, text).

- Input: - a greyscale image
- Output: - a binarized (black-and-white) image

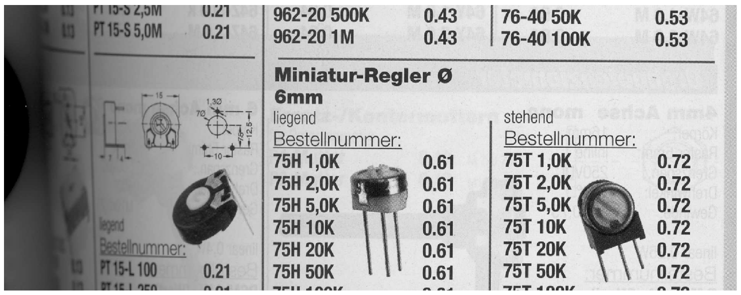
Figure 2.1: Greyscale values, black is value 0, white value 255

2.1.2 Global Thresholding

Binarization can be done by a global threshold. An example for global thresholding is given by the following algorithm:

```
INT threshold = 128 ; /*suppose 8bit greyscale images*/
FOR every pixel DO
    IF (value of pixel >= threshold) THEN set pixel to white ;
    ELSE set pixel to black ;
```

In other words: if a pixel is darker as our threshold, it is set to black. In the other case it is set to white. As shown in the figure 2.2, this method fails if the illumination of the scanned document is not uniform.



PT 15-S 2,5M	0.21
PT 15-S 5,0M	0.21
962-20 500K	0.43
962-20 1M	0.43
76-40 50K	0.53
76-40 100K	0.53

Miniatur-Regler Ø 6mm

liegend

Bestellnummer:

75H 1,0K	0.61
75H 2,0K	0.61
75H 5,0K	0.61
75H 10K	0.61
75H 20K	0.61
75H 50K	0.61

stehend

Bestellnummer:

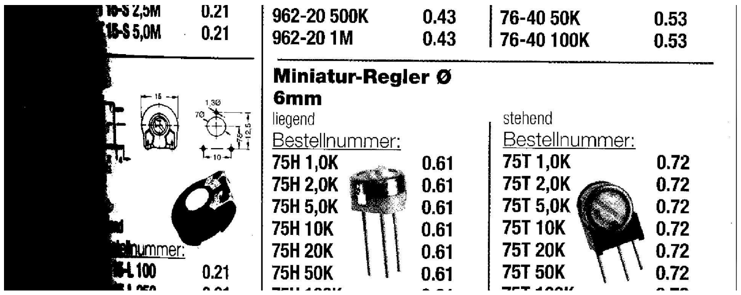
75T 1,0K	0.72
75T 2,0K	0.72
75T 5,0K	0.72
75T 10K	0.72
75T 20K	0.72
75T 50K	0.72

liegend

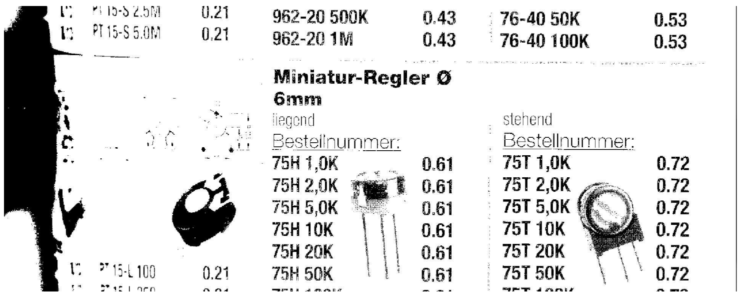
Bestellnummer:

PT 15-L 100	0.21

(a) Original greyscale image [Rei]



(b) Binarized with global threshold 128



(c) Binarized with global threshold 64

Figure 2.2: Example of global thresholding

2.1.3 Local Adaptive Thresholding

Basic Algorithm Instead of using one global threshold for all the pixels, local thresholding calculates a local threshold for every pixel. An example for this adaptive thresholding is the following algorithm [Dav97]:

```
minrange = 255/5 ;
FOR each pixel DO
   find minimum and maximum of local intensity distribution ;
   range = maximum - minimum ;
   IF range > minrange
       THEN T := (minimum + maximum)/2 ;
       ELSE T := maximum - minrange/2 ;
   IF value of pixel > T
       THEN thresholded pixel = 255 ; /*white*/
       ELSE thresholded pixel = 0 ; /*black*/
END FOR ;
```

The idea behind this algorithm is the following: first determine a value "*minrange*" that gives us the least likely difference in intensity between a dark pixel (foreground, text) and a bright pixel (background, paper) in a window (i.e. 40x40 pixels).

Then every pixel is examined: first we get the maximum and minimum in intensity of the neighbouring pixels. The method we used to get the maximum and the minimum for this window is explained in the next paragraph. Then we get the difference between the maximum and the minimum.

If this difference (range) is greater than the minrange then we have the case that we have some foreground and some background pixels in our window we examined. Therefore we set our threshold T to the mean of the sum of the maximum and the minimum. This means that our pixel has to be darker than this mean in order to be identified as black pixel.

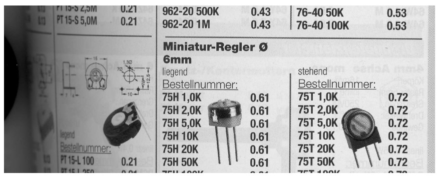
In case that our difference is smaller than the minrange we suppose that we only have background pixels that only differ a little from each other. Therefore our threshold is set to a lower value, namely $maximum - minrange/2$. This allows pixels to turn black that are relatively dark in comparison with their surroundings, e.g. if the background is quite dark.

The last step is simply comparing the value of our input (greyscale) pixel to the threshold we calculated for this pixel and its window, and setting the corresponding output pixel to black or white.

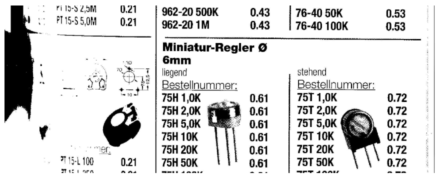
Finding Minima and Maxima The first method used was to check a two-dimensional window with size $l \times l$ for minimum and maximum. This method worked well but was very slow. For a picture of $m \times n$ pixels we get approximately $m \times n \times l^2$ passes of the inner loop.

We then tried to do this calculation only for a predefined grid. All the other values were interpolated linearly. This worked faster and the results were still good.

The second method works in two steps: first it checks for each row in a one-dimensional window of size l for the maximum and the minimum and saves these for each pixel. Then in the second step, it checks for each column the maximum and minimum in a window of size l out of the maxima and minima found in the first step. So we get the maximum and minimum for a two-dimensional window of size $l \times l$ but with less computational effort. For a picture of $m \times n$ we get $m \times l \times n + n \times l \times m = 2 \times l \times n \times m$ passes of the inner loop.



(a) Original greyscale image



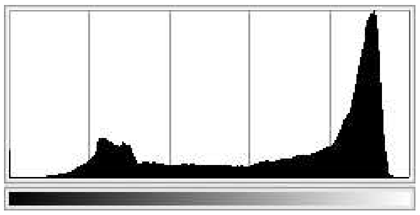
(b) Adaptively binarized, with window-size 40pix

Figure 2.3: Example of adaptive thresholding

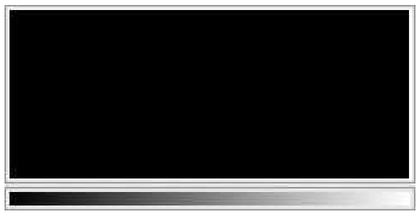
2.1.4 Difference image

Another method that has been tried, is to calculate an approximate background image, an approximate foreground image and then to calculate the difference image.

This method uses histograms. A histogram of a greyscale image is nothing else but a summary of how often what greyscale pixel (what value) appears. Figure 2.4(a) shows us the histogram of 2.2(a). As one can see, there is a lot of darker pixels and a peak with light grey pixels. Figure 2.4(b) shows us the histogram of 2.1. As every value of pixel appears equally often, this histogram looks like a constant function with the number of pixels of every greyscale value as height.



