RECOGNITION OF MATHEMATICAL HANDWRITING ON WHITEBOARDS

by

BEHRANG SABEGHI SAROUI

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
August 2015

Abstract

Automatic recognition of handwritten mathematics has enjoyed significant improvements in the past decades. In particular, online recognition of mathematical formulae has seen a number of important advancements both for pen input devices as well as for smart boards. However, in reality most mathematics is still taught and developed on regular whiteboards and offline recognition remains an open and challenging task in this area.

In this thesis we develop methods to recognise mathematics from static images of handwritten expressions on whiteboards, while leveraging the strength of online recognition systems by transforming offline data into online information. Our approach is based on *trajectory recovery* techniques, that allow us to reconstruct the actual stroke information necessary for online recognition. To this end we develop a novel recognition process especially designed to deal with whiteboards by prudently extracting information from colour images. In particular, we have applied an innovative combination of preprocessing techniques to clean images. For segmentation of characters we have experimented both with techniques from the literature and a novel approach that employs a stroke reconstruction technique based on rigorous mathematical definitions of stroke artefacts. Finally, we have developed and compared three methods to reconstruct characters from recognised strokes: one exploiting colour information, one based on search and combination of both as a heuristic search technique.

To evaluate our methods we use an online recogniser for the recognition task, which is specifically trained for recognition of maths symbols. We not only present a comprehensive comparison of different segmentation and reconstruction techniques, but also experiments varying the quality and sources of images. In particular, we have used our approach successfully in a set of experiments using Google Glass for capturing images from whiteboards, in which we achieve highest accuracies of 88.03% and 84.54% for segmentation and recognition of mathematical symbols respectively.

Acknowledgements

First and foremost, I would like to express my deepest appreciation to my supervisor, Dr. Volker Sorge, who has been a tremendous supervisor as well as a friend, for encouraging my research and helping me to grow as a scientist.

I am indebted to Dr. Behzad Bordbar, Prof. Achim Jung and Dr. Mark Lee for their aspiring guidance and help throughout, and Dr. and Mrs. Ramsay for their financial support at difficult times.

In addition, I thank Dr. Stephen Watt, Prof. Peter Tino and Alan Sexton for their invaluably constructive criticism and Dr. Ian Styles for providing me with facilities when needed.

I am grateful to Yongqin for her love and support, and all my friends, whom unfortunately I could not fit their names in this short note, for the invaluable discussions and fun that we have had in the past few years. I am certain that you would know you are one of them when reading this note.

Last but not least, a special thanks and my eternal gratitude to my family. Words cannot express how grateful I am to to my parents, Issa and Shokoufeh, and also my brother Bobak, for their endless love and support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

Over many centuries handwriting was the prevalent, if not only, means of communication and preserving knowledge. Even today, handwriting has not yet been fully supplanted by electronic forms of communication, and in many areas of life much information is still communicated in handwritten form. Automatic handwriting recognition aims at transforming this information into a processable electronic format, which gives us the opportunity to store, organise, retrieve and analyse knowledge electronically. Automatic recognition of handwriting allows easier access to an increased amount of information, making it available not only to millions of users but also amenable to electronic search, data mining, etc. In addition, having handwritten information available in processable electronic formats enables easy transformation into formats that are beneficial to user communities that would otherwise be excluded from that knowledge, for example, by making information available to visually impaired readers via screen reading systems. Consequently, handwriting recognition has attracted much attention by researchers over the past few decades and in the light of the prevalence of hand-held devices like smart phones and tablets that enable easy handwritten input, we envision the research area to become even more important in the future.

The motivation behind this research is to support the scientific community in tackling the automation of transformation of mathematical handwritten images into processable formats. As well as the general benefits of handwriting recognition systems, this can have

other significant impacts for science and engineering fields that have maths as a core part, such as automatic generation of lecture notes. In regards to accessibility, such systems can also be used as teaching and learning real-time tools for the visually impaired.

Traditionally, handwriting recognition is divided into two main categories; online and offline. In online recognition a digitised surface, such as a PDA or a tablet, takes samples of the movements of a stylus at constant intervals that contain information on pen placement, speed, angle, pressure, etc. In contrast, in offline recognition all that is available to the recogniser is essentially a static two dimensional image, either taken from a scanned document or photographed by a digital camera. Research in the area has proven that the extra information available to online recognition systems provides a clear advantage over offline recognition. Consequently, the accuracy of offline recognisers is generally lower than that of their online counterpart.

Much research has been carried out on online handwriting recognition and successful technologies such as tablets and smart-boards have been introduced, which allow another form of input to computers rather than the old-fashioned methods, such as keyboard and mouse. On the other hand, offline recognition systems have been less successful in comparison although they possess a much broader application domain. Moreover, it feels much more natural to write on normal paper with pens of your choice, not to mention cheaper, if one could transform the writing into a processable format.

In spite of considerable advances in analysis and recognition of handwriting, recognition of mathematics is still in its early stages, especially in the offline form. Mathematics is one of the oldest studies and subsequently there exist tremendous amounts of mathematical notes (handwritten and otherwise), which adds great importance to automatic recognition of mathematics due to the fact that it improves usability of scientific documents by allowing the retrieval, and generally more efficient analysis, of maths content through search engines and other services.

The recognition of handwritten mathematics is generally considered a difficult problem due to the two dimensional nature of mathematical notation. Nevertheless significant

advances have been made in recent years in online recognition of mathematical formulae. And while there have been some attempts at recognising handwritten mathematical documents from static images, as mentioned above, most of the research in that area in recent years has concentrated on online recognition, where detailed information on strokes can be taken into account.

In this thesis we focus on the offline recognition of handwritten mathematics on whiteboards. In addition, we investigate alternative approaches in various stages of the recognition process and evaluate each approach as necessary. In particular, we demonstrate an alternative approach to traditional offline handwriting recognition and transform offline data to be recognised by an online recogniser.

There have indeed been some attempts on the recognition of handwriting on whiteboards and smart-boards before (cf. [LB07, LB08, GLF+09]), however, in those experiments, online data is captured by a device attached to the whiteboard and then transformed to offline data to be passed to an offline recogniser. These systems achieve recognition accuracies of 61.4% for offline words and, 79.7% and 88.5% for online words and characters respectively. We effectively take the opposite approach: We use images of whiteboard content to recover the original stroke patterns and feed the results to an online recogniser for the symbol identification task. Moreover, in another study [SSR08], once again online data is captured and Hidden Markov Models are used to recognise the handwritings. This system has highest accuracies of 64.6% and 66.1% for characters and words respectively. Overall, neither the systems that we know of at this time work with offline static images from whiteboards, nor with mathematical notations. Please note, hereafter, the distinction between the terms "whiteboard" and "smart-board" throughout this thesis: the term smart-board is only used if there is a digitised mechanism taking samples of the movement of the pen while the note is being written.

Furthermore, most mathematical recognition systems at this time work with online data and they also restrict the number of strokes to a maximum of four. Plus, they can only deal with consecutive strokes. In this thesis we try to address these issues.

In the course of this research we have published a paper in DAS proceedings [SS14], which presents our maths character recognition methods through the *colour-driven* reconstruction method and another has been accepted and is to be presented at ICDAR [SS15], which introduces our *brute-force* and *informed* reconstruction methods. In the next section we describe how this thesis is laid out.

1.1 Overview of Thesis

Part I is an overview of related techniques, algorithms and research to the work in this thesis and handwriting recognition in general. More precisely, Chapter 2 is a review of both online and offline handwriting recognition and section 2.4, in particular, looks at the state of the art approaches taken in recognition of handwritten mathematics.

Part II presents and evaluates our approach to recognition of handwritten mathematical characters. Chapter 3 demonstrates the overview of our approach together with the techniques used to transform offline data to be recognised by an online recogniser. These techniques are broken down to three steps: preprocessing, local examination and global reconstruction. We mainly discuss the first two steps in that chapter. Then in Chapters 4, 5 and 6 we introduce three different methods for global reconstruction and evaluate our character recognition method with each of these approaches.

In Part III the focus is on segmentation of mathematical expressions. Chapter 7 presents our segmentation method which is based on the average distance between characters in expressions. In Chapter 8 another segmentation method based on Dynamic Programming is discussed. Then, in Chapter 9 we present a case study and evaluate our segmentation methods and further evaluate our character recognition methods with a lower resolution camera.

Finally, Part IV is the conclusions of this thesis; with Chapter 10 outlining the contributions of this thesis and Chapter 11 looking at how our work can be extended and improved.

Part I

Background

CHAPTER 2

RECOGNITION OF HANDWRITING

Over the past decades, research on text and handwriting recognition has made significant progress. This research was initially motivated by application areas such as automated postal address recognition, car number plate recognition, forensic document examination and signature verification, and over the years has found numerous new applications; so far that commercial handwriting recognition systems are now available in the market. These systems perform impressively for recognition of printed text and online recognition of handwritten characters – and words to a certain degree. However, regardless of the impressive progress in these areas, offline recognition of handwriting and particularly recognition of mathematics still remains a challenging task.

But what is Handwriting Recognition? Plamondon and Srihari [PS00] formulate it as "*the task of transforming a language represented in its spatial form of graphical marks into its symbolic representation*". They also go on to state that the symbolic representation is typically the ASCII or Unicode representation of characters. As briefly mentioned in the previous chapter, handwriting recognition systems are generally divided into two main categories:

- Online handwriting recognition systems
- Offline handwriting recognition systems

In the next few sections, we will discuss these systems, and some common techniques that are used within them, in more detail.

2.1 Overview

In online handwriting recognition, data are collected while they are being generated on a digitising surface (e.g. Tablet PCs). In addition to the spatial positions indicated by the tip of the writing tool on the surface, the data may also include velocity and acceleration of the stylus' movement, angle at which the stylus is held, pressure of the pen on the surface and others. Based on this information, the recognition algorithm then infers the written characters or words in real time [Imp94, LCB03].

However, in offline handwriting recognition, all which is available to the recognition system is the two-dimensional spatial information, e.g. the image of the address scanned from an envelope or an amount shown on a cheque [LCB03] (see Figure 2.1). Naturally, the lack of dynamic information has made offline handwriting recognition a more difficult task than online.

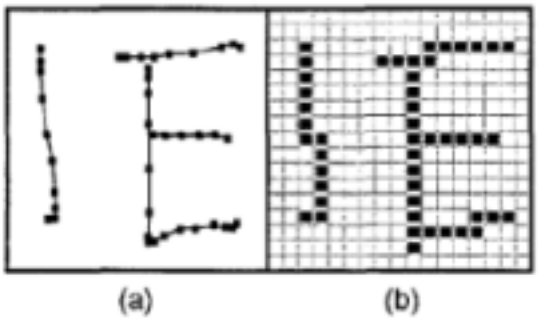


Figure 2.1: Online and offline data. (a) Online data consist of a sequence of time-ordered coordinate points, the writing trace is known. (b) Offline data consist of space-ordered pixel image data, the original time information is not available. [SSE96]

Comparing handwriting recognition systems – whether online or offline – is generally not an easy task. This is predominantly due to the fact that to be able to directly compare systems, you would need to test the systems on the same database; with the same lexicon size, writers, writing tools, etc. which is not the case all the time. The other factor that makes this comparison difficult is that systems are usually designed and built with a specific purpose. For instance, only in the case of offline recognisers systems deal with different problems such as different languages, handwritten versus machine-printed,

word versus character, forms and diagrams versus text versus mathematics, and they even extend to the writing surface i.e. paper versus whiteboard and others.[1]

However, what is easy to see is that systems that deal with machine-printed text can outperform the ones dealing with handwriting. These systems, widely regarded as OCR or Optical Character Recognition, have managed to reach accuracy levels of 99% for many languages such as English [CMUV97]. In addition, online systems also generally perform better than their offline counterparts [PS00]. Online recognisers have long managed to achieve results of over $90 - 95\%$ [GLF+09, Oh01, TSW90], while only offline recognisers that deal with a very limited number of lexicons have been nearly as successful [Pan12, PSH11, KSS03, Vin02].

But, many methods and techniques have been introduced to help tackle this complicated task, such as, Feature Extraction Methods and Pattern Recognition Methods. Although these methods have made offline handwriting recognition systems more accurate, offline recognition of handwriting still remains an open and challenging task.

2.2 Preprocessing and Feature Extraction

Although various methods are used in recognition of handwriting by different researchers and depending on their needs, some methods seem to be more common than others in the literature. In this section we aim to summarise some of the most common techniques.

Generally, it has been accepted as a de facto standard that the data firstly need to be prepared for further processing and this step is referred to as *preprocessing*. Preprocessing and feature extraction techniques are exceptionally important in handwriting recognition and optical character recognition (OCR) systems [Lee99] and are believed to be necessary in order to achieve high recognition accuracies. There are various types of features that can be used to recognise handwritten words and characters. But, one important factor to remember is that good features should enable the system to discriminate different classes

[1] No system yet exists that can deal with all or even some of these types together.

effectively, to reduce redundancy in representation and be robust to noise and deformation [LCB03]. Feature extraction is commonly performed on characters or words. There are a number of techniques that stand out in the literature and we will have a look at some of these techniques in the next few sections.

Generally, feature extraction for offline handwriting recognition systems is affected by the following factors:

- Different backgrounds of documents
- Noise introduced by electronics (cameras, scanners etc.) and writing tools
- Different qualities and colour of writing surfaces, and types of pens and ink.

Moreover, the feature extraction techniques in offline handwriting recognition are somewhat dependant on the type of image used by the system. There are three image types that are mainly used in handwriting recognition systems:

- Binary images
- Grey-scale images
- Colour images

Each of these types have their advantages and disadvantages. To put in a nutshell, the main disadvantage of binary and grey-scale images is that a great deal of information is eradicated (especially in binary images), which can be valuable in the recognition process. However, it is much faster to process these image types. In contrast, colour images contain too much data and it can be difficult to extract information from them if not handled properly, but there is much more information available.

2.2.1 Binary Images

A binary image is a digital image that has only two possible values for each pixel, i.e. zero and one. Although any colour can be chosen as the two possible colours, black and white

have become the two colours used in typical binary images and in most cases black as the foreground and white as the background colour. Therefore, it is much easier and faster to perform mathematical operations on a binary image as opposed to grey-scale and colour images. In addition, there are already many good feature extraction algorithms available for binary images.

Binary images are very useful for segmentation and tilt correction processes through *projection histograms*. There are several feature extraction methods available for binary images, to name a few, Projection Histograms, Zoning Templates, Moment Invariants, Geometric Moments, Template Matching and others. In the rest of this section we will discuss some of the most common feature extraction techniques used for binary images.

Projection Histograms

Projection histograms are mostly used for segmenting characters, words, and text lines. Since they are very sensitive to rotation, they are also used to detect whether or not an input image of a scanned text page is rotated. It is worth mentioning that the vertical projection is slant invariant, however, the horizontal projection is not. (See Figures 2.2 and 2.3.)

Given the Boolean foreground mask $F(x, y)$, the projection histograms θ (vertical) and π (horizontal) can be mathematically defined as: [VBC10]

$$\theta(x) = \sum_{y=0}^{F_y} \phi(F(x, y)) \qquad \pi(y) = \sum_{x=0}^{F_x} \phi(F(x, y)) \tag{2.1}$$

Where the function ϕ is equal to 1 if $F(x, y)$ is true and equal to 0 otherwise, while F_x and F_y are the width and the height of the foreground mask F respectively.

Figure 2.2: Horizontal and vertical *projection histograms* [TJT95]

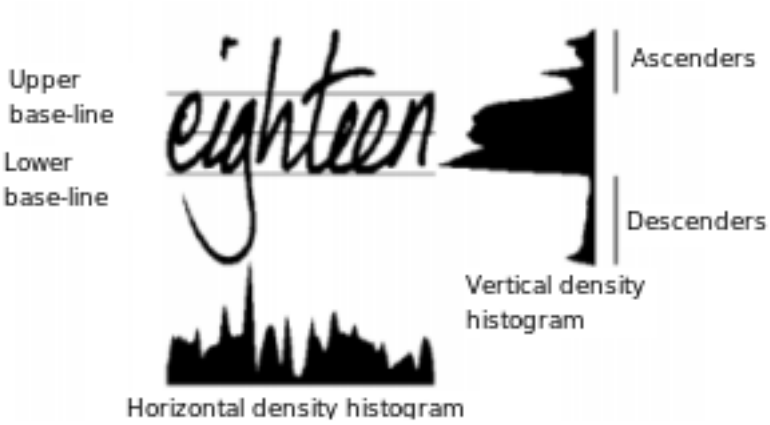


Figure 2.3: *Projection histograms* and baselines of a word [SR98]

Another common and very useful technique which is mainly used to reduce the amount of resources required to describe a large set of data and is performed on binary images is *Skeletonisation* as we will describe next.

Skeletonisation (Thinning)

Thinning or *skeletonisation* has been very popular in the previously published literature since you can obtain thin-line representation of images for data compression purposes. Thin-line representations are more flexible to extracting critical features such as end points, junction points, and connections between the components. Generally, thinning algorithms fall into three different categories:

1. Sequential Thinning

2. Parallel Thinning

3. Non-iterative Thinning

Algorithms that belong to the first two types produce skeletons by examination and deletion of contour pixels. On the other hand, the third family of algorithms is non-pixel based; they produce a median line of the pattern directly in one pass without the need to examine every individual pixel.

Moreover, *sequential* algorithms examine the boundary pixels of objects in an image and if they satisfy specific conditions, the pixels are removed. In this method the process is sequential, meaning, the pixels are checked and removed one at a time. Therefore, the removal of a pixel in an iteration can essentially depend on the other already processed pixels in the same iteration as well as the result of the previous iteration. Note that in this method visiting pixels in different orders might yield skeletons that are not identical. In contrast, *parallel* algorithms remove all pixels – that meet the deletion criteria – in one iteration at once and hence, the removal of a pixel in an iteration only depends on the result of the previous iteration. Therefore, visiting pixels in different orders does not vary the results of the process. The third category, *non-iterative* algorithms, have been suggested to be the closest to how human beings would perform thinning [Bar88] and the simplest category of non-iterative algorithms would scan lines and connect the mid-points of each line to form a skeleton. Although, these methods are computationally efficient, they also have a disadvantage that they might produce noisy branches in certain situations, especially when a median line cannot be obtained. We will discuss thinning in more detail later in Chapter 3.

2.2.2 Gray-Scale Images

The range of measured values of monochromatic light from black to white is usually called the grey-scale, and monochromatic images are frequently referred to as grey-scale

images [GW08]. Many feature extraction methods that are widely used in binary images can also be utilised for grey-scale images with minor modifications. These methods include *Zoning Templates*, *Geometric Moments*, *Template Matching* and *Unitary Transforms*.

As [LCB03] report: "*Binarisation by any method is a lossy operation. When binarisation is performed on grey-scale images by thresholding, especially at the early stage of the recognition process, it may lead to the loss of some significant information. Adaptive thresholding can be used to reduce information loss, in which the threshold level of binarisation needs to be continuously adjusted across the image to adapt the changing intensity and illumination. However, no matter how good the binarisation methods are, they often produce broken and touching line-segments*". Therefore, they believe that extracting features from grey-scale images can overcome some of the disadvantages caused by binarisation.

2.2.3 Colour Images

Colour images are predominantly left out in most handwriting recognition related research. This is mostly due to the fact that colour image processing is time and space consuming [CKLS07]. However, they can be very useful and certain features are lost when images are transformed from colour to either binary or grey-scale. We will see later in Chapter 4 how the information in colour images can be successfully utilised.

White Balancing and Colour Normalisation

White balancing and colour normalisation techniques are amongst the most common methods that are frequently utilised in image processing to preserve independence from illuminant variations. When a photo is taken by a digital camera, the values of the pixels in the photo depend on the response of 3 sensors which are affected by the illumination of the surroundings and a distinct colour cast presents itself over the captured photo, which is due to the colour temperature of the light source [CF06]. The lower the temperature

of this light source the more red a white object would appear in the photo. In similar fashion, illuminate a white object with a high colour temperature light source and the object will appear bluish in the photo. *White Balancing* is the process of removing this sort of unrealistic colour cast. There are many white balancing algorithms in the literature such as *Gray World Method*, *Perfect Reflector Method*, *Fuzzy Rules Method*, *Chiou's White Balance Method* and others. For more information about white balancing see [CF06, LCC95, WCF05].

Colour normalisation is another technique used to eradicate the effect of the illuminant on the image. This technique is usually performed on pixels or colour channels, but the combination of the two methods is known as *comprehensive normalisation*, the details of which are discussed in [FSC98, VLP+01]. Note that, colour normalisation should not be confused with image normalisation, which is the task of normalising the size of the character images [SLF95].

Edge Detection

Until recently, edge detection was almost entirely ignored in preprocessing of handwriting, nevertheless, edge detection process is extremely efficient for reducing the amount of available data and therefore simplifying the analysis of image. Moreover, this process maintains the original structure of the object boundaries at the same time.

More generally, edge detection is a fundamental tool in image processing particularly in feature detection and extraction. Edge detection aims at identifying points at which the image has discontinuities.

There are several edge detection algorithms in the literature such as Canny's [Can86] and Lindeberg's [Lin96] methods. In Figure 2.4 you can see how edge detection can reduce the amount of data to be processed while keeping the useful structural information.

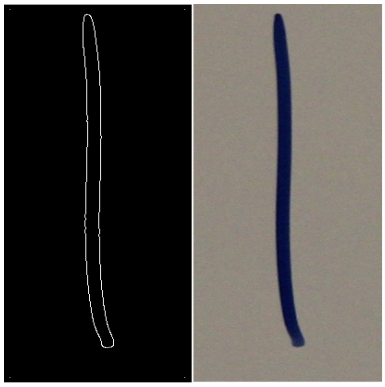


Figure 2.4: *Edge detection* example with Canny's method

According to John Canny [Can86], an optimal edge detection algorithm should satisfy the following three criteria:

1. Good detection:

   There should be a low probability of failing to mark real edge points, and low probability of false marking non-edge point.

2. Good localisation:

   The points marked as edge points by the operator should be as close as possible to the centre of the true edge.

3. One response per edge:

   A given edge in the image should only be marked once, and image noise should not create false edges.

Therefore, edge detection is a very effective method to reduce the amount of information, especially in colour images. In addition, it can also help to reduce the noise in the image.

2.3 Recognition Methodologies

Recognition of writing has come a long way and various methods have been tried, many of which have had at least satisfactory results. Moreover, systems are usually designed

for isolated characters [NT03] or unconstrained words [BS89, BLC97, SR98]. However, in online [Lee99], offline – handwriting [VGLK05, LCB03, NT03] or text [EBCBGMZM11] – and a combination of both [SSE96], Hidden Markov Models (HMM) seem to be a prevalent and successful technique. In general, HMMs allow easy statistical and stochastic modelling of temporal evolution of single or a set of numeric features which is why they are also widely used in other types of recognition such as action recognition [VBC10]. Markov Models are outside the scope of this thesis, but a survey of MMs for offline handwriting recognition can be found in [PF09].

In spite of that, many researchers believe that a better way to tackle the problem of offline recognition is by closing the gap between online and offline systems via recovering temporal clues [DR92, DR95] from the static images and trajectory recovery techniques [QNY06, KY00, L'H99, NV, RAC05, Jag98, QY06], and problems such as recovery of hidden loops have also been considered [DIRS02], however, no research that we know of on trajectory recovery has focused on mathematical symbols.

The predominant idea behind Trajectory Recovery (TR) is to extract the original path that the pen followed on the surface (i.e. paper, whiteboard, etc.) from the image of the writing. It is believed that automatic recognition of offline handwriting could be improved significantly if the trajectory of the writings can be extracted thoroughly [NB10]. TR is widely divided into three main parts in the literature: *Preprocessing*, *Local Examination* and *Global Reconstruction*. *Preprocessing* usually involves tasks such as binarisation, grey-scale conversion, noise removal and others; the purpose of which is to prepare the image for further processing. In the *local examination* stage, the handwriting *skeleton* or *contour* is computed and then used to detect ambiguities or *ambiguous zones*. These ambiguities include *double-traced* writing and *hidden loops*. We will discuss *local examination* and TR in general in more depth throughout this thesis and particularly in Chapters 3, 4, 5 and 6. In the final stage of TR (*global reconstruction*), the ambiguities that cannot be addressed locally are considered and solved, which can include identification of the direction and chronological ordering of the strokes.

On a different note, it needs to be mentioned that the applications of offline handwriting recognition are much greater than online and there are areas which only exist in the offline form such as historical document recognition [LRM04], forensic sciences/ signature verification [Fra08] and word spotting [RM02, RM03]. In addition, there are problems that also do not necessarily exist, or are essentially much easier to solve due to the one dimensional nature of data, in the online form such as segmentation [DW92], text line extraction [SES11] and even from psychological points of view how humans can perform the recognition task so well [BF88], which make offline handwriting recognition a broader area of research.

2.4 Recognition of Mathematics

Research into the online recognition of mathematical formulae has enjoyed much attention over the past decade mainly due to the important role it plays in transcribing documents in scientific and engineering disciplines into the electronic format [CY00] and recognition systems for both pen-based input on tablets or hand-held devices [Hu13, HW13a, GC04] as well as for smart boards [TR03, SW08, SSR08] have made significant improvements. In particular, the latter can have significant input on how mathematics is recognised and digitised in teaching and learning environments. However, in reality smart boards are not that prevalent to be found in every class room or mathematician's office and therefore most mathematics is still taught and developed on regular whiteboards (or blackboards). And enabling digitisation by recognising this content is still a challenging task.

Recognition of mathematics is a burdensome task due to many factors. As a matter of course, all problems of handwriting recognition, such as noisy input and others that we mentioned previously, are inherited naturally, but it extends much beyond that [Wat10]; maths notation can contain many small symbols that are difficult to distinguish from noise. Therefore, preprocessing and segmentation of maths notation can be an intimidating procedure. In addition, more ambiguities arise in symbol recognition of mathematical

content. This is, on one hand, due to the large character set in mathematics, and on the other hand, because of the ambiguities in the role of many of the symbols. For example, in handwritten maths it can be difficult to distinguish between "1", "l" and "|" or "9" and "q" or "0", "o" and "O". Furthermore, a dot can represent a multiplication operator or a decimal point in different contexts and it could also be hard at times to distinguish it from noise. Besides, in mathematics the spatial relationship between symbols, such as horizontal adjacency as opposed to super/subscripts, can also be difficult to identify, which are very important since they can represent functions or change the meaning of the expression entirely. And there can be little or no clue in the writing to resolve these ambiguities at times. In addition, Watt [Wat10] also points out that: "*there is no fixed vocabulary of mathematical words that can be used to disambiguate symbol sequences*".

Zanibbi and Blostein, identify four key problems in the automatic recognition of mathematical content [ZB12]:

1. Expression detection
2. Symbol extraction or symbol recognition
3. Layout analysis
4. Mathematical content interpretation

More precisely, in the first stage the expression has to be located and extracted. Then, each symbol needs to be segmented and recognised. Afterwards, the layout of the expression has to be analysed and the spatial relationship between the symbols need to be determined. This spatial structure is often represented as a tree. In the final stage, we are concerned with the semantics of expressions which involves interpretation of the tree and mapping of symbols in order to establish the variables, constants, operands and relations. The result of this stage is usually an *operator* tree.

