UNIVERSITY OF CALGARY

Deformation and Distance in Symbol Recognition

by

Juraj Pivovarov

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

APRIL, 2002

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Deformation and Distance in Symbol Recognition" submitted by Juraj Pivovarov in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor, Jim Parker, Department of Computer Science

Jeff Boyd, Department of Computer Science

Karen Seyffarth, Department of Mathematics and Statistics

2002-04-30

Date

Abstract

Many different approaches have been tried to build accurate, efficient symbol recognizers. Under the umbrella of prototype-based symbol recognizers, there is a spectrum ranging from rigid template matching to matching via deformable templates. However, the former is fast but not so accurate and the latter are very accurate but also very slow. We consider merging the best of both worlds into a new prototype-based classifier, one that is fast and robust.

We present an efficient, adaptable representation of prototypes as vector templates, and an image metric, the Inkwell Hausdorff distance, that is fast yet tolerant to small misalignments. This technique is shown to be faster that existing techniques, with only a slightly lower accuracy.

Acknowledgements

Many important people contributed to the completion of this work. First and foremost I would like to thank my supervisor, Jim Parker, not only for the many great conferences we attended together including Kelowna, Barcelona, and Vancouver, but for providing insight into difficult problems, and for encouraging me during the struggles. I would like to thank Jeff Boyd and Karen Seyffarth for their willingness to help, and for the knowledge they imparted in me through the courses I took with them. Mike Boyle was an indispensable friend during many of the technically frustrating experiences, and as an editor, always willing to go out of his way to help me. Besides the ACM contest, and doing our grad studies together, it was always rewarding to have a good chat at the Den. I would like to thank Sonny Chan for his big help in editing, and also other colleagues I've shared ideas with over the last few years including Shane Dorosh, Dave Gomboc, Peter Pivovarov, Mark Baumback, Prasanna Govindankutty, Korado Percic, and Roger Curry. You've all helped open my eyes.

Table of Contents

List of Tables

List of Figures

List of Equations

Chapter 1
Introduction

1.1 Introduction to Symbol Recognition

A post office can receive an enormous amount of mail in a day, which poses problems for its ultimate delivery. The letter mail must be sorted into piles based on which area it is intended for. This is usually done through the use of some kind of special code on each envelope that designates where the mail is going to, called a ZIP Code in the United States or a Postal Code in Canada. People often sort this mail by hand. Wouldn't it be nice if computers could automate this mundane task? What makes the problem difficult is that this information is not always in the same place on the envelope, and once it is found it can be hard to read. This is because it can be hand-written and thus in many different styles.

Consider another problem involving the use of "Multi-Nova" cameras for automatic detection of speeding cars. They photograph the license plates of cars traveling faster than the speed limit, and then save this image to possibly give the traffic violator a ticket. We say *possibly* because there could be some level of uncertainty in the image. When the image is taken, it may happen that the license plate has been obscured for some reason (perhaps it was very dirty), or was difficult to read due to glare, reflection, etc. Because of this, a small panel of people review each image taken by the camera, and decide whether the license plate is legible enough to avoid any mistaken identity. They want to ensure that one person doesn't get a ticket for someone else's fast driving. Even though the detection of speeders has been automatic, their identification has not. Why should a group of people have to perform this repetitive task? Because *symbol recognition* is a hard problem for computers to solve.

This thesis is about symbol recognition. The problem of symbol recognition is that of being able to understand what object an image represents; the task of assigning labels to objects. In general for people, this is an easy task. For example, it is that of recognizing a single letter, machine printed or handwritten. It is that of recognizing a "smiley face" and knowing what emotion it indicates.

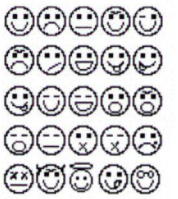


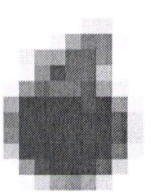
Figure 1 Smiley-faces are symbols

This is a seemingly easy problem for people to solve – we can easily look at a page of text and recognize letters on it. If we consider the words on a page as the symbols, even in the presence of spelling errors, we can easily classify the words into their proper symbol class. We can look at a stained napkin from a restaurant with a phone number scribbled on it with a splotchy fountain pen and still decipher the digits. For machines this problem is much more difficult. The nature of the difficulty comes from the underlying digital representation of an image. Here is an example of an image (Figure 2) that would be easily recognizable to humans, if it were not in computer form.

255	255	255	255	255	255	243	239	255	255
255	255	255	255	246	187	149	158	252	255
255	255	243	186	179	129	128	164	255	255
255	255	248	160	107	128	130	207	255	255
255	248	165	124	136	136	113	175	247	255
255	182	96	97	97	97	109	98	173	251
252	140	97	97	97	97	97	96	123	240
254	153	97	96	97	97	97	97	117	236
255	214	97	97	96	97	96	98	144	247
255	254	168	104	96	96	99	124	225	255
255	255	251	220	177	157	181	231	255	255

Figure 2 Digital Image - what is it?

But in this format, how do we know what it represents? A computer sees an image in the format that comes from computer graphics. It divides up a page into many small squares, much as if it was drawn on graph paper. A single number is used to represent each such square, or pixel (picture element). This format is well suited for creating photographic images but is an extremely poor representation for computer vision. All objects contained in a scene in this type of format must be recognized from only this view of the data. The matrix of numbers from Figure 2, when plotted as a greyscale image is shown in Figure 3 (a):



(a)

(b)

Figure 3 A greyscale apple at various resolutions.

In Figure 3 (a), we have a scaled down version of an apple at 10x11 resolution. This apple was obtained from the image in (b), which is at 53x56 resolution, by scaling it down. The numbers in Figure 2 represent the grey values for the pixels. They can range from 0 to 255 and 0 indicates a black pixel and 255 indicates a purely white pixel.

Another example of symbol recognition can be found below. The next two images, Figure 4 and Figure 5, contain both a library of symbols (the *alphabet*), and then some noisy *text* that contains letters from this alphabet. In this case the symbol set is that of 25 electric symbols that may be found on schematic diagrams from electrical engineering.

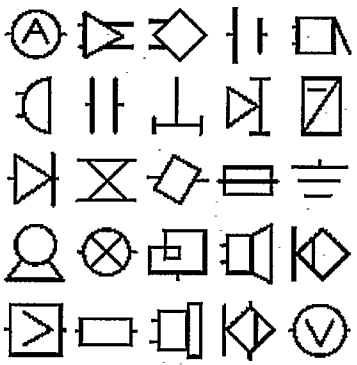


Figure 4 Library of electric symbols

Given this library, suppose we have an image of a large schematic diagram that connects these symbols in some way. It would be desirable to be able to identify the symbols in the image and therefore understand the nature of how the symbols are connected to each other. This allows potential for a richer user-interaction with the schematic, as well as allowing analysis of the schematic itself. Images of these schematics can be of a rather poor quality. Every time a schematic is photocopied, a certain level of error is introduced. Even in the face of noise, we would like to able to reliably identify the symbols on the page. Noise in an image can be defined loosely as any extraneous or missing pixels that obscure the real contents of the image. Consider such a noisy image in Figure 5. The symbols on the page were machine-generated, but the entire image has had artificial noise added. There are two kinds of noise present. The first kind is called *salt and pepper* noise and was generated by randomly toggling bits in the image with a fixed probability. The second kind of noise was boundary noise. This was generated by randomly toggling pixels with a probability that was dependent on how close the pixel was to boundary pixels. The closer a given pixel is to a boundary, the more likely it would be flipped.

This type of noise was intended to simulate the effects of photocopiers, where the edges sometimes become blurry in poorer quality machines.

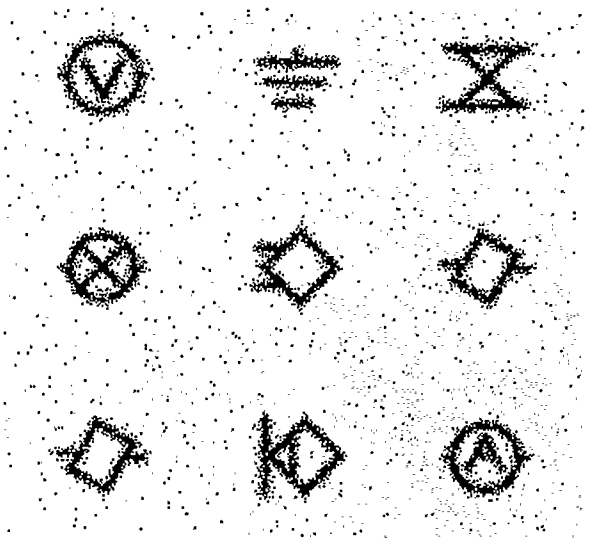


Figure 5 Noisy image with electric symbols.

A person can still easily tell which symbols are contained in the image, but with the extra noise it can become a tough problem for computers to solve.

The next example shows some of the problems inherent with symbols that are hand-drawn. In Figure 6, we see an image of a hand-drawn phone number. Notice the variation in all of the digits, even in the first two instances of the digit '2'. These digits were randomly selected from a hand written digit database, comprised of digits from many different authors.

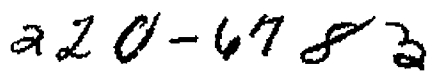


Figure 6 A handwritten phone number

All of the examples so far have shown symbol recognition examples where segmentation was not an issue. That is, it was easy to find the set of symbols that had to be recognized, and no sophisticated techniques for separating adjacent symbols was necessary. Often times this can be one of the most difficult problems that a symbol recognition system has to solve. Not only do we have to separate a connected region into symbols, sometimes we have to do the reverse, combine two unconnected regions so they are treated as a single symbol. Symbols made of multiple parts can be seen in the electric symbol library of Figure 4. The problem appears in two different ways as well. In one case, one region completely surrounds another, as in the case of the "ammeter" or

the "voltmeter". In the case of the "ground" symbol (three horizontal bars), the symbol is made of three regions that appear to all be independent of each other. For cases like the "ground" symbol, a decision has to be made as to whether the segmentor or the recognizer is responsible for this work. In this thesis, we will assume it is the segmentation algorithm.

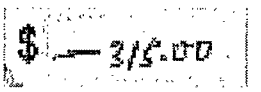


Figure 7 Example of segmentation difficulties - the two '0's are connected to each other

Advances in symbol recognition could also improve user interfaces, allowing people to use natural writing to communicate information to computers. However, this problem brings up an interesting difference between *off-line* and *on-line* algorithms. On-line algorithms are able to watch a user as he is drawing something. This means that time data is available in addition to the final image, you know when and how quickly a certain portion was drawn. In off-line symbol recognition, it means that the timing information is not available, only the final image. It turns out the extra information present from the on-line situation can improve recognition rates. The Palm Pilot, for example, boasts relatively high recognition rates using on-line algorithms. It also "cheats" a little by changing the way users do their handwriting. Since humans are so good at learning and adapting as opposed to our ability to instill these attributes to software, it is an efficient solution to the problem. It introduces a new way to draw letters, numbers, and some punctuation by a method called "Graffiti". Without going into to much detail, this new system of handwriting has made it easier for them to distinguish classes of letters that could otherwise appear quite similar. This better separation of classes is what gives better recognition rates.

Having surveyed some of the difficulties in symbol recognition, we are now ready to form our problem statement.

1.2 Problem Statement

For the purposes of this thesis, we will not discuss segmentation techniques, and assume that the images we process have undergone the segmentation process already. Thus we will consider only the problem of classifying isolated symbols. We will not consider on-line recognition algorithms, as we would like to solve the more general off-line recognition problem.

How can we recognize symbols efficiently using a prototype-based classifier, improving the recognition rate over traditional template matching algorithms while at the same time making use of advances in 'deformable prototypes?'

1.3 Goals

Our goals will be two-fold. Primarily, we would like to develop a prototype-based symbol recognition algorithm that performs well: achieving not only high recognition rates but doing so efficiently. We will attempt to do this by extending the work done on Vector templates [Parker1995]. We would like to improve over traditional prototype-based techniques (defined in Chapter 2) while still retaining the adaptability of current techniques of deformable prototypes. Secondly, we would also like to investigate the problem of comparing two binary images, and propose some new techniques for measuring distance between images.

1.4 Overview of the Thesis

This thesis makes contributions to the field of symbol recognition on two levels. First, we review and extend the work on vector templates, an elastic prototype representation that is well suited to handle variation in handwritten symbols. Second, some new techniques for comparing images are introduced, borrowing ideas from graph theory and string matching. Since the algorithms for comparing images are in fact a component of the vector template work, we will present those first in Chapter 3, deferring the talk of vector templates to Chapter 4.

The literature survey will be presented in Chapter 2. As there are many different techniques published for symbol recognition, we will have to focus on a few of the most representative ones. We will discuss the difference in a prototype-based and a prototype-free approaches to the problem, and then survey five key papers on prototype-based algorithms in detail.

In Chapter 3, new approaches to measuring distance between images will be discussed. Among these, the Inkwell Hausdorff distance is the most significant, and yields some of the best results compared to other metrics when used in handwritten symbol recognition. The technique itself was discovered before; but its position in the literature was somewhat hidden. It is now clear that the distance measure we use, which we have named the Inkwell Hausdorff distance, is really a special form of the Hausdorff distance. Another technique presented is called the Longest-Common-Subsequence (LCS) distance. It applies a well known algorithm for approximate string matching to solve the problem of measuring image similarity (which can be

viewed as 2D string matching). Finally, an algorithm that relies on matching theory will be presented to compare images. Any of these algorithms, or a combination of them, can be used in conjunction with vector templates in a larger symbol recognition system.