(a) Histogram of 2.2(a)



(b) Histogram of 2.1

Figure 2.4: Example of histograms generated with GIMP

The foreground 2.5(b) and background image 2.5(a) are calculated by a kind of histogram filter. For a window of size $l \times l$ the local histogram is calculated. A certain fraction (e.g. 10%) of the upper values of the histogram are taken as a mean value for the value of the background. The same is done with the lowest values of the histogram. The images obtained are shown in figure 2.5.

The next step consists of comparing fore- and background images. The following three cases have to be distinguished:

1st: Foreground image is white: there is no text, the binarized image in this point will also be white

2nd: if the original value is darker than the mean of foreground and background, then the binarized image in this point will be black; this means that the original image is probably containing text in this point

3rd: if none of the conditions above applied, set the binarized image on this point to white; if the mean of background and foreground is darker as the original image, the original image will probably be white in this position.

An example of this method can be found in figure 2.5

2.1.5 Results

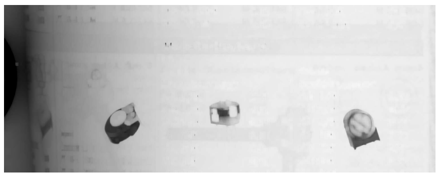
The global thresholding method is very simple and very fast, but it gives bad results when the illumination of the image is not uniform, as it is the case in figure 2.2(a).

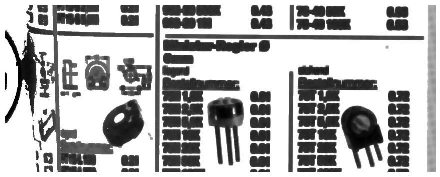
Figure 2.2(b) shows figure 2.2(a) after global thresholding with thresholding value 128. On the left side of the figure, the fine structures, used to distinguish foreground and background, are completely lost.

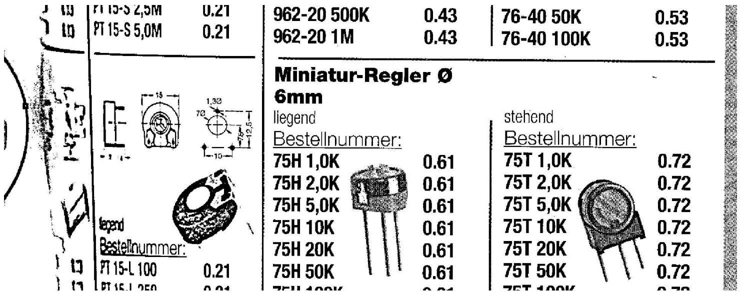
Figure 2.2(c) shows figure 2.2(a) after global thresholding with thresholding value 64. On the right side, a lot of pixels turned to white, which makes the image very noisy.

For local adaptive thresholding we have the problem that the size l of the window we examine, has to be big enough to ensure that the size of a character is smaller than the size of the window. Due to this restraint it is clear that this binarization method is not necessarily useful for other kind of pictures. But for this purpose, namely for scanned documents it works well enough and fast enough.

Although the difference method gave reasonable results and worked fast enough in our tests, we opted for the adaptive local thresholding due to its simplicity and its speed.


(a) Background Image


(b) Foreground Image


PT 15-S 2,5M 0.21
PT 15-S 5,0M 0.21

liegend
Bestellnummer:
PT 15-L 100 0.21

962-20 500K 0.43
962-20 1M 0.43

Miniatur-Regler Ø 6mm
liegend
Bestellnummer:
75H 1,0K 0.61
75H 2,0K 0.61
75H 5,0K 0.61
75H 10K 0.61
75H 20K 0.61
75H 50K 0.61

76-40 50K 0.53
76-40 100K 0.53

stehend
Bestellnummer:
75T 1,0K 0.72
75T 2,0K 0.72
75T 5,0K 0.72
75T 10K 0.72
75T 20K 0.72
75T 50K 0.72

(c) Binarized with Difference method

Figure 2.5: Example of difference method for binarization

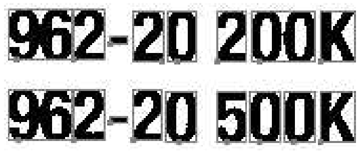
2.2 Connected Components

2.2.1 General Description

The aim of the connected components part is to extract the bounding boxes (grey boxes in figure 2.6) of the connected components out of a binary image. A connected component is a set of pixels that are "connected". Two pixels are connected if they are direct neighbours, or more intuitively, if they are touching each other. A bounding box is the smallest rectangle that fits around the connected component.

Ideally one would obtain one bounding box for every character. Due to noise and blurred printing it often happens, that a few characters are connected together, even though they were originally independent. An example for this can be found in figure 2.6(a) where "9" and "6" are recognised as one connected component "96".

The grey 3×3 pixel blocks give the seed pixel. This is a pixel that is a part of the main connected component of the bounding box. This information is necessary to identify unambiguously the connected component. An example for the case where this seed pixel is absolutely needed can be found in figure 2.6(b). The bounding box that surrounds the "f" includes three different connected components.



(a) Connected components with bounding boxes

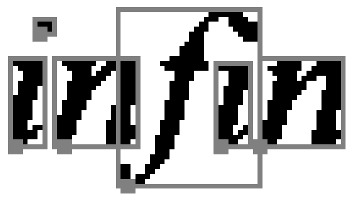

(b) Bounding box containing more than one connected component, [Ebb87]

Figure 2.6: Example of connected components with their bounding boxes

2.2.2 Region Growing

A simple method to get the connected components is the so called “region growing” method.

```
FOR each pixel
    IF (pixel has not been analysed)
      AND (pixel is black) DO BEGIN
        Check_Neighbours(current pixel) ;
        Read the marked pixels and calculate bounding box ;
        Save a seed pixel ;
    END IF ;
END FOR ;

...
Check_Neighbours(pixel)
    mark pixel as analysed ;
    mark pixel as part of the connected component ;
    FOR each neighbour of pixel DO
        IF (neighbour_pixel is black)
          AND (neighbour_pixel has not been
            analysed) DO Check_Neighbours(neighbour pixel)
    END FOR
END Check_Neighbours ;
```

The method begins with an arbitrary black pixel of the image, and looks for all the black pixels that are connected to this starting pixel. After finding all these pixels recursively, it calculates the size of the box and saves also the seed pixel. These steps are repeated until no unanalysed black pixel is left.

2.2.3 Labelling

This method uses another approach as region growing: instead of taking one pixel and checking what pixels are connected to it, this method scans the image line by line and labels the pixels so that every connected component afterwards has its own label (number). For this labelling process we use a data structure called “union-find” structure. This is a tree-structure for saving the labels: it allows us to find the root label given a certain label in the tree, and it allows us to append a tree to another tree as subtree.

- Input
  - a binarized image
- Output
  - a residual image containing all the large connected components. This image is used in the reconstruction step to reconstruct the original image.
  - a text file containing the lower left and upper right coordinates of every bounding box and a seed pixel.

```
abbreviations used:
l_n = left neighbour pixel
u_n = upper neighbour pixel
root(lab1) = root label of label lab1
bbox[] = Array of bounding boxes

01 FOR every row DO
02     FOR every pixel in this row DO
03         IF pixel is black THEN DO
04             IF l_n has label THEN label pixel with l_n label ;
05             ELSE label this pixel with a new label ;
06             IF this pixel has an u_n with label THEN
07                 lab1 = root(current pixel label) ;
08                 lab2 = root(u_n label) ;
09                 IF lab1 != lab2 THEN
10                     set root of current label as
                         child of u_n label ;
11         END IF
12     END FOR
13 END FOR
...
20 FOR every row DO
21     FOR every pixel in this row DO
22         set pixel labels with same root label to root label ;
...
30 FOR every row DO
31     FOR every pixel in this row DO
32         IF pixel is black THEN DO
33             add pixel to bbox[root(label of current pixel)] ;
34             IF (seed is not set) DO set seed ;
```

The first part of the algorithm does the scanning part: it scans the binarized image line per line from top to bottom, and scans the lines from left to right.

If a black pixel is found it checks if this pixel has a left neighbour pixel that has a label (or: if it has a black neighbour pixel). If this is the case (*line 04*), the label of our current pixel gets the label of its left neighbour. This is equal to say that the current pixel is a part of the connected component that started somewhere left of our current position. In the other case (*line 05*) our pixel gets a new label as it is probably the beginning of a new connected component.