There are numerous studies on online recognition of mathematics [DLZ14, ZMB11], from segmentation of maths [HZ13] to identification of super/subscripts [HW13b] and even

systems for solving mathematical equations [LLM+08] which involves layout analysis and content interpretation. On the other hand, recognition of offline mathematics has enjoyed more success on printed mathematics [SKOY04, UNS05, Bak12, OZ09b, OZ09a] than on that of handwriting [And68, ZBC02]. Nevertheless, recognition of mathematics in all areas still remains an open and challenging problem.

Part II

Recognition of Characters

CHAPTER 3

CHARACTER RECOGNITION KERNEL

As discussed in Part I, different techniques exist in the literature for recognition of characters and although some work can be found on trajectory recovery techniques, it has not been fully exploited. In particular, as discussed in Chapter 1, no research that we know of in handwriting recognition on whiteboards or mathematical handwriting recognition in general, relies on trajectory recovery techniques. However, we base our work on the foundations of trajectory recovery and as we will see in this part, promising results are achieved for the offline recognition of mathematical characters on whiteboards.

As basis for our work we use images of characters from whiteboards taken by a high resolution camera. We apply a number of image processing techniques, presented in Section 3.1, both to detect edges of strokes and to obtain an image skeleton via thinning. We then employ trajectory recovery techniques to segment the skeleton and to recover the original pen strokes, which are discussed in Section 3.2. Subsequently, the strokes have to be put in the right chronological order and the direction of each stroke needs to be determined as well, we discuss this in Section 3.3. Finally, to recognise the symbols, as we present in Section 3.4, we apply an online recogniser to the resulting data from the reconstruction phase. The limitations of our approach are addressed in Section 3.5. We then summarise the chapter in Section 3.6.

Figure 3.1 depicts an overview of the entire character recognition process. The intermediate outputs of the middle stages of the process have been omitted from this overview.

The output of the preprocessing step is essentially the skeleton of the character, which is passed together with the original coloured image to the Local Examination module. Then, after this stage the trajectory of the strokes are passed to the Global Reconstruction unit, where we use one of the three reconstruction methods, i.e. *Colour-driven*, *Brute-force* or *Informed*, to gain insight as to the direction and the chronological ordering of pen strokes. Although, we will have a brief discussion about these methods in Section 3.3, they are discussed in detail in Chapters 4, 5 and 6 respectively. Finally, the result of the Global Reconstruction unit, which is the ordered list of strokes, is then passed to the online recogniser for the recognition task.

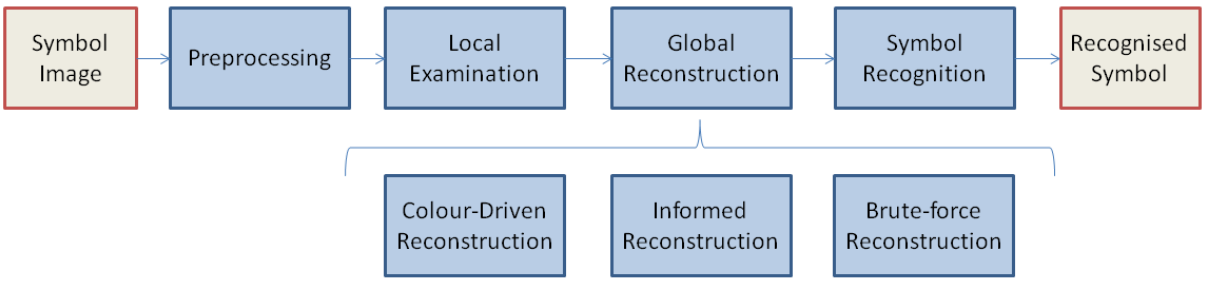


Figure 3.1: An overview of the character recognition process

3.1 Preprocessing

While it is generally the case, a successful handwriting recognition system has to perform some preprocessing techniques to prepare the image for further analysis [PS00]. Some of these techniques, such as binarisation and grey-scale conversion, can eradicate valuable information that could succour in the process of image analysis and consequently trajectory recovery [NB10]. We present our preprocessing techniques that aim to carefully avoid the loss of valuable information. Figure 3.2 shows the major courses of action in preprocessing.

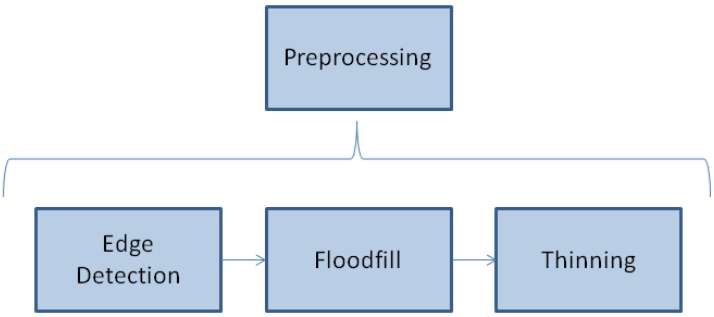


Figure 3.2: An overview of the main courses of action in *preprocessing*.

Figure 3.3 shows an example of a character image, which was obtained by a digital camera. We will be using this example frequently throughout this chapter to illustrate the output of various stages in the process of character recognition.

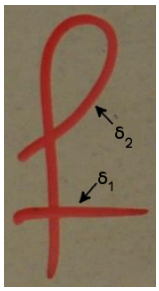


Figure 3.3: An image of a character

3.1.1 Edge Detection

In order to recover the trajectory of a stroke, first the stroke has to be located in the image. Traditionally, this is done through binarised images, but – as discussed in Chapter 2 – this can eradicate valuable information. Moreover, a single background pixel mistakenly placed on a stroke in the binarisation process can result in an artificial loop being created after thinning [NB10]. In case of whiteboards, this would be very common as our initial experiments revealed.

Consequently, we have chosen edge detection for the purpose of locating objects, i.e. pen strokes, in images. The advantage of edge detection is that it rarely produces un-

necessary noise on or around the pen strokes especially in high quality images and given that the thresholds of the edge detector have been accordingly adjusted. While there are many edge detection algorithms in the literature, our experiments with some of these algorithms proved that Canny's method was best suited for our purposes [Can86]. The reasons for this are twofold: firstly, Canny's method produced the least noise in our images of whiteboards, and secondly, it was best at identifying discontinuities in the image, hence, producing the least broken gaps on the detected edges, or stroke boundaries in our case. Both these factors can make the preprocessing of images much easier as we will see throughout this thesis. In Figure 3.4, you can see the Canny edge-detected image of 'f'. Note that, edge-detection has produced some noise to the right of this image.

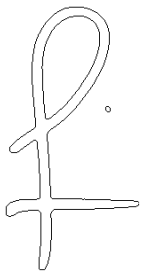


Figure 3.4: Edge-detected image of character 'f' and some noise to the right.

Edge Detection Rectification

Although, as discussed in the previous section, edge detection generally performs well for detecting pen strokes on whiteboards and in fact can help to reduce noise in the image, sometimes the process will fail to produce a closed shape, which will make it difficult to detect the entire boundary of strokes, and as we will see in Section 3.1.2, this can essentially be a deciding factor for the success or failure of the trajectory recovery process. Nevertheless, often this lapse is minor and only small gaps, sometimes as small as 1 pixel, are impeding the algorithm from producing a perfectly closed shape as you can see in Figure 3.5. We have therefore added a preprocessing step that aims to occlude small gaps, in our case up to 20 pixels, by connecting the *terminating vertices*[1] that are

[1]See Definition 5 in Section 3.2.2.

near each other after the edge detection process. This number of pixels is based on the heuristics that we gathered through experiments.

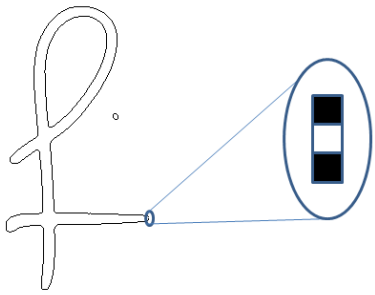


Figure 3.5: Example of *edge detection* failing to produce a closed shape.

Bearing in mind that extracting the skeleton from the character image is the ultimate and sheer purpose of preprocessing, having a fully closed shape plays an important part in that. It is only upon achieving closed shapes that flood-filling techniques can be used to fill out the shape before thinning can be applied to extract the skeletons.

3.1.2 Floodfill

At this stage, before we can apply thinning on the image and extract the skeleton of the character, we need to fill the resulting shape from edge detection. This process is known as *floodfill*, *boundary fill* and also the *grass-fire* approach [Pit93]. In essence, floodfill is the process of filling out a closed area. The algorithm is initiated by locating a pixel and then searching its neighbourhood to obtain other pixels, with the same value, that connect to it. The neighbourhood of these pixels is then searched to find similarly-valued pixels that connect to them and so on until the boundaries of the shape, or in other words differently-valued neighbours, are spotted. All these pixels' values are then replaced with a particular chosen value. In the same way as many other image processing algorithms, floodfill is applied considering 4 or 8-connectivity of pixels, see Figure 3.8 in Section 3.1.3 for a visual depiction and the difference between the two. It should be noted that since we are dealing with lines with minimum width that are at times slanted, the floodfill

algorithm must assume the 4-connectivity of pixels.

In Figure 3.6 you can see the floodfilled image of Figure 3.4 after the edge detection rectification process. Furthermore, the automation of floodfill is necessary in order to have an automated recognition system.

Figure 3.6: Example of *floodfill* results

Automation

The automation of floodfill is a matter of finding the right point at which to initiate floodfill. In our method, we initially find the *connected components*[1] in the image. Subsequently, we try to find the top left-hand pixel in each connected component and then initiate floodfill from a point to the right or possibly bottom of that point depending on the configuration of pixels in the neighbourhood of that point. The top left-hand pixel here refers to the point on the edge-detected boundary that has the lowest x and y - with preference of x over y - according to the Cartesian coordinates of computer graphics with the origin at the top left of the image. After finding the top left-hand pixel, which is a foreground (or black) pixel, we essentially need to locate a background (or white) pixel inside the closed shape that as explained will be to the right or bottom of this point depending on how the pixels are laid in the neighbourhood. We can then initiate floodfill from this background point.

[1]We will discuss connected components and how they are located in more detail in Chapter 7 Section 7.1

3.1.3 Thinning

The thinning procedure repetitively removes unnecessary pixels from the outer layer of the shape until a pixel wide line - a.k.a. *skeleton* - remains. This process will not alter the topological structure of the shape [QNY06] and will make the recovery of the original trajectory of strokes possible. Additionally, the thinning algorithm should ensure that the procedure wanes the image constituents from all sides so that when superimposing the outcome on the original image, the skeleton is equidistant to the edges of the original components as much as possible. This will help in calculating the width of the strokes as we shall discuss later in Section 3.2.1. Figure 3.7 depicts the thinned image of our example.

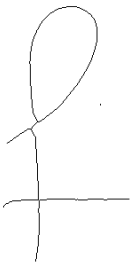


Figure 3.7: Thinning example

There are many thinning algorithms in the literature [LLS92]. We have chosen the algorithm by Guo and Hall [GH89], which has the advantage that it produces the thinnest line possible, i.e. one pixel wide. Some other algorithms tend to produce two pixel wide lines when the lines are slanted. One pixel wide lines are ideal for our purposes since an extra single pixel on the skeleton can in fact exacerbate the situation by creating a *junction*[1], as we will discuss later in Section 3.2.

In this method, the image is divided into two distinct sub-fields in a checker-board manner and the algorithm runs in two sub-cycles. Considering 8-connectivity (see Figure 3.8 to the right), in each iteration a foreground pixel P_0 is removed or kept based on the following conditions:

[1]See Definition 6 in Section 3.2.2

1. Sub-iteration 1:

   a. $X_H(P_0) = 1$

      where, $X_H(P_0) = \sum_{i=1}^{4} b_i$

      and,

$$b_i = \begin{cases} 1, & \text{if } P_{2i-1} = 0 \text{ and } (P_{2i} = 1 \text{ or } P_{2i+1} = 1) \\ 0, & \text{otherwise} \end{cases}$$

   b. $2 \leq min(n_1(P_0), n_2(P_0)) \leq 3$

      where, $n_1(P_0) = \sum_{k=1}^{4} P_{2k-1} \vee P_{2k}$

      and, $n_2(P_0) = \sum_{k=1}^{4} P_{2k} \vee P_{2k+1}$

   c. $(P_2 \vee P_3 \vee \bar{P}_8) \wedge P_1 = 0$

2. Sub-iteration 2:

   a. Same as sub-iteration 1a.

   b. Same as sub-iteration 1b.

   c. $(P_6 \vee P_7 \vee \bar{P}_4) \wedge P_5 = 0$

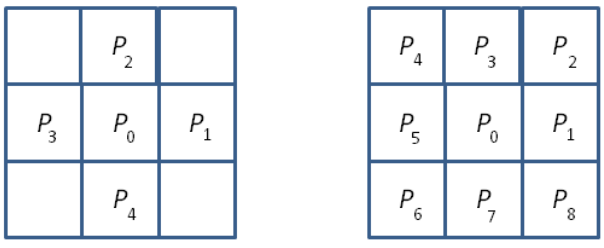


Figure 3.8: 4-connectivity (to the left): each pixel is considered to be connected to pixels that share an edge with the corresponding pixel. 8-connectivity (to the right): each pixel is considered to be connected to all its 8 neighbours.

It has to be noted that a single iteration consists of the two sub-iterations above. The iterations in this algorithm should continue until the resulting image is unaffected by an entire iteration.

3.2 Local Examination

As mentioned earlier the entire purpose of the preprocessing is to extract the skeleton of the character. The skeleton will make the recovery of trajectory of strokes possible as we will see shortly in this section. In order to reconstruct the stroke, however, we need to gather more information about the body of the actual strokes at various points. Therefore, we superimpose the skeleton on the original image. This can succour the system in solving ambiguities by allowing the extraction of more information such as the width and value of the pixels on the original strokes. These ambiguities can include noise reduction, junctions, start and end identification of strokes, and others. We will spend the following sections going into more details about the aforementioned obscurities.

In this section, we present the local examination of a symbol that determines single *segments* of a stroke and how they are connected via *vertices*. As mentioned, it is carried out on the thinned image that we have obtained from the preprocessing step, which is essentially just a two dimensional array of binary pixels. A foreground pixel is a pixel on the skeleton (usually black), whereas a background pixel is effectively the rest of the pixels (usually white). As we are normally only interested in the former, in the following we will generally refer to foreground pixels simply as pixels. An overview of the Local Examination process is portrayed in Figure 3.9.

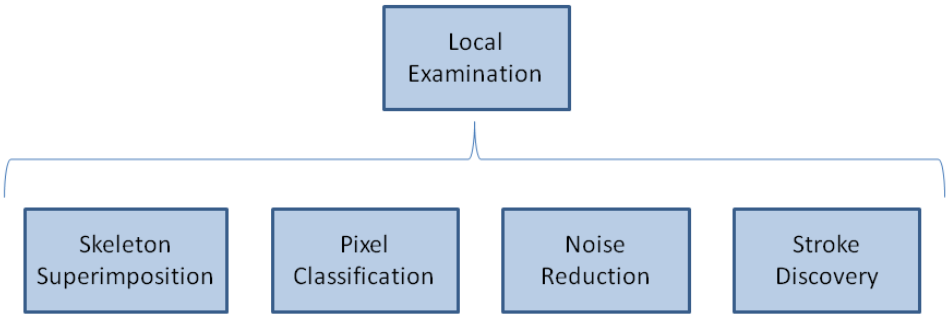


Figure 3.9: An overview of the main courses of action in *local examination*

In contrast to the previous overviews in this thesis, no arrows are put between the processes here. This is because the actions in local examination are interchangeable to an extent; but as we will see, Pixel Classification and Skeleton Superimposition stages will need to be finalised before Stroke Discovery.

3.2.1 Skeleton Superimposition

After the *skeleton* of the character has been extracted, we superimpose the skeleton on the original image, as shown in Figure 3.10. One of the purposes of the superimposition is that it assists us in calculation of the width of stroke at various points. The details of this procedure are discussed in the following section.

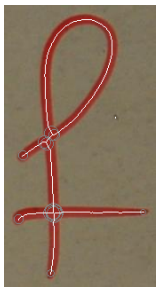


Figure 3.10: The superimposition of the skeleton on the original image

Stroke Width Calculation

In our method, the width of the stroke is calculated at diverse certain points by fitting a circle with a centre on the skeleton and the width of the stroke at that point would be the diameter of the circle. This circle will have its circumference points as close as possible to the edges of the stroke in the original image. Figure 3.11 depicts an example of how the circles are fitted on the strokes. This example uses random points for the purposes of demonstration. In addition, to avoid difficulties in finding the edges of the original image, it is easier to superimpose the skeleton on the floodfilled image for the purposes of calculating the width.

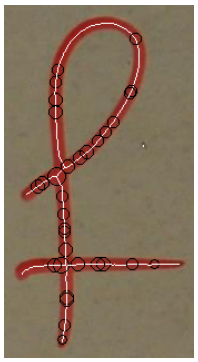


Figure 3.11: Calculating the width of strokes at various points

We use an optimised algorithm which takes advantage of the symmetrical shape of circles to calculate the points on the circumference in one octant of the circle and copying the points to the remaining octants. In this method, we first start with a circle with the smallest radius (one pixel) and then we increase the size of the radius gradually until at least one of the points on circle's circumference is no longer on the original stroke. This can obviously be decided by checking the value of the pixels on the circumference against either the value of the pixels on the stroke or the background. Algorithm 1 shows how the circumference points are calculated. Note that $\ll$ and $\gg$ are the (signed) left and right shift operators respectively.

Clearly, our width calculation method relies on the thinning process to produce a skeleton that is as close as possible to the middle point of strokes at any point.

Algorithm 1: How to calculate the points on the circumference of a circle.

Data: Radius of circle + x,y Cartesian coordinates of the centre

Result: The set of points on the circumference of the circle

discriminant = $(5-$ radius $\ll 2) \gg 2$;

$i = 0$ and $j =$ radius;

while $i \leq j$ **do**

  add points$\{(x+i, y-j)(x+j, y-i)(x+i, y+j)(x+j, y+i)(x-i, y-j)(x-j, y-i)(x-i, y+j)(x-j, y+i)\}$;

  $i = i + 1$;

  if *discriminant* < 0 **then**

    discriminant = discriminant $+(i \ll 1) + 1$;

  else

    $j = j - 1$;

    discriminant = discriminant $+(1 + i - j) \ll 1$;

3.2.2 Pixel Classification

In order to commence the discovery of strokes in the character image, we first need to define what a stroke and its constituents are. Thus far, all we have is the character image, which is essentially just a two dimensional array of pixels, and its skeleton in the same fashion. As we have already explained what we mean by a pixel, we shall now define the neighbourhood of a pixel. Note that, we use ■ to indicate the end of a definition in the rest of this chapter.

Definition 1. *The* neighbourhood *of a pixel p is defined as the 8 immediately adjacent pixels and the set of pixels directly adjacent to p as the* neighbour set $N(p)$. ■

Figure 3.12 presents a schematic depiction of an example of such a neighbour set. On the left, the neighbourhood of a pixel in an 8-connectivity manner are shown and on the right, an example of a pixel with three points in its neighbourhood is presented.

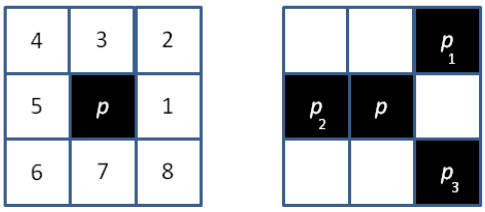


Figure 3.12: (i) The neighbourhood of pixel p (8-connectivity).
(ii) $N(p) = \{p_1, p_2, p_3\}$

Through the definition of neighbourhood of a pixel, we now define a concept of *connectivity* between two pixels recursively.

Definition 2. *Let p, q be pixels, then p and q are* connected*, written $p \bowtie q$, if*

(i) $p \in N(q)$, or

(ii) $p \in N(p')$ and $p' \bowtie q$. ■

Note that this is a *partial equivalence* relation, i.e. $\bowtie$ is:

- Irreflexive: there exists no element p such that $p \bowtie p$.
- Symmetric: if $p \bowtie q$, then $q \bowtie p$.
- Transitive: if $p \bowtie q$ and $q \bowtie q'$, then $p \bowtie q'$.

We can now also define a notion of *path* between two *connected* pixels:

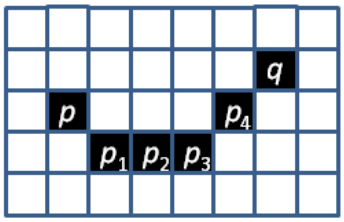


Figure 3.13: Example of *path* and *connection*. In this example we can see that p and q are *connected*, denoted $p \bowtie q$. And, the path between them is $\mathcal{P}(p, q) = \{p, p_1, p_2, p_3, p_4, q\}$.

Definition 3. *A* path *between $p \bowtie q$ is defined as a list of contiguous pixels* connecting *p and q: $\mathcal{P}(p, q) = \{p, p_1, \ldots, p_n, q\}$.* ■

See Figure 3.13 for an example of *connectivity* and *path*. Subsequently, we can render the notion of *vertex* precisely:

Definition 4. *A* vertex*, V, is defined to be a non-empty set of pairwise connected pixels:*

$$V = \{p_1, \ldots, p_n\}$$

where,

$$V \neq \emptyset \text{ and } N(p_i) \neq 2 \ \forall \ i \in \{1, \ldots, n\}.$$

■

We then distinguish the following two types of vertices:

1. Terminal Vertex
2. Junction Vertex

Definition 5. *A* terminal vertex*, V^T, is a single pixel with only one neighbour:*

$$V^T = \{p\} \text{ where } |N(p)| = 1.$$

■

Definition 6. *A* junction vertex*, V^J, is a maximal set of pixels and any adjacent pixel is either a terminal vertex or part of a segment that we will define later:*

$$V^J = \{p_1, \ldots, p_n\},$$

where,

$$n \geq 1 \text{ and } |N(p_i)| > 2 \ \forall \ i \in \{1, \ldots, n\}.$$

Furthermore,

$$\exists! q : q \notin V^J \text{ and } q \in N(p_i) \text{ for any } i \in \{1, \ldots, n\} \text{ with } |N(q)| > 2.$$

■

We will use $\mathcal{V}_T$ and $\mathcal{V}_J$ to denote a set of all *terminal* and *junction* vertices in an image, respectively. We also define the *boundary* of a vertex:

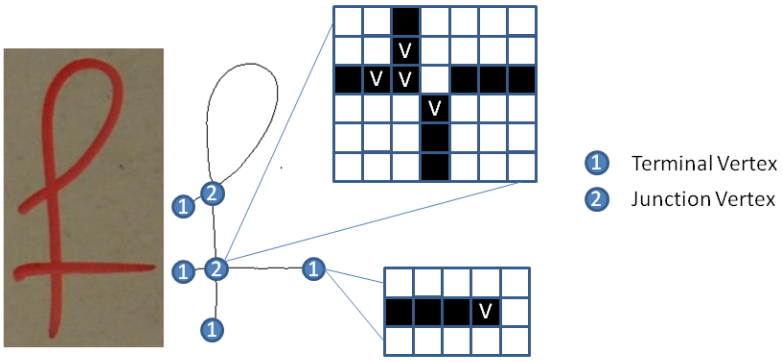


Figure 3.14: Example of *terminal* and *junction* vertices.

Definition 7. *The* boundary *of a vertex, $\mathcal{B}(V)$, is defined as:*

$$\mathcal{B}(V) = \{p \mid p \notin V \text{ and } p \in N(q) \text{ for some } q \in V\}$$

One can easily verify that $\mathcal{B}(V)$ is non-empty. ■

Subsequently, we define *segments*. Intuitively, a segment S corresponds to a part of the skeleton that connects two vertices V_1 and V_2 together without passing through other vertices. Needless to say that V_1 and V_2 can each be either terminal or junction vertices. More formally:

Definition 8. *A* segment*, S, is defined as:*

$$S = V_1 \cup V_2 \cup P$$

where,

$$P = \{p_1, p_2, \ldots, p_n\}$$

such that,

$$|N(p_i)| = 2 \ \forall \ i \in \{1, \ldots, n\}.$$

We further require that:

$$\exists \ p_i \bowtie p_j \ \forall \ i, j \in \{1, \ldots, n\} \text{ and } i \neq j$$

and,

$$\forall q \in \mathcal{P}(p_i, p_j) \Rightarrow q \in P \text{ and } |N(q)| = 2.$$

That is, two elements of P can not be connected via a vertex and we can not have any gaps in the segment. ■

Finally, we classify segments depending on the type of vertices they connect – and occasionally their length – into six different categories:

1. Isolated Segments
2. Dot Segments
3. Terminal Segments
4. Connecting Segments
5. Round Segments
6. Loop Segments

We shall now define each of these segment classifications:

Definition 9. *An* Isolated *segment is a segment that connects two different terminal vertices together.*

$$V_1, V_2 \in \mathcal{V}_T \text{ and } V_1 \neq V_2$$

■

Definition 10. *A* Dot *segment is a special type of* isolated *segment where the segment does not consist of more than one or at most two pixels. Therefore, this segment can only consist of a single terminal vertex or two terminal vertices without any connecting points in between, i.e.* $P = \emptyset$. ■

Figure 3.15 presents examples of an isolated segment and a dot segment. These segments are the result of processing the character 'j'.

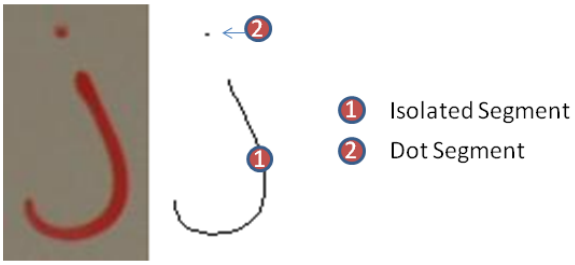


Figure 3.15: Example of *isolated* and *dot* segments.

Definition 11. *A* Terminal *segment is one that connects a terminal vertex and a junction vertex together.*

$$V_i \in \mathcal{V}_T, V_j \in \mathcal{V}_J$$

where,

$$i, j \in \{1, 2\} \text{ and } i \neq j$$

■

Definition 12. *A* Connecting *segment refers to a segment that connects two different junction vertices together.*

$$V_1, V_2 \in \mathcal{V}_J \text{ and } V_1 \neq V_2$$

■

Definition 13. *A* Round *segment connects the same junction vertex to itself.*

$$V_1, V_2 \in \mathcal{V}_J \text{ and } V_1 = V_2$$

■

Figure 3.16 presents examples of a terminal, connecting and dot segments. These segments are the result of processing a variation of the character 'f'.

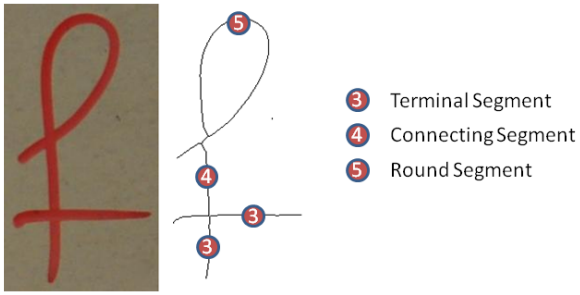


Figure 3.16: Example of *terminal*, *connecting* and *round* segments.