In Chapter 4, a flexible prototype model will be presented for use in prototype-based symbol recognition. A *vector template* is a dynamic prototype representation, and can match a variety of pattern variations, thereby making it suitable for handwritten symbol recognition. A system employing vector templates renders each prototype to match the *style* parameters of the unclassified image. In this way, an optimized template is used in each symbol comparison. The original work is extended in a number of ways. The utility of computing local stroke-width estimates is discussed and solutions to some common special case problems with vector templates are presented. Finally we will show how the distance algorithm we use with vector templates makes them into efficient deformable prototypes.

Chapter 5 will contain a description of the system design, and how object orientation was used to build reusable components for future symbol recognition systems. Primarily the modeling of vector templates and image metric functions will be covered.

In Chapter 6 we will discuss the experiments that were conducted and the datasets that we used. These datasets include both handwritten and machine printed symbols. The handwritten databases consist of four databases of isolated handwritten digits. The CENPRMI dataset consist of about 14,000 images, the ETL dataset consists of about 2,000 images, and the and also the USPS database of 9,300 images. Also, we use the large MNIST database, which is a benchmark dataset in symbol recognition, and contains about 70,000 images. Although most of the datasets are normalized to some degree, the MNIST database was normalized for size and orientation to a higher degree than the others. In the case of machine printed text, we examine an application of reading a few pages of a chess openings book, NCO, as well as a dataset of noisy symbols at various sizes and orientations from the electrical engineering domain. Vector templates were used to recognize this latter dataset with 100% accuracy, winning the symbol recognition contest of ICPR 2001.

Finally, Chapter 7 presents some concluding remarks about the use of vector templates in symbol recognition, and suggests some new unexplored areas.

Chapter 2
Symbol Recognition Algorithms

Many different techniques have been tried to solve symbol recognition problems. It would be impossible to survey them all, and we do not need to, as there is of course some overlap in the different approaches. In this section, we present a survey of a small, but representative, sample of prototype-based symbol recognition algorithms. We will also start with a more concrete statement of what the isolated symbol recognition problem is.

In essence we are given two sets of pre-classified data, called training data and testing data. Using only the test data, we have to devise a function, the symbol recognition algorithm, that, for any image, will determine which symbol it represents. This is done by learning from the training data and then evaluating its performance on the test data.

Formally, let $I_{m\times n}$ represent the space of all binary images, and suppose that we are given a subset of these images, $X = \{x_1, ..., x_r\}$. Each image is associated with a certain symbol $\sigma \in \sum = \{\sigma_1,...,\sigma_k\}$, where $\sum$ is sometimes called the *alphabet*. This association of images to symbols is represented by a classification function, $f_x : X \rightarrow \sum$. When $f_x(x_i) = \sigma_j$, that means that image x_i represents symbol σ_j. The ordered pair (X, f_x), i.e. both the set of images and the ground truth data, is called the *training data*, and r, the number of images in X, is the *size* of the training data. It will be used to develop a symbol recognition algorithm. A symbol recognition algorithm is a function from $I_{m\times n}$, *all* images, to a symbol $\sigma \in \sum$;

$$f : I_{m\times n} \rightarrow \sum. \qquad \textbf{(Eq 1)}$$

To evaluate our symbol recognition function, we are given an ordered pair (Y, f_y), where $Y=\{y_1,...,y_s\}$ of $m \times n$ images, and f_y is the classification function for Y. This ordered pair (Y, f_y) is called the *test data*. The images from the two sets X and Y are usually disjoint and the amount of training data is normally much larger than the amount testing data. We evaluate the symbol recognition algorithm f, by comparing how well it does with the ground truth function f_y. The recognition rate of f is computed as follows

$$R(f) = \frac{\sum_{i=1}^{s} \delta(f(y_i), f_y(y_i))}{s} \qquad \textbf{(Eq 2)}$$

where

$$\delta(a,b) = 1, \text{ if } a = b \text{ and } \delta(a,b) = 0 \text{ if } a \neq b.$$

Notice the expression in (Eq 2) simply counts how many symbols from the test data we classify correctly, as a percentage of the size of the test data.

Before getting into the bulk of the literature survey, it is important to define a few other key terms used in this section, including exemplar, prototype, and template. An *exemplar* is a specific image representing one particular symbol. A *prototype* is some representation of an exemplar (i.e. as an image, as a shape signature, as a set of lines). A prototype encodes the essential qualities of the symbol to facilitate robust recognition. A *template* is one particular kind of prototype; one that represents symbols by images. Templates were one of the first methods of prototype representation. A *model* is some representation of the total knowledge that describes a particular symbol. In a prototype-based system, a model of a symbol is made up of the set of prototypes for that symbol.

We will now discuss the dichotomy that symbol recognition algorithms fall into. In a *prototype-based* classifier, we compare an image against a set of prototypes. Whichever prototype most resembles the image decides its symbol class. For example, in the context of digit recognition, when classifying an image of an isolated digit, we would compare it with every prototype digit we have in our database. A quite different approach is to somehow learn what it is that distinguishes members of the various symbol classes. This knowledge can usually be compactly represented and allows the program to easily decide which symbol an image represents. A good example of this type of classifier would be a decision tree or a neural net. This type of approach yields a *prototype-free* algorithm – it is impossible to factor out which parts of the system are based on any one particular training prototype; all of the knowledge is combined into a single black-box unit.

Thus, symbol recognition algorithms are primarily either *prototype-based* or *prototype-free*. Of course, one can imagine a hybrid system as well, but if we omit discussion of multiple classifiers that combine many different strategies, a given algorithm can generally be classified as one or the other.

2.1.1 Prototype-Free Algorithms

Prototype-free algorithms can be thought of as rule-based classifiers. They store, rather than a set of prototypes, a set of rules about certain features of images of the various symbols. Neural nets and decision trees are good examples of prototype-free algorithms. The algorithms under this broad grouping attempt to summarize the properties of a given class of patterns. A set of features are computed for each training image and this information updates the current state of the classifier. Classifying an image involves computing the features for that image only, and then performing a small computation to determine the image class; (e.g., by feeding the values through a neural net or decision tree).

Since we are primarily concerned with prototype-based symbol recognition, we will point the reader to a good survey of some prototype-free algorithms. Yann LeCun, who manages the benchmark MNIST database for hand-written digit recognition, composed a survey of many learning algorithms, comparing not only performance, but training time, recognition time, and memory requirements [LeCun1995].

2.1.2 Prototype-Based Algorithms

The prototype-based approach uses a library of prototypes and compares an image with each one. The results of these comparisons are given to a classifier, which can simply be the *nearest-neighbour* strategy, where only the best prototype is reported, or a more sophisticated classifier. We show an example of a prototype-based classifier in Figure 8, where we have 5 symbols {K, Q, R, B, N}. We store 6 prototypes for this set, one for each symbol, except a Q, which has two prototypes. These symbols represent the pieces used in the game of chess, {K = King, Q = Queen, R = Rook, B = Bishop, and N = Knight}. If we are trying to classify the image of a hand-drawn Rook, we compare it to all prototypes in the library (in this case shown visually as images) and select the one that most closely matches our image. We report the symbol it represents as the classification of the image.

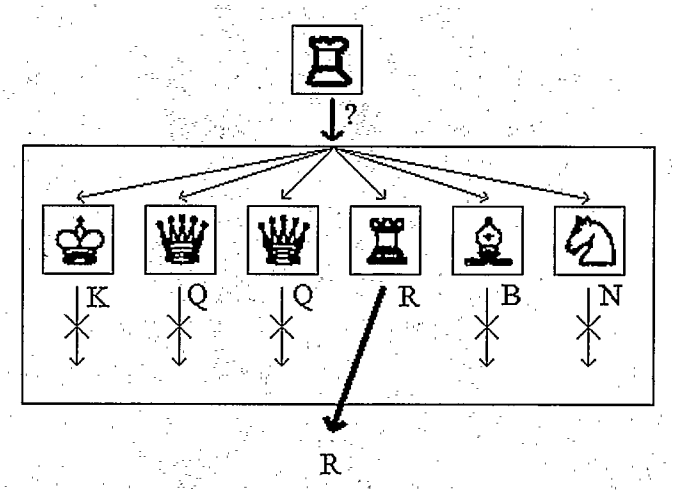


Figure 8 A Prototype-based classifier.

In any case, the running time for classifying an image is proportional to the number of prototypes in the library. If the alphabet is large, or each symbol requires numerous prototypes, performance could degrade rapidly. This makes the prototype-based approach well suited for distinguishing between a relatively smaller number of symbols, such as digits, or Latin-alphabet characters, but unwieldly for larger alphabets, such as Chinese ideographs.

In a prototype-based system, the questions of how to represent prototypes and how to measure distance (dissimilarity) between a prototype and an image need to be addressed. The literature survey in the remainder of this chapter discusses different attempts at answering these questions.

In addition to those representations that we will look at in detail, there are many other choices for prototype representation in the literature. To list a few, prototypes are represented as images, as feature vectors, as shape signatures, using a set of moments, slope-histograms, etc. Depending on this representation, one decides how to measure the similarity between a prototype and an image.

Furthermore, prototype representations can be grouped loosely into two categories: graphical and statistical. Other authors note this division as well, "we propose a dichotomy of matching in pattern space versus matching in feature space" [Wakahara1994]. A graphical (or pattern-space) representation could be an image or a shape signature. It can be defined as such a representation as that from which one can construct an exemplar of the symbol it represents.

Thus, one can often visually see the similarity in graphical representations. An example of a statistical (or feature-space) representation is storing an abstract feature vector for each prototype.

Various values can be used for features such as area, perimeter, circularity, moments, girth, etc. It is usually not possible to produce a sample image of what symbol this prototype represents.

After deciding on the prototype representation, we need to answer the question "How similar is a given image compared with certain prototype?" Measuring similarity of course depends heavily on the prototype representation. For pictorial prototypes, we have many spatial functions that can be used. Some of them insist on a direct overlap of pixels while others have a more lenient measure, one that is tolerant to small perturbations in the image. Many such techniques will be discussed in Chapter 3. For statistical prototypes, often a measure, like Euclidean-distance, is computed in the feature space of the prototypes. The same set of features is computed for the given image, and it is compared with each prototype. Of course some normalization for different coordinates of the features may take place as some features may have a higher weight than others.

2.1.3 A Comparison of Prototype-Based vs. Prototype-Free Algorithms

In general, prototype-free algorithms use far less memory and have smaller running times compared to their prototype-based counterparts. However, they can be difficult to train and this may require a lot of processing ahead of time. Also, not only is this training difficult and time consuming, but has to all be redone in order to extend the system. With a prototype-based approach, the set of prototypes can be changed very easily. A few new prototypes can be added and the system immediately "knows" more patterns.

Discussing both approaches thoroughly would be outside the scope of this thesis. In what follows, we will only provide a survey of prototype-based systems. It might be a natural question to ask which of these two broad categories is more effective at handwritten symbol recognition. Unfortunately, this is an extremely difficult question to answer. Yan LeCun described the "Boosted LeNet-4" (prototype-free) algorithm and, on the benchmark MNIST database, reported an error rate of only: 0.7%, yielding a recognition rate of 99.30%. This number has been recently surpassed by a prototype-based algorithm [Belongie1999], who report a recognition rate of 99.33%. This type of "leap-frogging" indicates that it is not yet apparent which types of algorithms, prototype-based or prototype-free, are best suited for handwritten symbol recognition.

2.2 A Survey of Prototype-Based Symbol Recognition

2.2.1 Template Matching [VanderBrug1977]

One of the oldest symbol recognition applications is OCR – Optical Character Recognition, where the problem is to reconstruct the text (in ASCII) from an image representing a page of machine printed text. One of the first approaches to solve this problem was that of storing a small bitmap for each possible character [VanderBrug1977]. That meant one for each lowercase letter, uppercase letter, digit and punctuation character. Since the font was uniform throughout, it was possible to enumerate in this way all the possibilities that would be encountered. Not only was the font-face uniform but all symbols were the same size and at the same orientation. One could segment the image into distinct symbols and classify these individually. An image was classified by comparing it to every template and reporting the identity of the best matching template as the symbol associated with that image. When an image and template were compared, the simple approach of counting which proportion of the total foreground pixels overlapped could be used to measure similarity.

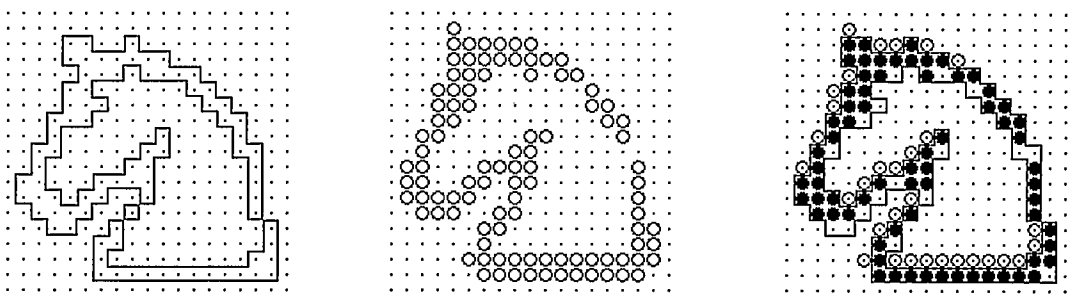


Figure 9 Example of a Knight symbol from a chess openings book.