In *line 06* it checks if there is an upper labelled (black) neighbour. If this is the case, the pixel we just labelled is a part of the connected component that started in the upper row. Then we have to check the root labels of our current pixel and our upper neighbour. If both are equal we are not allowed to add the subtree of our current label to the subtree of our upper neighbours label because we then would get cycles in the tree. If both roots are different, we have to add the subtree of the current pixel to the subtree of the upper neighbour, as the current label then is connected to all the pixels with labels in the tree of our upper neighbour. This is done by *line 07* to *line 09*. In other words: the pixel we just examined an all the pixels connected to this one, are part of the connected component we identified in the upper row.

Lines 20 to *22* set the labels for every pixel to its root label. So every pixel in a connected component has the same label.

Lines 30 to *34* create the bounding boxes by adding consecutively all the pixels that belong to the specified connected component.

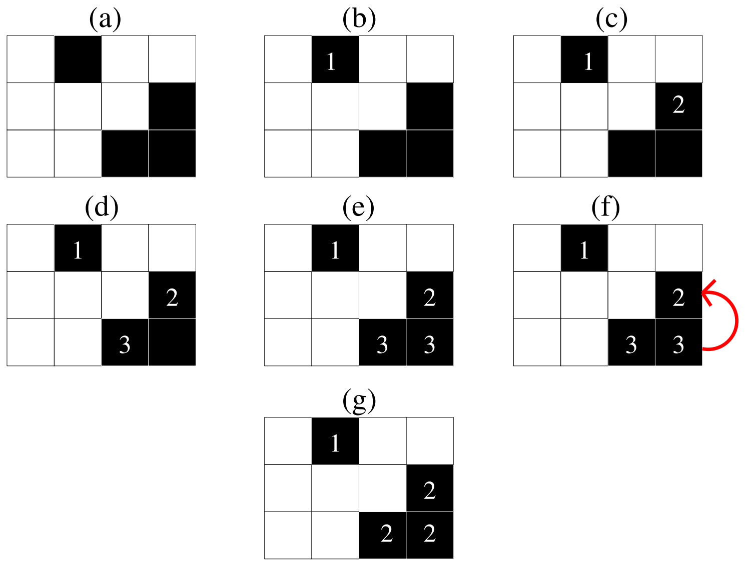


Figure 2.7: Example for the labelling method for extracting connected components

Example The initial binarized image can be seen in 2.7 *(a)*. In the first step the first pixel is labelled with 1 *(b)* by *line 05*. The same is done until we reach step *(e)*. There we can see that the lower right pixel inherits the label of its left neighbour *line 04*. Then immediately after this has been done, *line 06* to *line 10* add the tree of label 3 to the tree with root 2 *(f)*. In the last step *(g)* the labels are renamed, so that every pixel in a connected component has the same label (*line 20 to 21*).

2.2.4 Results

Because of the fact that, despite the overhead of the union-find data structure, the labelling method works faster in normal cases, we opted for the labelling method.

2.2.5 Creation of the Residual Image

The aim of this post-processing is to discard huge elements (e.g. pictures, lines, etc.) from being processed further because they are not appropriate for token-based compression.

After having run the labelling method, the bounding boxes are checked for boxes that are larger than a certain amount of the size of the page. All boxes that are larger than this size (e.g. 10% of the width of the page) are deleted, and their content is immediately copied to the residual image, as may be seen in figure 2.8. This image is used in the reconstruction step to reconstruct the original image. It contains everything that is not compressed by token-based compression.

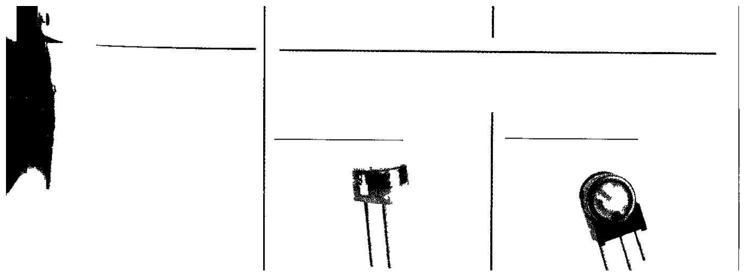


Figure 2.8: Example of a residual image of image 2.2(a)

2.3 Noise Removal

2.3.1 General Description

The only thing this part of the process does, is to eliminate connected components that are likely to be only noise. This method scans all the connected components and deletes those that only consist of a small number of pixel (e.g. 4 pixels). Instead of deleting these pixels in the binary image it simply deletes the corresponding bounding box.

The reason why this method was introduced is that, if a greyscale image is binarized, one obtains lots of connected components consisting of a few pixels only. This can be due to noise, or be the result of the binarization process for graphics or pictures. These connected components cannot be compressed reasonably by token-based compression. Furthermore the token-based compression is slowed down very much by these hundreds of "useless" connected components.

```
01 T = 4 ; //or: T = 8, depending on the size of the characters
```

```
02 FOR every bounding box DO
03     IF number pixels of con.comp. of current b. box < T DO
04         delete current bounding box ;
```

2.3.2 Results

This method is not very sophisticated. Instead of setting the threshold T as a fix value it might be useful to determine the ideal size of T for each image (e.g. the half of the pixel size of a "."). Nevertheless this method is used in order to keep the method working, even though there might be a lot of tiny connected components coming out of a binarized greyscale image.

2.4 Similarity Measurement

2.4.1 General Description

These methods represent the main part of this project thesis. It is their duty to scan for connected components that look somehow similar and to group them accordingly ("clustering"). Therefore we need to measure the degree of similarity between two connected components. For reasons of simplicity we call the "connected components" from now on "tokens".

The measurement of similarity of two tokens can be done by several methods. In this project thesis three of them were implemented and a combination of two methods has been retained.
Here is a rough sketch of the algorithm:

```
01 possibly read prototypes from an re-used dictionary;
02 read bounding boxes and create tokens ;
...
11 FOR every token DO
12     FOR all prototypes DO
13         IF prototype is similar to token THEN
14             save token in dictionary as member
                 of the class of the prototype;
15         ELSE /* if token is new, unknown token /*
16             make new class and save token as prototype
                 of this class ;
17         IF no prototypes exists THEN DO
18             save token as prototype ;
19     END FOR
20 END FOR
...
30 save information about positions of the
     tokens (reconstruction inform.) ;
31 sort and save prototypes ;
32 save substitution information ; /*used for finding the right
       prototypes after sorting */
33 updated residual image; /* for lossless compression */
```

- Input
  - a binary image
  - a text file containing information about the bounding boxes
  - possibly binary residual image obtained, for lossless compression
  - possibly a binary image containing prototypes, for re-use of dictionary
- Output
  - a binarized image containing all the prototypes
  - a text file containing information about the substitutions
  - a text file containing the coordinates of the places where the prototypes have to be placed for reconstruction
  - possibly an updated version of the residual image (used for lossless compression)

A prototype is a token, that has been chosen to be the representative of a certain class of tokens. In figure 2.9 the first token in each row (each row is one class) is the representative of this class (prototype).

The dictionary here means an image where the prototypes are saved in. Figure 2.18 shows a part of the dictionary.

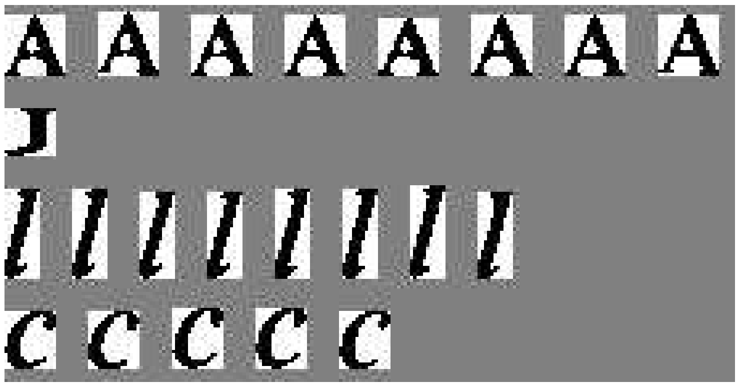


Figure 2.9: Example of 4 classes of tokens. The first token in each row is the prototype

2.4.2 Similarities

In this part the three methods used for measuring the similarity between two tokens are explained. The three methods used are pixelwise comparison, comparison of their moments, and comparison of their Fourier coefficients.