Definition 14. *A* Loop *segment is one with no vertices, i.e.:*

$$V_1 = V_2 = \emptyset$$

■

Figure 3.17 presents an example of a loop segment, which is produced as a result of processing the character 'o'.

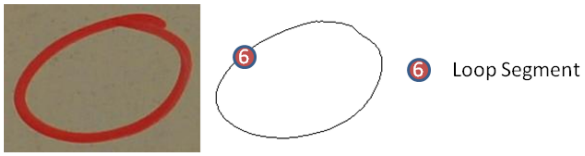


Figure 3.17: Example of *loop* segments.

We call the vertices of a segment its *terminating points* and refer to it as S^{V_1} and S^{V_2}.

This classification makes it easier for us to to solve problems when detecting and classifying junctions as we will discuss in the following sections. Consequently, we can finally define *strokes*:

Definition 15. *A* stroke, δ, *is defined as a* list *of segments:*

$$\delta = [S_1, \ldots, S_n]$$

where,

$$S_i^{V_2} = S_j^{V_1} \ \forall \ i \in \{1, \ldots, n-1\}, j = i + 1$$

■

This concludes the basic definitions required to begin the search for strokes. Next, with the additional information extracted from the image, we can try to eliminate any possible noise.

3.2.3 Noise Reduction

Noise reduction plays an important role in the process of handwriting recognition. Although, using high quality images will assist in minimum noise being introduced in the image, the images are still prone to noise. Furthermore, additional noise could also be produced during the preprocessing procedure.

Whiteboard State

The noise could be due to deterioration and damage to the whiteboard or also cleanness of it as sometimes whiteboards are not wiped properly. Therefore, the colour[1] and also the average width of strokes are taken into account to verify whether detected pixels on the skeleton are noise or otherwise part of a real stroke. To achieve this, once again we take advantage of superimposition of the skeleton on the original image and discard any pixels that do not qualify as a part of a stroke. For example, in Figure 3.10 the noise in the character image belongs to this category and is discarded after considering the pixel values on the noise in comparison to that on the strokes.

Artificial Loops

There is also possibility of small loops being created by the thinning or edge detection procedures due to, as little as, a single pixel mistakenly placed on or around the strokes. In this case the length of the loop segment can usually confirm whether we should consider it as a real stroke or otherwise an artificial loop. More precisely, any loop with a length less than $\pi \times width$ of stroke, could not have been produced by the pen.

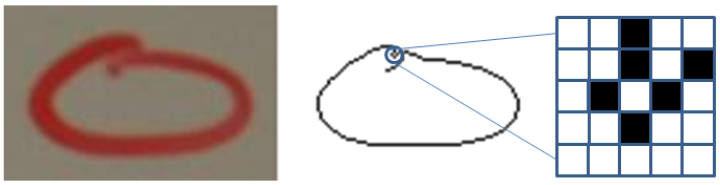


Figure 3.18: Example of an *artificial loop*

Spike Removal

In the thinning process, sometimes irregular components or segments can take shape. These segments are usually in the form of spikes and they beget genuine segments to be split into two segments, which can potentially make the process of stroke reconstruction

[1]By the colour we actually mean the value of the pixels in a colour model such as RGB or HSB.

unsustainable. Therefore, a proficient offline handwriting recognition system will need to deal with them appropriately. These spikes are often very short and commonly less than the average stroke width in length. Figure 3.19 depicts an instance of such component.

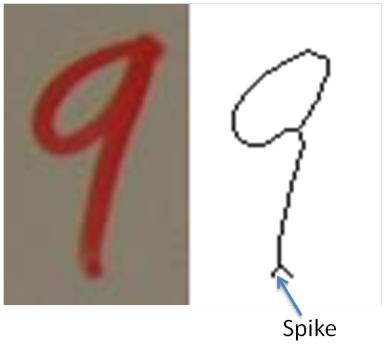


Figure 3.19: Example of a *spike*

The effort to eliminate such component is often denoted *pruning* [Sad13]. Although Sadawi suggests the blurring of the image – to smoothen the rough edges – and redoing binarisation and thinning processes, in our methods we simply erase the extra pixels from the skeleton. In this method, care must be taken to avoid splitting the two adjoining segments – that are part of a genuine stroke – by removing the *junction* pixels. In this scenario, given that all spike pixels apart from the junction pixels have been removed, there are two possible outcomes:

1. Firstly, the removal of the *spike* will remove the bogus junction automatically.

2. Secondly, the removal might leave a junction of degree 2, which we will discuss in Section 3.2.4.

As we will see later, neither of these cases will affect the reconstruction of the strokes.

Spurious Segments

Additional analysis should be performed before we can further examine the image and skeleton for solving junction ambiguities. There is a particular type of *connected* segment that in fact never actually exists in the original drawing of stroke or the character

image and can occasionally appear in particular situations as a result of the thinning process [QNY06]. These segments are called *spurious* segments.

In our methods, *connected* segments are examined for their length and if the length of the segment is below a certain threshold, which depends on the average width of strokes, the two junctions at each end of the connecting segment are merged. An example of a spurious segment is shown in Figure 3.20.

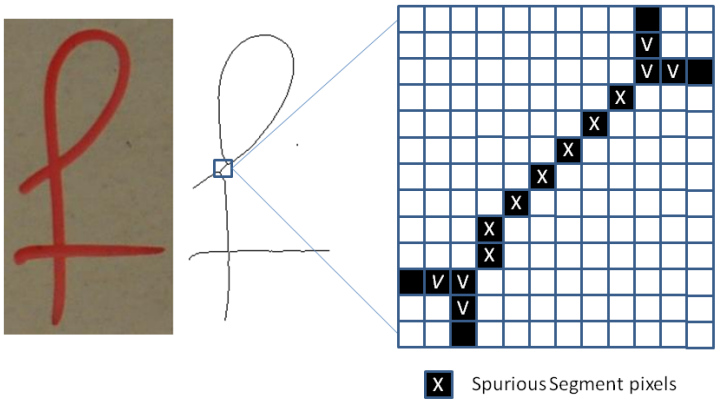


Figure 3.20: Spurious Segment caused by thinning

Following noise reduction and identification of spurious segments, the ambiguities that can occur at junctions in the skeleton image should be inspected in order to reconstruct the strokes.

3.2.4 Stroke Discovery

Although we have now defined what a stroke is in our scenario, we have not yet described a method on how to actually compute a stroke. In particular, we have to further classify junctions in order to determine how segments are connected in a junction. This will allow us to determine how junctions are actually traversed and therefore how to assemble the overall stroke and recover its trajectory. We now define a concept of *degree* for a junction vertex.

Definition 16. *The* degree *of a junction vertex, $\mathcal{D}(V^J)$, is defined as the number of elements in the junctions boundary set, i.e. $|\mathcal{B}(V)|$.* ■

While, in theory, there is no particular limit on the size of this set, in practice we are mostly interested in junctions of degree three, four, or five and above. Therefore, we classify junctions into the following five categories:

1. Junction of *degree* One, i.e. $\mathcal{D}(V^J) = 1$
2. Junction of *degree* Two, i.e. $\mathcal{D}(V^J) = 2$
3. Junction of *degree* Three, i.e. $\mathcal{D}(V^J) = 3$
4. Junction of *degree* Four, i.e. $\mathcal{D}(V^J) = 4$
5. Junction of *degree* Five and above, i.e. $\mathcal{D}(V^J) \geq 5$

We shall now investigate what situation each of these categories represent and also how to deal with them. But before that, we need to know how to calculate the gradient of segment at a junction.

Calculating Gradient of Segments Around Junctions

To calculate the gradient of segments at junctions we choose two points on the segment and calculate the gradient by $\frac{\Delta Y}{\Delta X}$. The two points, i and j, are chosen based on the length of the segment:

$$\begin{cases} i = (L - N) \text{ and } j = (L), & \text{if segment's length} \geq L \\ i = (\text{segment's length } -N) \text{ and } j = (\text{ segment's length }), & \text{if segment's length} > N \\ i = 0 \text{ and } j = (\text{ segment's length }), & \text{otherwise} \end{cases}$$

Where L and N are pre-defined constants such that $L > N$. L is defined to be the minimum number of points that a segment should contain in order to be considered a long enough segment to allow a fair estimation of the gradient of the segment and N is

the needed gap between the two points in order to calculate the gradient reasonably. We choose $L = 50$ and $N = 20$ in our experiments. It should be noted that i and j are the indices of the desired points in the segment. Furthermore, we should assume that points in the segment are ordered in a way that the first point in the segment's list is the closest to the specific junction and the last point in the list the furthest point away in the path from that junction.

Note that here we will be discussing the categories in the order in which they should be looked at to clear the ambiguities prudently. We discuss the reasons later on in this section.

Junctions of degree 1

Junctions of degree one are generally due to thinning or cleaning mistakes and we can dispose of them. A junction of degree one is effectively only a knob at the end of a line and it can simply be replaced by a single *terminating* vertex.

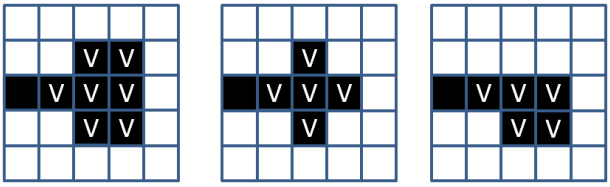


Figure 3.21: Junctions of degree 1

Junctions of degree 2

Junctions of degree two are also generally due to thinning or cleaning mistakes. They can be viewed as a bulge leftover by thinning or as a result of cleaning a *spike*, as was explained before, and we can merge the two segments it connects into a single one.

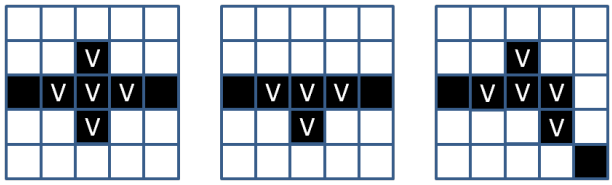


Figure 3.22: Junctions of degree 2

Subsequently, junctions are checked and strokes are reconstructed in the following order:

Junctions of degree 5 and above

For junctions of degree 5 and above, two of the segments that have the closest gradient at the junction are merged and this is repeated until we are left with a junction of degree 3 or 4.

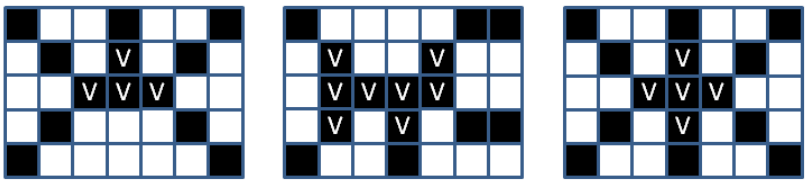


Figure 3.23: Junctions of degree 5 and above

Junctions of degree 4 - Crossing or touching strokes

A junction of degree four, would suggest two crossing or touching strokes coming together. Qiao et al. report [QNY06] that 95.1% of all junctions of degree four are of crossing type. However, we match the segments according to the gradient at which they enter the junction.

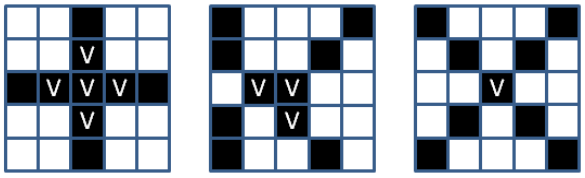


Figure 3.24: Junctions of degree 4

Junctions of degree 3 - Double-traced segments and hidden terminating points

Following [DR95, QNY06] junctions of degree three, suggest one of the following scenarios:

(i) There is at least one *hidden terminating point*, as shown in Figure 3.26, or

(ii) there exists at least one *double-traced segment*, as shown in Figure 3.27

The width of the segments around the junction can assist in determining which scenario applies to the junction. Our experiments show that double-traced segments are often thicker than other segments. Therefore, we distinguish hidden terminating points from double-traced at junctions of degree 3 by examining the thickness of the merging segments at the junction. In the case that all the segments have a thickness in the same range then we would identify the hidden terminating point by considering the angle at which the segments leave the junction. Evidently, in the case that one of the segments is thicker than the other two, we treat the thicker segment as a double-traced one.

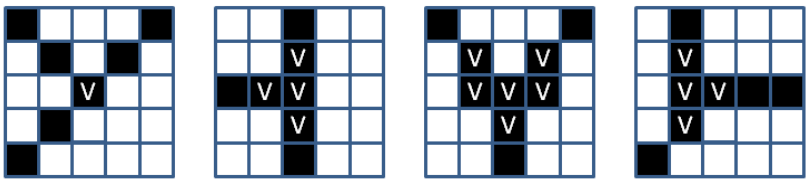


Figure 3.25: Junctions of degree 3

Junctions of degree three are the last ones to be checked since a double-traced segment might cross another segment, and therefore it is necessary to have resolved the crosses

and generally junctions of higher degree first. Similarly, hidden terminating points should be resolved before double-traced segments.

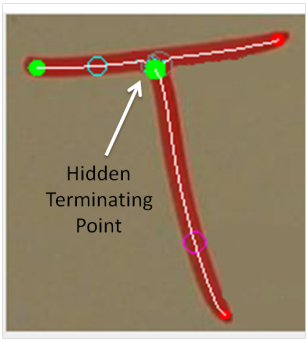


Figure 3.26: Junctions of degree 3 - Sample of a handwritten T, this sample includes a hidden terminating point and it has been identified correctly.

Consider Figure 3.26, the skeleton of the character in this image has a junction. The three terminal segments leaving the junction are all of similar width – 10 pixels on average – and therefore the gradient of the segments is taken into account to determine the hidden end-point. In this case, gradients of the segments are as follows:

1. Segment ending on $(16, 29)$: between the two points $(53, 29)$ and $(33, 30) = -0.05$
2. Segment ending on $(154, 14)$: between the two points $(115, 22)$ and $(135, 18) = -0.2$
3. Segment ending on $(124, 170)$: between the two points $(91, 60)$ and $(95, 80) = 5.0$

Therefore, the last segment is identified as the segment with a hidden end-point.

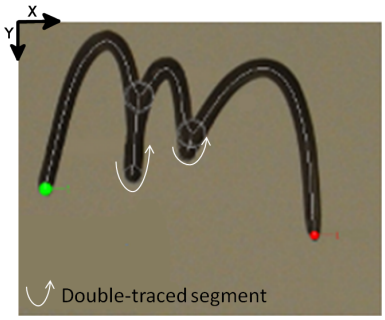


Figure 3.27: Junctions of degree 3 - Sample of a handwritten m, this sample includes two double-traced segments and they have both been identified correctly.

Moreover, in Figure 3.27 there are two junctions of degree three. The junction to the left, V_{J_1}, is at coordinates $(269, 188)$ and the junction to the right, V_{J_2}, is at $(352, 247)$. There are also four terminal vertices; from left to right $V_{T_1} = (122, 335)$, $V_{T_2} = (259, 312)$, $V_{T_3} = (345, 275)$ and $V_{T_4} = (544, 408)$. Consequently, we have the following five segments:

1. S_1 connecting two vertices V_{T_1} and V_{J_1} with average width of 25 pixels.
2. S_2 connecting two vertices V_{J_1} and V_{T_2} with average width of 30 pixels.
3. S_3 connecting two vertices V_{J_1} and V_{J_2} with average width of 28 pixels.
4. S_4 connecting two vertices V_{J_2} and V_{T_3} with average width of 34 pixels.
5. S_5 connecting two vertices V_{J_2} and V_{T_4} with average width of 26 pixels.

Therefore, S_2 and S_4 are classified as the double-traced segments at junctions V_{J_1} and V_{J_2} respectively.

Loop Segments

As previously mentioned, *loop* segments do not actually have a vertex when analysing the skeleton and in fact have hidden start and end points, which have to be found before we can make a valid stroke. We once again find the hidden terminating points by considering the width of the segment at various points. In this method, we essentially split the segment at the point that is thickest in the segment, i.e. has the highest width, which creates an *isolated* segment and can be added to the list of strokes.

After we have matched (merged) segments appropriately at the junctions, given that all the junctions are resolved – and loop segments have been dealt with – we essentially end up with a list of strokes. We then pass these strokes to the Global Reconstruction module for further analysis.

3.3 Global Reconstruction

After the local examination is concluded and all strokes have been reconstructed, we now need to find out for each stroke in which direction it was drawn and for a collection of strokes in which chronological order they were drawn. This is achieved in the global reconstruction phase (see Figure 3.28), which we discuss in more detail in Chapters 4, 5 and 6.

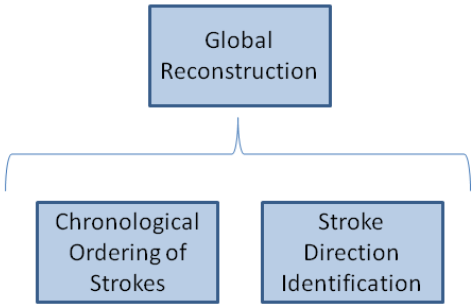


Figure 3.28: Two constituents of the Global Reconstruction process

In the aforementioned chapters, we precisely discuss three approaches for the reconstruction of strokes, namely:

Colour-Driven Reconstruction, which takes advantage of the colour information on terminating points of strokes to determine their direction. In this method a heuristic is used to prioritise strokes for ordering. (Chapter 4)

Brute-force Reconstruction, that utilises brute-force search - or exhaustive search - to choose the most probable combination for the provided strokes both for their direction and order. (Chapter 5)

Informed Reconstruction, which is a synthesis of the previous two approaches. (Chapter 6)

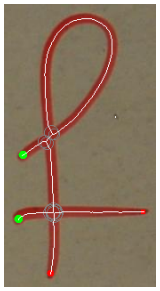


Figure 3.29: Example of sampling and end-point identification in Global Reconstruction.

After the direction and order of the strokes have been determined, we use an online recogniser for the identification of the character.

3.4 Online Recogniser

Subsequent to global reconstruction, we pass the list of strokes to the online recogniser which is presented in [HMW12]. Each stroke is essentially itself a list of ordered pixels, i.e. x, y Cartesian coordinates of skeleton pixels, from the beginning of stroke to the end it. Note that the origin of this coordinate system is considered to be at the top left hand-side of the image – which is the case for most standards in computing such as inkML. So in Figure 3.26, the x and y coordinates of the starting point of the vertical stroke are (84,32) and the end point is at (124,170).

The factor that makes this recogniser suitable for offline recognition of handwriting is that this recogniser only requires the trajectory points of each stroke in the desired symbol. That is to say there is no need to provide other dynamic information associated with strokes such as speed, angle and timing information.

It is also worth mentioning that this recogniser would usually return multiple results for a given set of strokes together with the recognition confidence for each result. The returned confidence can be used for further analysis and we will explain later how we successfully use this information to improve our methods. Figures 3.30 and 3.31 show examples of the results returned by the recogniser for the corresponding symbols.

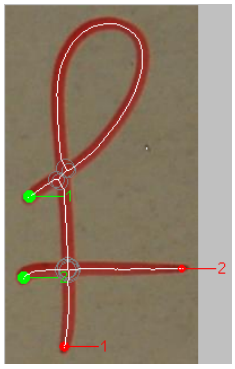


Figure 3.30: Result of recognition for the character pictured above: $\begin{cases} \text{'f'} = 91.56\% \\ \text{'1'} = 72.39\% \end{cases}$

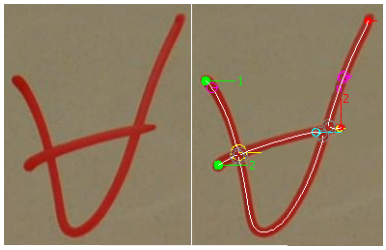


Figure 3.31: Result of recognition for the character pictured above: $\begin{cases} \text{'}\forall\text{'} = 81.59\% \\ \text{'}\theta\text{'} = 68.25\% \end{cases}$

3.5 Limitations and Discussions

Our experiments show that there are a number of exceptions and cases that the aforementioned procedure will fail to cope with and therefore, it was deemed necessary to discuss them here. This section is designated to discussions about these cases, and offering solutions to rectify the shortcomings where appropriate. We have divided these into categories based on the point at which they occur in the process.

3.5.1 Preprocessing

One of the main reasons behind failures in the preprocessing stage of many image analysis applications is the failure in detection of the entire component, which traditionally is done via binarisation and in our methods edge detection. Although, as discussed earlier, edge detection performs well in most situations, in our experiments it was noticed that sometimes it fails to detect the entire boundary and the gaps it leaves is more than our rectification method (see Section 3.1.1) can handle. One possible way to improve the edge detection results is by using a method that automatically adjusts the threshold of the edge detection to get the best possible result for any given image.

In addition, automatic floodfill could be another factor that causes preprocessing and hence character recognition to fail. Although our methods deal with multiple connected components in characters, if two perfectly closed shapes are touching one another at some point – which essentially makes them a single *connected component* – then the automatic floodfill method that was discussed in this chapter will fail to fill one of the shapes and consequently, the symbol will be mis-recognised. This could be a tricky issue to handle, however, one could try to tackle this problem by tracking loops instead of connected components and floodfill the loops instead.

3.5.2 Local Examination

As seen previously, in most cases the ambiguities at junctions can be resolved by taking into consideration the degree of the junction and acting appropriately. However, sometimes this can be trickier than what we have discussed so far. There are exceptional cases where without further analysis the mentioned methods would fail to deal with.

False Junctions

Have a look at the examples in Figure 3.32. The two junctions to the left of this figure should in fact be end points. Instead we are essentially left with junctions of degree 3.

It has to be said that you should not usually end up with a junction such as the one to the right of the figure, because the thinning algorithm should take care of such case, however, in certain situations it is possible, and in fact it can be caused by cleaning or noise reduction. Note that in this case also we are left with a junction of degree 3, which in fact should have been a degree 2 junction.

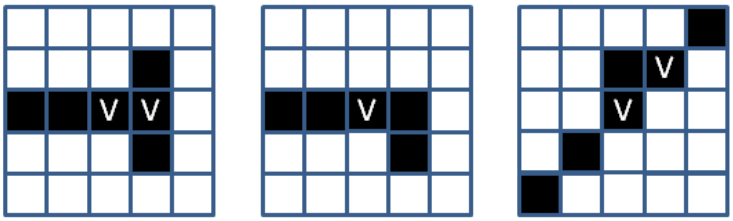


Figure 3.32: Example of false junctions.

We need to take care of these potential pitfalls. The way we resolve these issues is by examining the set of all junction vertices, $\mathcal{V}_J$, that is to say, examination of each point in junctions' boundary sets, $\mathcal{B}(V)$, by an attempt to follow the boundary points away from the junction and eliminating all the points in the segment from the skeleton if it is less than the average width of strokes.

False Terminal Points

Also there could happen to be false *terminal* points. These are very similar to *spikes* with an obvious difference in length. See Figure 3.33 for an example of false terminal points.

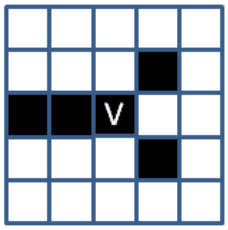


Figure 3.33: Example of false terminal points.

False terminal points are similar to false junctions in that they too occur around

junctions and therefore can be dealt with in the same way.

Segment Matching at Junctions

Although our segment matching technique, which was explained in Section 3.2.4, generally works well. In our experiments we came across cases where it failed to match the right segments and caused the mis-recognition of the character. One possible way to solve this problem could be by using machine learning, perhaps *statistical classifiers*, to match the segments at junctions.

Additional type of Hidden End-Points

In addition to the types of *hidden end-point* that we have discussed so far, there is yet another type, which has proven much more difficult to locate in the skeleton or the original image. We call this type *overlapping* hidden end-points where a hidden end-point sits on top of another. This is a similar case to *loop* segments with the difference that there could in fact be other end points and junctions involved which is what makes them difficult to locate. Figure 3.34 presents an example of such case.

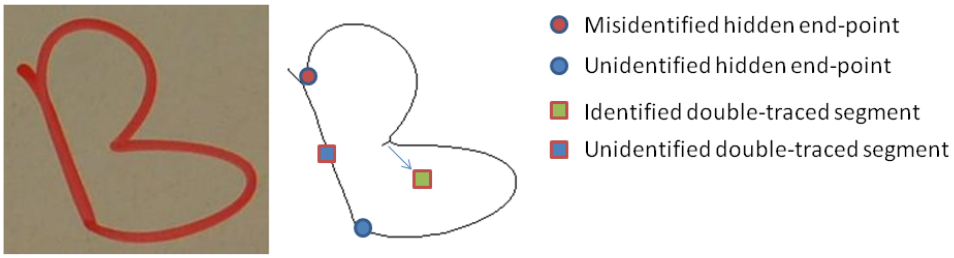


Figure 3.34: Example of an *overlapping* hidden end-point. As you can see because the overlapping hidden end-point was not identified, it has lead to mis-identification of other segments and vertices.

In this example, since our algorithm failed to identify the overlapping hidden end-points, a segment that should have been marked as a double-traced one is instead considered as a part of another connecting segment. This is because the average width of the

stroke is no longer greater than the other segments joining the junction at the top left of the symbol. This is also why we end up with a mis-identified hidden end-point at that junction. Consequently, the character is also mis-recognised.

However, this might not always cause problems in writing of Latin and Maths since it does not arise in most characters. Particularly when the sloppiness factor of writing on whiteboards is taken into account. In addition, in many cases when the hidden end-point is not identified, the recogniser will still be able to identify the character correctly. See Figure 3.35 for an example.

It goes without saying that in languages such as Chinese overlapping hidden end-points happen very regularly and therefore they must be dealt with if any meaningful result is to be achieved through trajectory recovery techniques.

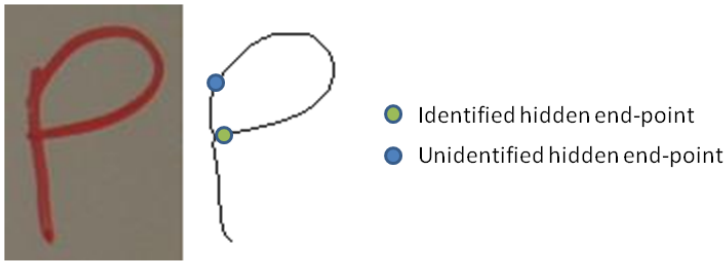


Figure 3.35: Example of an *overlapping* hidden end-point. In this example, although the overlapping hidden end-point was not identified, we are able to obtain the correct character. (Recognition results: 1.'p' 95.12% , 2.'P' 94.93%, 3.'rho' 92.44%

3.5.3 Recognition

Our experiments showed that in some cases, although the trajectory of the strokes were extracted successfully with the correct ordering of the strokes, the wrong results were achieved. Moreover, our online recogniser was not specifically trained for recognition of characters on whiteboards and generally, writing on whiteboards can be quite different to writing on paper; since there are factors such as resting of hand on the writing surface and writing when in a stand-up position, which can make handwriting somewhat sloppier.

These factors indicate that further training of the online recogniser could increase the recognition accuracy of the procedure.