In Figure 9, we see (a) a template of a Knight symbol, (b) an image of a Knight symbol (that we are trying to classify), and (c) both images super-imposed to see the extent of their overlap. The pixels are drawn as dark discs where the two overlap. This represents a pretty good match, which is not surprising, as both were obtained from the same machine printed page.

This technique works well when recognizing symbols from such a controlled environment as machine-printed text, yet problems arise when *any* parameter of the input changed. If the font-face, font-size, or orientation of the page happened to change, a new set of templates had to be produced. From Figure 10, consider how difficult it is to get a good match with a template and an

image under some transformations: (a) scaling, (b) rotation, and (c) line-thickening. In each case, many pixels are mismatched.

Thus, extending this technique to work for hand-printed symbol recognition was not so easy. In hand printed symbols, the variation applied to the symbols is not uniform: each symbol, each time it appears, is slightly different. Even a given author often does not draw the same symbol in the same way. To illustrate the amount of variation one can encounter, consider Figure 11; which contains (a) machine drawn O's and (b) hand-printed O's.

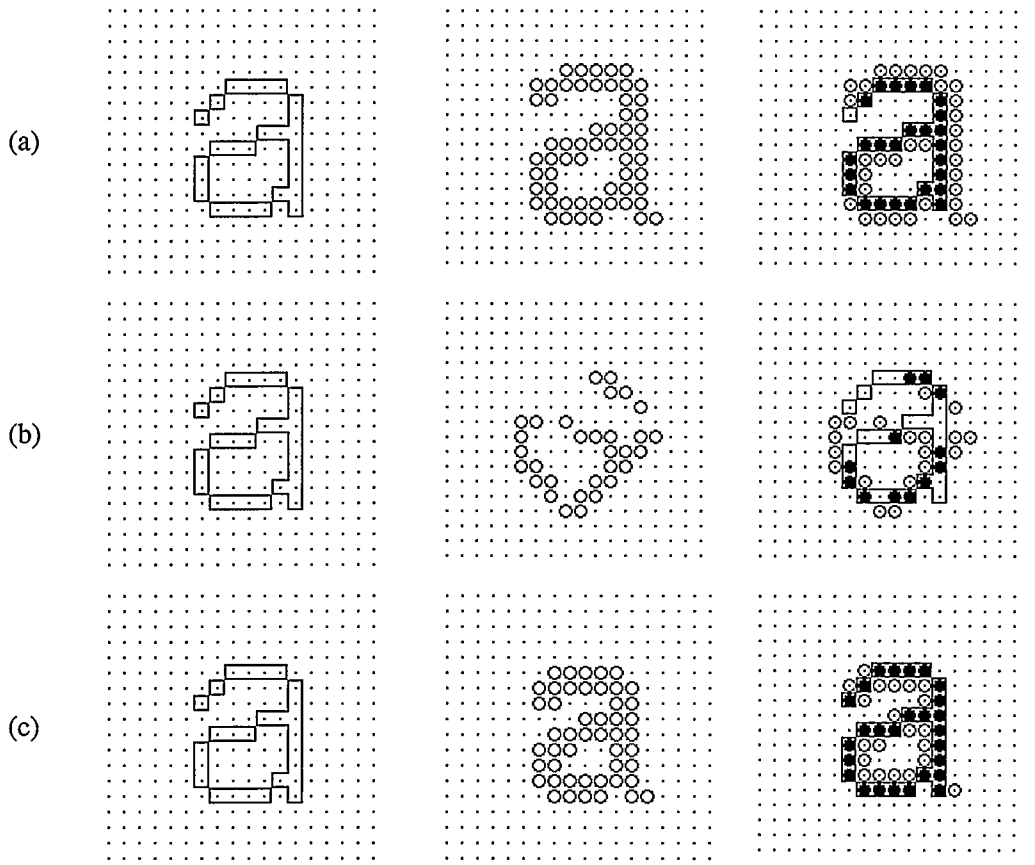


Figure 10 Template Matching under Transformations

(a) (b)

Figure 11 Variation in the capital letter 'O'.

The templates just presented are seen as too rigid when applied to handwritten symbol recognition. In an effort to make prototypes more elastic, many techniques were tried. The following sections gradually introduce the notion of deformable prototypes, until we arrive at the current state of the art.

2.2.2 Angle-of-Sight Shape Signatures [Tchoukanov1992]

Another guise that symbol recognition appears under is that of shape matching. A popular technique in shape matching is to use *shape signatures*. A shape signature is a 1D function that captures the *shape* of an object, usually as some parameterization of the boundary of the object.

The concept of shape can be defined in many ways, but regardless of the exact definition, it should be invariant to rotation, to scaling, and translation. The authors in [Tchoukanov92] define shape as "a simply connected compact region in a 2D Euclidean space." This definition is not invariant to the aforementioned properties but the shape signature they define is.

They provide a short survey of shape signatures, and include polar representation, the centroidal profile representation, the rectangular representation, the tangential representation, the curvature representation, the normal-contour-distance (NCD) signature, the slope density description, "signature", the angle-and-length chord distribution, and the gradient encoding scheme.

It is insightful to see how a typical shape signature is computed. Often an origin is selected and then a measurement is made for each point along the boundary relative to this origin. In the polar representation, one can perform a radial sweep of the object from its centroid and record (r, θ) values of the boundary points.

(a) (b)

Figure 12 Performing a radial sweep along the boundary.

This can lead to a couple of problems. First, some objects have a center of mass that is not contained in the object itself. Sometimes the center of mass is a boundary point. This can cause problems when doing a radial sweep since the θ-value is not well-defined at these points.

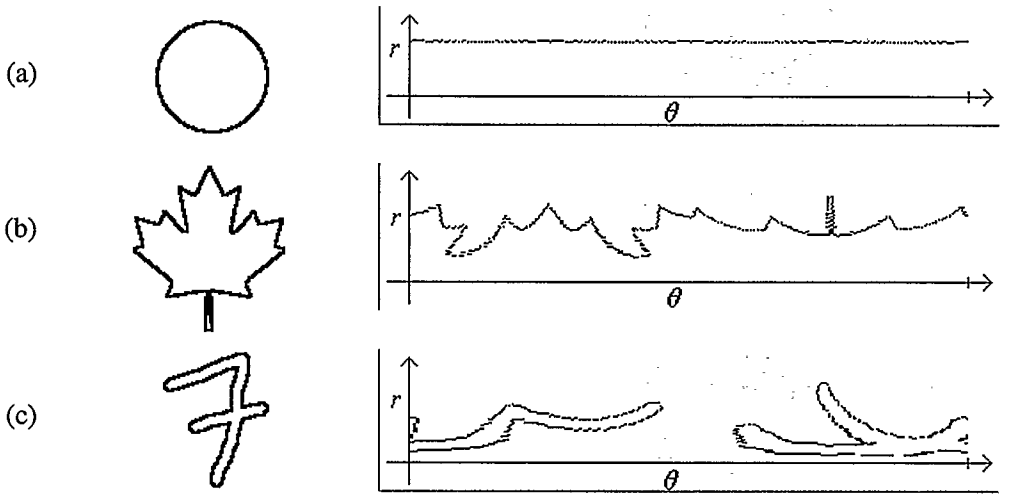
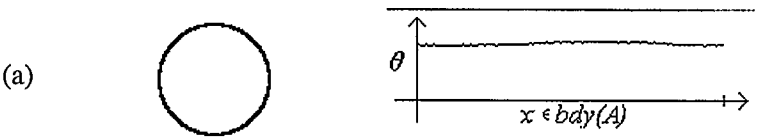


Figure 13 Shape signatures parameterized with (r,θ).

The curves in Figure 13 show shape signatures computed using the (r,θ) parameterization. The x-axis shows θ values, while the y-axis shows r values. It is evident from Figure 13 that the (r,θ) representation has some undesirable properties. The relationship between r and θ is not even a function, and at some points is not defined.

Tchoukanov et al. propose a new scheme that is based on the parameterization of the boundary as a periodic function of one variable (a signature). They record the angle-of-sight, or AOS, of each boundary point to an origin that hovers above the image plane. A similar effect could be achieved by following the boundary and recording distance to the centroid as well.



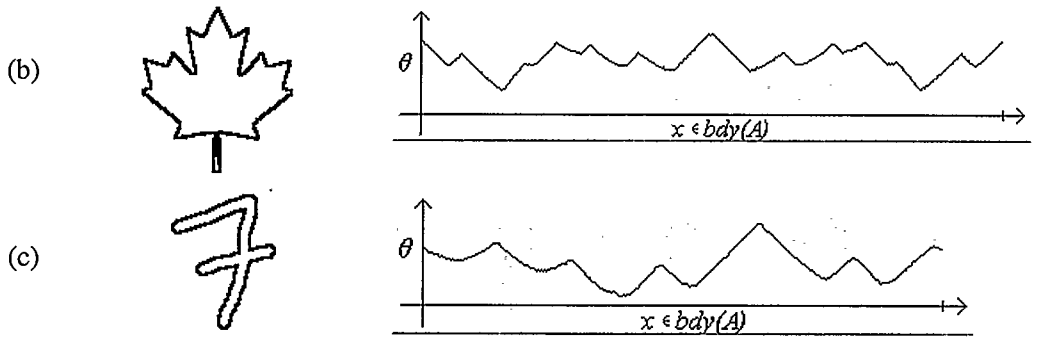


Figure 14 AOS Shape Signatures for some sample objects

In Figure 14, AOS signatures are computed for a set of objects. In part (a) of Figure 14, we see a circle and its shape signature (a flat line), in (b) we see a maple leaf and its signature, and in (c) we see a '7' and its signature. The AOS signature has some important properties. It is a single-valued function, defined at all points, and does not have abrupt changes in the signature amplitude. The signature is made rotation invariant by standardization of the starting point, a rotation of the image corresponds to a shift in the shape signature. It is made scale invariant by normalization of the function values, a change in scale in the image corresponds to a scaling of the function values.

2.2.3 Local Affine Transformation [Wakahara1994]

This work presents a more flexible prototype representation; one that is 'deformable'. Thinning is performed on both the input image and the templates, so the following discussion applies to skeletons. Wakahara et al. describe the application of Local Affine Transformations (LATs) to shape matching and handwritten numeral recognition. An LAT is a 2D transformation that can include rotation, scale change, shearing and/or translation. Each prototype is iteratively deformed in a way that maximizes similarity to the input image. An optimal, (local) affine transformation is computed for each point in the prototype. The goodness of fit is evaluated for a small circular neighbourhood. The points are weighted using a gaussian window function and a Hausdorff-distance is used to measure dissimilarity between the point sets. Multiscale, or coarse-fine matching is achieved in this way, as the effective interacting neighbourhood decreases with each iteration. The end goal of this method is to provide a point-wise correspondence between the two images. (In actuality, two separate correspondences are found, so this is not a *matching* in the strict sense.)

At the end of an iteration, a transformation for each point in the prototype has been found. After applying the deformations to the prototype, they define the dissimilarity measure as

$$D(A,B) = \frac{\sum_{a \in A} \min_{b \in B} \| a-b \|^2 + \sum_{b \in B} \min_{a \in A} \| a-b \|^2}{|A|+|B|}, \quad \textbf{(Eq 3)}$$

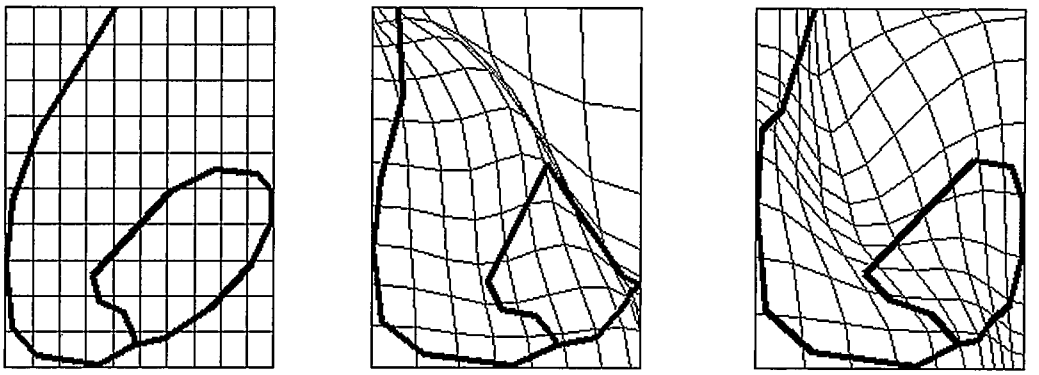
where A is the input image and B is the deformed version of the prototype after a number of iterations. This equation is really the Inkwell Hausdorff distance with some normalizing constants.

The authors also make use of topological and geometric features as structural information. They rely heavily on the crossing index of each pixel. Recall the intuitive definition of crossing index: a value of 0 indicates an isolated point, a value of 1 indicates an endpoint, a value of 2 indicates a link, and finally values of 3 and 4 indicate more complex intersections. Rules are put in place that strongly encourage the crossing index to agree between corresponding pixels. That is, that endpoints should map to endpoints, intersections to other intersections, etc.

However, the combination of thinning algorithms and a strict reliance on crossing indicies can be dangerous. Thinning produces many artifacts, often creating additional branching points (i.e. "necking"). This can create some undesirable correspondences in the output. Thus the algorithm is especially sensitive to noise, in particular boundary noise.