Pixelwise Comparison

This method is very intuitive: given two tokens it checks if the tokens have approximately the same height and width. If this is the case it compares the pixels. If one token has pixels in a range that is not defined for the other token, these pixels are considered as different. And if the number of differing pixels is small enough, the two tokens are considered similar. An example can be seen in figure 2.10. Here the pixelwise difference is 2. The pixels $(2,2)$ and $(2,3)$ in the two images are different.

After having count the number of differing pixels, the decision has to be taken if the two tokens are similar or not. Therefore two methods have been tried:

- Compare with the size of the surface: e.g. if the number of different pixels is less or equal than 10% of the minimum of the surfaces of the two tokens, then the tokens are similar. This method gave bad results for tokens having a big surface but only consisting of few black pixels as the "f" in figure 2.6(b). The token "F" in figure 2.10 would be seen as a "P": the surface size is 15, 10% of 15 equals 1.5 (if we round up we get 2) and 2 is less or equal than 2. Because of the bad results a better method is needed.
- Compare with the number of black pixels: e.g. if the number of different pixels is less than 10% of the minimum of the number of black pixels of the two tokens then the tokens are similar. This method gave better results because it only considers the important pixels, namely the black ones. This time the "F" in figure 2.10 would be seen as a different token, because the minimum number of black pixels equals 8 and 10% of 8 is 0.8. If we round up we get 1 as the maximal allowable difference. So the two characters are different.

N.B. it might be useful to set the percentage of differing pixels not to a fix value, as done in this project, but to adapt it somehow to the mean size of the characters.

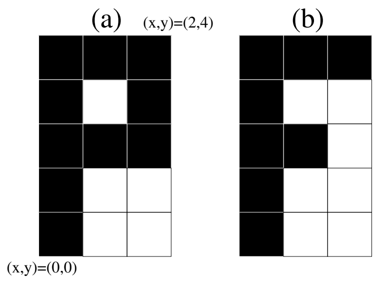


Figure 2.10: Example for pixelwise comparison

Although the fact of comparing the number of differing pixels to the minimum number of black pixels gave good results, one problem still persists: it often happens that two tokens are nearly identical, except for the fact that one of them has on one of the 4 sides one ore more pixels in addition. This can be due to noise, blurred printing, problems with binarization, etc. Then it might be, that the number of counted different pixels is high, although they vary only in one black pixel, as in figure 2.11. The red pixels are all the pixels that differ from the corresponding pixel in the other token.

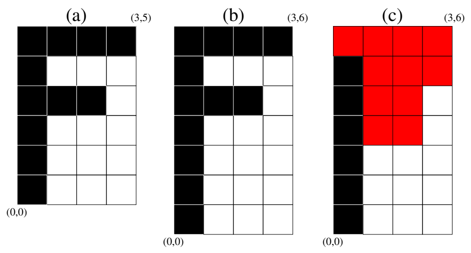


Figure 2.11: Example for problem with naive pixel difference method

Pixelwise difference with superimposing of greyscale barycentre

This method utilises moments in order to calculate the greyscale barycentre of the two tokens, then superimposes the two greyscale barycentres, and then calculates the pixel difference considering the necessary translation to superimpose the two tokens.

Greyscale Barycentre The greyscale barycentre can be seen as the centre of gravity of an greyscale image. Although we only have binarized images we can calculate the centre of gravity of this binary image. The values of the greyscale have the role of the masses for calculating the centre of gravity. To calculate this centre, moments are utilised.

Moments In the continuous case, the moment of order $(p+q)$ for a two-dimensional function $f(x,y)$ is defined as follows:

$$m_{pq} = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} x^p y^q f(x,y)\,dx\,dy \tag{2.1}$$

for p, q = 0, 1, 2, ...

For the discrete case we get: let $f(x,y)$ be a two dimensional function that gives us for a certain position (x,y) the greyscale value of that pixel. Then the moment of order $(p+q)$ is defined as follows:

$$m_{pq} = \sum_{-\infty}^{+\infty}\sum_{-\infty}^{+\infty} x^p y^q f(x,y) \tag{2.2}$$

Central moments in the continuous case are defined as

$$\mu_{pq} = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} (x-\bar{x})^p (y-\bar{y})^q f(x,y)\,dx\,dy \tag{2.3}$$

where

$$\bar{x} = \frac{m_{10}}{m_{00}} \qquad \textit{and} \qquad \bar{y} = \frac{m_{01}}{m_{00}} \tag{2.4}$$

Central moments in the discrete case are defined as

$$\mu_{pq} = \sum_{-\infty}^{+\infty}\sum_{-\infty}^{+\infty} (x-\bar{x})^p (y-\bar{y})^q f(x,y) \tag{2.5}$$

where

$$\bar{x} = \frac{m_{10}}{m_{00}} \qquad \textit{and} \qquad \bar{y} = \frac{m_{01}}{m_{00}} \tag{2.6}$$

$\bar{x}$ and $\bar{y}$ are the coordinates of the greyscale barycentre in the local coordinate system of the token. Central moments are used in the next paragraph for calculating the central moments of a token.

Our method now calculates the translation that is needed to superimpose the greyscale barycentre of one token to the greyscale barycentre of the other token. It then calculates the number of differing pixels.

Because of the rounding operation, we not only checked the pixel difference for this translation, but also for its 8 neighbouring pixels. The minimum of this 8 pixel difference calculations is returned as result.

For our example 2.12 we get for token (a):

$$\begin{aligned} \bar{x_A} &= \frac{m_{10}}{m_{00}} \\ &= \frac{\sum_0^3 \sum_0^5 x f(x,y)}{\sum_0^3 \sum_0^5 f(x,y)} \\ &= \ldots \\ &= \frac{6 \times 0 \times 255 + 2 \times 2 \times 255 + 2 \times 3 \times 255 + 1 \times 4 \times 255}{(11 \times 255) + (13 \times 0)} \\ &= \frac{2295}{2805} \\ &\simeq 1 \end{aligned} \tag{2.7}$$

N.B.: in order to find the barycentre of the black pixels we changed for the calculation of the moments the value for white to 0 and the value for black to 255. The same has been done for all the values of the greyscale scale in case we don't have a binary image.

$$\begin{aligned} \bar{y_A} &= \frac{m_{01}}{m_{00}} \\ &= \frac{\sum_0^3 \sum_0^5 y f(x,y)}{\sum_0^3 \sum_0^5 f(x,y)} \\ &= \ldots \\ &= \frac{1 \times 1 \times 255 + 1 \times 2 \times 255 + 3 \times 3 \times 255 + 1 \times 4 \times 255 + 4 \times 5 \times 255}{(11 \times 255) + (13 \times 0)} \\ &= \frac{9180}{2805} \\ &\simeq 3 \end{aligned} \tag{2.8}$$

So we get for our greyscale barycentre of token (a) $B_A(x,y)$ the coordinates $B_A(1,3)$. The same way we get the coordinates for our token (b) $B_B(1,4)$.

Now we can superimpose the two tokens and we get a pixel difference of 4 pixels. In figure 2.12 the greyscale barycentre is coloured in blue. In figure 2.12(c) you can see the tokens after superposition. The red pixels are the pixels that are not the same for the two tokens. Instead of 11 different pixels we find here only 4 different pixels.

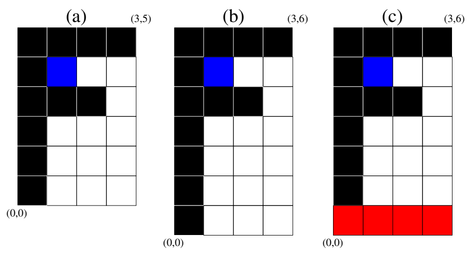


Figure 2.12: Example for superimposing two tokens

Comparison using moments

Instead of doing the pixelwise comparison, we also tried to compare the moments of the token to find out whether two tokens are similar or not. The order of the central moments we calculated was 2. So we calculated μ_{00}, μ_{01}, μ_{10} and μ_{11} for every token. As $\mu_{01} = 0$ and $\mu_{10} = 0$ we only needed to calculated $\mu_{00} = m_{00}$ and $\mu_{11} = m_{11} - \bar{x}m_{01} = m_{11} - \bar{y}m_{10}$. This calculations can be found in [GW02].