3.6 Summary

In this chapter we discussed the methodology of our character recognition method. We have shown how the trajectory of a symbol can be extracted in three steps, namely **preprocessing**, **local examination** and **global reconstruction**. The main goal of preprocessing is to obtain the *skeleton* of a symbol – which can also be deemed as the original trajectory of the strokes drawn by the writer. In this research we extract the skeleton of a symbol through a combination of preprocessing methods that were specifically designed and adjusted for whiteboard images. These methods include **Edge Detection**, **Floodfill** and **Thinning**. After the skeleton has been prepared, it needs to be analysed and the ambiguities have to be resolved in the local examination stage. We have defined the necessary basics and components required for this analysis, such as **vertices**, **segments** and **strokes**. After the analysis, the trajectory of the strokes can be restored where we also take advantage of the information in the original image. Some **noise reduction** techniques were also discussed, which can be a decisive factor in the success or failure of the recognition system. We have also briefly touched upon Global Reconstruction and showed the two constituents of this stage, i.e. **Chronological Ordering of Strokes** and **Stroke Direction Identification**. In the rest of this part of the thesis we will discuss the three different reconstruction methods that we introduced in this chapter, i.e. *Colour-driven*, *Brute-force* and *Informed* Reconstruction methods.

After the specifications on how the trajectory of the strokes can be recovered from static images of symbols, we introduced an **online recogniser** that we use throughout our research for the recognition task. However, it is worth noting that this online recogniser – trained by Hu *et al.* [HMW12] – was not specifically trained for our research on

whiteboards.[1]

The limitations of our methods at various stages in the process, and possible solutions where appropriate, together with extensions and future work were also discussed in section 3.5. The future works include automatic adjustment of edge detection threshold, dealing with false junctions, false terminal points and overlapping terminal points. In the next chapter we will see how colour images can be exploited in order to establish the original direction of strokes as well as a set of experiments in which we put the methods that are explained so far to test.

[1]Both our system and the recogniser are implemented in *Java* programming language.

CHAPTER 4

COLOUR-DRIVEN RECONSTRUCTION

While trajectory recovery has been investigated before [NB10, DR95, KY00, QNY06], the previous literature generally focuses on working with binary or grey-scale images and methods that rely on having so much information as these image types permit. In our work, however, we use additional colour information that was formerly often neglected in offline recognition of handwritten documents. This was partially due to hardware limitations, which today can be safely ignored, and mainly because it was deemed that the amount of information that colour images can reveal could not justify the processing costs. However, we use colour information successfully, in particular for noise reduction – as discussed in the previous chapter – as well as for the *statistical classification* techniques that help us to determine the direction of the recovered strokes, which we describe in this chapter.

In colour-driven Reconstruction, we first determine the end points of single strokes as means to retrieve the direction of the original pen movement - described in Section 4.1 - and then order them to gain insight on how the entire symbol was drawn, which is discussed in Section 4.2. This data is then collated and given to an online recognition system specialised on mathematical symbols for the final analysis of the characters, as discussed previously in Section 3.4.

Our experiments – on which we report in Section 4.3 – on a sample set obtained from different writers, using different pen colours, have yielded accuracies of 94.46% and

87.91% for the recovery of strokes and their direction, respectively. Our overall recognition accuracy for the entire process is 78.84%.

4.1 Stroke Direction Identification

In the colour-driven reconstruction method, we determine the direction of the strokes by analysing the colour and width information around the terminating points of a stroke. We have trained a *statistical classifier* to determine the difference between start and end points, or in other words, for the identification of the start and end point of a stroke, which we shall discuss in more detail in the next section. But, in order to analyse the colours, since we are dealing with digital images we first need to know how colours are modelled and represented in computer graphics.

4.1.1 Colour Models

In computing and electronics, colours are modelled using mathematics and are represented as tuples of values or colour components/channels. RGB and CMYK are the two most commonly known colour models, with the first model representing each pixel as a tuple of Red, Green and Blue and the latter as Cyan, Magenta, Yellow, and Key (black) values. Each of these colour models has its strengths and weaknesses, and is therefore used in various fields accordingly.

For instance, the RGB colour model is ideal for representation of colour images and therefore it is widely used in various electronic displays such as TVs (from CRT to LED), computer and mobiles, and also other electronic devices such as digital cameras and scanners. Moreover, RGB is an additive colour model, meaning that the three light channels (Red, Green and Blue) are added together to reproduce a colour. This model implies that the absence of colour components (light) would yield the colour black and to produce white colour all three colour channels would have to be combined at their full intensity. Therefore, devices that transmit light can generally accomplish their aim using

this colour model.

On the other hand, CMYK colour model is a subtractive colour model, meaning that white is the natural colour of the background, while black results from a full combination of all colours (CMY). Therefore, this model is ideal for describing colour printing, where the natural background is normally white (paper) and therefore, it is used in most printers today. See Figure 4.1 and 4.2 for graphical depiction of the RGB and CMYK models respectively.

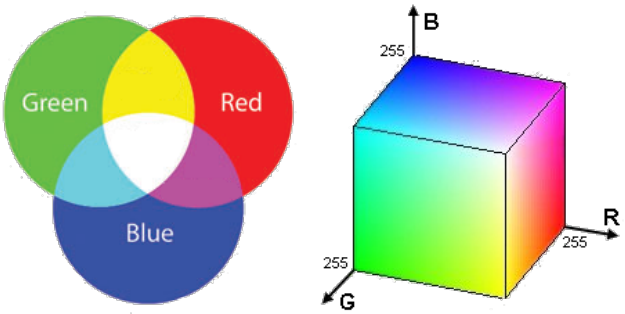


Figure 4.1: RGB Colour Model

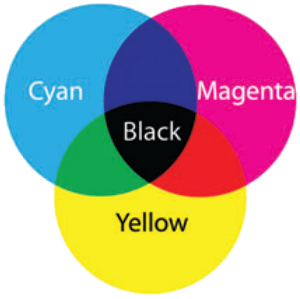
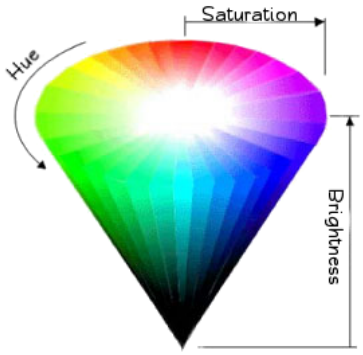


Figure 4.2: CMYK Colour Model

Although the RGB colour model is in some ways similar to how humans perceive colours, hence the reason why they are good for representation of colour images, it is not directly aligned with the colour-making attributes recognised by the human vision. Therefore, other representations of this model have been developed such as HSB[1] and HSL (Hue, Saturation, Brightness or Lightness). HSB and HSL improve on the colour

[1] A.k.a HSV i.e. Hue, Saturation and Value

cube representation of RGB by arranging colours of each hue in a radial slice, around a central axis of neutral colours, which is essentially the brightness and ranges from black at the bottom to white at the top. See Figure 4.3 for an illustration of the HSB model and some sample values compared to RGB. Note that RGB values range from 0 to 255 for all three channels. However, in HSB, hue ranges from 0° to 360°, and saturation and brightness are percentage values.



Colour	Name	(R,G,B)	(H,S,B)
	Black	(0,0,0)	(0°,0%,0%)
	White	(255,255,255)	(0°,0%,100%)
	Red	(255,0,0)	(0°,100%,100%)
	Green	(0,255,0)	(120°,100%,100%)
	Blue	(0,0,255)	(240°,100%,100%)
	Yellow	(255,255,0)	(60°,100%,100%)
	Cyan	(0,255,255)	(180°,100%,100%)
	Magenta	(255,0,255)	(300°,100%,100%)
	Gray	(128,128,128)	(0°,0%,50%)

Figure 4.3: HSB Model and example values compared to RGB

It should be noted that the perception of colour highly depends on the illuminant of the environment and one could consider other criteria such as the lighting and contrast. However, these methods have already been exploited in numerous previous work, and methods such as binarisation and gray-scale conversion have been introduced to deal with such problems. In our research, we attempts to show that, under normal artificial lighting conditions in research offices and lecture theatres, colours can reveal valuable information, which will in turn help in recovering the trajectory of strokes.

4.1.2 Classifier Training

To determine the direction of a stroke, we use the average width of strokes near each terminating point, and also the colour information of the pixels around terminating points and train a *classifier* with this information. The colour information is gathered according to the HSB model, which during our experiments proved to reveal much more information

than normal RGB. The value for each of the channels in this model is a floating-point number between 0 and 1.0 in our experiments.

400 handwritten samples were used to train the classifier and the samples included over 70 Latin, Greek, and mathematical symbols. Our experiment proved that the saturation channel, specifically, is a deciding factor for revealing the start and end – or in other words the direction – of the strokes. We train the classifier with the following 7 features:

1. Hue at the beginning of stroke, denoted by H_b
2. Saturation at the beginning of stroke, denoted by S_b
3. Brightness at the beginning of stroke, denoted by B_b
4. Hue at the end of stroke, denoted by H_e
5. Saturation at the end of stroke, denoted by S_e
6. Brightness at the end of stroke, denoted by B_e
7. Width, denoted by W

And we show the feature vector of a stroke, δ, as:

$$F_\delta = \{H_b, S_b, B_b, H_e, S_e, B_e, W\}$$

The colours are gathered around each end[1] of the stroke by firstly gathering a certain number of circles fitted in the original stroke[2] and then collecting all the HSB colour information for every pixel in those circles. The average of each channel, i.e. H, S and B, is then calculated for each end of the stroke.

The width feature is the result of comparison between the average width, or diameter of the gathered circles, at each end of stroke. This was done to avoid scaling issues for the classifier. We use 1, 0.5, 0 for the width depending on whether the average width at the beginning of stroke is larger, equal or smaller than the end of it.

[1]Although, at this point we have not set the direction of the stroke, a stroke is essentially a list of pixels. Therefore, here by the beginning and end of a stroke we simply mean the points at the beginning or at the end of the list.

[2]See Section 3.2.1 for how the circles are fitted.

The calculated coefficients for each of the features are as follow:

1. Beginning Hue: 3.8259,
2. Beginning Saturation: 23.1035,
3. Beginning Brightness: -18.3359,
4. End Hue: -4.9505,
5. End Saturation: -22.3298,
6. End Brightness: 17.0013,
7. Width: 3.2737

4.1.3 Terminating Point Analysis

Let's consider an example of feature vectors of strokes. To follow our example in the previous chapter, we consider the two strokes of the character 'f' in Figure 4.4. The horizontal stroke, which we call δ_1 here, has a feature vector as follows:

$$F_{\delta_1} = \{H_{b_1}, S_{b_1}, B_{b_1}, H_{e_1}, S_{e_1}, B_{e_1}, W_1\}$$

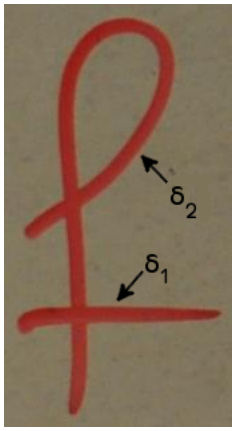


Figure 4.4: Image of a character with two strokes.

This stroke also has ends at $p_1 = (16, 241)$ and $p_2 = (156, 233)$ and the average of the diameter of the first forty circles from p_1 is 6 pixels compared to that of p_2 which is 4 pixels and therefore $W_1 = 1$. So the feature vector for this stroke is as follows:[1]

$$F_{\delta_1} = \{0.9570, 0.7232, 0.5747, 0.9805, 0.6992, 0.5330, 1\}$$

[1]Note that the average HSB values are gathered from the same circles that we used to calculate the width

These values are then passed to the classifier and in this case the classifier returns the value of 78.6357%. A value greater than 50% would indicate that the list of points in the stroke is already in the correct order or else the list has to be inverted.

The second stroke then, δ_2 has a feature vector: $F_{\delta_2} = \{H_{b_2}, S_{b_2}, B_{b_2}, H_{e_2}, S_{e_2}, B_{e_2}, W_2\}$. And its ends are located at $(21, 169)$ and $(52, 302)$ which are both 5 pixels wide on average over the first forty circles and therefore $W_2 = 0.5$. And the feature vector for this stroke is:

$$F_{\delta_2} = \{0.9770, 0.7187, 0.5675, 0.9763, 0.6478, 0.5584, 0.5\}$$

After passing the values to the classifier the value 79.7542% is returned that means there is no need for the stroke to be inverted.

Next, the strokes need to be put in the correct chronological order before they can be passed to the recogniser.

4.2 Chronological Ordering of Strokes

After the analysis of terminating points, in case there is more than one stroke in the character image, the strokes have to be put in the right order. Having the strokes in the right order is quite significant and failing to find the right order can cause the recogniser to mis-recognise the character. The algorithm we use prioritises the strokes based on the following heuristics:

(i) Near-vertical strokes (angle $> 79\,^\circ$) have the highest priority,

(ii) A stroke to the left of another detached stroke has a higher priority. This is based on the general ordering of characters in Latin and mathematics, and also the principle of least effort that is naturally often chosen by people when writing.

(iii) Longer strokes are prior to shorter ones.

Calculating the angle at which the stroke is drawn is a difficult task, since it would be difficult to define the angle due to the fact that the strokes are not necessarily straight

lines. We use a somewhat naive method for this calculation based on the bounding box of the stroke:

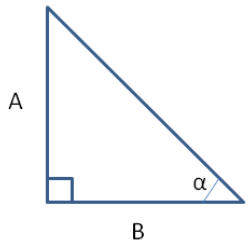


Figure 4.5: Stroke angle calculation.

In Figure 4.5, we can calculate the angle α by the formula $\tan\alpha = \frac{A}{B}$ and therefore, $\alpha = \tan^{-1}\frac{A}{B}$. Accordingly, for α to be roughly greater than 79° we need $\frac{A}{B}$ or $\frac{height}{width}$ to be greater than or equal to 5. This is obviously a naive method because it assumes that the stroke is a straight line. However, in our experiments it proves to be sufficiently accurate. This is because for non-straight strokes, the stroke's *height* would rarely be five times greater than its *width*.

Correspondingly, in this method we first sort the list of strokes based on their positioning (left-to-right), and then their length. Finally, we look for near-vertical strokes and move them up in the list.

To follow the example in the previous section, in Figure 4.4, δ_1 is a short horizontal stroke and δ_2 is a longer one. Therefore, δ_2 takes priority over δ_1 and then the strokes are passed to the recogniser in their new order.

4.3 Experiments

In this section we present our experiment which was designed to confirm the potential of our methods. More precisely, the objectives of this experiment are four-fold; to verify the effectiveness of the following techniques:

1. Preprocessing

2. Stroke recovery

3. Stroke direction identification

4. Symbol recognition

We ran experiments on sets of colour images photographed with a high quality camera — a Nikon $\mathcal{D}90$ with an AF-S Nikkor 18-105mm lens — from characters written on whiteboards by a number of writers in different colours.[1] As recogniser we use the system presented in [HW13a]. We first trained our classifier on a set of 400 images taken from 3 different writers written in red (160 images), black (140), blue (100). The set contained 250 Latin characters and 150 mathematical symbols.

We then ran experiments on a distinct set of 600 samples from 6 different writers, containing 200 maths symbols and 400 Latin characters, where 250 were in red, 200 in black and 150 in blue ink. The results of the experiments are given in tables 4.1 and 4.2, where the former gives a breakdown on the recognition rate of maths symbols versus Latin characters, while the latter breaks down with respect to colour.

We were able to achieve an overall recognition rate of 78.84%. This does not include preprocessing failures, which were due to failed edge detection, wrong flood filling, or thinning problems. Note, that the recogniser sometimes suggested a preference list of results (for example see Figure 3.31), and we define a successful recognition if our character was in that list. The rate for first preferences was 69.66%. In general, the online recogniser was not specifically trained for whiteboard characters, and thus there were a substantial number of cases where even though the strokes and their direction were recovered correctly, the result of the recognition was incorrect. We believe that the recognition rate will be increased when the recogniser is trained accordingly.

As for a difference in recognition between the different symbols and colours, a slight variation we can observe is that, although our methods deal with black ink better in the preprocessing stage, they are not as effective for direction identification compared to the

[1] All experiments were carried out with round headed whiteboard pens.

other colours. This is predominantly due to the fact that the saturation and brightness channels for black inks do not necessarily experience a change from the beginning to the end of stroke as other colours do. On the other hand, the red pen seems to reveal the most clues in this regard and that is why direction identification works best when the symbols are drawn in red.

Moreover, in the preprocessing stage, our system evidently performed better for maths than Latin characters. Yet, direction identification deals with Latin symbols much more robustly. However, we could not find any indication as to why this is the case in our results and therefore, concluded that it could be merely coincidental.

Table 4.1: Character recognition with Colour-driven reconstruction: Maths vs. Latin characters

	Maths	Latin	Overall
Preprocessing Failure	2.34%	6.14%	4.87%
Stroke Recovery	96.86%	93.26%	94.46%
Direction Identification	83.78%	89.97%	87.91%
Recognition Accuracy	77.91%	79.30%	78.84%

Table 4.2: Character recognition with Colour-driven reconstruction: Accuracy broken down by colour

	Red	Black	Blue
Preprocessing Failure	4.97%	4.40%	6.87%
Stroke Recovery	95.51%	92.68%	100%
Direction Identification	92.73%	85.42%	86.36%
Recognition Accuracy	82.83%	75.40%	83.63%

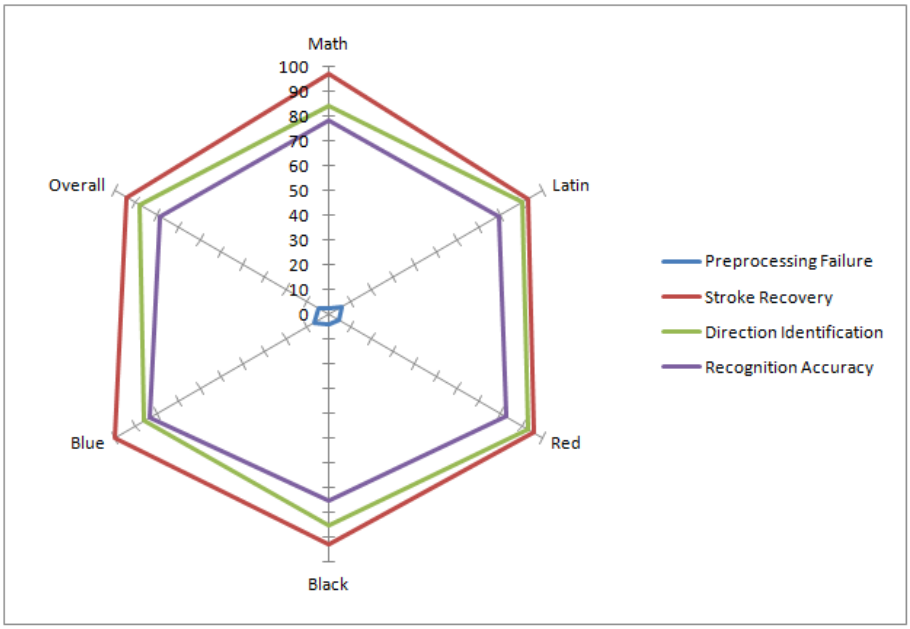


Figure 4.6: Character recognition with Colour-driven reconstruction results.

Additionally, correlations between failures of the method at different stages of the process and the number of strokes could not be found nor with the left/right handedness of the writer. However, "bad" writing style does seem to play a part but can be partially amended by further training of the online recogniser. Have a look at Figure 4.7 and 4.8 for examples of successful and unsuccessful characters respectively.

Symbols that consisted of multiple strokes and had multiple hidden end-points – for example see character 'p' in Figure 4.8 – or had multiple overlapping end-points were generally amongst the worst performing characters. Unsurprisingly, single and multiple stroke symbols – with detached or simply crossed strokes – were quite successful, even though a couple of exceptions can be seen in Figure 4.8.

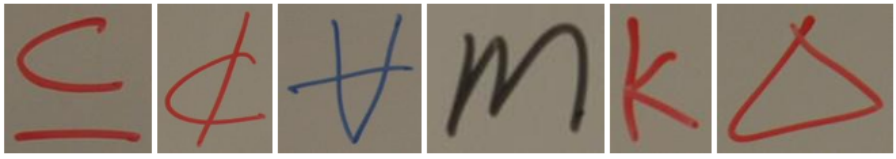


Figure 4.7: Examples of characters that were recognised correctly.

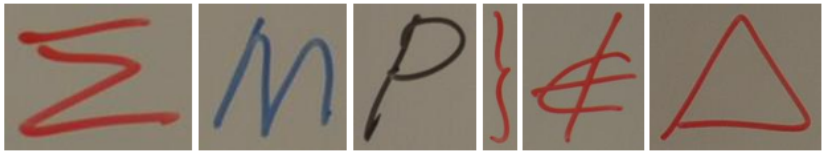


Figure 4.8: Examples of characters that failed to be recognised correctly.

In Figure 4.8, the first character on the left, *Sigma*, was not successfully recognised because the online recogniser has failed to recognise a two-stroked Sigma character most likely due to lack of training, otherwise the strokes had been recovered successfully. The second character from the left in this figure has failed due to a preprocessing mistake. The edge detection algorithm fails to produce a fully closed shape and also the edge detection rectification algorithm fails to close the gap, which causes the misrecognition of the character. In the third example in this figure, the system fails to recognise the character p, due to a mistake in identifying the end points. In the next example the curly bracket was misrecognised due to the system failing to spot the double-traced segment and instead misinterprets it as a hidden terminating point. Similarly, for the second character from right, the system fails again at junctions but this time, although all the spurious segments were identified correctly, a connecting segment is also mistakenly taken as a spurious segment. Finally, for the last example similar to the Sigma character situation, the single stroke in character Δ was recovered successfully, however the recogniser misrecognised the character as a zero.

4.4 Limitations and Discussions

Our experiments show that, although our stroke direction identification technique works well for both maths and Latin symbols, the heuristics we use for the chronological ordering of strokes works better for Latin than maths symbols. This could be due to the fact that maths symbols generally contain more strokes and therefore we need a more robust method than the heuristics we have. This was a motivation that lead to our brute-force

reconstruction method that we will discuss in the next chapter.

Also, our classifier could have possibly performed even better if the length of the strokes was included in the feature set. This is because of the wet nature of whiteboard pens and the fact that especially the saturation channel can drop dramatically if the length of the stroke is greater. However, credibility of this notion needs to be verified through rigorous experiments.

4.5 Summary

In this chapter we described the colour-driven (global) reconstruction method. The Global Reconstruction method generally consists of two major parts; **Chronological Ordering of Strokes** and **Stroke Direction Identification**, that had been introduced in Chapter 3.

The colour-driven reconstruction method takes advantage of the strength of *statistical classifiers* in order to set the direction of the recovered strokes. We have discussed in detail how our classifier was trained including the features used in the process and the calculated coefficients. These features include the colour information (based on the **HSB** colour model), together with the width information near each end of a stroke. Also, the details of how the colours can be gathered from the original image using the skeleton of the symbol were discussed.

We then explained the threefold heuristics that we use in order to get an insight into the chronological ordering of the strokes in a symbol, which to summarise depend on:

1. Posture (vertical strokes),
2. Position (left-to-right), and
3. Length of strokes.

Subsequently, we reported on our experiments that were designed to verify the effectiveness of the methods that we have described so far. The overall recognition accuracy

of the system was 78.84% when using this reconstruction method. Although, as explained earlier in Chapter 1, our method is quite different in nature to other proposed offline recognition methods on whiteboards, it outperforms the aforementioned techniques. Moreover, in comparison to other systems that rely on trajectory recovery techniques – cf. [NB10, RCA06] – our method with 94.46% overall stroke recovery outperforms most of these systems as well, which report between 80.2% and 97.6% stroke recovery accuracy, although non of those systems work with mathematical content. Finally, in the last section of this chapter, we discussed the shortcomings of our colour-driven method and also identified possible future work. In the next chapter, we introduce another and rather different reconstruction method.

CHAPTER 5

BRUTE-FORCE RECONSTRUCTION

Another method that we introduce to reconstruct the strokes in the Global Reconstruction phase deploys Brute-force or Exhaustive search. Brute-force search is a typical method of problem-solving that involves systematical generation and examination of all possible permutations in order to find the desired solution. In case of reconstruction of a set of strokes, using brute-force search means that all possible permutations for the strokes, including the inversion of each stroke, have to be generated to allow selection of the most plausible case. Therefore, although a brute-force search is simple to implement and will consistently find the best possible solution for a given problem, its processing cost is proportionate to the number of candidate solutions. And in many practical problems, the size of the result set tends to grow rapidly, i.e. exponentially, as the size of the problem increases. Therefore, brute-force search is usually utilised when the problem is of a manageable size. And the problem of finding the correct order and direction of strokes, is no exception and if the number of strokes that are being considered to form a character is more than a few we will be facing a prohibitive result set.

However, generally speaking, in Latin and mathematics the number of strokes that form a character is not more than a few strokes as we will discuss in the following sections. This reconstruction method clearly differs from our colour-driven method in that it does not analyse the colour or width information.

In this chapter we will discuss the methodology of brute-force search for the recon-

struction of the strokes, i.e. both direction identification and stroke ordering, in Section 5.1 . Then, we present our experiments in Section 5.2, which were performed on a similar sample set as we discussed in Section 4.3 but with a few hundred more samples. In addition, these samples were taken from new writers and Chisel marker pens were used in a small portion of them. The overall recognition accuracy that we obtain from this method is 81.51%.

5.1 Methodology

In order to reconstruct the strokes using brute-force search, all permutations of drawing the strokes must be generated. The 8 possible permutations of the two strokes in the letter 'A' are depicted in Figure 5.1. Consequently, each possibility is passed to the recogniser and the possibility with the highest confidence from the recogniser is selected.

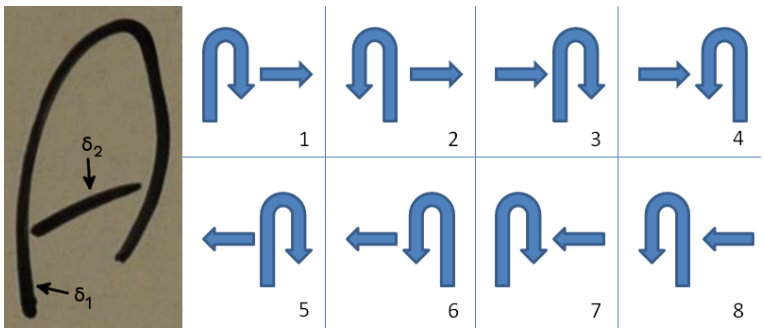


Figure 5.1: Brute-force search for drawing the two strokes in the symbol 'A' to the left: the 8 possibilities are shown on the right.

So in this example we have the two strokes δ_1 and δ_2, each of which are essentially an ordered list of pixels with two possible direction:

$$\delta_1^{\uparrow} = [(21,242)(21,241)(20,240)...(96,197)(95,197)(94,198)]$$

$$\delta_1^{\downarrow} = [(94,198)(95,197)(96,197)...(20,240)(21,241)(21,242)]$$

Where $\delta_1^{\uparrow}$ and $\delta_1^{\downarrow}$ denote the two possible directions of stroke δ_1. And similarly:

$$\delta_2^{\uparrow} = [(24,180)(25,179)(26,178)...(104,144)(105,144)(106,143)]$$

$$\delta_2^{\downarrow} = [(106,143)(105,144)(104,144)...(26,178)(25,179)(24,180)]$$

After generating all permutations we pass each of the possibilities to the recogniser and the following are the highest accuracy of each case respectively:

1. 69.48%
2. 72.26%
3. 68.75%
4. 69.65%
5. 69.65%
6. 69.9%
7. 75.08%
8. 66.41%

Consequently, permutation number 7 is returned and in this case the result set returned by the recogniser contains a single character which is the character 'A'.