2.2.4 Deformable Templates [Jain1997]

Another approach to deformable prototypes was presented by Jain et al. in [Jain1997]. Although the original paper called them "Deformable Templates", we will use the term prototype to remain consistent with the rest of the work. The motivation for deformable prototypes came from the fact that often an image is similar to a template, but it may have certain parts that are slightly different. It would nice to have a more elastic prototype, one that could account for small differences between observation and a template. If two parts of an image seemed close, then it may be wrong to describe their resemblance as either "they match" or "they don't match". It would be good to have some shades of grey in this area. Thus, with these elastic templates it would be important to understand how much effort was spent bending a template into a specific contortion. If the amount of work is ignored, then any prototype could conceivably be molded to match any pattern. Also, by having this measure described by a parameter, the prototypes could be made more elastic or more rigid as needed.



(a) (b) (c)

Figure 15 Deformations applied to a '6'

In part (a) of Figure 15, we see a digit '6', represented as a set of line segments. In (b) we have created a deformation of the plane and moved *only the endpoints* of the line segments. In (c), we have done this again, for another (random) deformation. The original paper applied the deformation to all points in the prototype, not just endpoints of lines.

Here is a technical (while brief) explanation of their way of modeling deformations. First, a deformation is defined as any function that maps the unit square to itself in a continuous fashion. The points on the boundary must remain fixed while the inner points are free to go to any other point. The entire set of deformations can be thought of as an infinite-dimensional vector space with a countable set of basis vectors. In the enumeration they give, only a small set of the lower frequency basis functions are needed to capture most of the characterizing information of a typical deformation. This model is sometimes referred to as a *rubber sheet model* since it makes the plane appear as a rubber sheet; it appears distorted as if by pulling and twisting.

A search is performed to find the coordinates, in this truncated deformation space, of an optimal deformation of a template to make it match an input image. An objective function is then formulated as a linear combination of the image dissimilarity and the deformation cost. Thus they measure both how well the pictures match each other as well as how much work is required to achieve this deformation. Again, it is important to keep a balance as most prototypes can be deformed to an arbitrary image if enough energy is spent bending the prototype.

Here is a mathematical description of the aforementioned basis, and how it was used to implement deformable templates.

The image is scaled to a unit square. Then a displacement function $D(x, y)$ is defined, and the mapping $(x, y) \rightarrow (x, y) + D(x, y)$ is a continuous function which maps the unit square to itself.

Displacement functions have the property that they are zero on the boundary of the square and only displace inside the square. The space of displacment functions has an infinite orthogonal basis:

$$e^x_{mn}(x,y) = (2\sin(\pi nx)\cos(\pi my),0) \quad \textbf{(Eq 4)}$$

$$e^y_{mn}(x,y) = (0,2\cos(\pi mx)\sin(\pi ny)) \quad \textbf{(Eq 5)}$$

For m, $n = 1,2, \ldots$. Low values of m and/or n correspond to lower frequency components of the deformation in the x and y directions respectively.

Thus an arbitrary deformation is a linear combination of these basis vectors:

$$D_c(x,y) = \frac{\sum_{n=1}^{N}\sum_{m=1}^{M}(c^x_{mn}e^x_{mn} + c^y_{mn}e^y_{mn})}{\lambda_{mn}} \quad \textbf{(Eq 6)}$$

Where the parameters $\lambda_{mn} = \alpha\pi^2(n^2+m^2)$.

The authors take the components with values of m, $n \in \{1,2,3\}$. This means that only 9 low order functions are used and that a linear combination of these represents a deformation (of 18 parameters, as each basis function has an x and y component). Thus, when searching for a deformation, they must solve for the following 18 variables:

$$c^x = \begin{bmatrix} c^x_{11} & c^x_{12} & c^x_{13} \\ c^x_{21} & c^x_{22} & c^x_{23} \\ c^x_{31} & c^x_{32} & c^x_{33} \end{bmatrix},\ c^y = \begin{bmatrix} c^y_{11} & c^y_{12} & c^y_{13} \\ c^y_{21} & c^y_{22} & c^y_{23} \\ c^y_{31} & c^y_{32} & c^y_{33} \end{bmatrix} \quad \textbf{(Eq 7)}$$

We implemented this system and used values of m, $n \in \{1,2,3\}$ such that $m+n \leq 4$. This reduced the search space to only 6 functions and thus finding 12 coefficients. The choice to not take a square matrix was made because the sum of m and n indicate the frequency of the deformation. Each diagonal line through the matrix where $m + n = c$, a constant, indicates a family of functions with similar frequency domain. It makes more sense to select functions from an entire frequency domain, than to take all the low order ones and only a few from the higher domains.

$$c^x = \begin{bmatrix} c^x_{11} & c^x_{12} & c^x_{13} \\ c^x_{21} & c^x_{22} & 0 \\ c^x_{31} & 0 & 0 \end{bmatrix},\ c^y = \begin{bmatrix} c^y_{11} & c^y_{12} & c^y_{13} \\ c^y_{21} & c^y_{22} & 0 \\ c^y_{31} & 0 & 0 \end{bmatrix} \quad \textbf{(Eq 8)}$$

Each template can thus be deformed with these 12 degrees of freedom. In addition to the deformation model explained up to this point, the authors take into account prior information on the deformation coefficients for each template. They assume all the coefficients are independent and identically distributed, with a mean of 0 and variance of σ^2. The value of σ^2 reflects the confidence about the template, with large values of σ^2 allowing more deformation. This essentially controls how "rubbery" the templates are. To measure distance between a template and an image, the template is transformed via the deformations and some linear transformations such as rotation and scaling. The template is then drawn to produce an image and this is compared to the original image. The comparison is done by measuring the distance to the nearest edge pixel taking into account not only position, but also alignment of tangential edge directions.

When searching for an "optimal" deformation of a template with respect to a particular image, a two term objective function is minimized. The two terms are a model-based term, which measures the deviation of the deformed template and the prototype template (i.e., how much bending was necessary), and a data-driven term which describes the fitness of the deformed template contour to the boundary in the image (i.e., how good was the final match / how close were we able to get to the given image).

Although this technique produces high recognition rates, a lot of computational effort must be spent. The algorithm for finding the optimal basis coefficients is an EM (Energy Minimization) algorithm and this has notoriously bad running times. The authors admit that the computational burden is high, but since the matching is so good, one can conceivably reduce the set of templates. They implemented an advanced template selection strategy using *complete-link hierarchical clustering* to refine the size of the template set. [Jain1998]

2.2.5 Shape Matching with Shape-Contexts [Belongie1999]

Belongie et al. introduce a new approach to shape matching based on weighted bipartite matching. Two sets of points are sampled from the image boundaries and then a matching is found between them. The matching process is guided, not solely by distance, but by a heuristic called a *shape-context*. The shape-context at a point captures the shape of the global object from that point's perspective, thereby providing a rich local shape description. The distances between the shape -contexts, rather than distances between the points themselves, are used to build the weight matrix for the bipartite matching algorithm. The algorithm uses a nearest-neighbour classifier to produce the final classification. The shape-context algorithm has been shown to be

effective at recognition of hand-written digits as well as 3D objects. Exactly the same distance function is used for both 2D and 3D recognition.

To avoid excessive computing time, the entire point set of each image cannot be considered. Thus, as a preprocessing stage, a random and uniformly distributed sample of points along the shapes' boundaries is determined, each with approximately the same cardinality.

Furthermore, when determining which points should be matched, it is important to pick points that somehow represent the same "shape portion" of their respective images. The endpoint of a stroke should probably be matched to another endpoint. However, using only interesting points like endpoints and intersections would leaves shapes like a circle without representation. To describe each point of the point set, regardless of its topology, the authors define a *shape-context*. A shape-context defines the local perspective of the overall shape of the image. Different points on the same image will have different a shape-context, but corresponding points in similar images should have similar shape-contexts. The shape-context of a point is computed as a histogram of the relative distribution of other points on the shape from its local perspective.

In Figure 16, we show an example of a shape-context. In (a), (b) we see two images of an 'A', subsampled along the edge points. In part (c) we see a diagram of the log-polar histogram bins used in computing the shape-contexts. There were 5 bins used for $log\ r$ and 12 bins used for θ. In parts (d), (e), and (f) we see example shape-contexts for reference samples marked by the circle and of image (a), the diamond of part (b), the and triangle of part (a), respectively. Each shape-context is a log-polar histogram of the coordinates of the rest of the point set measured using the reference point as the origin. Dark shading implies a large value. Note the visual similarity of the shape-contexts for the circle and diamond, which were computed for relatively similar points on the two shapes. By contrast, the shape-context for the triangle is quite different. Finally in (g) we see the correspondences found using bipartite matching, with costs defined as follows: the shape-contexts are viewed as grey-scale images, and the distance between two shape-contexts is the sum of the squares of the differences in grey-value.

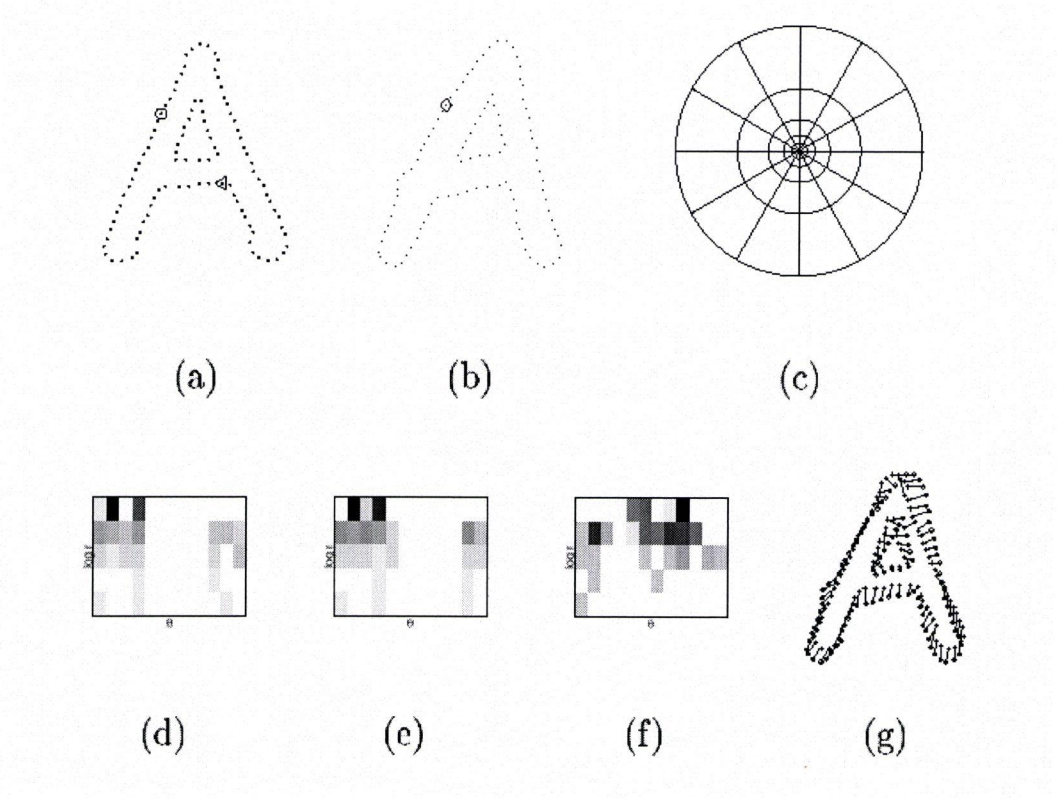


Figure 16 Shape-Contexts

A distance function is also defined to compare how similar two shape-contexts are. Using this function, and n points on each image, an $n \times n$ matrix of distances can be created. To determine the optimal weighted matching between these point sets, we can use the well-known *Hungarian method*. This algorithm runs in $O(n^3)$, where n is the number of points in each set, but it is possible to do better by taking advantage of the fact that the graph is dense [Jonker1987]. Jonker's algorithm is a bit complex to analyze for its asymptotic behavior; while it performs better than $O(n^3)$ algorithms in practice, it does not seem asymptotically better.

Since it is desirable that two points on similar parts of two images have similar shape-contexts, it would be convenient to define it such a way as to have these properties: translation invariance, scale invariance, rotation invariance, and higher sensitivity to local shape information. Translation invariance is automatic since coordinate calculations are relative to the defining point of the shape-context. Scale invariance is accomplished by making the unit of measurement relative to the image size. The median of the set of pairwise distances between points was used as this reference point.

Rotation invariance can be added by computing shape-contexts and changing the frame of reference for each point. A tangent vector to the boundary of the shape at that point can be used as the x-axis. Thus each point has it's own coordinate system, and this yields a rotation-invariant representation for a shape description. Although rotation-invariance is possible, it is not always desired or even necessary. For example, in digit recognition, a 6 and a 9 are equivalent under rotational invariance and we still would like to distinguish between them. Finally, the $log\ r$ axis is used so that the shape that is local to that point has more influence than parts of the shape that are much farther away.

How do we measure the similarity between images based on this matching? One way would be just to use the weight of the optimal matching determined above. However, this would not take into account any measure of how much effort is required to transform one image into the other. Thus the authors use a more sophisticated approach. The point correspondences determined by the matching algorithm are used to estimate the optimal thin-plate-spline (TPS) transformation of one image to the other. These transformations describe the effort required to perform the bending of one image into the other, while maintaining certain constraints, as avoiding self intersection. However, such a transformation may not be able to send all points to their corresponding match in the other image. Thus, the transformation is applied to the first point set, and then the matching algorithm is executed again. In this way, the matching and transformation calculation form one step that can be repeated in an iterative process. Three iterations of first a matching then a transformation calculation are performed.