In this case, two tokens are considered similar if their vector of the two moments is similar. In our case this vector contains μ_{00} an d μ_{11}. The problem here is to define when these two vectors are similar and when not. We tried to set the similarity as a threshold: if a the number of differing elements is less than the threshold, they are considered similar. We set this threshold to 2. This means that the two elements of vector (a) have to be similar to the two elements of vector (b). Two elements of the vectors are considered similar, if their difference is less than a threshold T. This threshold is fixed by the user. It might be suitable to set this threshold more “intelligent” than by setting it to a fix value.

So in our case two vectors are considered similar if $abs(\mu_{01_a} - \mu_{01_b}) < T$ and if $abs(\mu_{11_a} - \mu_{11_b}) < T$.

Comparison using Fourier coefficients

The idea behind this method is, not to compare the surface distribution as in the moments method, but to consider the contour of the character, and then to compare values that have a relation with the contour, instead of comparing values that give us only information about the distribution of the pixels.

The idea is to consider the coordinates of the contour pixels of a token as a complex function $f(x, y) = x + \imath y$. Then this function is Fourier transformed in order to get the Fourier coefficients. These coefficients can be interpreted as information about the contour of the character. The token representing an “o” will probably have a high part of low frequencies, because there are not a lot of direction changes. So an “o” and an “O” are likely to have similar Fourier coefficients. This will not be the case for the token representing an “E”. Here a lot of high frequencies can be found due to the high frequent change of direction.

So in this method the Fourier coefficients are compared. As only the coefficients representing the low frequencies are interesting to get a rough impression of the contour, we only compare the coefficients representing low frequencies. If these are similar enough for two tokens, this two tokens are considered as being similar.

The first step is to find the coordinates of the boundary pixels. We used the Pavlidis Algorithm [Pav82a].

Pavlidis Algorithm The idea is quite simple: choose a starting point, and then walk in a fixed direction around the token until you are back at your starting point.

The first step is to choose a starting pixel. As this starting pixel should be the same for two identical tokens we chose the lowest leftmost pixel as starting pixel because it is unique in every token and it is the same for two equivalent tokens.

The easiest way to explain the algorithm is the following: imagine a little turtle being set on this first starting pixel. The aim of the algorithm is to let this turtle walk all the way around till it is back on the starting point.

Therefore we need first need to define the directions in which the turtle may look: because we only check for the four neighbours, we only need to have four directions. These are defined as in figure 2.13. Direction 0 points upwards. The other directions are given counter clock wise: direction 1 will point to the left of the token, direction 2 will point to the bottom of the token and direction 3 to the right side. These directions are relative to the orientation of the token, not to the orientation of the turtle. The direction in which our turtle is looking

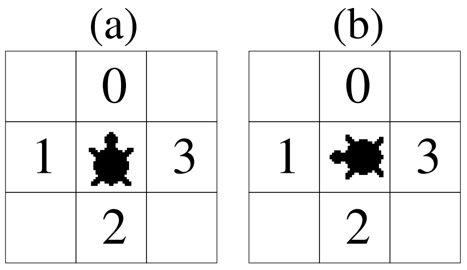


Figure 2.13: An example for a turtle looking in direction 0 (a), and one for looking in direction 1 (b)

at the beginning is fixed with 0. So the turtle looks up to the upper side of the token.

We then define the pixels that the turtle will have to check in this order: P1, P2 and P3. P1 is the pixel on the upper left side of our turtle, relative to the turtles orientation. P2 is the pixel in front of our turtle and P3 if the upper right pixel. P1, P2 and P3 are illustrated in figure 2.14. These pixels are chosen relative to the orientation of the turtle. The algorithm works as follows:

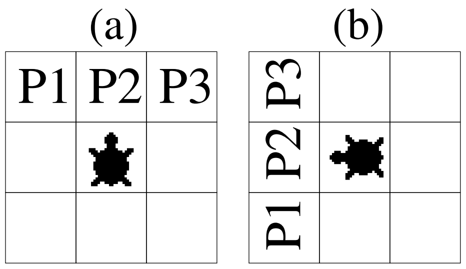


Figure 2.14: Example for the three different pixels that need to be checked for the Pavlidis algorithm

```
REPEAT
  IF P1 is a black pixel THEN go to pixel P1
  ELSE IF P2 is black THEN go to P2
  ELSE IF P3 is black THEN go to P3
  ELSE then turn 90 clockwise
  IF number of consecutive rotations = 4 THEN stop.
UNTIL stopped OR current position = starting position
```

To illustrate the algorithm an example can be found in 2.15. In (a) the turtle is in its starting position. It then moves to its upper left neighbour (b). From this position an direction no black pixel can be reached, so the turtle turns 90 clockwise (c). It then moves to P3 (d). These steps are repeated until it reaches the starting pixel again.
The algorithm will return a vector containing the coordinates of the pixels that are part of the boundary.

Fourier Transform As seen before, the Pavlidis algorithm returns us a vector of the coordinates of the boundary pixels. These pixels can be seen as pair $(x_0, y_0), (x_1, y_1), \ldots, (x_{k-1}, y_{k-1})$. Consider the function $s(k) = [x(k), y(k)]$ where $x(k) = x_k$ and $y(k) = y_k$. We then can consider $s(k)$ as complex function $s(k) = x(k) + \imath y(k)$.

The advantage of this method is to reduce a two dimensional problem to a one dimensional problem.

The Discrete Fourier Transform (DFT) is defined as follows:

$$a(u) = \frac{1}{K} \sum_{k=0}^{K-1} s(k) e^{-\imath 2\pi uk/K} \qquad (2.9)$$

The calculation of the DFT is done by the Fast Fourier Transform (FFT). An introduction about how the FFT works can be found in [Pav82b]. The algo-

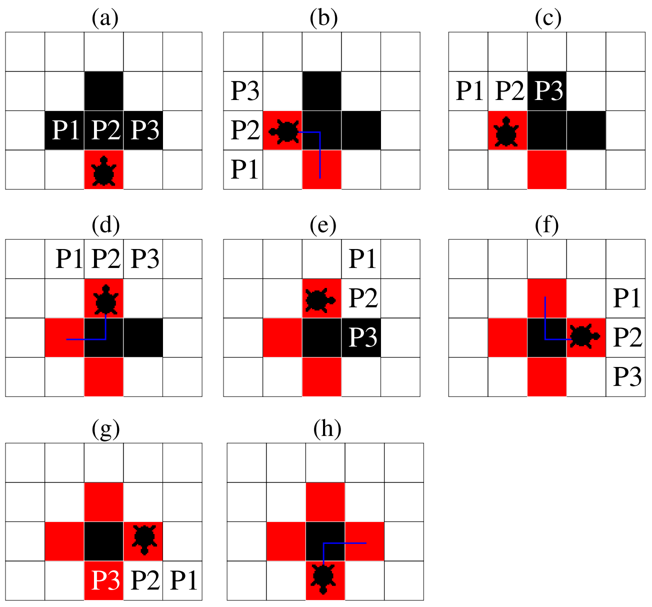


Figure 2.15: Example for the Pavlidis algorithm for extracting the contour of a token

rithm gives us as output a vector of all the Fourier coefficients of the current contour. The vector is sorted, so that the first entry is equal to $a(0)$, the second to $a(1)$, etc.

If we have to compare two tokens, t1 and t2, we first calculate the boundary of the two tokens. Then the boundaries are Fourier transformed. Now we have the Fourier coefficients of the two tokens and can compare these. We defined two coefficients as similar if the following condition is true: $|a_{t1}(k) - a_{t2}(k)| < 1$.

In our implemented version of the methods, we coupled the Fourier method with the pixelwise method. So the Fourier comparison has not to be too restrictive. That is why we defined two tokens as similar if 2 out of the 8 first coefficients are similar.

Results

The pixelwise method works well but has one big disadvantage: it needs a lot of time. For every pair of prototype and unanalysed token the difference has to be calculated. And the more prototypes we get, the more pixelwise comparisons have to be done. This is the reason why pixelwise comparison was not chosen as a first step to find out if two tokens are similar or not. However is was retained as second step after finding out with a first step that two tokens are potentially similar.

The moments method worked fast but not well enough. There were a lot of misclassifications. Also is it difficult to find a good method to decide whether two vectors of moments are similar or not. Due to the fact that the Fourier method worked better, without having to set a lot of parameters, this method was not withheld, although it might be possible to get better results if an adapted method for comparing the moments vectors is found.