As mentioned before, this method can be computationally very expensive as the possibilities of drawing the strokes have an exponential growth. More precisely, the number of permutations can be calculated by $\prod_{i=1}^{n} 2i = 2^n \times n!$ where n is the number of strokes. Therefore, the complexity of this algorithm is $\mathcal{O}(2^n \times n!)$.

Proof. Prove that $\prod_{i=1}^{n} 2i = 2^n \times n!$ for any $n \in \mathbb{N}$

Base case $n = 1$: Then, $2 \times 1 = 2^1 \times 1! = 2$. So, it holds for $n = 1$.

Inductive hypothesis: Suppose the theorem holds for all values of n up to some k, $k \geq 1$

Inductive step: Let $n = k + 1$. Then,

$$\begin{aligned}\prod_{i=1}^{k+1} 2i &= 2(k+1) \prod_{i=1}^{k} 2i, \text{ by our inductive hypothesis} \\ &= 2(k+1) \times (2^k \times (k!)) \\ &= 2 \times 2^k \times (k+1) \times k! \\ &= 2^{(k+1)} \times (k+1)!\end{aligned}$$

So, it also holds for $n = k + 1$. Therefore, by the principle of mathematical induction, it holds for all $n \in \mathbb{N}$. □

It has to be noted, when generating all possible ways for ordering a set of strokes, that each stroke can be drawn in two ways. Therefore, brute-force reconstruction can only be effectively employed - on a regular modern computer - if there are four or less strokes in

the symbol. With four strokes there are 384 possibilities to consider. But to generate and traverse through $3,840$ or $46,080$ cases, which is the number of possibilities for only five and six strokes respectively, would take a considerable amount of time. However, as far as we are aware, there are no maths symbols with more than five strokes, such as $\nVdash$ and $\nVDash$, in the list of maths symbols in LaTeX, apart from the symbol $\boxtimes$ which could possibly be drawn with six strokes.

In the next section we will have a quick look at the algorithm we use to generate all possible cases, given a set of strokes.

5.1.1 Algorithms

The brute-force search algorithm is quite straightforward. We essentially need two simple functions; one to generate all possible permutations and the other to discover the permutation with the highest accuracy from the recogniser. Here is the former algorithm:

Algorithm 2: Generating all possible permutations for a list of strokes.

Input : List of strokes.

Output: All possible permutations as a list of list of strokes.

```
1  allPermutations(strokes) begin
2      if strokes contains one element only then
3          add single stroke
4          add invert of single stroke
5      else
6          removed ← remove the first element from strokes
7          removedInv ← the reverse of removed
           // Depth First
8          df ← allPermutations(strokes)
9          foreach stroke in df do
10             currentStroke ← current stroke in df
11             for j ← 0 to number of segments in currentStroke do
12                 newStroke ← insert removed at j
13                 add newStroke
14                 newStrokeInv ← insert removedInv at j
15                 add newStrokeInv
16             end
17         end
18     end
19 end
```

Consequently, we traverse the result – the list of list of strokes – returned by this function and pass every list of strokes to the recogniser, tracking the one with the highest accuracy and eventually returning the permutation with the highest accuracy. To follow the example in Figure 5.1, permutation number 7 is returned as the most plausible

candidate.

5.2 Experiments

In this section we present our brute-force experiments. The objectives of this experiment are to verify the effectiveness of the following two techniques:

1. Preprocessing

2. Symbol recognition using brute-force reconstruction

Similar to our experiment that was discussed in Section 4.3, we ran experiments on sets of colour images photographed with the same camera from characters written on whiteboards by a number of writers in different colours. This experiment was designed to verify the effectiveness of our brute-force search method. In addition, as well as round headed marker pens that were used in the previous experiment, we added further symbols drawn with Chisel marker pens in this experiment which can further complicate the preprocessing step due to sharp edges that are produced by this type of pen, which can possibly cause edge detection to fail to detect entire boundary of strokes.

We then ran experiments on a set of 907 samples of characters on whiteboards from 10 different writers. This sample set contained 392 maths symbols and 515 Latin characters. Moreover, 513 of the samples were in red, 260 in black, 112 in blue and only 22 in green ink. The results of the experiments are given in tables 5.1 and 5.2, respectively. Table 5.1 gives a breakdown on the recognition rate of maths symbols versus Latin characters, whereas Table 5.2 breaks down with respect to colour of the pens.

We were able to achieve an overall recognition rate of 81.51%. This does not include preprocessing or reconstruction failures as discussed in the previous experiment. Furthermore, we also used the same online recogniser, which as explained was not specifically trained for whiteboard characters.

Table 5.1: Character recognition with Brute-force reconstruction: Maths vs. Latin characters

	Maths	Latin	Overall
Preprocessing Failure	1.27%	5.05%	3.42%
Recognition Accuracy	85.71%	73.40%	78.72%
Accuracy (Not including failures)	86.82%	77.30%	81.51%

Table 5.2: Character recognition with Brute-force reconstruction: Accuracy broken down by colour

	Red	Black	Blue	Green
Preprocessing Failure	2.92%	3.46%	6.25%	0%
Recognition Accuracy	81.48%	74.61%	73.21%	90.91%
Accuracy (Not including failures)	83.93%	77.29%	78.09%	90.91%

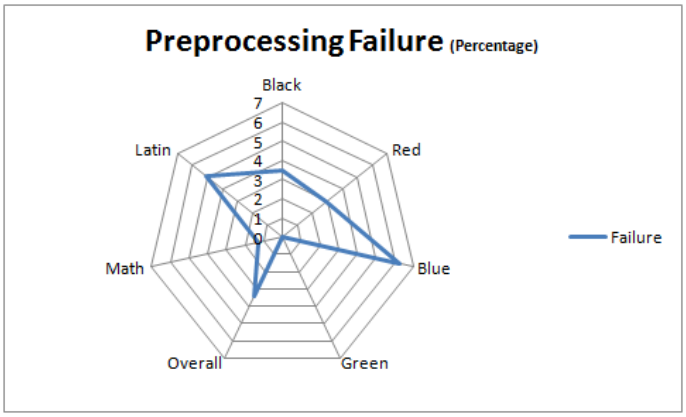


Figure 5.2: Character Recognition with Brute-force Reconstruction: Failure

As for a difference in recognition between the different symbols in this experiment, a variation we can observe is that, our methods deal with maths better than Latin characters both in preprocessing and recognition stages, and the latter is possibly due to the recogniser specialising on math. Furthermore, as for a variation between the colours, our methods clearly work better with red inks compared to black and blue, especially in

preprocessing. This is mainly due to the contrast between the stroke and the background colour. Although the green ink shows promising results, however, the number of samples in this ink were much less than the rest and therefore additional experiments should be performed to verify the propitious results. Figure 5.3 and 5.2 can better represent this difference in success and failure of our system for various colours.

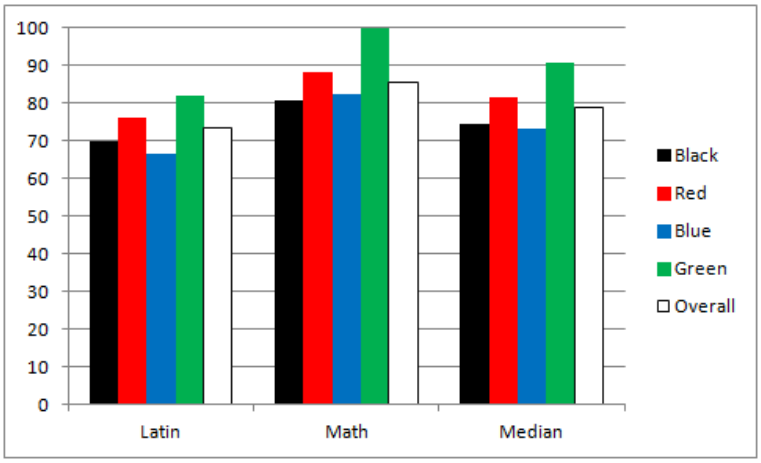


Figure 5.3: Character Recognition with Brute-force Reconstruction: Results

Furthermore, Figures 5.4 and 5.5 depict some examples of characters that were successful and unsuccessful during the recognition process respectively.

Figure 5.4: Examples of characters that were recognised correctly.

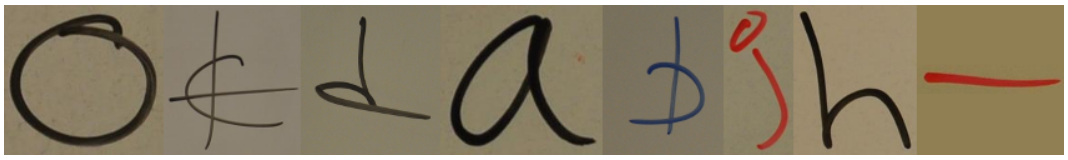


Figure 5.5: Examples of characters that failed to be recognised correctly.

As you can see in these figures, the intricate network of strokes would not necessarily cause the failure of a character in being recognised, however, unusual writing styles –

such as the “i” and “b” symbols in Figure 5.5 – does play an important role. However, further training of the online recogniser would improve the system’s ability to deal with such cases.

5.3 Limitations and Discussions

Although brute-force search outperforms our colour-driven reconstruction method, it comes with the cost of expensive processing resources if there are many strokes in the character as was discussed. Therefore, this method can suit Latin and mathematical symbols. But, without optimisation it would not be effective for languages in which having more strokes in a character is very common, such as Chinese. However, brute-force search can be optimised through the use of heuristics to reduce the set of candidate solutions to a manageable size. We will discuss this further in the next chapter.

The timing comparison of the colour-driven and brute-force reconstruction methods is presented in the table below:

Table 5.3: Timing comparison between Colour-driven and Brute-force reconstruction methods in milliseconds.

Number of Strokes	Brute-force	Colour-driven
1	91	93
2	94	57
3	120	96
4	833	101
5	7297	193

It must be noted that the timing of the colour-driven reconstruction highly depends on the size of the images – due to the fact that the colour of the pixels on the strokes need to be gathered – and therefore, it does not necessarily increase as the number of strokes increases. In contrast, brute-force reconstruction does not depend on the size of

the images and it is clear to see the dramatic increase in the timings as the number of strokes increases.

The above timings were gathered from a total of 150 samples; 30 characters in each category, i.e. characters with one through five strokes. Figure 5.6 shows the comparison of the *normal distribution* between colour-driven and brute-force reconstructions in this experiment.

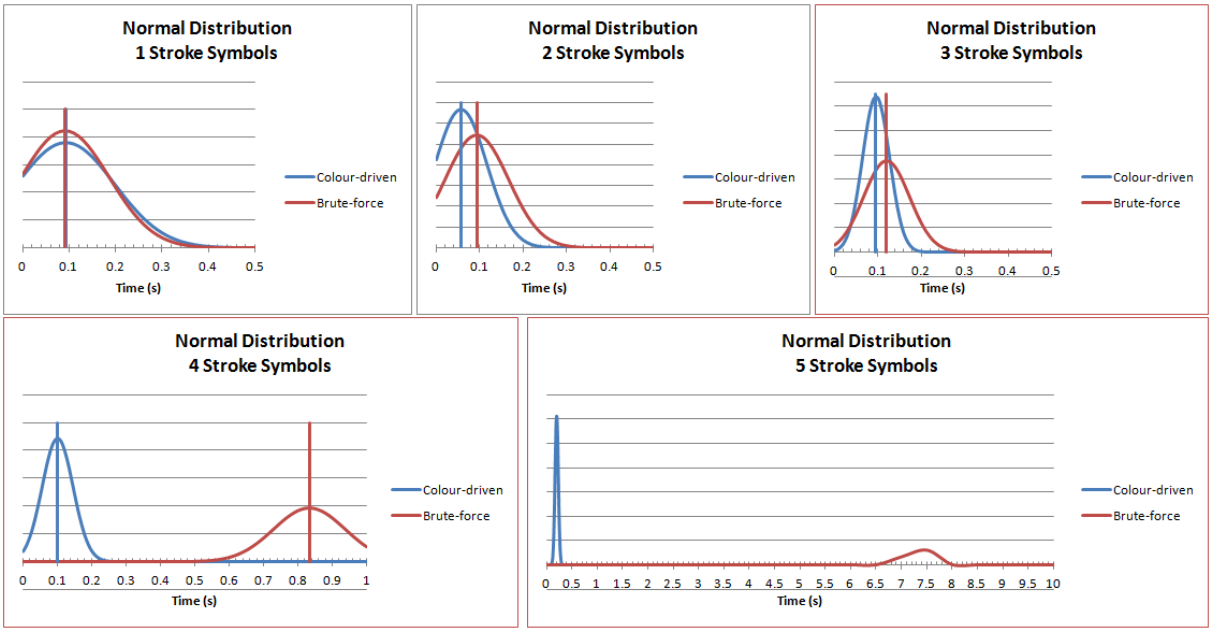


Figure 5.6: Comparison of the Normal Distribution of timings for *colour-driven* and *brute-force* reconstructions.

On a different note, since many of the online maths recognition systems restrict the number of strokes to four and work only with consecutive strokes, this method could be useful for online maths symbol recognition systems as well; to simplify the variations in writing of maths symbols (stroke ordering), by making the training for these systems easier and increasing their recognition accuracy. In addition, this method can be optimised in such systems because the direction of the strokes is already a given.

5.4 Summary

In this chapter, we discussed the details of our *Brute-force Reconstruction* method. More precisely, we presented how *brute-force search* can be used to generate all permutations for a set of strokes forming a character, which in fact combines the two stages of Global Reconstruction, i.e. **Chronological Ordering of Strokes** and **Stroke Direction Identification**. In addition, a simple algorithm was presented to generate all permutations using this method. We then showed the effectiveness of our approach through experiments.

Subsequently, the main limitation of this method was discussed; the resources required to compute a result in a timely manner as the number of strokes increases. Also, we presented a timing comparison between Colour-driven and Brute-force reconstruction methods and showed how quickly the process can become interminable as the number of the strokes increases. In the next chapter we present our *Informed Reconstruction* method that will specifically address this issue.

As for the performance of Brute-force compared to our Colour-driven method, we have seen that the former – with 81.51% accuracy – outperforms the latter method – with an accuracy of 78.84% – which as mentioned previously in Chapter 5, outperformed other similar offline recognisers for whiteboards. However, it should once again be noted that these systems have been tested on very different datasets and a direct and meaningful comparison cannot be established without testing the systems on the same sample sets.

CHAPTER 6

INFORMED RECONSTRUCTION

As observed in Chapter 5, although *Brute-force Reconstruction* outperforms our *Colour-driven Reconstruction*, it is a very computationally expensive method. Therefore, in order to enhance the reconstruction of strokes, it was intuitive to try to synthesise the two methods to produce one that can take advantage of each algorithm's strength.

On that account, we introduce two new reconstruction methods, namely, *Randomisation* and *Gradatim* methods. The former tries to produce random permutations from an initial state and choose the best one, while the latter aims to use the recognition and terminating point analysis accuracy results, in a step-by-step fashion, to find the best solution.

In the rest of this part we will discuss the methodology of our *Gradatim* technique in Section 6.1 and *Randomisation* technique in Section 6.2. We will then explore the limitations of the methods in Section 6.3. The experiments with our *informed* reconstruction method will be discussed later in Chapter 9.

6.1 Gradatim Methodology

One of the ways that seemed intuitive to significantly speed up the brute-force reconstruction algorithm, was to design a step by step method that would take advantage of the accuracy (percentage) of the results returned by the classifier in colour-driven recon-

struction[1] – which was trained to indicate the direction of strokes – and also the online recogniser[2]. The purpose of this design was two-fold; firstly, it would speed up the reconstruction process by testing each permutation on the bounce, without having to initially generate many permutations, and secondly, this efficiency would also allow the method to be effective for symbols with more strokes, especially to be used for symbol recognition in other languages.

However, it soon became apparent that neither of the methods – the classifier nor the online recogniser – reveal any clues as to whether or not the permutation under analysis is getting closer to the optimum result. More precisely, a permutation that would arrive at the optimum with a single swap between two strokes can receive a lower recognition accuracy than one which would need many more by the online recogniser. We experienced a similar situation with our *stroke direction identification* method. This is because the *classifier* that was trained can return a high percentage value where the stroke should in fact be inverted or vice versa.

Therefore, in order to optimise our methods, it was clear that we had to generate a number of permutations before selecting the best out of them, which lead to our *informed randomisation* method.

6.2 Randomisation Methodology

In order to optimise our reconstruction methods, since the ideal Gradatim method was not successful, it was decided that we should take advantage of speed of the Colour-driven Reconstruction, considering that it does not actually perform very poorly compared to the decisive albeit very expensive Brute-force counterpart[3]. Therefore, in this method we do exactly as explained in Chapter 4 initially and then try to produce an acceptable amount of informed permutations at random from there. An acceptable amount is regarded as

[1]See Chapter 4 Section 4.1 for more details.
[2]See Chapter 3 Section 3.4 for more details.
[3]See the results of the experiments in Section 4.3 and 5.2.

a number of possibilities that can be generated and examined within a few seconds on a regular contemporary computer.

In addition, after further analysis of data from our earlier Colour-driven experiments, it was realised that the ordering of the strokes is the main cause of failure in the recognition process rather than the direction of strokes in characters with multiple strokes. Therefore, it was intuitive to have two separate algorithms to generate permutations for a set of strokes at random; one that only generates random permutations without changing the direction of the given strokes – $A1$ –, and the second to only generate permutations by only changing the direction of strokes at random – $A2$ –. This distinction can help us to produce more permutations based on reordering of strokes, rather than changing their directions.

So with that in mind, we design the two algorithms and generate permutations in such a way that we first generate n permutations through $A2$, where n is the number of available strokes, and then generate a further n^2 permutations through $A1$ for all n permutations that have been produced in the first stage. For example, if the symbol we are considering has four strokes, then we first produce four permutations through $A2$ plus the initial state itself which was the result of the colour-driven technique. And then we produce a further 16 permutations for the five permutations that we already have, which makes a total of 80. Hence, we essentially produce $(n + 1) \times (n^2)$ possibilities. It is clear to see that the number of permutations generated in this method is much less than that generated with brute-force search, which for four strokes would be 384. It must be noted that for symbols with less than 4 strokes because on one hand, it would be impossible to generate $(n + 1) \times (n^2)$ possibilities in this method, and on the other hand brute-force search would not actually be slow to generate all permutations, we will still be using our brute-force method for symbols with 1 and 2 strokes, and generate $(n + 1) \times (2n)$ permutations for symbols with 3. Table 6.1 shows the number of permutations generated with this method compared to that of our Brute-force Reconstruction method.

Table 6.1: Number of possibilities generated with Brute-force and Informed reconstruction methods.

Strokes	Brute-force	Informed
1	2	Same as brute-force
2	8	Same as brute-force
3	48	$(n+1) \times (2n) = 24$
4	384	80
5	3,840	150
6	46,080	252
10	3,715,891,200	1,100
n	$2^n \times n!$	$(n+1) \times (n^2)$

We will have a look at the two algorithms in our *informed randomisation* method in more detail in the next section.

6.2.1 Algorithms

In this section we present the two algorithms that were discussed in the previous section. As you can see in the following algorithms, these methods contain two levels of randomisation:

- One that decides how many strokes will be affected by the procedure, i.e. how many times the state of the symbol is changed.
- The other to decide which strokes should be swapped or inverted in each iteration.

Although we only use the following methods for symbols containing 3 or more strokes, since the required number of permutations can be passed as a parameter, they can be used for any symbol given that the number of required permutations does not exceed $n!$ in the first algorithm and 2^n in the second, where n is the number of strokes in the character.

Algorithm 3: Generating random lists of strokes (No inversion).

Input : List of strokes (strokes) & required number of permutations (n)

Output: n random permutations as a list of list of strokes.

```
1  randomPermutationsNoInversion(strokes, n) begin
       // generate n non-equal random permutations
       // n must not be greater than factorial of (length of strokes)
2      while size of results < n do
3          numberOfChanges ← a random number between 0 and length of strokes
4          current ← strokes
5          for i ← 0 to numberOfChanges do
6              a ← a random number between 0 and length of strokes
7              b ← a different random number between 0 and length of strokes
8              swap a'th and b'th strokes in current
9          end
10         if results do not include current then
11             add current to results
12         end
13     end
14     return results
15 end
```

Note that, in both of these algorithms each randomisation iteration starts from our initial state which was produced through the Colour-driven Reconstruction method. This is to make sure that the generated permutations do not stray too far from our initial state for the reasons that we discussed already in the previous section.

Algorithm 4: Generating random lists of strokes (Inversion only).

Input : List of strokes (strokes) & required number of permutations (n)

Output: n random permutations as a list of list of strokes.

```
1  randomPermutationsInversionOnly(strokes, n) begin
       // generate n non-equal random permutations
       // n must not be greater than 2^(length of strokes)
2      while size of results < n do
3          numberOfChanges ← a random number between 0 and length of strokes
4          current ← strokes
5          for i ← 0 to numberOfChanges do
6              a ← a random number between 0 and length of strokes
7              invert a'th stroke in current
8          end
9          if results do not include current then
10             add current to results
11         end
12     end
13     return results
14 end
```

6.3 Limitations and Discussions

Although this method has done well on our sample set as we will see in Chapter 9, it should still be tested for characters that actually have more than four strokes to see how effective it can actually be for such characters. But since there are only a few maths symbols with such criteria and plus the online recogniser that we used was only trained to deal with symbols with four or less strokes, unfortunately we could not test this method further.

Moreover, it has to be noted that since this method produces somewhat random[1] permutations, it might not consistently produce the same final result, which might be desired in certain applications.

The timing comparison between the Informed (randomisation) and Brute-force Reconstruction methods is presented in the table below, we have omitted comparison between the two methods for symbols with one and two strokes since both use the same method:

Table 6.2: Timing comparison between Informed and Brute-force reconstruction methods in milliseconds.

Number of Strokes	Brute-force	Informed
3	120	105
4	833	367
5	7297	518

It is worth noting that a fluctuation in timings is observed occasionally for *informed* reconstruction, which is due to the fact that if a permutation is generated that has already been encountered, it is discarded and the iteration starts over. Nonetheless, this incident is expected to be decreased as the number of strokes in symbols increases and besides, it would not cause the algorithm to be as slow as Brute-force as our experiments confirm.

The same sample set as our experiment in Chapter 5 were used for this experiment – total of 150 images; 30 characters in each category. Figure 6.1 shows the comparison of the *normal distribution* of the timings between informed and brute-force reconstructions in this experiment.

[1]Clearly, the permutations generated through the *randomisation* method that we discussed here are not entirely random, since we have a set initial state and we start from that state in each iteration.

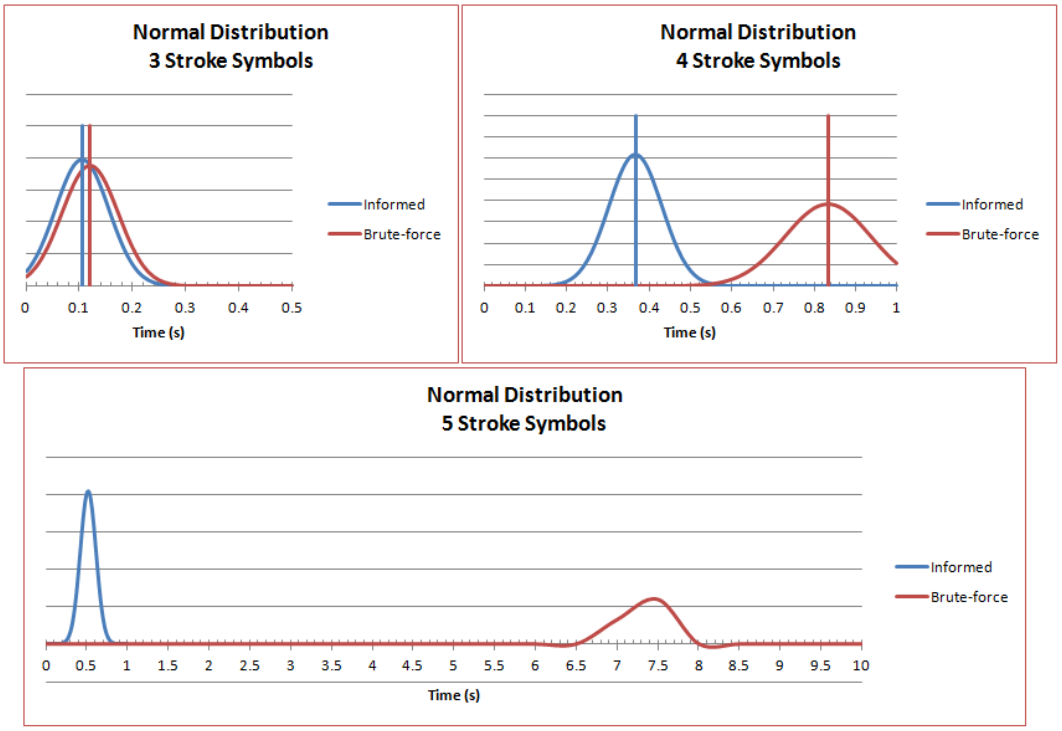


Figure 6.1: Comparison of the Normal Distribution of timings for *informed* and *brute-force* reconstructions.

6.4 Summary

The details of our *informed reconstruction* method was discussed in this chapter. We initially described two informed reconstruction methods, i.e. **Gradatim** and **Randomisation**, that aim to improve the complexity of the Brute-force Reconstruction. We explained the reasons why the former method was not successful and how the latter could be implemented. In addition, *randomisation*'s complexity was compared to that of Brute-force, both in the number of permutations generated and timing-wise. We also identified some limitations of this method in our experiments, the most important of which. We will later on see the results of our experiments with *informed* reconstruction method in Chapter 9, where we also compare the results of our methods on the same sample set.

Part III

Segmentation and Recognition of Maths Expressions

CHAPTER 7

COMPONENT-BASED SEGMENTATION

In the previous part of this thesis we looked at how images of single characters on whiteboards can be transformed into a computer processable format. However, in order to recognise characters inside a mathematical formula and eventually transform an entire formula into a processable format more analysis is required, which starts with the segmentation of symbols.

Unlike the recognition of characters, words and in turn sentences in Latin and many other languages, where on one hand you can essentially take a linear approach and on the other hand make use of dictionaries to help solve ambiguities to fulfil this task, the complex two-dimensional structure of mathematics debars one from deploying either of those approaches.

Also, in mathematics the spatial relationship between the symbols plays an important role or in fact act as an implicit function. Take the two symbols "a" and "b" as an example. Depending on the configuration of these symbols, we could have a_b, a^b and ab, which fundamentally have very different meanings. Moreover, configurations such as these make segmentation and layout analysis of handwritten mathematical formulae even more difficult, where differences in writing styles make distinguishing between these cases, i.e. horizontal adjacency and super/subscripts, a daunting task. And in case of whiteboards, handwritings tend to be more sloppy due to the fact that you would not normally have your hand firmly rested on the surface and also because of writing in a stand-up position,

that can further complicate the process.

Furthermore, many symbols in mathematics have ambiguity in their capacity; for instance, a horizontal line can represent:

- a subtraction function ($a - b$),
- a vinculum placed over a set of symbols indicating that all the symbols make a single group ($\overline{ab}$),
- a negation function (a bar on top of a symbol, $\overline{a}$) in some context, or
- a fraction line ($\frac{a}{b}$).