The measure of image similarity is then determined by a weighted sum of three terms: shape-context distance, image appearance, and the bending energy. The shape-context distance is the value of the optimal weighted matching between one image and the other image under the best transformation. The image appearance similarity is defined by the sum of squared brightness differences in gaussian windows around corresponding image points. Lastly, the bending energy corresponds the magnitude of the transformation required to align the shapes.

2.3 Conclusions

We have now looked at various ways of representing prototypes, starting from basic bitmap templates. Although these are an acceptable representation in some very controlled environments, a more flexible template is needed for handwritten symbol recognition. We have shown how to represent prototypes as shape signatures, which provide rotation and scale

invariance in a novel way. We then examined local affine transformations to produce local template deformability. Along this same direction came 'deformable templates' and then another elastic representation using shape-contexts in unison with bipartite matching. The techniques presented achieve high recognition rates, but suffer from a common drawback of high computational burden.

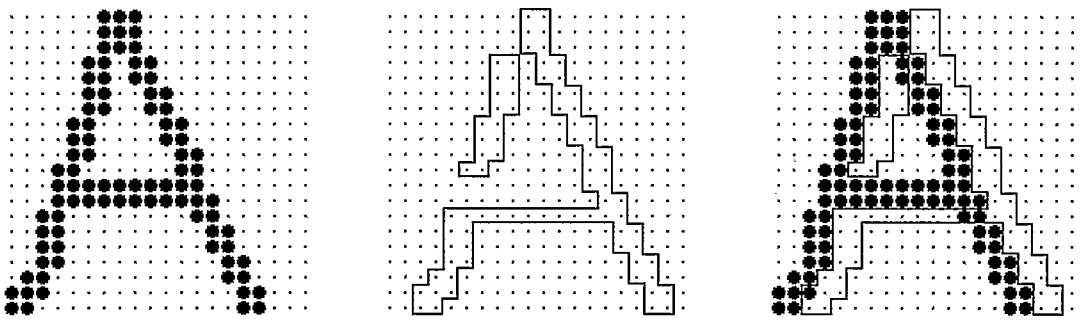
Before presenting vector templates in Chapter 4, we would like to discuss methods of bilevel image comparison. We mentioned the distinction between matching in pattern space and matching in feature space; we will be concerned with matching in pattern space and the distance algorithms of the next chapter will be the vehicle through which this is done.

Chapter 3

Measuring Distance Between Images

In many symbol recognition algorithms, especially template-based algorithms, it is very important to know how similar two images are. If the template is stored as an image, or converted to one at a later stage, we need to compare it against the unclassified image and report how good the match is. When we introduce *vector templates* later on, this will be a key part of the classification process.

We can view the act of measuring distance between as two images that of accumulating penalties for their local differences. An obvious way to measure dissimilarity is to overlap the two images and count how many pixels are different. This can produce acceptable results in some very controlled cases, ones that require precise image alignment, identical orientation, as well as uniform stroke width.



(a) (b) (c)

Figure 17 An image and a template with almost no overlap, and thus a large distance from each other, yet with similar shape.

However, it can happen that two images with seemingly identical shape have a very poor amount of direct overlap. In Figure 17 above, we see (a) an image of an 'A', (b) a template of an 'A', and (c) both images super-imposed to see the extent of their overlap. They have very little direct overlap (5 pixels only) but have similar shape. It is true that these images could be aligned better, but locally this phenomenon could still happen.

In handwritten pattern matching, we cannot control all of these parameters. And so, for more difficult datasets such as these, we would like to have a more tolerant dissimilarity measure, one that can identify similar shape in two images that have little or no direct overlap in their graphical

representation. We shall formalize the concept of distance between images and then describe some techniques that compute such a distance, and evaluate them. We will define and explain what a distance transform is and show how it can be used to compute a more tolerant distance measure.

3.1 Definitions

We define some of the vocabulary used in subsequent sections. We define the important terms *image*, *metric function*, *pixel metric*, and *distance transform*, in addition to a few others.

3.1.1 Image

We are concerned primarily with bilevel images, and thus we can view an *image* as an $m \times n$ binary matrix over $\{0, 1\}$. Pixels with value 0 are called *background* pixels and pixels with value 1 are called *foreground* pixels. The image height corresponds to the number of rows, m, and the image width corresponds to the number of columns, n. Sometimes it is convenient to treat an image as a point set. This point set of an image $A = [a_{ij}]$ is the set of (i,j) coordinates of foreground pixels in A. The rows are numbered from 0 to $m - 1$, where the columns are numbered 0 to $n - 1$. The point set is defined mathematically as

$$P(A) = \{(i, j) \mid a_{ij} = 1\} \qquad \textbf{(Eq 9)}$$

We use the notation A to mean both A and $P(A)$, as we trust the meaning will be clear from the context.

Since the above definitions are important in the context of measuring distance between images, we will state a few additional conventions regarding distance. When measuring distance between two images, we mean to say we are measuring dissimilarity. The higher the value we compute, the more unlike two images are, and the closer to zero it is, then the more identical they are. We assume we are always comparing two images of the same size. If this was not the case, we could always scale the larger image down to the size of the smaller, or perform some scaling operations to equalize the image sizes.

3.1.2 Metric Functions

We would like to make precise the notion of a function that measures distance. In this regard, we can talk about *metrics* in the sense of *metric spaces*. A metric space is defined as a set together

with a function called a metric that assigns a real number to every pair of elements in the space. This function is everywhere positive, has the properties of identity, symmetry and triangle inequality. These properties correspond to our intuitive notions of similarity. Identity means an object is similar to itself (i.e., the distance from it to itself should be 0), symmetry means the order of comparison does not matter, and the triangle inequality means that if two different objects resemble a third, then they also resemble each other. Formally, we would like metric d to satisfy the following properties: (for objects x, y and z in the space)

$$d(x,x)=0, \quad \forall x \tag{Eq 10}$$

$$d(x,y)\geq 0, \quad \forall x,y \tag{Eq 11}$$

$$d(x,y)=d(y,x), \quad \forall x,y \tag{Eq 12}$$

$$d(x,z)\leq d(x,y)+d(y,z), \quad \forall x,y,z \tag{Eq 13}$$

For our purposes, the domain points x, y, and z are primarily images. However, we make use of metrics as well when measuring distance between actual pixels themselves.

3.1.3 Pixel Metrics: L_1, L_2, L_∞

There are many different ways to measure the distance between two points in a 2D vector space (we are primarily interested in the space of 2D points with integer coordinates). The following norms are three commonly used measures, and are actually metrics in the sense of metric spaces.

For the following definitions, consider

$$p_1=(x_1,y_1) \text{ and } p_2=(x_2,y_2), \tag{Eq 14}$$

and define

$$d_x=|x_1-x_2| \tag{Eq 15}$$

$$d_y=|y_1-y_2|. \tag{Eq 16}$$

The L_1 metric is defined as

$$L_1(p_1,p_2)=d_x+d_y \tag{Eq 17}$$

which could be interpreted as the distance on a 4-connected grid. It is sometimes referred to as *4-distance.*

The L_2 metric is defined as

$$L_2(p_1, p_2) = \sqrt{d_x^{\,2} + d_y^{\,2}}\,, \qquad \textbf{(Eq 18)}$$

which is just the Euclidean distance.

The L_∞ metric is defined as

$$L_\infty(p_1, p_2) = \max\{d_x, d_y\}\,, \qquad \textbf{(Eq 19)}$$

which could be interpreted as the distance on a 8-connected grid. It is sometimes referred to as *8-distance.*

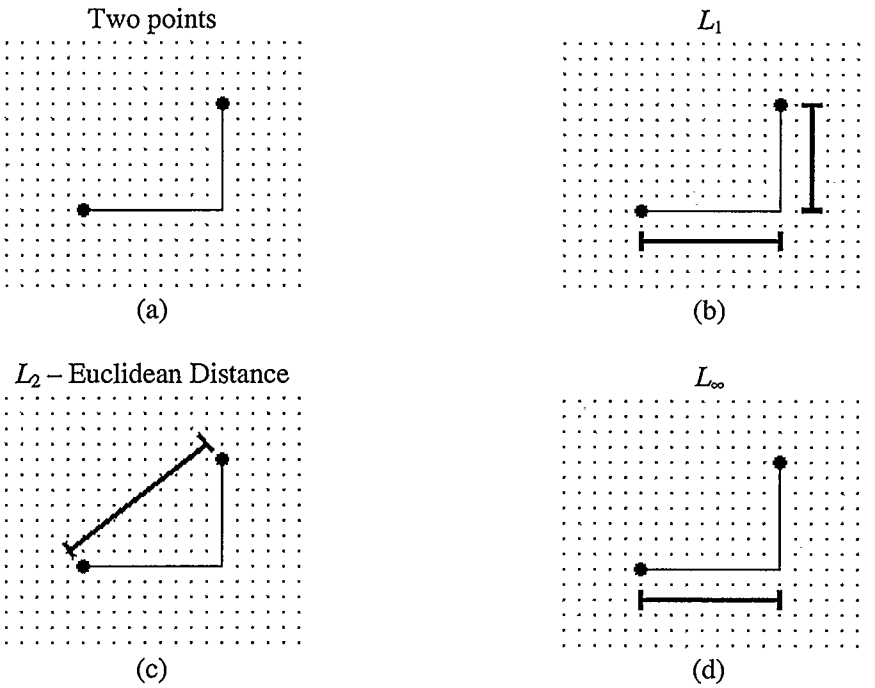


Figure 18 Comparison of Pixel Metrics

In Figure 18, we see two points in (a) and we measure the distance between them using different pixel metrics. In (b), we use L_1, yielding $d = 9+7 = 16$, in (c) we use L_2, giving $d = \sqrt{130}$, and finally in (d) we use L_∞, giving $d = \max\{7,9\} = 9$.

These three metrics can be related graphically by the following diagram. We plot circles of a fixed radius using each of the three metrics to measure distance.

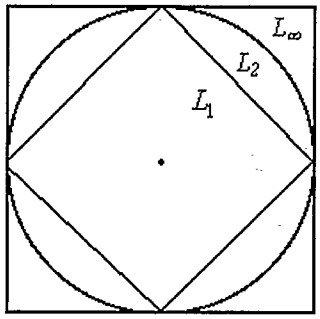


Figure 19 Circles of constant radius under different point metrics

Figure 19 compares the pixel metrics L_1, L_2, and L_∞. For each metric, we trace out a "circle", the set of all points that are a fixed distance away from the center. We see that L_2, Euclidean distance, produces a standard circle. L_1 measures steps in the vertical direction separately from steps in the horizontal direction, and so underestimates the Euclidean distance (creating a smaller circle), and the L_∞ metric overestimates the Euclidean distance by counting a diagonal step (of length $\sqrt{2}$) as a unit step.

The following inequality can be easily proven:

$$L_\infty(p_1, p_2) \le L_2(p_1, p_2) \le L_1(p_1, p_2) \quad \textbf{(Eq 20)}$$

Proving this is equivalent to showing the following two inequalities hold:

$$\max(|dx|, |dy|) \le \sqrt{dx^2 + dy^2} \le |dx| + |dy| \quad \textbf{(Eq 21)}$$

3.1.4 Distance Transforms and Feature Transforms

A *distance transform* of an $m \times n$ image A, is an $m \times n$ matrix T, of real numbers, and the value of t_{ij} indicates the distance from the point (i,j) to the nearest foreground pixel in A. The distance from one point to another can be defined by an arbitrary point metric. An example of a distance map will be shown in Figure 21 (a).

$$T_D(A) = [d_{ij}] \text{ where } d_{ij} = \min_{a \in A} |(i, j) - a| \quad \textbf{(Eq 22)}$$

A *feature transform* of an $m \times n$ image A, is an $m \times n$ matrix F, of coordinates, and the value of $f_{ij} = (u,v)$ indicates that a nearest point to (i,j) in A is (u,v). Certainly the point (u,v) need not be distinct, as many pixels (u',v') may be the same minimal distance from (i,j) as (u,v) is. An

example of a feature transform (although vectors are truncated to unit length for clarity) is shown in Figure 21 (b).

$$T_F(A) = [f_{ij}] \text{ where } f_{ij} = \arg\min_{a \in A} |(i, j) - a| \qquad \textbf{(Eq 23)}$$

The difference between a distance transform and a feature transform is that a distance transform tells how far away the nearest pixel is, and thus is (scalar-valued). A feature transform tells which pixel is the nearest pixel (actually *a* nearest pixel, as there may be more than one nearest neighbour), and thus is vector valued. Given a feature transform, it is possible to determine the distance transform but not vice versa. Also, we can view a distance transform as being the discrete case of a Voronoi diagram. A Voronoi diagram for a given (2D) point set partitions the plane into regions, such that (i) each region contains exactly one point x from the point set X, and (ii) each region R_x (a convex polygon) contains all points in the plane that are nearest to it's point x. From this perspective, a distance transform of an image A partitions the image into regions of pixels such that each region contains one point a of A and all pixels in that region have a as their nearest neighbour from A.

Both types of transforms depend on the underlying norm to measure distance between pixels. Any one of L_1, L_2, and L_∞ can be used; the exact choice depends on the application circumstances.