The method comparing Fourier coefficients works fast if the number of coefficients to be compared is low. However, if we only compare a few number of coefficients the number of misclassified tokens increases. If we compare many coefficients, this number decreases, but the algorithm becomes slower.

Two steps method

This method is composed of two steps: the first one is a fast Fourier coefficients comparison method, the second one is the pixelwise comparison method.

```
get contour of token ;
do FFT with the contour and get vector of coefficients ;
int similar = 0 ;
FOR the n first four.coeff. DO
    IF both coefficients of the two tokens are similar THEN
        similar++ ;
IF (similar > k) THEN
   compare the two tokens pixelwise ;
   IF (number of different pixels < l) THEN
       two tokens are similar
   ELSE tokens are different
ELSE tokens are different
```

In our case $n = 8$ and $k = 2$ gave good results. The problem, that these values allow a lot of misclassifications for the Fourier method, is rectified by the pixelwise comparison. The value for l is can have two values in our case, depending on whether we have lossless compression l = 0.5 x min(number black pixels of the two tokens) or we have lossy compression l = 0.3 x min(number black pixels of the two tokens). The reason why we have two different values is to allow the lossless compression to misclassify some tokens. This misclassification is undone by the fact that we save the difference pixels in order to reconstruct the original image. In the case of lossy compression no misclassification is allowed, because the text will loose its readability if e.g. the “e” is replaced by a “c”.

2.4.3 Lossy/Lossless Compression

The aim of this part of the project was to implement the ability to compress an image without loss of information. This means that the original image had to be reconstructed without any change of any pixel.

For this reason the residual image has to be modified. The idea is to virtually reconstruct the image after compression, and to scan the original image and the virtually reconstructed pixelwise and to save the differing pixels in the residual image. So, if a pixel in the reconstructed image differs from the one in the original image, the concerning pixel in the residual image is set.

Instead of really reconstructing the image, we update the residual image by simply comparing the original tokens with the prototypes that will replace these tokens.

An example for how this works is given in figure 2.16. The blue pixels should be the greyscale barycentres, but in this case they are not. It is only used to create a simple example to illustrate our technique. Let (a) be the prototype and (b) be the token similar to the prototype. So the token (b) is substituted by token (a). To calculate the residual image we superpose the tokens so that the greyscale barycentres are superposed (blue pixel). As the tokens have not necessarily the same size, they usually don't overlap in all their points. Therefore, three cases have to be considered:

1st: we are somewhere in the overlapping area. If the pixel (x, y) of the original token differs from the pixel (x, y) of the prototype then the pixel $(origposX + x, origposY + y)$ of the residual image is set (it is set to 0, black), where origposX and origposY are the coordinates of the position of the original token in the original image. When we are reconstructing, we can check if the pixel of the residual image in this position is black and then we can set the resulting pixel to the inverse of the prototype pixel (black to white resp. white to black).

2nd: if we are in the area of the original token but outside the area of the prototype, then we simply copy the pixel from the original token to the residual image. These areas aren't checked while reconstructing. So we simply have to save all information that these areas contain.

3rd: if we are in the area of the prototype but outside the area of the original token: then we copy the value of the prototype pixel to the residual image. Here we have to save the pixels from the prototype in order to delete them during reconstruction.

In figure 2.16 the numbers in (d) indicates what case was used to determine the value of the pixel.

An example of a residual image can be found in figure 2.17.

2.4.4 Compression of multiple pages with one dictionary

The idea was to increase the compression rate by reutilising one dictionary a for compressing several pages. The hope was that the number of new prototypes for each page would tend to zero, the more pages we compressed, and that the only new information would be the positions informations and the residual images.

The first step to do is to read such a dictionary and to extract the tokens and save them as new prototypes (as they have already been identified as prototypes and because these prototypes are needed to reconstruct the images). In the second step the normal image would be read, the tokens extracted and the program would continue normally.

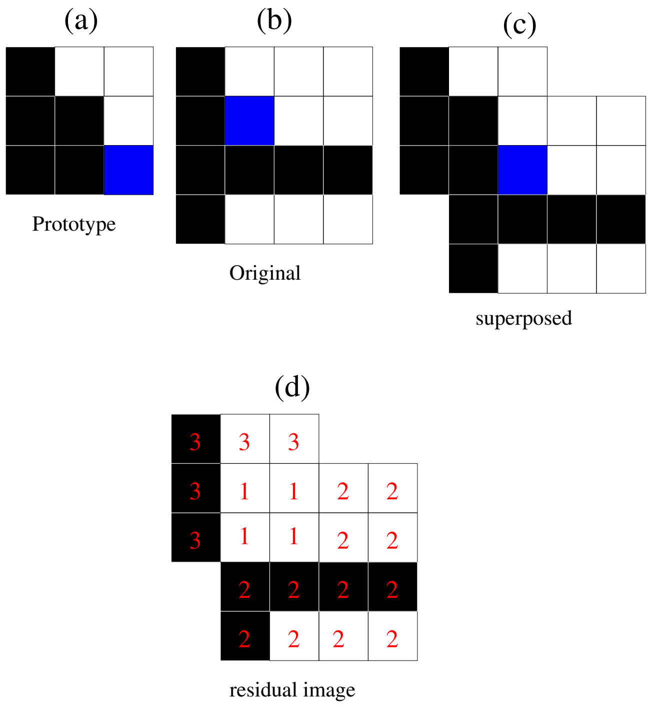


Figure 2.16: Example for updating the residual image

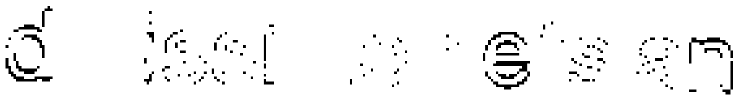


Figure 2.17: Example of a part of a residual image

2.4.5 Implementation Details

Saving of prototypes

In a first phase the prototypes were all saved in an image, all prototypes in one row, one after the other. In order to find the prototypes back we needed an additional text file containing the bounding boxes of the prototypes. Because of this and because of the fact that disk space is wasted due to the different heights of the prototypes, another format for saving the prototypes had to be found.

The format we used incorporates the information of the bounding boxes into the image of the prototypes: first the prototypes were sorted according to their height. Then an arbitrary width of the image is fixed, e.g. 1'000 pixels. Then we start to put the prototypes into the image, beginning with the smallest and ending with the highest. If the end of one row is reached, a new row is inserted and so on. In order to allow us to distinguish the different prototypes and in order to find back the rows, black control pixels are introduced. The leftmost pixel column of the prototypes image contains a column of control pixels. They mark the different rows of the prototype image. These rows contain also control pixels: these define the beginning and the end of each prototype. An example for this prototype image can be found in figure 2.18.

This figure is only a part of a complete prototype image in figure 2.19.

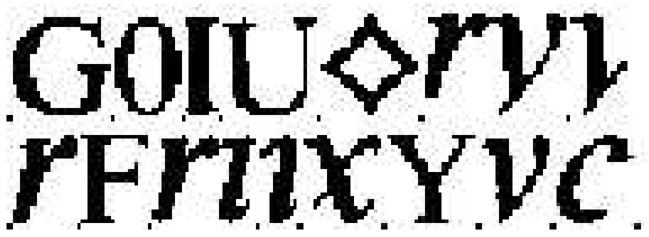


Figure 2.18: Example for the format of a prototype image

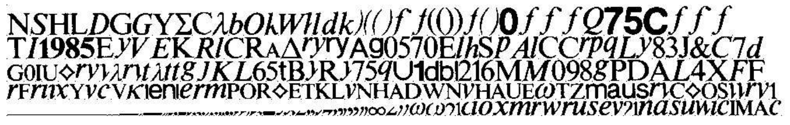


Figure 2.19: Example for prototype image

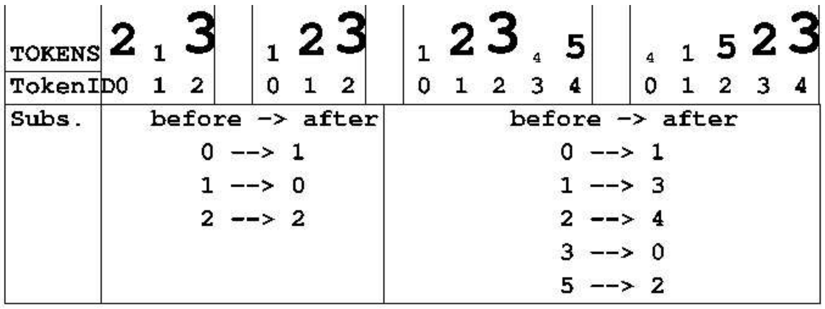
This method has one disadvantage: the original height of our token is lost, because all tokens in one row get the same height. In order to extract the original height of the prototype we have to scan the bounding box, given by the control pixels, for the highest black pixel.