All these factors can make the recognition of maths symbols difficult.

Figure 7.1 depicts an overview of mathematical expression recognition excluding *maths content interpretation* with the four stages of our *component-based segmentation*.

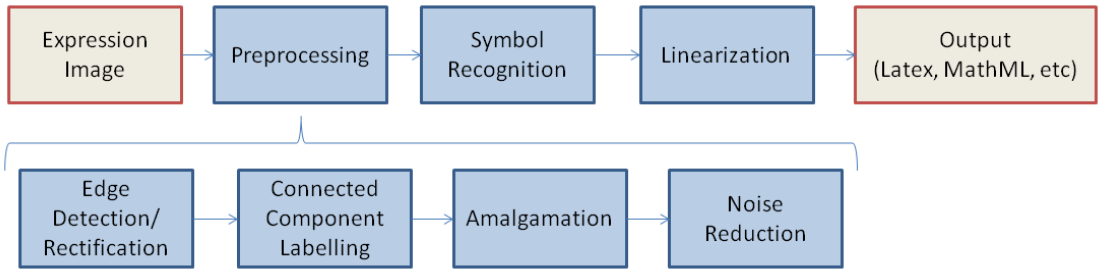


Figure 7.1: Overview of mathematical expression recognition.

In the rest of this chapter, we will discuss this segmentation method which relies on the distance between the symbols to perform this operation, skipping over Edge Detection that was already discussed in Chapter 3. More precisely, we will first discuss *Connected Component Labelling* in Section 7.1. Then *Amalgamation* techniques, which play an important role in the segmentation of maths symbols, will be explained in Section 7.2. Afterwards, in Section 7.3, we will look at (global) *Noise Reduction*, which differs from noise removal at symbol level (local) that was explained earlier. We will also discuss a

Linearizer in Section 7.4 and finally, the discussions about these methods will be presented in Section 7.5.

7.1 Connected Component Labelling

The idea in the *Component-based Segmentation* method is that we can segment the symbols according to the structure of the *connected components* inside a mathematical formula. Therefore, in this method the first step would be to find the connected components in the image. Bearing in mind that all we have initially is a digital colour image of the entire maths expression, once again we use *edge detection*[1] to essentially transform our colour image into a binary image of only the edges of the components or symbols. We can then extract the connected components from the binary image.

A *Connected Component* in the image processing field refers to a set of *connected pixels*[2] and the process of distinguishing all sets of connected pixels is called *Connected Component Labelling* or CCL. Similar to thinning algorithms, CCL algorithms also work on pixels based on 4 and 8-connectivity of neighbours. In the same way as our thinning process, we are interested in the 8-connectivity of pixels for our CCL algorithm. This is because in images of handwriting we do end up with many diagonal lines, and in certain configuration of pixels, single components could be broken into many components in the CCL process with 4-connectivity of neighbouring pixels.

There are many CCL algorithms in the literature. Here we will summarise a few approaches:

Two-pass with equivalence class resolution:[HS91]

This algorithm essentially scans the image twice starting from the top left-hand of the image and scanning in a left-to-right and top-to-bottom fashion in that order.

In the first round, temporary labels are assigned to each pixel. This is done in such a way that each pixel's surrounding neighbours are examined and if no neighbour has been

[1]See Chapter 2 Section 2.2.3 for more details.
[2]See Chapter 3 Section 3.2 for the definition of connectivity of pixels.

visited during the scan, the corresponding pixel is assigned a new label value or *class*. Otherwise, in the case that one of the neighbours has been visited before and therefore, already has been labelled, we assign the same label value to the corresponding pixel. If a pixel is encountered that has more than one neighbour which has been visited, the smallest value amongst the neighbours is chosen and assigned to the corresponding pixel. In the latter case, note is taken of the equivalent classes that are encountered.

In the second round, we check the value of each assigned label and compare to the noted equivalent class list. If a smaller equivalent class exists, we assign the smaller value to the corresponding pixel.

Grass-fire approach:[Pit93]

This approach essentially uses the *floodfill* algorithm to find the similarly coloured neighbours of a pixel and likewise for the neighbours, until there are no more new neighbouring pixels, i.e. the entire connected component is obtained. You would then try to find other components in the same way.

Graph data structure approach:[HLP10]

In this method, each pixels is represented as a node in a graph and then lists are made out of the connected nodes to form the connected components. However, this method could be inefficient if implemented using object-oriented languages.

The result of a CCL algorithm is the bounding box information of the connected components in the image. The bounding box information include the minimum x, y coordinates, meaning the minimum x and minimum y amongst all pixels contained within the connected component. Note that the x and y do not necessarily belong to the same point. It also includes either the maximum x, y coordinates or the $width$ and $height$ information of the connected components. Figure 7.2 depicts an example of the bounding box information for the character C. In this example x_1 and y_1 refer to the minimum, and x_2 and y_2 to the maximum values.

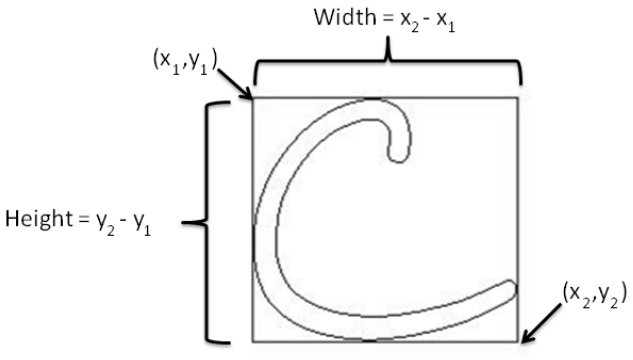


Figure 7.2: Example of the bounding box of a character.

7.2 Amalgamation

In mathematical handwriting recognition, one of the main challenges in successful segmentation, and in turn recognition, of characters is to identify and merge connected components that belong to the same symbol. There are various types of connected components that should be merged carefully in order to avoid extra gratuitous processing and additional noise being introduced. Here we break down the required amalgamations into two main categories, namely *internal* and *external* amalgamation. We then further classify the external amalgamation into *horizontal* and *vertical*. We will spend the next few sections exploring these approaches.

7.2.1 Internal Amalgamation

After the connected components have been located in the image, since we use edge detection, there may be connected components inside one another that are in fact parts of the same symbol, such as inner loops, or artefacts created during the edge detection process. See Figure 7.3 for some examples of internal components that need to be merged with the enclosing component. This example includes an inner loop (Connected Component #3) and a small artefact (Connected Component #2).

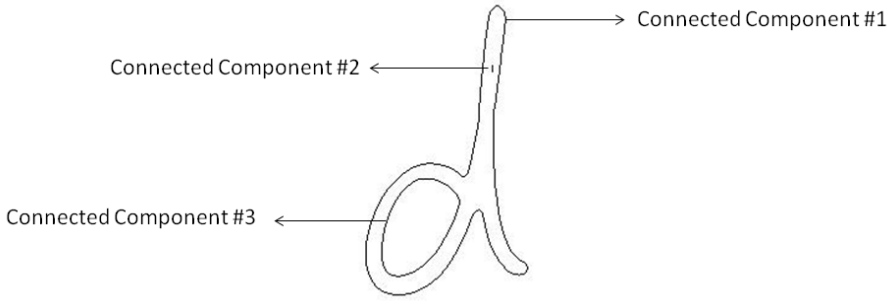


Figure 7.3: Example of multiple connected components forming a character.

In order to identify such instances we examine connected component bounding boxes and mark the ones that are situated within another for further investigation. In the next step, we check whether the smaller components' points are actually within the boundaries of the larger component's edge detected area – not the bounding box. Note also that by points here we mean the points of the connected components rather than their bounding boxes. We use Algorithm 5 to decide whether the points of one connected component lie within another.

Algorithm 5: Checking if a point lies within a connected component.

Input : Connected component points (ccPoints) & the point to be checked (p)
Output: Whether or not the point is inside the connected component

```
1  isPointInsideConnectedComponent(ccPoints, p) begin
2      left ← all ccPoints that their y is equal to p.y and their x is to the left of p.x
3      right ← all ccPoints that their y is equal to p.y and their x is to the right of p.x
4      up ← all ccPoints that their x is equal to p.x and their y is to the north of p.y
5      down ← all ccPoints that their x is equal to p.x and their y is to the south of p.y
6      Remove all the points in neighbourhood of one another in left, right, up and
       down, such that only one of the points is kept
7      if number of elements inside left, right are both odd or number of elements
       inside up and down are both odd then
8          The point is within the connected component.
9      end
10 end
```

Although rationally this method should be very accurate, and would be if perfect geometric shapes were used, in reality the complex shapes of the edge detected handwritten

characters frustrates the algorithm from producing accurate results all the time and for any given point. It is merely because of this that we use the logical *or* function on line 7 in Algorithm 5, otherwise, the *and* function would have been used instead. Moreover, because of this deficiency, when checking whether an entire connected component lies within another, we allow up to 15% shortage. In other words, we count the number of points that fall inside the connected component and even if 15% of the points are calculated to be outside the connected component, we still consider the smaller shape to be inside.

7.2.2 External Amalgamation

In spite of the fact that there can be complications in Internal Amalgamation, yet, it is considered to be a simpler task in comparison to the External counterpart. In offline handwritten maths recognition, amalgamation of external components is a very difficult task mainly due to the varieties in handwriting styles and here we divide them into two separate categories in order to simplify the task:

Vertical Amalgamation

Some symbols contain two or more separate connected components. In other words, there are symbols with multiple strokes where the strokes never intersect one another, such as "=" or "*i*". Most external amalgamations fall into this category and in order to identify the components that need to be merged vertically, we begin by finding small components, i.e. components with an area at least three times less than the average area of components. The area is calculated based on the bounding box of the components. Then, we locate the closest component to the north or south of the corresponding component, and if the distance between the two is less than half the average height of all components in the expression, the components are merged.

Horizontal Amalgamation

For horizontal amalgamation of external components, we are mainly interested in components that actually slightly overlap one another and not the ones that are contained within another. If such case is encountered we calculate the area of intersection. If the intersection area size and also the areas of each of the components alone are all less than the average size of the components, we mark them for further analysis. We check the areas of the components to avoid merging larger components such as brackets with smaller ones – See Figure 7.4 for an example of such case. This is because such symbols usually have a large bounding box and they can easily intersect others in an expression. In the last stage, we check the marked components and calculate the closest distance between any two points of the connected components themselves (and not the bounding boxes) and if they are closer than a predefined distance, the two components are merged. We have set this predefined distance to be twenty pixels in our experiments.

7.2.3 Cleaning Overlapping Bounding Boxes

In order to make the image of a component ready to be passed to the *Character Recognition Kernel*, we need to make sure that the image does not contain parts of image of other symbols. See Figure 7.4 for an example of a case where there are overlapping symbols in the image.

If such case is encountered we use Algorithm 5 that was discussed in Section 7.2.1 and remove any point that falls inside the intruding connected component from the corresponding image (see Figure 7.5). This ensures that the symbol recognition module would only examine the desired symbol and minimum noise is introduced in the given symbol image. But, as mentioned before the algorithm is not perfect and therefore there might be some residue pixels. However, these residues are usually not identified by edge detection algorithm because of the fact that they are often scattered and when they are the noise reduction module will take care of them.

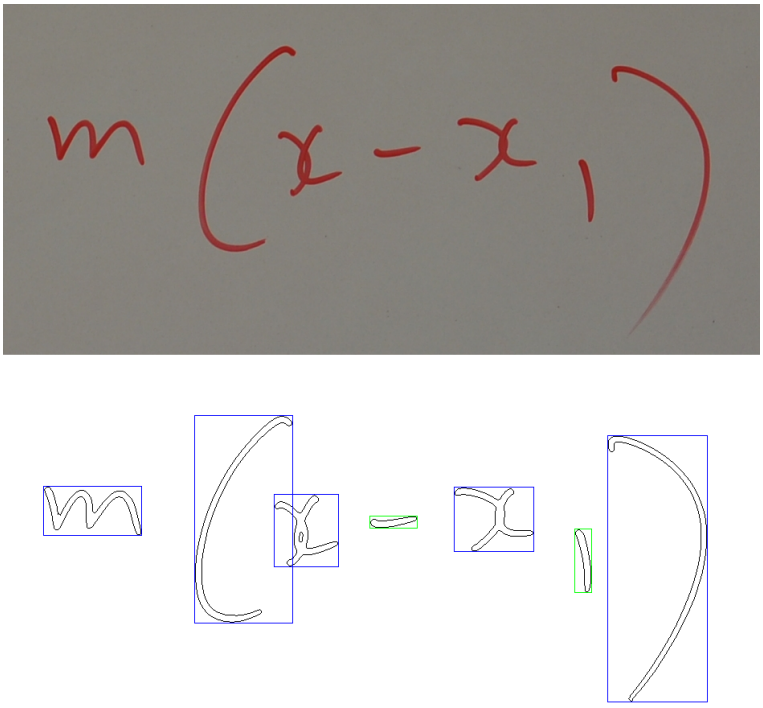


Figure 7.4: Example of symbol bounding boxes overlapping

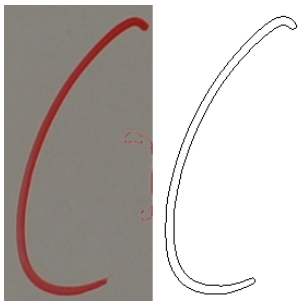


Figure 7.5: Example of clearing image from intruding components and the result of *edge detection*.

7.3 Noise Reduction

After clearing the symbol images from possible overlapping symbols, before we pass on the symbol images to the character recognition kernel we need to make sure that we reduce the noise in the images as much as possible. In particular, we want to make sure that we do not pass images containing only noise and no actual character.

As explained in the previous part of this thesis – in Section 3.2.3 –, the character recognition kernel has its own noise reduction module, which will attempt to take care of any possible merged noise with higher precision since it has more information, such as the width and colour of strokes, available to it. Therefore, at this point we are mainly interested in noises that fall outside the bounding box of other symbols and not really worried about any noise that might have been merged with or included in other symbols' images.

With that in mind, once again the resulting connected component bounding boxes are examined and small components that have not been merged with others will be identified and removed. These components are usually less than the average line width, which in our experiments were between 7 to 25 pixels depending on the size of the image and the marker pen used. These components include two common types of noise on whiteboards, namely *ink residual* and *deterioration marks*. It is very common even on a cleaned whiteboard to have some ink residual which may or may not appear in the image. Moreover, it is not atypical for small deterioration marks to appear on whiteboards and these marks are also often small in size. Figure 7.6 depicts examples of various noise that appear in images of mathematical expressions. We classify the noise in the images based on their position into three categories:

- Inner Noise
- Outer Noise
- Merged Noise

Subsequent to this process, we pass each symbol image to the *character recognition module* for further analysis and ultimately to identify the contained character.

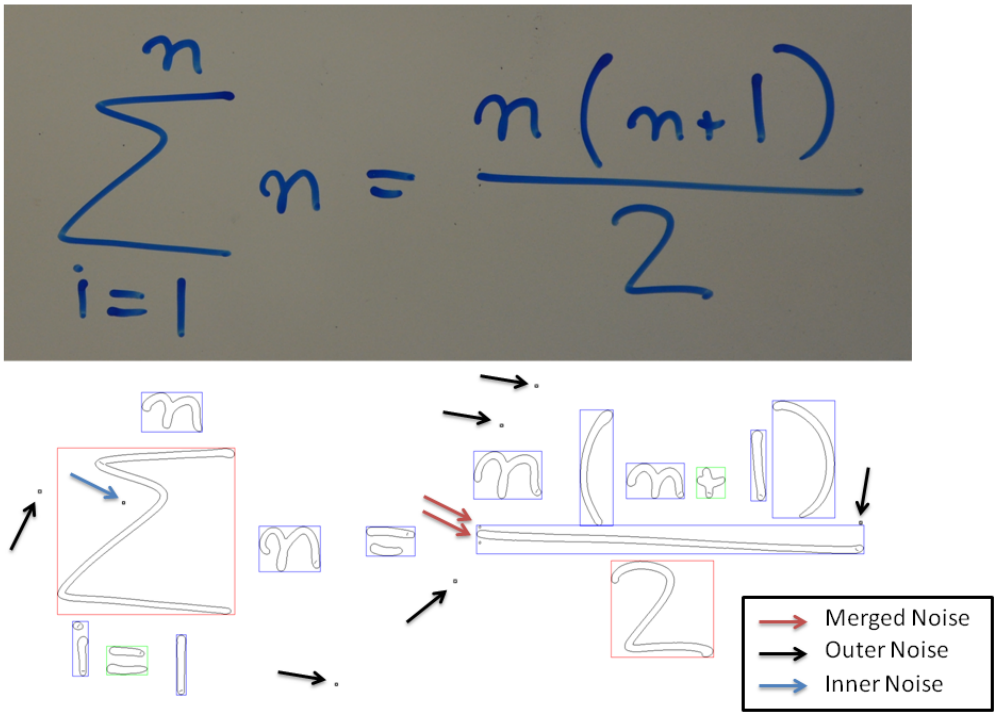


Figure 7.6: Example of various types of noise based on their position.

7.4 Linearizer

After the characters have been recognised, we essentially need to reconstruct the symbols in order to create a computer processable mathematical expression. For this purpose, one can use the *extended Linearizer* explained in [BSS09] which is a modified version of the *Linearizer* first introduced by Anderson in [And68]. Since Anderson's algorithm was only designed to work with relatively simple algebra, Baker et al. extend his algorithm to work with a larger range of mathematical content. This extended linearizer was originally designed to work with mathematical expressions in PDF documents.

However, all the *extended linearizer* essentially requires is the bounding box information for the symbols together with their Unicode metadata[1] with the exception of fraction lines, for which a *line* command is expected instead of the Unicode metadata. Therefore,

[1]The bounding box information and Unicode metadata are required in form of *JSON* files.

given that well positioned characters are passed to the linearizer, they have implemented two drivers namely LaTeX and MathML, in order to produce different outputs in the desired format depending on the requirements.

Though, producing well positioned characters, similar to that in machine printed expressions, for handwritten symbols is not a trivial task. More precisely, sometimes it can be very difficult to find out the spatial relationship between the symbols such as when a symbol acts as a sub or superscript to the previous symbol, which has been mentioned in many previous studies [ZBC02, CY00]. See Figure 7.7 for an example of bounding box ambiguities that can arise in handwritten mathematics.

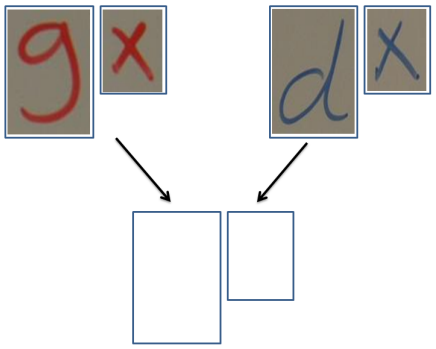


Figure 7.7: Spatial relationship of two characters in an expression according to their bounding boxes.

Typographical centre of symbols, a.k.a. vertical and horizontal base-points or x and y centre, is used in many studies in an attempt to solve the problem. However, it is not clear how different systems calculate this information. Anderson in [And68] explains that the x centre is always the average of minimum and maximum x for the character, and y centre is calculated from min and max y by a function which depends on the particular character. He also mentions that it is best for the character recognition engine to provide this information because of knowing the identity of the characters, however, he never actually explains how the horizontal base-point should be calculated. Baker et al. receive this information from the PDF documents and claim that this information is provided about every character and therefore, never calculate it.

In addition, as our experiments indicate, the size of the symbols cannot help uncover the spatial relationship between the adjacent characters either, due to varieties in handwriting styles. Moreover, an attempt was made to calculate an estimate font size for characters by the ratio between foreground and background pixels in the bounding box of the characters, however, this method has its limitations as well. To be specific, symbols such as "−" or "1" where there is only a straight horizontal or vertical line, would have a very high ratio. Besides, in some cases super/subscripts would not be any different to normal characters in terms of their font size. Therefore, these factors make this method impractical without further analysis of the symbols' identities at least.

Therefore, we made an attempt to directly pass the bounding box information together with the Unicode metadata of each symbol to the *linearizer* and as one can expect the results were amiss in most cases. Figure 7.8 shows a couple of our best generated examples of the recognised expressions. It is clear to see the semantical mistakes even in these samples that most of the characters have been recognised correctly.

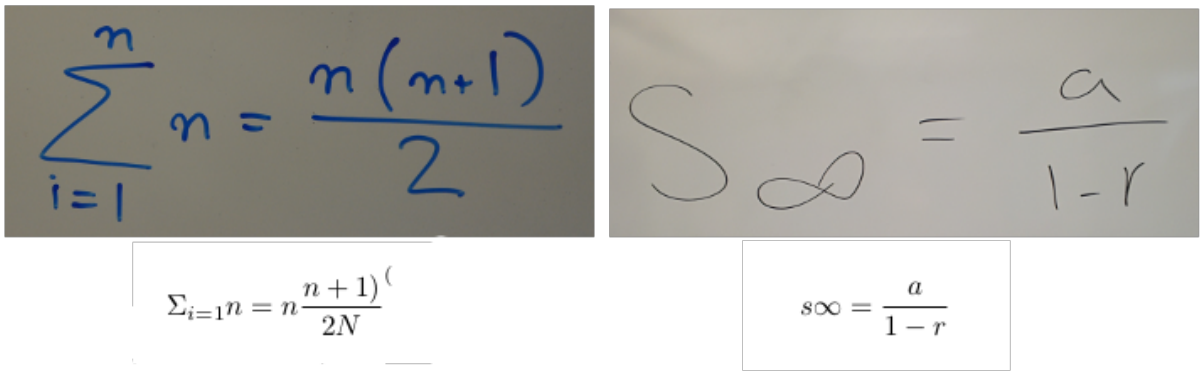


Figure 7.8: Example of PDF files produced through *linearizer*.

We have already seen an overview of the entire process, from the image of the expression to the LaTeX or PDF output in Figure 7.1. But to summarise, after the initial preprocessing steps, the outcome of which would be the the bounding box information of each character, the characters are identified in the *Symbol Recognition* phase. Subsequently, we pass these information to the *Linearizer*, which generates the LaTeX output.

7.5 Discussions

As explained in Section 3.4, our recogniser usually returns multiple results for a given set of strokes and as discussed in our experiments in Section 4.3, not always the first preference of the recogniser is the correct choice. Selecting the right choice from the returned list can be a tricky task. One possible solution to this would be to perform post-processing techniques such as *constraining outputs* to choose the best and not necessarily the most confident result returned by the recogniser.

It would be useful to have classes for characters so that a grammar could be established. This can then help to solve ambiguities by comparing the symbols to their neighbouring symbols in order to decide the likelihood of a symbol over the rest of the recognition set returned by recogniser.

A limitation of this method is that it cannot deal with touching characters in the expressions. We will address this problem and try to find a solution to it in the next chapter.

7.6 Summary

In this chapter we investigated a *component-based* segmentation method and how the connected components in an expression image can be used to segment the symbols. More precisely, we discussed **Connected Component Labelling** algorithms and the fact that symbols can consist of more than one connected component. Therefore, **Amalgamation** methods were explored to overcome the difficulties introduced by this fact. In order to simplify the problems in *amalgamation*, we then categorised them into **Internal** and **External** procedures. Then we considered overlapping bounding boxes and the way we can apply cleaning in order to simplify the task character recognition. We also looked at a simple *noise reduction* process before the image of the characters are passed to the *character recognition kernel* for the recognition task.

Subsequently, we discussed how we can use a *linearizer* in order to transform the

bounding box information together with the corresponding Unicode metadata into computer processable formats such as LaTeX and MathML. However, more work is needed on this part before meaningful results can be achieved. Finally, we had discussions about the methods introduced in this chapter.

In the next chapter, we will introduce another segmentation method which specifically will try to address some of the limitations of *component-based* segmentation such as touching characters.

CHAPTER 8

STROKE-BASED SEGMENTATION

In this chapter we introduce a segmentation method that, unlike *component-based* segmentation – which relied on connected components in order to perform segmentation – relies on strokes in order to segment the symbols. The main motivations behind this segmentation method are the following:

- To be able to deal with touching characters.
- To minimise the limitations caused by the differences in handwriting styles, such as the varying gaps between symbols within the same expression, which makes *component-based segmentation* difficult.
- To deal with noise more prudently by starting to extract information early in the process.

Figure 8.1 shows an overview of the Stroke-based Segmentation process.

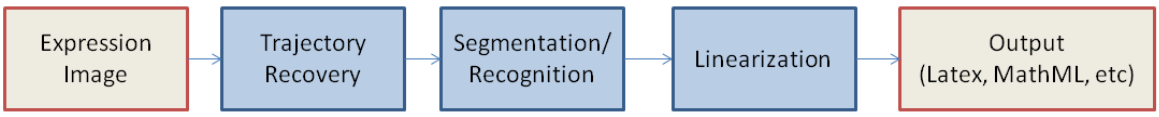


Figure 8.1: Overview of *Stroke-based Segmentation*.

A noteworthy difference between this method and *component-based* segmentation is that this method essentially combines the recognition and segmentation steps into a single

process.

In this method the strokes have to be reconstructed before the segmentation of the characters could take place and the trajectory recovery techniques discussed in Chapter 3 can be used in order to do so. Once the trajectory of all strokes in an expression have been recovered, we can initiate the segmentation and recognition tasks simultaneously.

In the rest of this chapter we will focus on the segmentation problem in Section 8.1 and the related discussions in Section 8.2.

8.1 Segmentation and Recognition

In this segmentation method, the idea is to use the recognition confidence together with a reconstruction method[1] to generate the best possible set of characters for a given set of strokes.

Since there can be a large number of strokes in an expression, i.e. an expression would usually consist of many symbols, it would both be inefficient and futile to try to start by considering all the strokes at once for this task. Therefore, for the purpose of making a more efficient algorithm, first we begin by calculating the average distance between the strokes.

So in order to calculate the average distance between the strokes, all strokes are traversed and for each stroke we find the closest horizontal and vertical strokes, ignoring the strokes that cross one another. We then take the highest value between the horizontal and vertical distances (if they both exist). Although, it would be inefficient to start from all strokes at once as mentioned, we still want to allow as many strokes as possible to fall within the average distance, which is why we take the highest value between the two. Note that to calculate the distance between the strokes, we essentially consider the distance between their bounding boxes. Once we have gone through all the strokes we calculate the average distance between them.

[1]Due to the limitations of *colour-driven* and *brute-force* reconstructions that were discussed in Chapters 4, 5 & 6, *informed* reconstruction is preferred to the other two in stroke-based segmentation.

As an example consider let's revisit one of the examples in the previous chapter. So in Figure 8.2, we firstly recover all nine strokes first and then calculate the average distance which in this case is 132 pixels.