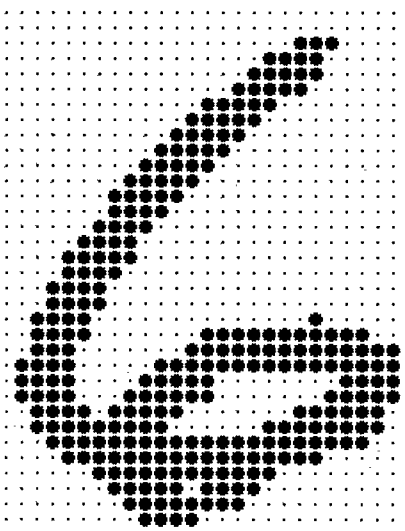
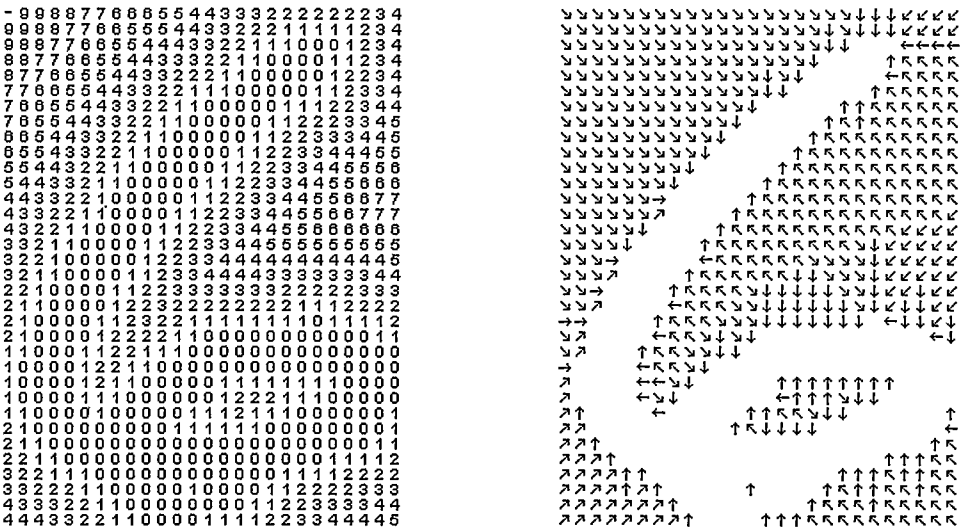


Figure 20 An image of a six

We compute both a feature transform and a distance transform for Figure 20. The foreground pixels (value 1) are the dark discs and the background pixels (value 0) are the light points.



(a) (b)

Figure 21 Distance Map and Feature Map using L_∞

Figure 21 shows us a distance transform in (a) for the '6' of Figure 20. At each pixel location, we have recorded the distance to the nearest foreground pixel. Recording a distance value of 0 means that location contains a foreground pixel. In part (b) we present a feature transform for the previous figure of a '6'. The feature map shows an arrow for each pixel location; each arrow points to the nearest foreground pixel. For purposes of clarity, all vectors are drawn with unit length, although their magnitude varies according to the distance map in (a). Tracing a path of arrows shows a shortest path to a nearest foreground pixel (since there may be more than one).

3.2 Binary Correlation

One of the simplest ways to measure distance between two images is to overlap them and count how many pixels disagree.

$$\beta(A,B) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\left|a_{ij} - b_{ij}\right| \qquad \textbf{(Eq 24)}$$

This works well when comparing machine drawn symbols as either they will usually overlap exactly or not. However, with some variation present in the symbols, this method penalizes all differences by the same amount. Two pixels that are a distance of 10 apart are treated in the same way as two adjacent (but not overlapping) pixels. A good example of how this could yield a misleading distance was in Figure 17. In this figure, although we could align the shapes better,

this type of misalignment can happen locally and would cause the same type of erroneous distance penalty. In summary, the method of binary correlation is not tolerant to small misalignments. From our experience, it does not generally give great results when used in hand-written pattern recognition.

3.3 The Hausdorff Distance Family

The Hausdorff distance is an attempt to fix some of these problems by providing a distance metric between images that is more discriminating. It is able to account for small distortions and penalize them in the distance values according to their magnitude. If two pieces of the image are greatly misaligned, a large term will be contributed to the final distance sum. However, if only a small difference is encountered, only a slight penalty will be incurred.

It may be easiest to explain the Hausdorff distance using an analogy to pizza delivery. Imagine a city of homes and pizza delivery restaurants. We have two maps, one contains all the homes, and the other the pizza places. We are interested in answering "What is the longest time anyone would ever have to wait for a pizza?" For each home we would like to associate with it its nearest pizza place, and over all homes we need to find the one that is farthest away from its pizza place. An answer to this question would tell us approximately how compatible these two maps are. Getting back to distance between images, we could think of one image representing the residences, and the other the pizza places. Computing the longest time that anyone would have to wait for a pizza is just computing the Hausdorff distance.

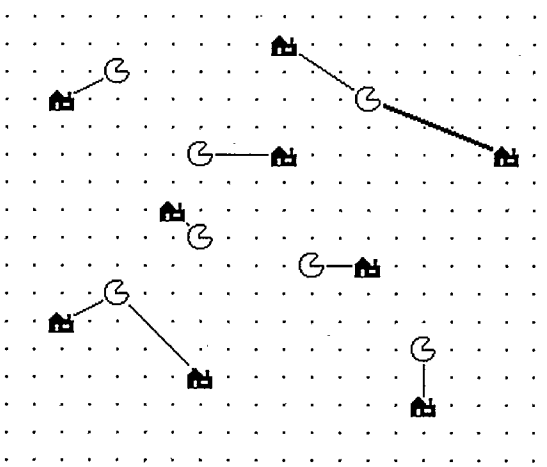


Figure 22 Pizza Delivery Example to Explain Hausdorff distance

Figure 22 shows the map of the homes and of the pizza places super-imposed on top of one another. The line segments show a line from each home to its nearest pizza place. The bold line shows the longest of these segments. The length of this line is the (directed) Hausdorff distance from the homes to the pizza places.

3.3.1 Hausdorff Distance

The Hausdorff distance measures the extent to which each point of a 'model' set lies near some point of an 'image' set and vice versa [Huttenlocher1991]. Thus this distance can be used to measure the degree of resemblance between two images that are superimposed upon one another.

To make a precise definition, let A and B be images, and d be a metric on the space of 2D points. (e.g. L_1, L_2, L_∞). Define $N_B(a)$, for each point a of A, to be the nearest neighbour b in B:

$$N_B(a) = \arg\min_{b \in B}\{d(a,b)\} \qquad \textbf{(Eq 25)}$$

Then, we could define a function (for each a) that measures the distance to this nearest neighbour b

$$\delta_B(a) = d(a, N_B(a)) = \min_{b \in B}\{d(a,b)\} \qquad \textbf{(Eq 26)}$$

Finally, the *directed* Hausdorff distance between the two images is

$$\vec{h}(A,B) = \max_{a \in A} \delta_B(a) = \max_{a \in A} \min_{b \in B}\{d(a,b)\}, \qquad \textbf{(Eq 27)}$$

measuring the largest distance any a is from a nearest neighbour b. This function however is not symmetric, i.e. there are cases where $\vec{h}(A,B) \neq \vec{h}(B,A)$.

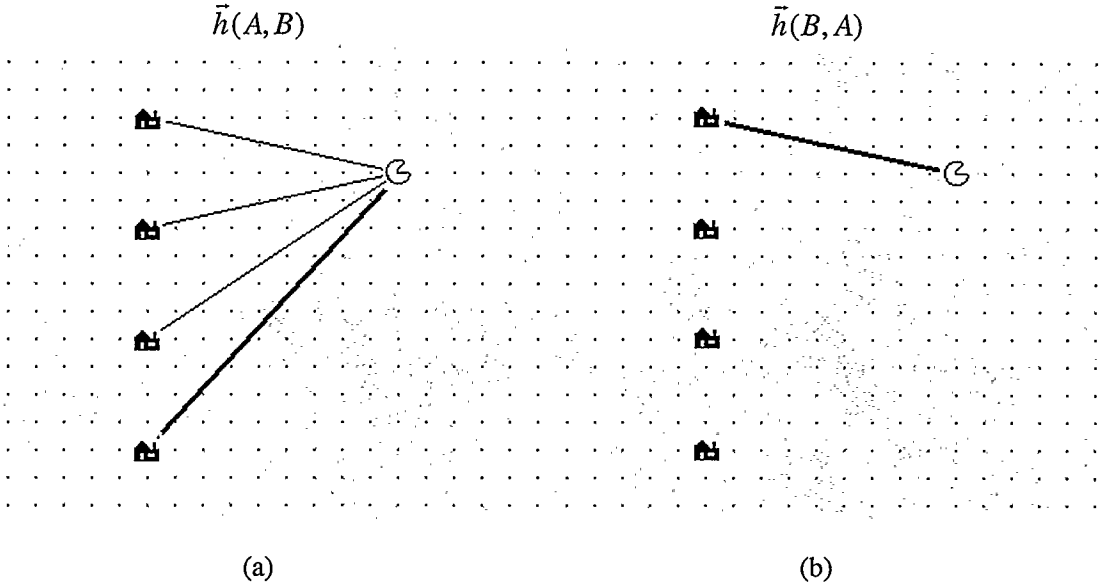


Figure 23 The Hausdorff distance is not symmetric.

In Figure 23 (a), on the left, we can see a calculation of $\vec{h}(A,B)$. We draw one line segment for each home to its nearest pizza place, and draw the longest of these in bold. In diagram (b) on the right we see a calculation of $\vec{h}(B,A)$. We draw one line segment for each pizza place, and draw the longest of these in bold. We define A as the set of homes and B as the set of pizza places. The length of the bold line shows the (different) directed Hausdorff distance in each case.

To avoid favoring one image in a comparison, and thus to make the function symmetric, it can be computed in both directions, to yield the *undirected* Hausdorff distance between A and B.

$$h(A,B) = \max\{\vec{h}(A,B), \vec{h}(B,A)\} \qquad \textbf{(Eq 28)}$$

3.3.2 Inkwell Hausdorff distance

The Inkwell Hausdorff distance measure is named this way since it can be explained with a metaphor involving an inkwell. Before explaining it this way, we will explain it first in terms of the pizza delivery example above, to relate it to Hausdorff distance. We are interested in measuring how much gas will be required to deliver pizzas to *all* the homes, assuming each driver can carry only one pizza at a time. This would measure the overall *total effort* required to service an area. If this total effort is low, then it means the two points sets are in rough agreement with each other, i.e. that the two images resemble each other. On the other hand if the total effort is quite high, it means that the images are quite different.

In terms of inkwells, the distance can be explained as follows: suppose an artist has a picture in front of him, image A, and wishes to draw a new picture, image B, but starting on a fresh canvas. Each of these are images in regards to the definition of binary pixels on a discrete grid. The artist has an ink-pen and treats each pixel in A as an inkwell, that is, it has an unlimited supply of ink. His pen can only hold enough ink however to color one pixel at a time. We wish to measure how much *work* the artist has to do to reproduce B given the inkwell locations on A. Also, we can assume the two images are superimposed on top of one another. To measure the amount of work, we will compute the total distance the pen has to move while carrying a dab of ink.

Contrasting this definition with the Hausdorff distance above, we are replacing a maximum operator with a summation operator. In the inkwell metaphor used above, the ordinary Hausdorff distance would simply measure the largest single distance the pen would have to travel to color a pixel.

Formally, the directed Inkwell Hausdorff distance between the two images is

$$\vec{H}(A,B) = \sum_{a \in A} \delta_B(a) = \sum_{a \in A} \min_{b \in B} \{d(a,b)\} \qquad \textbf{(Eq 29)}$$

Compare this to (Eq 27). A similar technique is used to make it symmetrical, we combine both terms as follows:

$$H(A,B) = \vec{H}(A,B) + \vec{H}(B,A) \qquad \textbf{(Eq 30)}$$

An algorithm to compute it efficiently was published by Parker. [Parker1991].

3.3.3 Computing the Hausdorff Distance Efficiently

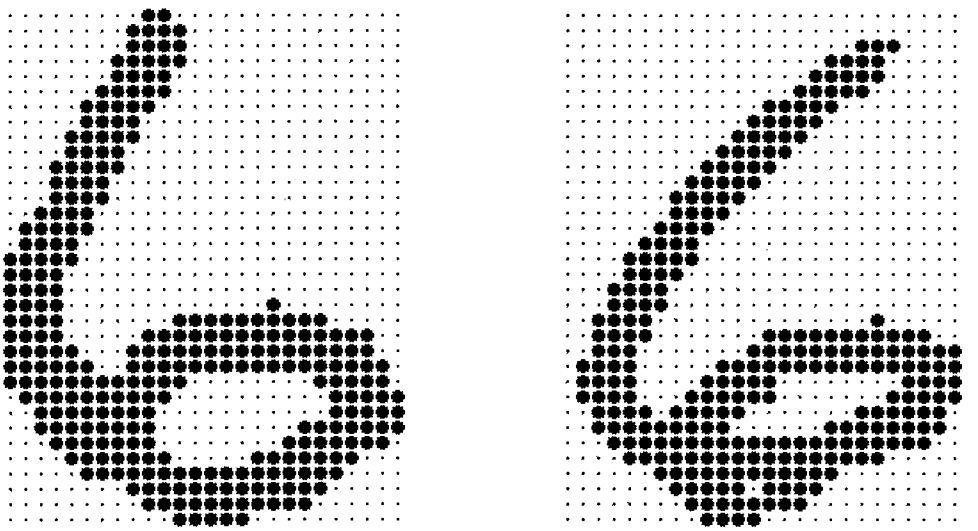
For the Hausdorff distance to be of practical use, we need to be able to compute it efficiently. In [Huttenlocher1991], a survey is performed of various algorithms to compute Hausdorff distance, as well as more general definitions of the distance itself. When the points are on an integer grid (as is the case with images) we can make use of distance transforms and feature transforms. See Section 3.1.4 for the corresponding definitions.