Substitutions

For reconstruction, we need to know on what coordinates what prototype has to be put. This prototype is identified by a number. This number is assigned when it is created. The first prototype found gets the number 0, the second the number 1, etc. These numbers are not saved in the prototype image. It was intended to get the number out of the position in the prototype image. But now that the prototypes are first sorted and then saved, this method does not work any more.

Therefore we introduced a new text file containing the substitution(s). In the case that we don't use an old dictionary, this means that our dictionary is empty at the beginning of the similarities check, this file only contains a mapping from the old prototype number to the new number. This new number is the place of the prototype in the sorted prototype list.

In case we use one dictionary to compress more than one image, we need to be able to reconstruct the original number even after several sorting actions. That is the reason why for every compressed image, a new vector of entries are appended to the substitutions file. A small example for how this works can be found in figure 2.20.



TOKENS	2 1 3	1 2 3	1 2 3 4 5	4 1 5 2 3
TokenID	0 1 2	0 1 2	0 1 2 3 4	0 1 2 3 4
Subs.	before -> after		before -> after	
	0 --> 1		0 --> 1	
	1 --> 0		1 --> 3	
	2 --> 2		2 --> 4	
			3 --> 0	
			5 --> 2	

Figure 2.20: Example for the use of the substitutions information

Suppose that image "test1.pgm" has been compressed to "test1pos.txt", "prototypes1.pgm" and "substitutions.dat". In the second step the dictionary "prototypes1.pgm" is reused to compress the image "test2.pgm". Here we get the new dictionary "prototypes2.pgm", a new positions data file "test2pos.txt" and the substitutions file is extended with a few new entries.

We now want to reconstruct the image "test1.pgm" by using the "test1pos.txt" and the new dictionary "prototypes2.pgm". In order to find the original number of the token "3" we take the number of the token "3", in this case 4 and substitute it with its precedent number, namely 2. Then we go to the first substitution and replace the number 2 by it precedent number, in this case also 2. So 2 is the original number of token "3" in the first compression step. This works the same with more than two steps.

In order to distinguish the different entries in the substitutions file and to allow us to find what entries have to be composed, to get the original prototype number, we needed a unique identifier that tells us what image has been compressed with what dictionary. In case we are using an existing dictionary we save the name of the new dictionary (e.g. "prototypes2.pgm"), the name of the old dictionary ("prototypes1.pgm") and the name of the positions data file. In case we are not using an existing dictionary we save the name of the new dictionary file (prototypes image) twice and the name of the positions data file.

Compressing the output files

In order to compress the output files, a script creates a zipped file that contains all the output files needed for reconstruction. These files are the following:

- prototype image: this image is in the "pgm" format
- residual image: this image is compressed as a "TIFF G4" file
- positions data: a text file containing data needed for reconstruction
- substitutions data: a text file containing data needed to find the original unique number of a prototype (in case of compressing more than one image with one dictionary)

2.5 Reconstruction

2.5.1 General Description

The aim of this part is to reconstruct the original image (or an image that is quite similar to the original image) out of our saved data:

- Input:
  - residual image
  - a text file containing the positions data
  - substitution data

- Output:
  - the original image

The algorithm works as follows:

```
extract tokens from prototype image
read residual image
read positions data
read substitutions data
IF image compressed with reused dictionary THEN
    Compose different substitutions steps to one substitution ;
ELSE
    Read the one and only substitution step ;
FOR each entry in the positions data DO
    reconstruct on position x,y the original token by using
```

```
           the concerning prototype and the residual image ;
20 save the output image ;
```

2.5.2 Composition of the substitutions

As seen before, it may happen that a series of substitutions has to be composed in order to get the original number of the token as it has been saved in the positions text file. To illustrate how this works we will explain it with an example of a substitutions file, that can be found in table 2.1.

The starting point of the composition is given by the dictionary name we use to reconstruct the image, e.g. "pro03.pgm". The ending point is given by the positions data file name, e.g. "pos01.txt". What we need is the number of the prototypes as they were in dictionary "pro01.pgm", to reconstruct image 1. We then try to find a "way" from the beginning to the end. First we substitute the entries of the 2nd substitution by the ones of the 3rd substitution. Then we only need the step from "pro02.pgm" to "pro01.pgm". This step can be done by substituting the entries we got in the step before into the entries from the first substitution. So after the last step we have a mapping from the original number of a prototype in dictionary "pro01.pgm" to the number of this prototype in the new dictionary "pro03.pgm".

2.5.3 Lossless compression

If we are using lossy compression we simply put the right prototype on the right place. For lossless compression, we have to check the pixels of the residual image in order to merge the information of the prototype and the residual image to reconstruct the original token.

We have the following information: the place where the prototype has to be placed and the number of the prototype. Then we scan for all the prototype pixels the concerning pixels in the residual image. Three cases may happen:

1st: IF residual image in (x,y) is white THEN we simply copy the corresponding prototype pixel.

2nd: IF residual image in (x,y) is black AND IF concerning prototype pixel is white THEN save the pixel as black.

pro01.pgm pro01.pgm pos01.txt ...
pro02.pgm pro01.pgm pos02.txt ...
pro03.pgm pro02.pgm pos03.txt ...

Table 2.1: Example for the entries in the substitutions data file

3rd: IF residual image in (x,y) is black AND IF concerning prototype pixel is black THEN save the pixel as white.

Still there is one unsolved problem with this method: as seen before, it may happen that two or more boxes are overlapping. In this case it is not clear what residual information belongs to what token. This problem has not been solved in our method so there will still be differences between the reconstructed image and the original image. So this method is not lossless but only pseudo-lossless as there still exist pixels that differ from the original image. This problem can be avoided if the construction of the residual image would be done by comparing a reconstructed lossy image to its original image, and then saving the differing pixels.

Chapter 3

Results

The results were obtained on a Pentium4 2.5GHz System with 512MByte RAM. The test images are taken from the book [Ebb87], scanned with 300dpi. Thirty pages were compressed using the different methods.

We compared the size of our zipped file with the size of the image compressed with JPEG (least quality), with DjVu and zipped TIFF.

3.1 Lossy Compression with new dictionaries

Mean compression factor compared with JPEG	4.16
Mean compression factor compared with DjVu	0.32
Mean compression factor compared with zipped TIF	3.86

Table 3.1: Size of resulting ZIP-file compared to the size of different other image formats for lossy compression with new dictionaries

Mean time needed for similarity check	18s
Mean number of tokens per image	3713
Mean number of classes per image	496
Mean recognition factor	7.58

Table 3.2: Performance of the proposed method for lossy compression with new dictionaries

Mean size of compressed position data	21323 Bytes
Mean size of compressed prototyes image	20237 Bytes
Mean size of compressed residual image	614 Bytes
Mean size of compressed substitution data	895 Bytes

Table 3.3: Space needed for the different resulting files.

As one can see in table 3.2, the method needs quite a lot of time, nearly 20 seconds in average. Compared with JPEG in least quality (table 3.1), it creates files that are four times smaller and are better readable than the JPEG version of the image. But JPEG is not the best method for compressing bi-tone images of text. Compared to DjVu, our method gives a file that is three times bigger than the same image compressed with our method. For zipped TIFF the factor is also about four.

A conclusion that is quite straightforward is, that the compression rate increases if the recognition rate increases. This means that we have fewer prototypes and fewer prototypes means less space is needed to save them.

This is also the explanation why the method needs less time if the recognition rate is better: fewer prototypes are found and the new tokens have to be compared fewer times.

It is also clear that the size of the compressed residual image is very small (table 3.3). It contains only a few black lines that can be compressed well with standard compression techniques.