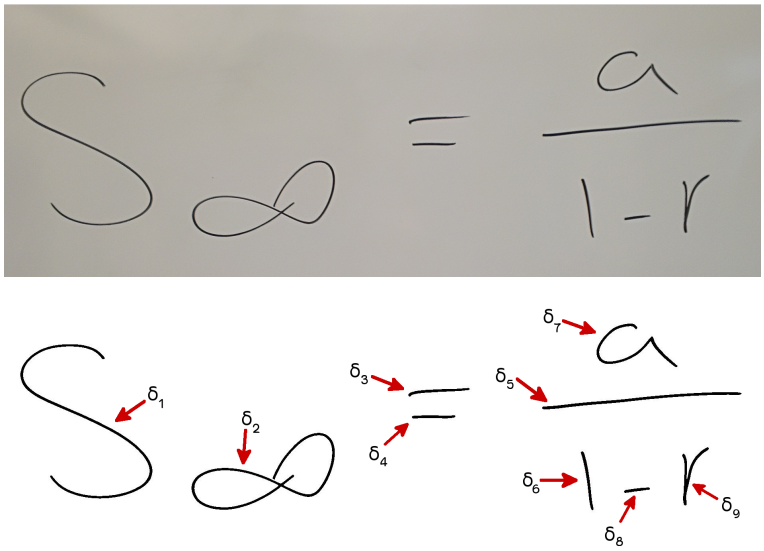


Figure 8.2: *Stroke-based Segmentation*: in this method the strokes have to be recovered first.

Afterwards, the strokes are sorted according to their position in a way that strokes towards the left of the expression take priority over the ones to the right. Subsequently, we start from the first (left-most) stroke, calculating the surrounding strokes using the average distance between the strokes that was calculated and try to determine a **best fit** based on the confidence of the recognition results returned by the recogniser.

The *best fit* is computed by taking a set of strokes and using a reconstruction method – preferably the *informed randomisation*[1] – to compute the best permutation for all the strokes in the particular surroundings. If the recognition accuracy of this best permutation is high enough (i.e. greater than 80%) then we return this set of strokes as the *best fit*. Otherwise, the stroke that is furthest away from our initial stroke is removed and the operation is repeated until either a high enough accuracy is encountered or the list is empty because the results were not satisfactory. In the latter case, we return the set of

[1] See Chapter 6 Section 6.2 for details.

strokes that had the highest accuracy during the entire operation. Afterwards, the strokes in *best fit* are removed from our initial list of strokes and the same procedure continues until a *fit* has been found for all the strokes.

To follow our example in Figure 8.2, we start off by considering stroke δ_1 and in its surroundings and in this case δ_2 falls inside the surroundings of δ_1. However, for these two strokes the recogniser returns "<<" with 65.94% accuracy and a few other symbols. Since the recognition accuracy is not high enough the furthest stroke is removed from the set and this time the recogniser returns character "s" with 98.7% accuracy. Afterwards, δ_1 is removed from the set and the operation continues. In the next instance, no other stroke falls within the surroundings of δ_2 and the recogniser correctly identifies the character "∞" with 85.78% accuracy and this stroke is also removed from the set. Afterwards, δ_4 is the only stroke that falls within the surroundings of δ_3 and the recogniser correctly returns character "=" with an accuracy of 96.34% and the operation continues until a *fit* has been found for all the rest of the strokes and regarding this example all the characters are segmented and recognised successfully.

Clearly in this method higher priority is given to *fits* with larger number of strokes. This is because in many cases if we simply allow the process to continue to the most basic *fits*, i.e. single strokes, they would receive a high accuracy from the recogniser. As an example, consider the three strokes in the character H; two vertical and one horizontal strokes. Every one of these three strokes can be a character on its own, such as "1" for the vertical and "−" (hyphen) for the horizontal strokes. Therefore, they would have high accuracies – possibly higher than the three strokes together. This is why prioritising *fits* with larger number of strokes is essential in this method.

However, our experiments prove that the opposite case can also happen (even though not as often), where a confident recognition is encountered even before we get to the same number of strokes as the originally intended symbol. As an example let's consider $\frac{a}{B}$, where the writer uses four strokes to write the expression: one for a, one for the fraction line and two for B. Depending on how the expression is written it is possible to end up

with the three characters: a, T and 3. We will see the results of our experiments with this method in the next chapter.

8.2 Discussions

The main limitation of this method is that many combinations of strokes in an expression can be translated into characters with high accuracies by the recogniser, which are neither the originally intended characters nor were they drawn in that order by the writer. This method could be possibly improved by performing more analysis before the final result is returned. That is not to return the first encountered high accuracy recognition. That being said, as discussed one must be cautious of the problems associated with that method; that *fits* with a single stroke can make a character on their own with high recognition accuracies. Therefore, different coefficients could be used to reconcile this difference; meaning, characters with more number of strokes would receive a greater coefficient compared to the ones with less number of strokes. But, regardless of the improvements that could be made to this method, generally speaking, the major problem in this method is that it is difficult to backtrack and find where the algorithm has failed to try to solve problems and ambiguities.

Therefore, it could be best to combine the two segmentation methods discussed so far, i.e. *component-based* and *stroke-based*. In other words, it could be best to perform segmentation through the *component-based* method and then using this method in ambiguous areas, that is in the surroundings of segmented areas where the recognition accuracy is low or touching characters are suspected.

As a side note, one particular problem that we encountered with this method or possibly with our recogniser is that some combination of strokes caused the recogniser to freeze. But unfortunately the reasons behind this problem could not be discovered.

8.3 Summary

In this chapter we investigated a novel segmentation method, namely *stroke-based segmentation*, which combines the segmentation and recognition tasks into a single recursive step. In this method, the strokes are used as basic elements for analysis in order to simultaneously perform symbols segmentation and in recognition in an expression. We discussed how this method could overcome some of shortcomings of the *component-based* segmentation, such as touching characters, and the methods that can be used to implement this method. We then explored the limitations and difficulties that one can expect to face with this method.

In next chapter we will see how the two segmentation methods compare to one another in a set of experiments with Google Glass. Although, as we will see this methods is outperformed by the component-based segmentation, it shows promising results.

CHAPTER 9

EXPERIMENTS: GOOGLE GLASS

So far in our experiments we have been using a high quality camera, however, in reality they are not as widely used or carried around as normal cameras, such as smart phone cameras and similar devices. Therefore, we design new experiments which aim to test the usability of our methods with a lower resolution camera. With that in mind, in the following experiments we use Google Glass, which has a 5-megapixel camera same as many smart phones available in today's market.

In the following experiments a dataset has been used that contains 53 mathematical expressions consisting of 440 characters taken from 5 different writers. There are 8.3 characters per expression on average. These expressions are from various branches of mathematics such as Algebra, Applied Mathematics, Arithmetic, Calculus, Geometry, Logic, Probability and Statistics. Moreover, the samples include 165 black, 96 blue, 93 red and 86 green characters; 244 of which are maths and 196 Latin characters.

The experiments in this chapter have been divided into two main categories, Characters (Section 9.1) and Expressions (Section 9.2). In the former section, we put to test the usability of our character recognition and reconstruction methods, while in the latter we present the experiments with entire mathematical expressions to test the effectiveness of our segmentation methods.

9.1 Characters

In this section we present our character recognition experiments. This section has also been divided into two main parts: experiments with the *colour-driven* reconstruction in Section 9.1.1, and the *brute-force* together with *informed* reconstruction methods in Section 9.1.2. The *brute-force* and *informed* reconstruction experiments have been combined because as explained previously, in the latter method *brute-force* is used for symbols with one and two strokes.

Figure 9.1 presents examples of character images taken by Google Glass.

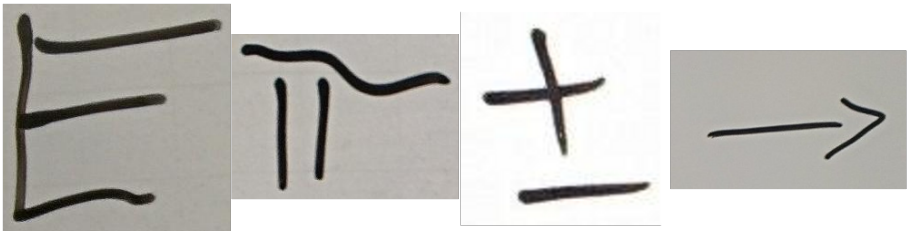


Figure 9.1: Example of character images taken by Google Glass.

9.1.1 Recognition with Colour-driven Reconstruction

The result of our experiments for the recognition of characters with the colour-driven reconstruction method are presented in the following figure and tables. Table 9.1 breaks the results down with respect to Latin and Maths characters and Table 9.2 with respect to the colour of the marker pens used.

Table 9.1: Character recognition with Colour-driven reconstruction: Maths vs. Latin characters

	Maths	Latin	Overall
Preprocessing Failure	2.05%	2.55%	2.27%
Recognition Accuracy	87.70%	73.47%	81.36%
Accuracy (Not including failures)	89.54%	75.39%	83.26%

Table 9.2: Character recognition with Colour-driven reconstruction: Accuracy broken down by colour

	Red	Black	Blue	Green
Preprocessing Failure	1.07%	1.21%	2.08%	5.81%
Recognition Accuracy	86.02%	78.79%	87.5%	74.42%
Accuracy (Not including failures)	86.96%	79.75%	89.36%	79.01%

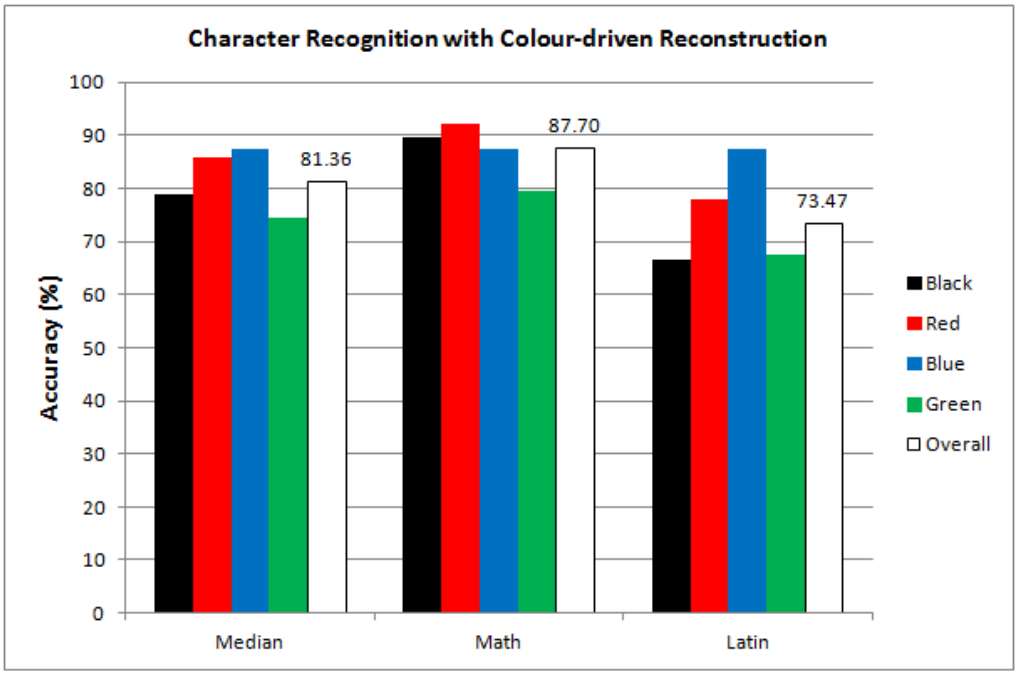


Figure 9.2: Character Recognition with Colour-driven Reconstruction: Results

As for a variation between the recognition of the symbols we can see that our methods clearly work better with maths than Latin characters. This is partly due to the fact that the recogniser is able to identify mathematical characters regardless of their direction in many cases, which is because of how the algorithm is trained and more precisely because of variations in writing styles in math. For example, the character + is drawn differently by different writers with variation in stroke ordering and direction. Therefore, the recogniser is able to pick up on that whereas, compared to math, these variations are seldom in Latin characters and consequently, if the algorithm fails to identify the direction

of strokes correctly, the recogniser will mis-recognise the symbol.

Furthermore, although our preprocessing methods work better with red and black inks, the highest accuracy belongs to the symbols drawn in blue. Moreover, the preprocessing methods do not work as well with the green ink due to the brightness of colour which can cause the edge detector to fail to fully detect the boundaries of the strokes. Figure 9.3 is a visual depiction of the preprocessing failure of the system.

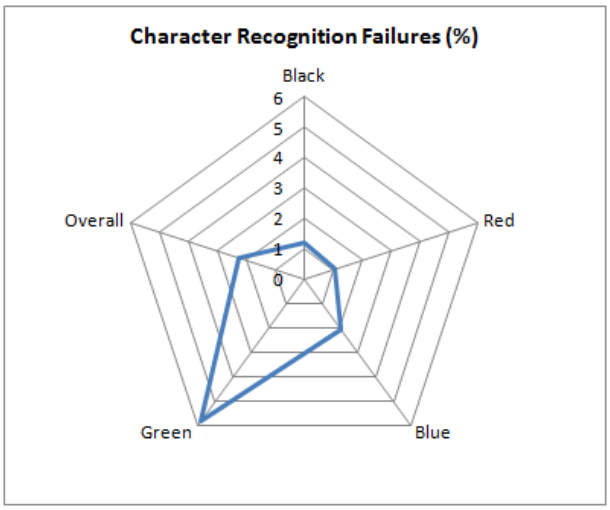


Figure 9.3: Character Recognition: Failure

9.1.2 Recognition with Informed/Brute-force Reconstruction

In this section we demonstrate the results of our character recognition method using brute-force reconstruction. Similar to the previous section the following tables break down the results according to the nature and colour of the symbols.

Table 9.3: Character recognition with Informed reconstruction: Maths vs. Latin characters

	Maths	Latin	Overall
Preprocessing Failure	2.05%	2.55%	2.27%
Recognition Accuracy	89.75%	78.06%	84.54%
Accuracy (Not including failures)	91.63%	80.10%	86.51%

Table 9.4: Character recognition with Informed reconstruction: Accuracy broken down by colour

	Red	Black	Blue	Green
Preprocessing Failure	1.07%	1.21%	2.08%	5.81%
Recognition Accuracy	87.1%	83.64%	89.58%	77.91%
Accuracy (Not including failures)	88.04%	84.66%	91.49%	82.72%

It is clearly observable that this method outperforms the colour-driven reconstruction and the black colour especially benefits from it more than the other colours. This is because the colour channels, i.e. HSB, are not able to reveal enough information in order for our trained classifier to determine the direction of black strokes as accurately as the others.

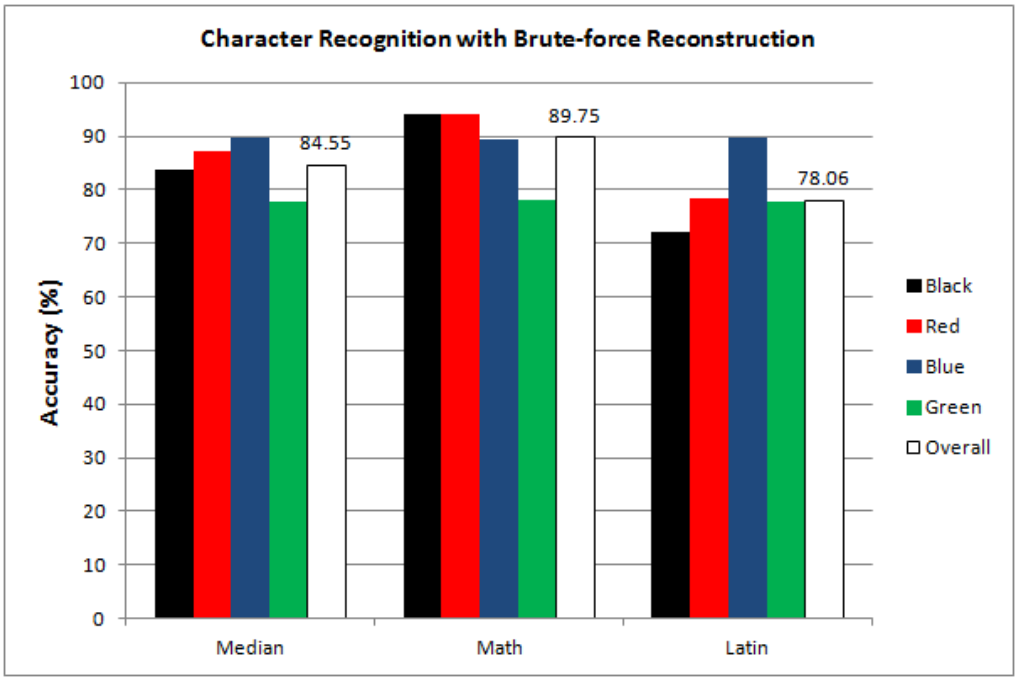


Figure 9.4: Character Recognition with Brute-force Reconstruction: Results

9.2 Expressions

In this section our experiments with segmentation of characters in mathematical expressions are presented. We demonstrate the outcome of our experiments with *component-based* segmentation in Section 9.2.1 and the results of experiments with *stroke-based* segmentation in Section 9.2.2.

Figure 9.5 depicts an example of an expression image taken by Google Glass.

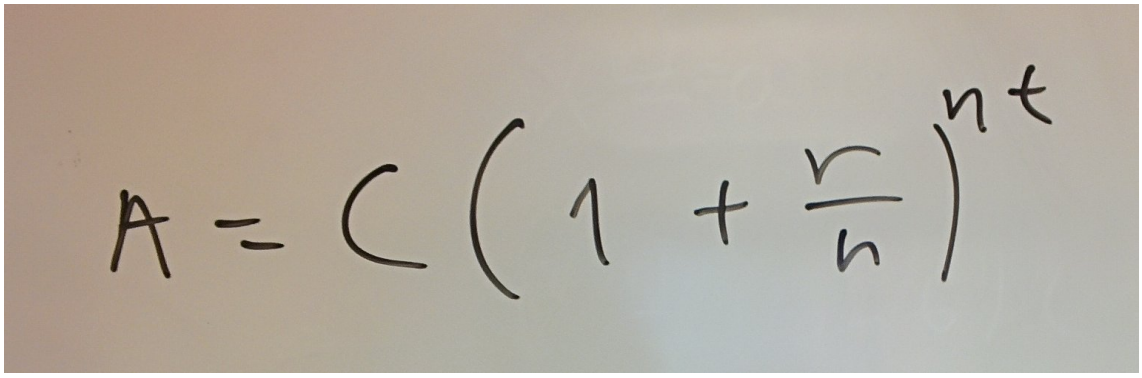


Figure 9.5: Example of an expression image taken by Google Glass.

9.2.1 Component-based Segmentation of Expressions

The following tables break down the segmentation accuracy of our *component-based* method. Again, Table 9.5 breaks down the results of experiments according to the type and Table 9.6 according to the colour of symbols.

Table 9.5: Component-based segmentation: Maths vs. Latin characters

	Maths	Latin	Overall
Segmentation Accuracy	88.67%	87.22%	88.03%
Accuracy (Not including noise)	90.65%	89.69%	90.23%

Table 9.6: Component-based segmentation: Accuracy broken down by colour

	Red	Black	Blue	Green
Segmentation Accuracy	94.68%	82.94%	96.91%	81.11%
Accuracy (Not including noise)	95.7%	85.45%	97.92%	84.88%

Here we classify a segmented character as successful if the entire character, i.e. all the strokes of the character, are contained within the cropped image by the algorithm with no other symbols in it.

As we can see, overall there is a negligible difference between the segmentation of mathematical and Latin characters, and the main difference is for the green colour – see Figure 9.6 for details. Moreover, as for the difference in segmentation of characters in specific colours we can see that the method is weakest in dealing with the colours green and black respectively. Regarding the colour green, it is once again, mainly due to edge detection failures, however, for black it is because of *amalgamation* mistakes, meaning that the algorithm failed to combine the different connected components together for various reasons and not necessarily because of the colour of the characters.

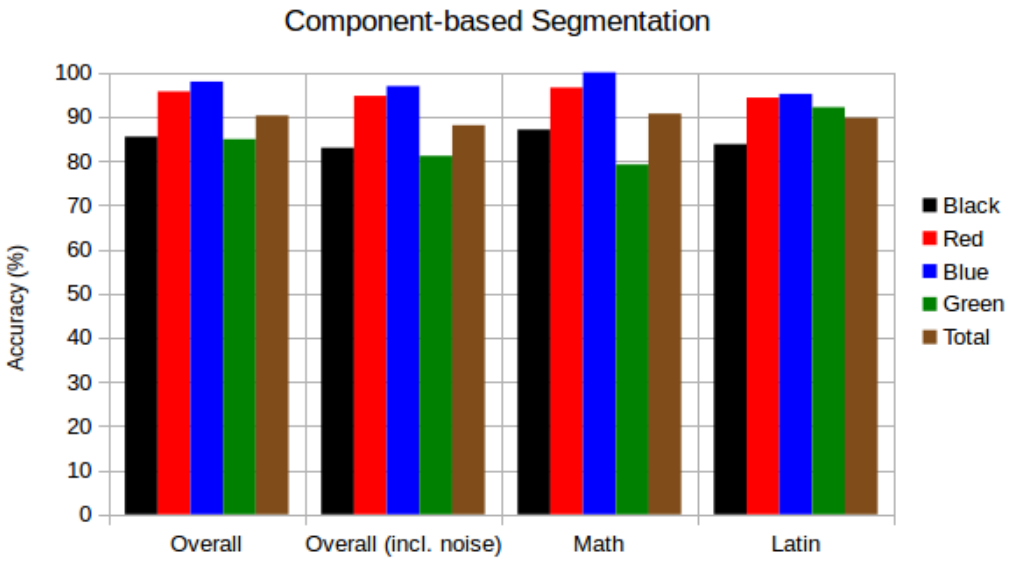


Figure 9.6: Component-based Segmentation: Results

9.2.2 Stroke-based Segmentation of Expressions

We present the segmentation accuracy of our *stroke-based* method in this section. Once again, the results are broken down according to the type and colour of symbols in Tables 9.7 and 9.8 and in Figure 9.7.

Table 9.7: Stroke-based segmentation: Maths vs. Latin characters

	Maths	Latin	Overall
Segmentation Accuracy	82.19%	74.26%	78.62%
Accuracy (Not including noise)	84.23%	76.53%	80.78%

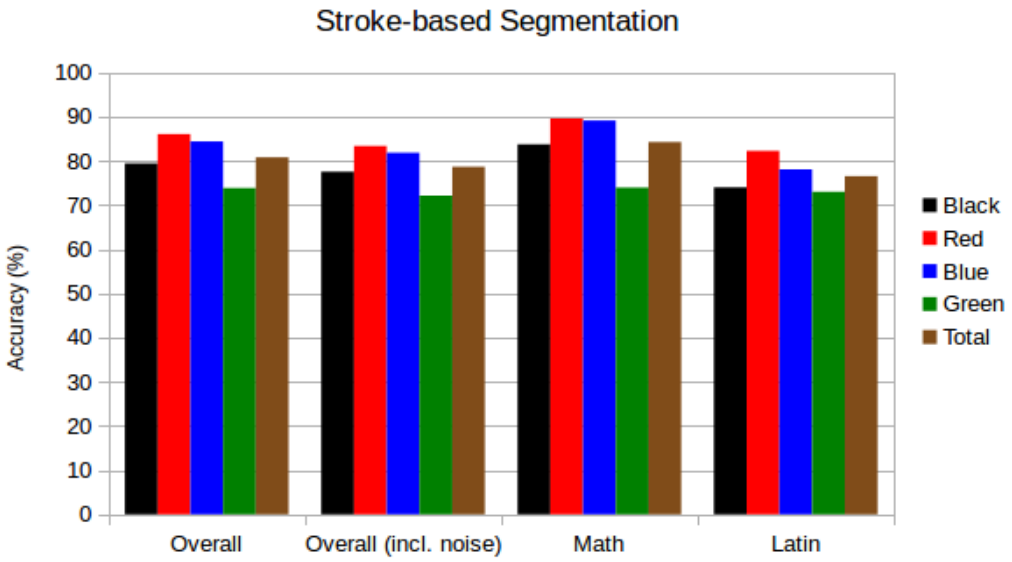


Figure 9.7: Stroke-based Segmentation: Results

We can explicitly see that this method is outperformed by the *component-based* method. However, it is worth mentioning that the low accuracy of this method is not necessarily because of it being unfit for segmentation purposes, but because this method fails immediately if the strokes are not recovered correctly. Furthermore, the more complicated and compact the expression the more difficult it gets for this method to produce the correct result. We have identified and discussed further complications for this method in Chapter 8.

Table 9.8: Stroke-based segmentation: Accuracy broken down by colour

	Red	Black	Blue	Green
Segmentation Accuracy	83.33%	77.51%	81.82%	72.09%
Accuracy (Not including noise)	86.02%	79.39%	84.37%	73.81%

9.3 Summary

In this chapter we presented a set of experiments that covered most of the techniques described in this thesis was presented in this chapter. More precisely, we used Google Glass instead of high resolution cameras in this case study to examine the effectiveness of our segmentation – **component-based** and **stroke-based** – and character recognition methods – such as preprocessing, local examination, global reconstruction (**colour-driven** and **brute-force/informed**), and recognition – in a real life scenario, since high resolution cameras are not as common or as useful as normal cameras, for instance the one fitted into the Glass.

A successful mathematical handwriting recognition system integrated in such devices can have an important impact such as helping the visually impaired students in maths lectures to make the most out of them, which unfortunately is not possible for them at the moment due to the complex two-dimensional nature of mathematics.

In summary, in regards to our character recognition methods, we have seen that the brute-force/informed reconstruction method clearly outperforms the colour-driven method by a convincing (over) 3% margin and we have managed to achieve recognition accuracy of 84.54% with the former method. Although, as we have mentioned throughout this thesis, we do now know of any homogeneous systems that work with mathematical content on whiteboards, there are systems that attempt to recognise characters and words on essentially smart-boards[1], i.e. they capture online handwriting data – though some transform the online data to offline later, our system has managed to outperform most

[1]See Chapter 1 for more details about these systems.

of these systems. This proves that trajectory recovery is amongst the most successful techniques that can be used for identification of offline handwriting characters on whiteboards.

As for the time taken to identify all characters in an expression, assuming that there are around 8 characters in every expression on average, with a further assumption that 6 of those would have three strokes or less, one four stroke and a single five stroke or above character, it would take the character recogniser with colour-driven reconstruction method to conclude the results in about 582 milliseconds, whereas with the brute-force and informed methods it would take around $8,490$ and $1,200$ milliseconds respectively. However, we have seen that brute-force reconstruction's timings begin to increase dramatically once the character has over four strokes and we have also seen that only a few mathematical symbols have more than four strokes, so far that most online mathematical character recognisers have limited the number of strokes allowed in characters to four. Therefore, in situations where the expression only contains symbols with less than four strokes, this immense gap between the timings would be reduced considerably.

In respect of the segmentation methods introduced in this thesis, although the component-based segmentation method performs much more robustly for the task than the stroke-based counterpart with an overall accuracy of 84.54%, it is unable to deal with touching strokes/symbols, whereas the latter can cope with such circumstances to a certain extent and as touching symbols do appear in mathematical expressions and particularly on whiteboards, due to the sloppiness factor that we have discussed, we have identified future works for this method in the next chapter so that it can be improved as the capability to deal with such expressions would be crucial for a successful mathematical handwriting recognition system.