We make use of known algorithms to efficiently compute these transforms. In [Parker1991] we find how to compute these transforms in linear time (with only two passes over the image) for the norms L_1 and L_∞. These algorithms rely on dynamic programming, and make decisions for

how far away a's nearest neighbours are, by looking at the nearest neighbour's of a's neighbours. In the first pass, we determine the nearest of a's neighbours that are above or to the left of a. In the second pass, we determine the nearest of a's neighbours that are to the right or below a. Finally, we just keep the best neighbour from either pass as a's nearest neighbour.

We can also compute the L_2 norm (or Euclidean-distance norm) distance transform in linear time, again with two passes through the image, as is shown in [Gavrilova2000]. We implemented the algorithm in that paper, and, with a few minor corrections, it worked as promised. The idea is again a two-pass algorithm, but uses some more sophisticated datastructures and some simple geometry to determine the nearest neighbour.

To give an example of computing the directed Hausdorff distance using distance transforms, consider the following (unclassified) image, of another six:



(a) (b)

Figure 24 Comparing distance between two images

Figure 24 shows an example of how to measure distance between two images, (a) and (b), using a distance transform. Both figures represent a hand-drawn digit '6'. We will compute the distance transform of figure (b).

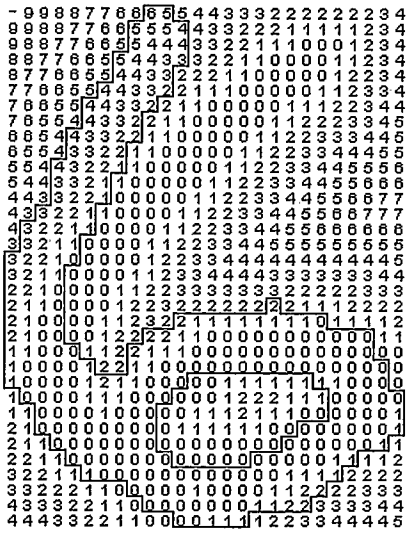


Figure 25 Overlaying both images.

The distance transform of Figure 25 (b) is shown as a grid of integers, and image (a) from the previous figure is drawn as an outline. The Inkwell Hausdorff distance would return the sum of all numbers inside the curve. The Hausdorff distance would return the maximum value inside the curve. The underlying point metric was L_∞.

3.4 Earth-Mover's Distance

The Earth-Mover's distance is another way to measure distances between images. For bilevel images, it turns out to be most similar to bipartite matching. We will compare and contrast it with bipartite matching, as well as the other techniques we have seen.

The Earth-Mover's Distance was originally proposed by Rubner [Ruber1998], as a metric between discrete distributions with weights associated to each data point. In [Cohen1999] the analogy to moving earth is explained well. For the two distributions, one distribution is seen as a set of piles of dirt, of varying sizes; the other is a set of holes, again of varying sizes. The Earth Mover's Distance (EMD) computes how much work is required to move the dirt from the piles into the holes. It is assumed that the size of the distributions is the same, that is the amount of dirt equals the volume of the holes. If this is not the case, we can take the larger distribution and label it as the dirt, and the smaller one as the holes. The EMD in this case computes how much work is required to fill the holes, and ignores the unused dirt. The meaning of "how much work" is the sum over all displacements of the amount of dirt multiplied by the distance it travels.

When comparing greyscale images, this analogy fits well. We sum up the grey values in both images and determine which image is larger, i.e. which has more "dirt". The smaller image will be treated as having holes with depths proportional to grey values. Each grey-scale value increment is one unit of dirt or one unit of volume in a hole. We compute how much work is required to move the dirt into all the holes.

When comparing binary images, a foreground pixel is a single unit of dirt. If the images have identical area, then the EMD really computes a weighted bipartite matching between two the point sets, with weights determined by distances between points.

3.4.1 Comparing the Hausdorff Distance to the Earth Mover's Distance

It is interesting to note the differences between the Hausdorff-distance and the Earth-Mover's Distance in the discrete case (i.e. comparing binary images). We will compare these algorithms using the earth-moving metaphor.

The Hausdorff distance only measures the largest distance a single pile of dirt would have to be transferred to fit into a hole. The Inkwell Hausdorff distance would allow us to "reuse" dirt. After a pile of dirt is used up, it would magically reappear again to be a candidate for filling other nearby holes. This is why it was explained in terms of a bottomless inkwell. And of course the Earth Mover's Distance itself does not allow reuse of dirt, but finds a direct correspondence between dirt and holes, although a pile of dirt may be shared between many small holes.

As for the required computational effort: for binary images, it requires computing a minimum weight bipartite matching, so $O(n^3)$.

3.5 Distance Viewed as Weighted Bipartite Matching

In this section, we will describe some areas that were explored in applying results and algorithms for bipartite matching to image comparison. The motivation for this was that the concept of detecting and aligning similar parts of two images to measure similarity between them seems like a natural one. The Hausdorff distance finds a correspondence between pairs of points in images but this correspondence is not a matching, it is much less stringent since points can be "reused" – many points can match to the same target. It is an interesting question to ask how the situation would change if we enforced a strict 1-1 correspondence between points in the images. Although bipartite matching is more expensive to compute than a distance transform, it may produce better results if used properly.

We will first define and describe what the weighted bipartite matching (WBM) problem is and then show how it can applied to image comparison.

3.5.1 Definition of Weighted Bipartite Matching

The weighted-bipartite matching problem is sometimes called the *assignment problem*, in which we look at a set of workers and set of jobs, as well as a productivity associated with each person-job pair. The productivity function describes which workers are better at what jobs and by how much. We are asked to find the best correspondence between workers and jobs, i.e., the one that maximizes the overall productivity.

Formally, let $A = \{a_1, ..., a_m\}$ and $B = \{b_1, ..., b_n\}$ be sets, and let $C = [c_{ij}]$ be the $m \times n$ matrix that represents the distance function, defining the distance or cost of matching each pair (a_i,b_j). Let M be a subset of $A \times B$. The set M is called a matching if each a_i appears in at most one (a,b) pair in M, and similarly for each b_j. If (a,b) is in M, it is said that a is *matched* with b. The number of elements in M is called the *cardinality* of the matching. In the weighted bipartite matching problem, we ask the question of "What is the minimum cost matching (of maximum cardinality) we can find?". The cost of a matching is just the sum of the costs of all the matched pairs:

$$c(M) = \sum_{(a,b)\in M} c(a,b) \qquad \textbf{(Eq 31)}$$

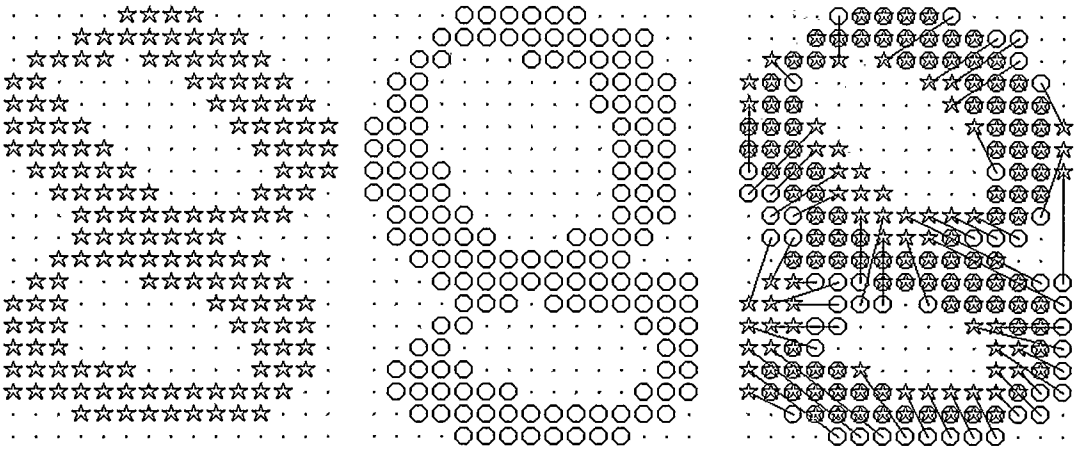
The problem of maximum weight matchings and minimum weight matchings are very closely related; one instance can easily be phrased as an instance of the other. For example, to solve a minimum weight matching problem, we can just negate each entry of the cost matrix of the maximum weight matching instance, and then add a fixed value to make all the weights positive. i.e. $c_{ij}' = k - c_{ij}$, where $k = \max_{i,j} c_{ij}$

3.5.2 Image Comparison using Matchings

Image comparison can be viewed as an instance of the minimum weight bipartite matching problem. The point sets of each image make up the respective bipartitions, and we seek to match each point in one image to it's "nearest" neighbour in the other. In this case though, we are looking only for a 1-1 correspondence, so each point can only be matched with exactly one other point in the other image. It is important to be clear on this, as in cases like the Hausdorff distance one pixel may be a nearest neighbour to many others. To determine distance we use the

Euclidean distance between the points, although more complicated distance functions are possible. For example we could take into account the local topology of a pixel making use of its crossing index, so endpoints are matched with endpoints, intersections with intersections, etc. The distance between the images is then reported as the cost of the minimum weight matching.

Consider the following two images, and then an optimal matching between them.



(a) Point Set *A*, 159 pixels (b) Point Set *B*, 154 pixels (c) Min. Cost Matching of cardinality 154

Figure 26 Example of a matching between the point sets of two images.

In Figure 26 (a) above we see an example of a hand-drawn '8', we will call this image *A*. In (b), we see another instance of an '8', we call this image *B*. Finally, we show the results of a minimum cost weighted matching between the two point sets. Since the sets are not the same size, 5 pixels from A were left out of the matching. With these basic parameters, we are in fact performing the Earth Mover's Distance calculation.

We can label the pixels in image (a) as $a_1,...,a_m$ and the pixels in image (b) as $b_1,...,b_n$ then we have used the weight matrix

$$c(a_i, b_j) = L_2(a_i, b_j). \qquad \textbf{(Eq 32)}$$

One problem that is evident from this diagram, is that during the optimization the algorithm allows two neighbours that are quite far apart to be matched. This is a problem because we would like to use this match to explain how to deform one image smoothly into the other. This is not very useful to us because we would like to view this matching as a distortion from one image into the other. Thus we would like one part to be smoothly transformed into its corresponding

part, and not twisted, if it can be helped. Thus the matching algorithm should prefer a set of average displacements to a set of small displacements combined with one large displacement. Before discussing how to change the cost function to more accurately reflect these goals, let us get a better feel for the amount of twisting and bending implicit from a matching. Consider that each matching induces a vector field on the plane. A vector is created for each point that is displaced in the optimal matching. Technically, the matching need not be unique, but we will assume we have any optimal matching.

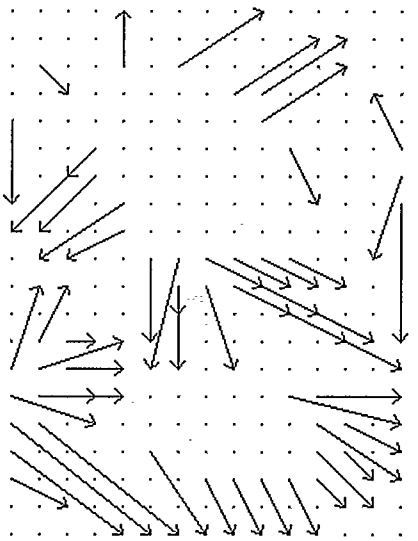


Figure 27 Vector field for matching using $c = L_2$.

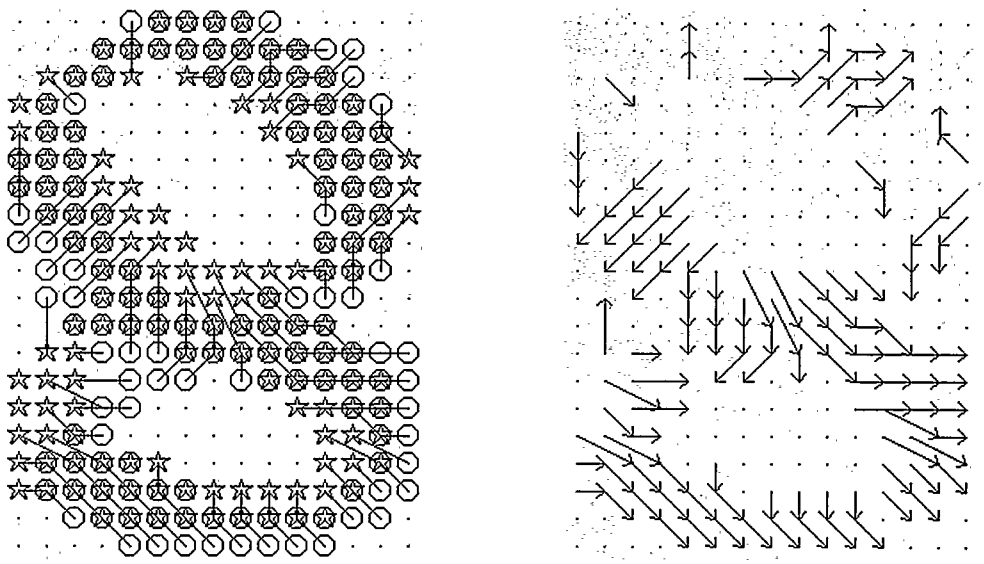
In Figure 27, one vector is plotted for each point that was displaced by the optimal matching. The vector shows where each point was carried to. The set of vectors plotted together gives an impression of the global effect on the plane.