3.2 Lossless Compression with new dictionaries

Mean compression factor compared with JPEG	1.10
Mean compression factor compared with DjVu	0.21
Mean compression factor compared with zipped TIF	1.01

Table 3.4: Size of resulting ZIP-file compared to the size of different other image formats for lossless compression with new dictionaries

Mean time needed for similarity check	8s
Mean number of tokens per image	3713
Mean number of classes per image	274
Mean recognition factor	13.83

Table 3.5: Performance of the proposed method for lossless compression with new dictionaries

Mean size of compressed position data	20197 Bytes
Mean size of compressed prototyes image	11425 Bytes
Mean size of compressed residual image	130527 Bytes
Mean size of compressed substitution data	490 Bytes

Table 3.6: Space needed for the different resulting files.

In table 3.5 we can see, that the method was faster in the lossless case than in the lossy case. This is due to the increased recognition rate: the factor of new tokens divided by prototypes rose to 13.83. So there are fewer prototypes, and fewer prototypes mean that the program has to compare fewer tokens.

The explanation why we have an increased recognition rate is, that in this case the program is allowed to classify tokens wrongly, e.g. an “e” as a “c”. These misclassifications are undone by the residual image. So the size of the residual image increases and the size of the prototype images decreases. It is necessary to find an optimum between the number of misclassifications and the size of the residual image in order to minimise the output files.

The size of our output files increased dramatically. Our method gives approximately the same size as one would obtain with the compressed TIFF file (table 3.2). The DjVu files are in mean 5 times smaller than ours.

The problem with our method is the residual image: it contains a lot of black pixels, distributed all over the image. These can be compressed very difficultly. This can be seen in table 3.6: the mean size of the compressed residual image is about 10 times the mean size of the compressed prototypes image. Compared with the lossy compression, the size of the residual image increased about 200 times.

A method to decrease the number of residual pixels would be to choose an ideal prototype such that the difference between all the tokens that are similar to their prototype will be minimal. So the number of residual pixels produced by this prototype will be minimal, and the fewer lonely black pixels there are, the better this image may be compressed.

3.3 Lossy Compression with re-used dictionary

Here one dictionary file is used for 10 images. This has been done twice, with two sets of 10 images, in total with 20 images. The size for JPEG, ZIP TIFF and DjVu is calculated by summing up the space needed for each single file. This is compared with our compressed file containing the information for all the 10 files. The time and numbers of tokens and number of prototypes are absolute values, and no mean values.

	Set 1	Set 2
Compression factor compared with JPEG	6.74	5.54
Compression factor compared with DjVu	0.55	0.42
Compression factor compared with zipped TIF	6.32	5.16

Table 3.7: Size of resulting ZIP-file compared to the size of different other image formats for lossy compression with re-used dictionaries

In table 3.8 we can see, that the recognition rate is higher than in the lossy case without re-used dictionary, and its compression rate is also better (table 3.3). So the re-use of a dictionary has had the impact we had planned, namely to increase the recognition rate and to increase the compression rate. But compared to the time needed to compress this 10 images, this improvement is not

	Set 1	Set 2
Total time needed for similarity checks	13m44s	13m26s
Total number of tokens	37355	38062
Total number of classes	1675	1548
Recognition factor	22.3	24.59

Table 3.8: Performance of the proposed method for lossy compression with re-used dictionaries

	Set 1	Set 2
Size of compressed position data files	251860 Bytes	251593 Bytes
Size of compressed prototyes image	67159 Bytes	61876 Bytes
Size of compressed residual image files	6259 Bytes	6145 Bytes
Size of compressed substitution data	7689 Bytes	7036 Bytes

Table 3.9: Space needed for the different resulting files.

relative to computational effort. It is simply too slow. The compression rate is even better than the rate obtained by compressing the images one by one. But it is still no match for DjVu, even if we simply sum up the size of the single compressed files DjVu files.

We can see here, that in the lossy case, the space needed by the residual images is nearly neglectable (table 3.9). Even the substitution file is larger than all the residual images.

3.4 Lossless Compression with re-used dictionary

Here one dictionary file is used for 10 images. This has been done twice, with two sets of 10 images, in total with 20 images. The size for JPEG, ZIP TIFF and DjVu is calculated by summing up the space needed for each single file. This is compared with our compressed file containing the information for all the 10 files. The time and numbers of tokens and number of prototypes are absolute values, and no mean values.

	Set 1	Set 2
Compression factor compared with JPEG	1.07	1.08
Compression factor compared with DjVu	0.21	0.20
Compression factor compared with zipped TIF	1.00	1.00

Table 3.10: Size of resulting ZIP-file compared to the size of different other image formats for lossless compression with re-used dictionaries

We can see in table 3.11, that even in this case our compression rate is not good, although the recognition rate for ten images is very high, due to a probably large number of misclassifications. Zipped TIFF and JPEG are,

	Set 1	Set 2
Total time needed for similarity checks	4m32s	6m37s
Total number of tokens	37355	38062
Total number of classes	777	740
Recognition factor	48.08	51.44

Table 3.11: Performance of the proposed method for lossless compression with re-used dictionaries

	Set 1	Set 2
Size of compressed position data files	223529 Bytes	225030 Bytes
Size of compressed prototyes image	30292 Bytes	28161 Bytes
Size of compressed residual image files	1423917 Bytes	1428918 Bytes
Size of compressed substitution data	3695 Bytes	3405 Bytes

Table 3.12: Space needed for the different resulting files.

compressed each file for its own, nearly as big as the file we get with our method (table 3.4). This is due to the residual image that is too large (table 3.12). As one can see, the major part of the space needed is used by the residual images. Compared with the sum of the sizes of the single DjVu Files this method brings no advantage for the compression rate.

Chapter 4

Conclusion & Open Problems

In this project thesis a token-based compression method has been implemented. The first step, consisting of binarization, has been done with an adaptive local thresholding algorithm. Then the connected components have been identified by a labelling method that uses a union-find data structure. After this, a naive noise reduction step deletes the boxes smaller than a certain threshold.

The main step consists of grouping the tokens together, so that similar tokens are in the same class. Pixelwise comparison, moments comparison and Fourier coefficients comparison of the boundary pixels have been tried. A combination of Fourier coefficients comparison and pixelwise comparison has been used.

The method allows to compress lossy and lossless. It also has the ability to re-use a dictionary (an image file containing prototypes in a certain format) in order to compress a number of pages with the same dictionary.

The last step is the reconstruction of the image out of the files created in the compression step.

Compared to JPEG an compressed TIFF our method has in some cases a better compression rate, although it does not reach the efficiency of the DjVu format .

One problem is the size of the residual image for lossless compression. It gets too big due to the fact that the choice of our prototype is not ideal. More work should be done to find a way to optimise the choice of the prototype. Maybe it would be useful to save a token as prototype that does not exist in the image but that has the least distance to all tokens similar to it.

In addition, the different parameters may be chosen in a more intelligent way. Especially the parameter of the allows pixel difference has in important role. It should be chosen to minimise the total size of the residual image and the size of the prototype image.

An other problem is the time the method needs: even though the Fourier method works faster than only comparing the tokens pixelwise, it has the problem that the larger the number of prototypes gets, the more time it need, because it has to compare every unknown token with all the prototypes. This also explains why the method reusing dictionaries works so slowly.

Bibliography

[AN74] R.N. Ascher and G. Nagy. *A Means for Achieving High Degree of Compaction on Scan-Digitized Printed Text*, volume C-23, No. 11, pages 1174–1179. November 1974.

[BHH^{+}98] L. Bottou, P. Haffner, P. Howard, P. Simard, Y. Bengio, and Y. LeCun. High quality document image compression with DjVu. *Journal of Electronic Imaging*, Vol. 7(No. 3):410–425, July 1998.

[Dav97] E. Roy Davies. *Machine Vision: theory, algorithms, practicalities*, page 97ff. Academic Press, San Diego, 2nd edition, 1997.

[Ebb87] Heinz-Dieter Ebbinghaus. *Ω-Bibliography of Mathematical Logic*, volume 3. Springer Verlag, Berlin, 1987.

[GW02] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Prentice Hall, Upper Saddle River, NJ, 2nd edition, 2002.

[Pav82a] Theo Pavlidis. *Algorithms for Graphics and Image Processing*, page 142ff. Springer Verlag, Berlin, 1982.

[Pav82b] Theo Pavlidis. *Algorithms for Graphics and Image Processing*, page 28ff. Springer Verlag, Berlin, 1982.

[Rei] Reichelt. Reichelt Katalog 2005/1.