Part IV

Conclusion

CHAPTER 10

CONTRIBUTIONS

In this thesis, we have presented novel techniques for recognition of mathematical characters from whiteboard images by reconstructing stroke information to make images amenable to online recognition systems. In particular, we have presented two novel approaches for the offline segmentation of mathematical characters and investigated several techniques for the reconstruction of stroke information, via taking into account information such as colour and stroke artefacts like varying width as well as uninformed and informed search techniques. In more detail, we believe our contributions to be as follows:

To our knowledge, this thesis presents the first comprehensive work on offline recognition of handwritten mathematics on whiteboards. While there has been research on recognition of general handwriting on whiteboards, this has been done mostly using online data. Similarly, most of the research on recognition of handwritten mathematics has been using online technology such as smart-boards and tablet PCs. Although the latter has allowed us to use a system developed in this context as recognition engine in our work, it has the unfortunate side effect that there are no systems that we can meaningfully compare our results with.

We have developed a novel combination of image preprocessing and noise reduction techniques designed specifically for whiteboards. As whiteboards can be very noisy environments, traditional preprocessing techniques are often not successful. Therefore, for preprocessing of characters we have developed techniques that aim to deal

with symbols prudently and reduce noise as much as possible. Although we reuse some well known image processing techniques from the literature, both their combination and application context are novel to our knowledge. Furthermore, our experimental results have lead to a number of directions for future work, how these methods can possibly be improved. We will discuss this in more detail in the next chapter.

Our work is the first time that *trajectory recovery* techniques were applied for the recognition of mathematical expressions. While trajectory recovery has been previously used in other contexts, in particular for the recognition of alpha numerical characters, as well as signatures and words, previously there has been no attempt to use the technique for a domain that is as diverse as that of mathematical symbols.

We have also introduced a number of novel *global reconstruction* methods. In particular, these are:

Colour-driven Reconstruction that uses a *statistical classifier* to determine the direction of a stroke using the colour information in the image. Colour images have previously been left out in handwriting recognition in general.

Brute-force Reconstruction that uses brute-force search or exhaustive search to generate all possible permutations and find the most likely permutation for a set of strokes using the feedback from the online recogniser.

Informed Reconstruction that combines the two methods above. This method attempts to optimise *brute-force* through the speed of *colour-driven* and heuristics that we gathered through our experiments.

Additionally, we developed novel segmentation methods suitable for mathematical symbols on whiteboards:

Component-based Segmentation which relies on the average size of the *connected components* in the expression to amalgamate the connected components as necessary.

Stroke-based Segmentation that makes use of trajectory recovery techniques to segment the symbols. More precisely, in this method the stroke trajectories of an entire expression are recovered before segmentation is performed. In addition, segmentation and recognition of symbols are in fact combined into a single process. This method has not been used in any context prior to this.

CHAPTER 11

FUTURE WORK

In this chapter we introduce possible improvements and extensions both for our methods and for the system. Accordingly, we will discuss them in more detail in the same respective order.

11.1 Improvements to Our Methods

In this section we discuss possible ways of improving the methods that we have described in this thesis.

A possible way to improve the **edge detection** results is by using a method that automatically adjusts the threshold of the edge detector, possibly based on the colours and illumination in the image, to minimise the amount of noise and open gaps that appear in the resulting image.

We strongly believe that further training of the **online recogniser** that we use for whiteboard characters would improve the recognition results. This is because of the fact that in our experiments we noticed many cases where the trajectory of the symbol was recovered successfully, however, the correct recognition results were not obtained.

The results of the **classifier** that was trained for **identification of direction** of strokes in colour images could possibly be improved with an additional *length* feature to specify the length of the strokes. The reason behind this idea is that, as mentioned before our experiments proved that the *saturation* channel is of particular importance for determining the direction and due the wet nature of whiteboard pens, we have noticed that the more a stroke continues on the surface, the more its saturation drops, which makes us believe that the length could also play an important role in the identification of start/end points of a stroke.

As discussed previously, the recogniser that was available to us restricts the number of strokes to four. Therefore, our **informed reconstruction** is yet to be tested for characters with more than four strokes, in order to verify the usability of this method for such characters.

In order to improve the results of recognition for expressions, it would certainly be useful to classify mathematical characters into specific classes, which would allow for the **constraining of the outputs** or in other words easier selection of the correct choice from the recognition results returned by the recogniser.

One possible way to improve the **segmentation** results would be to use a combination of the segmentation methods that we discussed in this thesis in such a way that we start by segmenting the characters through the *component-based* method and then use *stroke-based* segmentation in obscure areas. The obscurity could be decided by the recognition results returned by the recogniser or in crowded areas, where too many components are gathered in a specific region.

Also, an interesting factor that was observed during the course of experiments was that the confidence of the writers played an important role in the recognition process. More precisely, we noticed that when the writers were unfamiliar with the mathematical symbols or simply not used to writing certain characters, hesitation was observed by the writers, which was at times reflected in the recognition process. It

must be said that although most of our subjects were PhD students in computer science, not all were familiar with some Latin/Maths symbols, which worsened the hesitation. But, unfortunately our experiments were not designed to gather such data and therefore we cannot conclude this observation from a scientific point of view and would like to put it forward as a suggestion for further research.

11.2 System Extensions

In this final section, we explore some practicable applications and extensions for offline mathematical handwriting recognition on whiteboards.

Improving Handwriting Recognition Systems on Whiteboards: Our methods have not been tested for text recognition and similar applications on whiteboards, however, in theory it could work with such form of information by adapting the recognition module – whether online or offline.

Extraction of Expressions from Images: Since our system can only deal with images of single-lined expressions, a useful extension would be a system that could extract single expressions from images, which could then be passed to our system for recognition.

Author Recognition: Many applications such as signature verification systems also use trajectory recovery techniques, which leads us to believe that the trajectories recovered by our system can also be used to identify the writers in whiteboard images that could then be used in collaborative applications and identify which parts of a given image from whiteboard has been written by which writer.

Lecture Notes from Whiteboard Images: A certainly interesting application would be where lecture notes could be made out of entire images from whiteboards. Clearly such system would require dealing with images which could contain the surroundings of whiteboards as well as multiple expressions, text, diagrams etc. Therefore,

our methods could be integrated as a part of such system which will be of huge importance. Not only such system would be able to make it easier for both students and lecturers, but it could in fact be very useful for visually impaired students who could greatly benefit from such system.

LIST OF REFERENCES

[And68] Robert H. Anderson. *Syntax-Directed Recognition of Hand-Printed Two-dimensional Mathematics*. PhD thesis, Harvard University, January 1968.

[Bak12] Josef B. Baker. *A Linear Grammar Approach for the Analysis of Mathematical Documents*. PhD thesis, School of Computer Science, June 2012.

[Bar88] Orit Baruch. Line thinning by line following. *Pattern Recognition Letters*, 8:271–276, 1988.

[BF88] Mary K. Babcock and Jennifer J. Freyd. Perception of dynamic information in static handwritten forms. *American Journal of Psychology*, 101:111–130, 1988.

[BLC97] Richard Buse, Zhi-Qiang Liu, and Terry Caelli. A structural and relational approach to handwritten word recognition. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS*, 27(5):847–861, October 1997.

[BP97] Horst Bunke and P.S.P.Wang, editors. *Handbook of Character Recognition and Document Image Analysis*. World Scientific, 1997.

[BS89] Radmilo M. Bozinovic and Sargur N. Srihari. Off-line cursive script word recognition. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 11:68–83, 1989.

[BSS09] Josef B. Baker, Alan P. Sexton, and Volker Sorge. A linear grammar approach to mathematical formula recognition from pdf. In *Mathematical Knowledge Management*, 8th International Conference, LNCS 5625. Springer Verlag, 2009.

[Can86] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Trans. on*, PAMI-8(6):679 –698, November 1986.

[CF06] VARSHA Chikane and CHIOU-SHANN Fuh. Automatic white balance for digital still cameras. *JOURNAL OF INFORMATION SCIENCE AND ENGINEERING*, pages 497–509, 2006.

[CKLS07] Mohamed Cheriet, Nawwaf Kharma, Cheng-Lin Liu, and Ching Y. Suen. *Character Recognition Systems - A guide for students and practitioners*. John Wiley, 2007.

[CMUV97] Ron Cole, Joseph Mariani, Hans Uszkoreit, and Giovanni Batista Varile. Survey of the state of the art in human language technology. *Cambridge University Press*, 1997.

[CY00] Kam-Fai Chan and Dit-Yan Yeung. Mathematical expression recognition: A survey. *International Journal on Document Analysis and Recognition*, 3:3–15, 2000.

[DIRS02] David S. Doermann, Nathan Intrator, Ehud Rivlin, and Tal Steinherz. Hidden loop recovery for handwriting recognition. *IEEE*, 2002.

[DLZ14] Kenny Davila, Stephanie Ludi, and Richard Zanibbi. Using off-line features and synthetic data for on-line handwritten math symbol recognition. In *Int'l Conf. Frontiers in Handwriting Recognition*, 2014.

[DR92] David Doermann and Azriel Rosenfield. Temporal clues in handwriting. *IEEE*, pages 317–320, 1992.

[DR95] David S. Doermann and Azriel Rosenfield. Recovery of temporal information from static images of handwriting. *International Journal of Computer Vision*, 15:143–164, 1995.

[DW92] Christopher E. Dunn and P.S.P Wang. Character segmentation techniques for handwritten text - a survey. *Pattern Recognition*, II:577–580, 1992.

[EBCBGMZM11] Salvador Espana-Boquera, Maria J. Castro-Bleda, Jorge Gorbe-Moya, and Francisco Zamora-Martinez. Improving offline handwritten text

recognition with hybrid hmm/ann models. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33:767–779, April 2011.

[Fra08] Katrin Franke. Stroke-morphology analysis using super-imposed writing movements. *LNCS*, 5158:204–217, 2008.

[FSC98] Graham Finlayson, Bernt Schiele, and James Crowley. Comprehensive colour image normalization. In Hans Burkhardt and Bernd Neumann, editors, *Computer Vision - ECCV'98*, volume 1406 of *Lecture Notes in Computer Science*, pages 475–490. Springer Berlin / Heidelberg, 1998.

[GC04] Utpal Garain and Bidyut Baran Chaudhuri. Recognition of online handwritten mathematical expressions. *IEEE Trans. on Systems, Man, and Cybernetics*, 34(6):2366–2376, 2004.

[GH89] Zicheng Guo and Richard W. Hall. Parallel thinning with two subiteration algorithms. *Comm. of the ACM*, 32(3):359–373, 1989.

[GLF+09] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31:855–868, 2009.

[GW08] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing - Third Edition.* Pearson Education, 3rd edition, 2008.

[Hic08] Jason Hickey. *Introduction to Objective Caml.* Cambridge University Press, 2008.

[HLP10] Kenneth A. Hawick, Arno Leist, and Daniel P. Playne. Parallel graph component labelling with gpus and cuda. *Parallel Computing*, pages 655–678, 2010.

[HMW12] Rui Hu, Vadim Mazalov, and Stephen M. Watt. A streaming digital ink framework for multi-party collaboration. In *Proceedings of the 11th international conference on Intelligent Computer Mathematics*, CICM'12, pages 81–95. Springer-Verlag, 2012.

[HS91] Robert M. Haralick and Linda D. Shapiro. *Computer and Robot Vision*, volume 1. Addison-Wesley, 1991.

[Hu13] Rui Hu. *Representation, Recognition and Collaboration with Digital Ink*. PhD thesis, 2013.

[HW13a] Rui Hu and Stephen M. Watt. Determining points on handwritten mathematical symbols. In *Proc. of CICM 2013*, 2013.

[HW13b] Rui Hu and Stephen M. Watt. Identifying features via homotopy on handwritten mathematical symbols. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 61 – 67, September 2013.

[HZ13] Lei Hu and Richard Zanibbi. Segmenting handwritten math symbols using adaboost and multi-scale shape context features. In *Int'l Conf. Document Analysis and Recognition*, 2013.

[Imp94] Sebastiano Impedovo, editor. *Fundamentals in Handwriting Recognition*, volume 124 of *NATO ASI Series - Series F: Computer and Systems Sciences*. Springer, 1994.

[ISFU12] Y. Iwakiri, S. Shiraishi, Yaokai Feng, and S. Uchida. On the possibility of instance-based stroke recovery. In *Frontiers in Handwriting Recognition (ICFHR), 2012 International Conference on*, pages 29–34, Sept 2012.

[Jag98] Stefan Jager. *Recovering Dynamic Information from Static, Handwritten Word Images*. PhD thesis, Albert Ludwigs University of Freiburg, July 1998.

[KSS03] A. L. Koerich, R. Sabourin, and C. Y. Suen. Large vocabulary off-line handwriting recognition: A survey. *Pattern Analysis and Applications*, pages 97–121, 2003.

[KY00] Yoshiharu Kato and Makoto Yasuhara. Recovery of drawing order from single-stroke handwriting images. *IEEE*, pages 938–949, 2000.

[LB07] Marcus Liwicki and Horst Bunke. Combining on-line and off-line systems for handwriting recognition. *Ninth International Conference on Document Analysis and Recognition, ICDAR.*, 1:372 – 376, 2007.

[LB08] Marcus Liwicki and Horst Bunke. *Recognition of Whiteboard Notes - Online, Offline and Combination*, volume 71 of *In Series in Machine Perception and Artificial Intelligence*. World Scientific, 2008.

[LCB03] Zhi-Qiang Liu, Jinhai Cai, and Richard Buse. *Handwriting Recognition - Soft Computing and Probabilistic Approaches*, volume 133 of *Studies in Fuzziness and Soft Computing*. Springer, 2003.

[LCC95] Yung-Cheng Liu, Wen-Hsin Chan, and Ye-Quang Chen. Automatic white balance for digital still camera. *IEEE Trans. on Consumer Electronics*, 41:460–466, August 1995.

[Lee99] Seong-Whan Lee. *Advances in Handwriting Recognition*, volume 34 of *Machine Perception Artificial Intelligence*. World Scientific, 1999.

[L'H99] Eric L'Homer. Extraction of strokes in handwritten characters. *Pattern Recognition*, pages 1147–1160, 1999.

[Lin96] Tony Lindeberg. Edge detection and ridge detection with automatic scale selection. Technical report, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, January 1996.

[LLM+08] George Labahn, Edward Lank, Scott MacLean, Mirette Marzouk, and David Tausky. Mathbrush: A system for doing math on pen-based devices. In *Document Analysis Systems, 2008. DAS'08. The Eighth IAPR International Workshop on*, pages 599–606. IEEE, 2008.

[LLS92] Louisa Lam, Seong-Whan Lee, and Ching Y. Suen. Thinning methodologies - a comprehensive survey. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(9):869–885, 1992.

[LRM04] Victor Lavrenko, Toni M. Rath, and R. Manmatha. Holistic word recognition for handwritten historical documents. In *Document Image Analysis for Libraries, 2004. Proceedings. First International Workshop on*, pages 278 – 287, 2004.

[MPR05] Ramin Mehran, Hamed Pirsiavash, and Farbod Razzazi. A front-end ocr for omni-font persian/arabic cursive printed documents. *IEEE Computer Society*, 2005.

[NB10] Vu Nguyen and Michael Blumenstein. Techniques for static handwriting trajectory recovery: A survey. *International Workshop on Document Analysis Systems (DAS)*, pages 463–470, 2010.

[NT03] Hiromitsu Nishimura and Takehiko Timikawa. Offline character recognition using online character writing information. In *Document Analysis and Recognition, 2003. Proceedings. Seventh International Conference on*, volume 1, pages 168–172, 2003.

[NV] Ralph Niels and Louis Vuurpijl. Automatic trajectory extraction and validation of scanned handwritten characters.

[Oh01] Jong Oh. *An On-Line Handwriting Recognizer with Fisher Matching, Hypotheses Propagation Network and Context Constraint Models*. PhD thesis, Department of Computer Science, New York University, 2001.

[OK95] Lawrence O'Gorman and Rangachar Kasturi. *Document Image Analysis*. IEEE Computer Society Press, 1995.

[OZ09a] Ling Ouyang and Richard Zanibbi. Handwritten mathematical symbol classification using layout context. Int'l Workshop on Pen-Based Mathematical Computation, 2009.

[OZ09b] Ling Ouyang and Richard Zanibbi. Identifying layout classes for mathematical symbols using layout context. IEEE Western New York Image Processing Workshop, 2009.

[Pan12] Neeta Nain Subhash Panwar. Handwritten text recognition system based on neural network. *Journal of Computer and Information Technology*, 2:95–103, 2012.

[PF09] Thomas Plotz and Gernot A. Fink. Markov models for offline handwriting recognition: A survey. page 269298, 2009.

[Pit93] Ioannis Pitas. *Digital Image Processing Algorithms*. Prentice Hall, 1993.

[PS00] Rejean Plamondon and Sargur N. Srihari. On-line and off-line handwriting recognition: A comprehensive survey. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 22(1):63–84, January 2000.

[PSH11] J Pradeep, E Srinivasan, and S Himavathi. Diagonal based feature extraction for handwritten alphabets recognition system using neural network. *International Journal of Computer Science & Information Technology (IJCSIT)*, 3:27–38, 2011.

[QNY06] Yu Qiao, Mikihiko Nishiara, and Makoto Yasuhara. A framework toward restoration of writing order from single-stroked handwriting image. *IEEE*, 28:1724–1737, 2006.

[QY06] Yu Qiao and Makoto Yasuhara. Recover writing trajectory from multiple stroked image using bidirectional dynamic search. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, volume 2, pages 970–973, 2006.

[RAC05] Laetitia Rousseau, Eric Anquetil, and Jean Camillerapp. Recovery of a drawing order from off-line isolated letters dedicated to on-line recognition. In *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on*, pages 1121–1125, August 2005.

[RCA06] Laetitia Rousseau, Jean Camillerapp, and Eric Anquetil. What knowledge about handwritten letters can be used to recover their drawing order? *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

[RM02] Toni M. Rath and R. Manmatha. Word image matching using dynamic time warping, 2002.

[RM03] Toni M. Rath and R. Manmatha. Features for word spotting in historical manuscripts. In *Document Analysis and Recognition, 2003. Proceedings. Seventh International Conference on*, pages 218 – 222 vol.1, aug. 2003.

[Rus06] John C. Russ. *The Image Processing Handbook*. CRC Press, 5th edition, 2006.

[Sad13] Noureddin Sadawi. *A RULE-BASED APPROACH FOR RECOGNITION OF CHEMICAL STRUCTURE DIAGRAMS*. PhD thesis, School of Computer Science, The University of Birmingham, May 2013.

[SES11] Raid Saabni and Jihad El-Sana. Language-independent text lines extraction using seam carving. In *ICDAR'11*, pages 563–568, 2011.

[SHB99] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis, and Machine Vision (Second Edition).* PWS, 1999.

[SKOY04] Masakazu Suzuki, Toshihiro Kanahori, Nobuyuki Ohtake, and Katsuhito Yamaguchi. An integrated ocr software for mathematical documents and its output with accessibility. In Klaus Miesenberger, Joachim Klaus, WolfgangL. Zagler, and Dominique Burger, editors, *Computers Helping People with Special Needs*, volume 3118 of *Lecture Notes in Computer Science*, pages 648–655. Springer Berlin Heidelberg, 2004.

[SLF95] Geetha Srikantan, Dar-Shyang Lee, and John T. Favata. Comparison of normalization methods for character recognition. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 2, pages 719 –722 vol.2, aug 1995.

[SR98] Andrew W. Senior and Anthony J. Robinson. An off-line cursive handwriting recognition system. *Pattern Analysis and Machine Intelligence, IEEE Trans. on*, 20(3):309 –321, mar 1998.

[SS14] Behrang Sabeghi Saroui and Volker Sorge. Recognition of handwritten mathematical characters on whiteboards using colour images. In *Document Analysis Systems (DAS), 11th IAPR International Workshop on*, pages 91–95, April 2014.

[SS15] Behrang Sabeghi Saroui and Volker Sorge. Trajectory recovery and stroke reconstruction of handwritten mathematical symbols. *Document Analysis and Recognition (ICDAR), 13th International Conference on*, 2015.

[SSA05] Patrice Y. Simard, David Steinkraus, and Maneesh Agrawala. Ink normalization and beautification. In *ICDAR'05*, pages 1182–1187, 2005.

[SSE96] R. Seiler, M. Schenkel, and F. Eggimann. Off-line cursive handwriting recognition compared with on-line recognition. *IEEE*, pages 505–509, 1996.

[SSR08] Joachim Schenk, Stefan Schwärzler, and Gerhard Rigoll. PCA in online handwriting recognition of whiteboard notes. *ICFHR*, 2008.

[SW08] Elena Smirnova and Stephen M. Watt. Communicating mathematics via pen-based interfaces. In *Proc. of SYNASC'08.*, pages 9–18. IEEE, 2008.

[TJT95] Oivind Due Trier, Anil K. Jain, and Torfinn Taxt. Feature extraction methods for character recognition - a survey. *Pattern Recognition*, 29:641–662, 1995.

[TR03] Ernesto Tapia and Raúl Rojas. Recognition of on-line handwritten mathematical formulas in the e-chalk system. In *ICDAR*, volume 3, pages 980–984, 2003.

[TSW90] C.C. Tappert, C.Y. Suen, and T. Wakahara. The state of the art in online handwriting recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12:787–808, 1990.

[UNS05] Seiichi Uchida, Atsushi Nomura, and Masakazu Suzuki. Quantitative analysis of mathematical documents. In *IJDAR*. Springer-Verlag, 2005.

[VBC10] Roberto Vezzani, Davide Baltieri, and Rita Cucchiara. Hmm based action recognition with projection histogram features. pages 286–293, 2010.

[VGLK05] Christian Viard-Gaudin, Pierre-Michel Lallican, and Stefan Knerr. Recognition-directed recovering of temporal information from handwriting images. *Pattern Recognition Letters*, 26:2537–2548, 2005.

[Vin02] Alessandro Vinciarelli. A survey on off-line cursive word recognition. *Pattern Recognition*, 35(7):1433 – 1446, 2002.

[VLP+01] Maria Vanrell, Felipe Lumbreras, Albert Pujol, Ramon Baldrich, Josep Llados, and J.J. Villanueva. Colour normalisation based on background information. In *Image Processing, 2001. Proceedings. 2001 International Conference on*, volume 1, pages 874 –877, 2001.

[Wat10] Stephen M. Watt. On the mathematics of mathematical handwriting recognition. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2010 12th International Symposium on*, pages 17–17, 2010.

[WCF05] Ching-Chih Weng, Homer Chen, and Chiou-Shann Fuh. A novel automatic white balance method for digital still cameras. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 3801 – 3804 Vol. 4, may 2005.

[ZB12] Richard Zanibbi and Dorothea Blostein. Recognition and retrieval of mathematical expressions. *International Journal on Document Analysis and Recognition*, 2012.

[ZBC02] Richard Zanibbi, Dorothea Blostein, and James R. Cordy. Recognizing mathematical expressions using tree transformation. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 24(11):1455–1467, 2002.

[ZMB11] Richard Zanibbi, Harold Mouchere, and Dorothea Blostein. Stroke-based performance metrics for handwritten mathematical expressions. In *Int'l Conf. Document Analysis and Recognition*, pages 334–338, 2011.

[ZNM08] Mohsen Zand, Ahmadreza N. Nilchi, and Amirhassan S. Monadjemi. Recognition-based segmentation in persian character recognition. *World Academy of Science*, 38:183–187, 2008.

Appendices

APPENDIX A

CHARACTER SET

Table A.1: Character Set

Character	**Name**	**HTML (Unicode)**
!	Exclamation mark	&#x21;
)	Right parenthesis	&#x29;
(	Left parenthesis	&#x28;
+	Plus	&#x2B;
,	Comma	&#x2C;
–	Hyphen/Minus	&#x2D;
.	Full stop	&#x2E;
/	Slash	&#x2F;
0	Zero	&#x30;
1	One	&#x31;
2	Two	&#x32;
3	Three	&#x33;
4	Four	&#x34;
5	Five	&#x35;
6	Six	&#x36;
		Continued on next page

Table A.1 – Continued from previous page

Character	Name	HTML (Unicode)
7	Seven	&#x37;
8	Eight	&#x38;
9	Nine	&#x39;
:	Colon	&#x3A;
<	Less than	&#x3C;
=	Equals	&#x3D;
>	Greater than	&#x3E;
A	A	&#x41;
B	B	&#x42;
C	C	&#x43;
D	D	&#x44;
E	E	&#x45;
F	F	&#x46;
G	G	&#x47;
H	H	&#x48;
I	I	&#x49;
J	J	&#x4A;
K	K	&#x4B;
L	L	&#x4C;
M	M	&#x4D;
N	N	&#x4E;
O	O	&#x4F;
P	P	&#x50;
Q	Q	&#x51;

Continued on next page

Table A.1 – Continued from previous page

Character	Name	HTML (Unicode)
R	R	&#x52;
S	S	&#x53;
T	T	&#x54;
U	U	&#x55;
V	V	&#x56;
W	W	&#x57;
X	X	&#x58;
Y	Y	&#x59;
Z	Z	&#x5A;
[	Left bracket	&#x005B;
]	Right bracket	&#x005D;
a	a	&#x61;
b	b	&#x62;
c	c	&#x63;
d	d	&#x64;
e	e	&#x65;
f	f	&#x66;
g	g	&#x67;
h	h	&#x68;
i	i	&#x69;
j	j	&#x6A;
k	k	&#x6B;
l	l	&#x6C;
m	m	&#x6D;

Continued on next page

Table A.1 – Continued from previous page

Character	Name	HTML (Unicode)
n	n	&#x6E;
o	o	&#x6F;
p	p	&#x70;
q	q	&#x71;
r	r	&#x72;
s	s	&#x73;
t	t	&#x74;
u	u	&#x75;
v	v	&#x76;
w	w	&#x77;
x	x	&#x78;
y	y	&#x79;
z	z	&#x7A;
{	Left curly bracket	&#x7B;
|	Vertical bar	&#x7C;
}	Right curly bracket	&#x7D;
~	Tilde	&#x7E;
±	Plus-minus	&#xB1;
Δ	Delta	&#x394;
Σ	Sigma	&#x3A3;
α	Alpha	&#x3B1;
β	Beta	&#x3B2;
ε	Epsilon	&#x3B5;
θ	Theta	&#x3B8;

Continued on next page

Table A.1 – Continued from previous page

Character	Name	HTML (Unicode)
λ	Lambda	&#x3BB;
μ	Mu	&#x3BC;
π	Pi	&#x3C0;
ρ	Rho	&#x3C1;
ω	Omega	&#x3C9;
$\varkappa$	Kappa	&#x3F0;
ℓ	Ell	&#x2113;
$\mathbb{N}$	Natural Numbers	&#x2115;
$\mathbb{Q}$	Rational Numbers	&#x211A;
$\mathbb{R}$	Real Numbers	&#x211D;
$\rightarrow$	Right arrow	&#x2192;
$\forall$	Forall	&#x2200;
$\exists$	Exists	&#x2203;
$\emptyset$	Empty Set	&#x2205;
$\in$	Element of	&#x2208;
$\notin$	Not element of	&#x2209;
$\surd$	Square root	&#x221A;
∞	Infinity	&#x221E;
$\cap$	Intersection	&#x2229;
$\cup$	Union	&#x222A;
$\int$	Integral	&#x222B;
$\approx$	Almost equal to	&#x2248;
$\neq$	Not equal to	&#x2260;
$\leq$	Less-than or equal to	&#x2264;

Continued on next page

Table A.1 – Continued from previous page

Character	Name	HTML (Unicode)
≥	Greater-than or equal to	&#x2265;
⊂	Subset of	&#x2282;
⊄	Not a subset of	&#x2284;
⊆	Subset of or equal to	&#x2286;