This vector field in turn could be used to estimate a transformation of the plane that would carry one image into the other. When comparing images, we would like to measure the amount of twisting and bending that was needed to contort one image into the other. If the amount of bending is small, then the two images are likely a good match. If the amount of bending is excessive, then the distance should be penalized accordingly. In the vector field above, we see the transformation of the plane would be quite "messy".

To fix this situation, we can assign steeper penalties between points that are very far apart. We can achieve this by squaring the distance values between the points:

$$c(a_i, b_j) = L_2{}^2(a_i, b_j) \qquad \textbf{(Eq 33)}$$

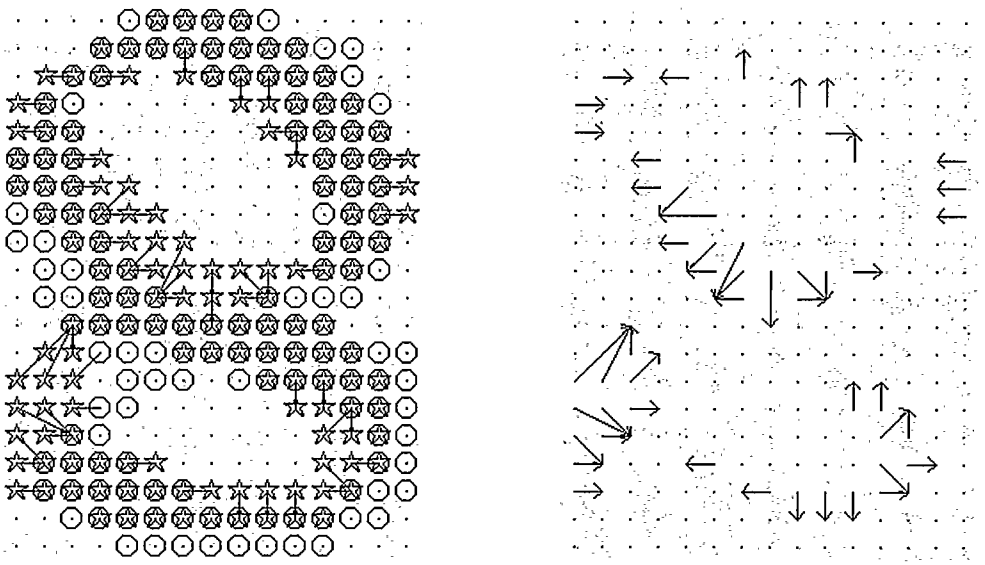
It turns out this new penalty function does solve the aforementioned problems, as is evidenced by Figure 28. All the longer line segments have disappeared, and only shorter vectors remain.



(a) (b)

Figure 28 Weighted Matching using improved penalty function $L_2{}^2$.

It may be interesting to compare this vector field with the one obtained from the Inkwell Hausdorff distance, where we measure the distance to *any* nearest neighbour, and more than one pixel can be "matched". Thus we really have a union of two mappings, from one image to the other image and vice versa.



(a) (b)

Figure 29 Matching up pixels under the Inkwell Hausdorff distance

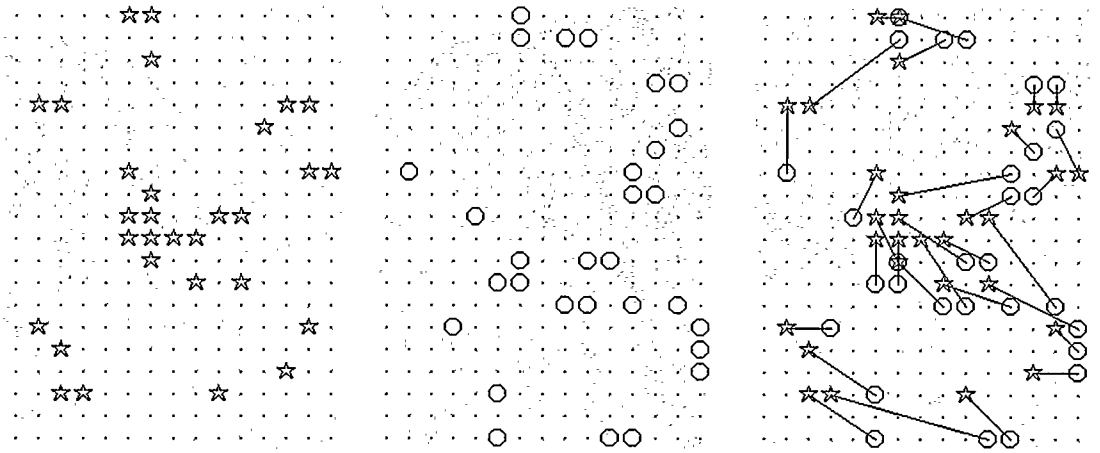
In Figure 29 (a), we see how pixels are mapped to their nearest neighbours under the Inkwell Hausdorff distance. We see that some pixels from image B are used more than once in the mapping. In (b) we see this mapping drawn as a vector field. We will call the vector field generated from two distance transforms in this way the 'mutual proximity vector field', since this concept will be used later on, particularly in the future directions section of Chapter 7. The magnitude and number of the vectors is greatly reduced from the vector field generated from the bipartite matching, where the matching is a one-to-one function.

3.5.3 Reducing the Number of Points

We have used the standard algorithm for weighted bipartite matching, called the "Hungarian Method" [Kuhn1955], and it borrows ideas from results in flow maximization, maximum cardinality matchings, and linear programming. However, the algorithm runs in time $O(n^3)$. This would be too slow for larger images, so we would like some way to reduce the number of points we must match. We tried two techniques.

First, we decided to subsample the patterns randomly and extract only about 30 points or so from each image. Besides leaving a manageable number of points, this had the added advantage of making both sets the same size so that a perfect matching could be found (A *perfect matching* is a matching that includes all vertices). Taking a random subsample is not an ideal subsample, since one area could be favored by the random-number generator. In fact, results looked quite

poor for the two given images. However, if we could ensure the points would spread out evenly, we think we would see that the shape could still be captured fairly well. This could be achieved by modeling the points as repelling particles, and allowing them to spread out by themselves. This would no longer lead to a random and independent distribution, but what is important is that we have captured a representative portion of the shape of the object with fewer points.



(a) Random subsample of 30 points

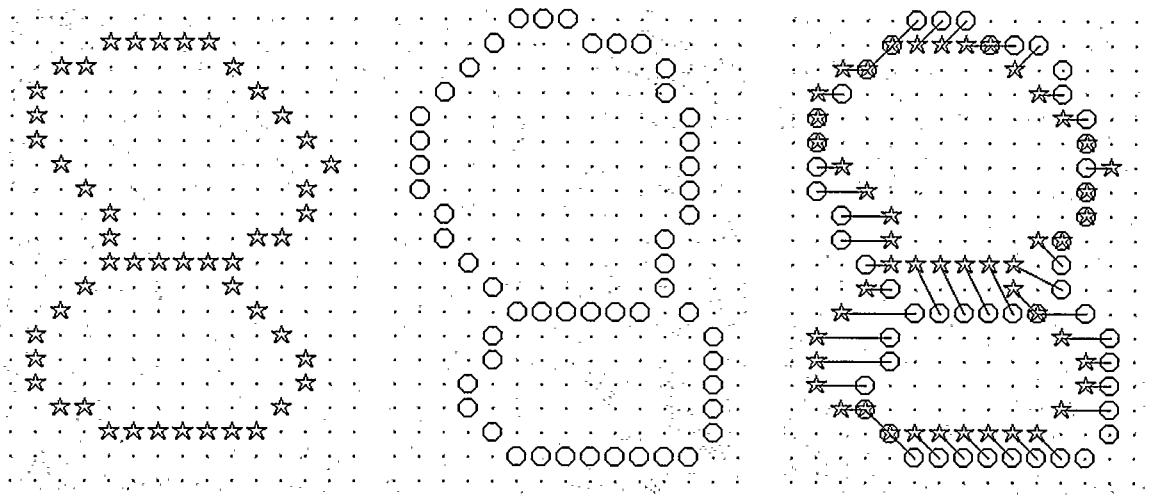
(b) Random subsample of 30 points

(c) Matching using improved penalty function

Figure 30 Random Subsampling (30 points each) without any redistribution.

In Figure 30 (a) and (b), we have sampled the patterns of '8's we have been using, but only with 30 random pixels. As is evident, the general shape of the original '8's is hardly recognizable.

Given the poor results from this technique, an improvement was necessary. Another way to reduce the number of points is to perform thinning on the images, and then perform the matching only on the skeletal pixels.

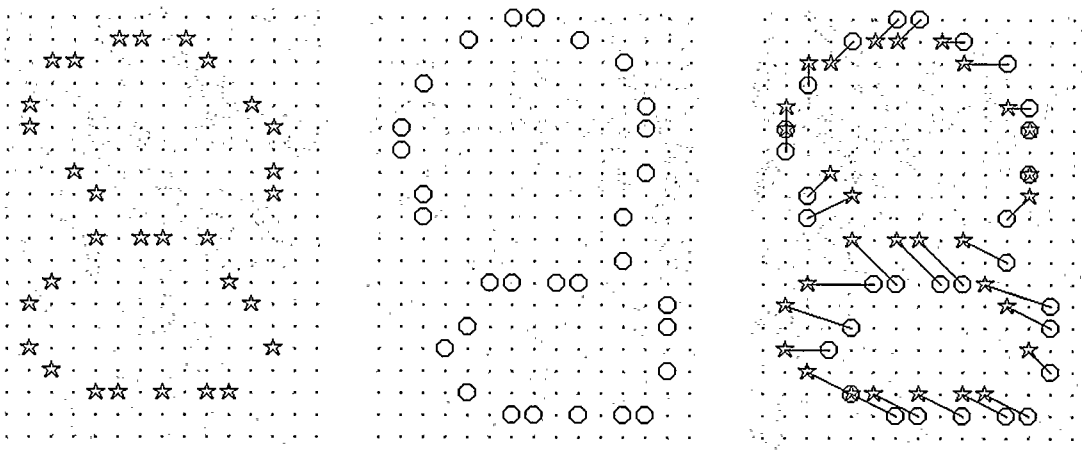


(a) Skeleton with 49 points (b)Skeleton with 52 points (c) Matching using L_2^2

Figure 31 Bipartite matching on skeletal pixels only

Shown in parts (a) and (b) of Figure 31 are the skeletons of the two '8' seen so far. In (c) we compute the optimal bipartite matching between the pixels.

This produces a much better results, as the transformation from one skeleton to the other is in fact quite smooth. This process could still, however, produce an excessive number of points. Since skeletons are made up of curves, defined as pixel sequences for our purposes, it is fairly straightforward to come up with a way of distributing k pixels uniformly onto each skeleton. Visually, and in the case of hand-printed digits, with 30 points it seemed that a sample representative of a digit's shape was captured. Also, by using a deterministic approach, the distributions are guaranteed to be uniform, and of course better-looking results are achieved. Here are some examples of subsampling the skeletons to a smaller value of 30 pixels each.



(a) (b) (c)

Figure 32 Low Resolution Skeletons

The images in Figure 32 represent subsampled skeletons of the images of '8's we have been working with. The overall shape of the 8's appears preserved.

One further step with this technique that we initially experimented with but didn't have time to take to completion is to analyze the connections between points in the skeleton of A and the induced connections in B. For example, if a_1 and a_2 were adjacent in A, then how about $M(a_1)$ and $M(a_2)$ – were they connected as well? Use of this information could be helpful in the final distance calculation. Images that are similar should, after determining a matching, preserve connectivity.

3.5.4 Conclusions on Matching

Although this technique gave promising initial results, we did not have the time to make it competitive with the Inkwell Hausdorff distance for digit recognition. However, Belongie et al. have shown that with some extra innovations the method can be made to work well. At the same time as we were exploring bipartite matching, independent authors [Belongie1999] were using Shape Contexts to perform a similar type of comparison. See the literature survey in the previous chapter on their approach for more details. They used a more sophisticated cost function between points, not relying just on Euclidean-distance. They computed a shape-context, i.e. a local perspective of the entire image's shape. These shape contexts drove the matching process, to make sure that similar features on the images were matched. Also, this was used to estimate an optimal transformation that would map one point as close as possible to the other. The overall

algorithm then interlaced matching iterations with iterations to determine transformations, until the system converged on the best match.

In any event, once the weights between points have been determined, and an optimal weighted matching has been found, there are still a few choices for the measure of distance. An obvious one is the cost of the optimal weighted matching, but it is also possible to estimate the associated transformation and factor in "how much bending energy" was required to align the shapes.

3.5.5 Generalizations of Matching

When comparing two symbols, the idea of *matching up corresponding portions* between the images in an optimal way is an intuitive one. But what we don't understand is exactly what to match up and how to evaluate the goodness-of-fit. We saw that we could use pixels and the distances between them, but perhaps more sophisticated features and distance measures are needed.

It is possible to generalize the previous attempts at applying matching algorithms to symbol recognition. Essentially, we need to specify *what* objects will be matched and *how* to determine the goodness-of-fit between them. From this generalized perspective, we could evaluate and compare different strategies quite easily. Once the objects and the function that measure dissimilarity are determined, a distance matrix can be built between the two sets. It is common for authors to choose a subset of the point-set of an image for their objects. Another approach is to identify features on the image. These can be topological features like endpoints, points of intersection, holes, etc. or they can be any other kind of spatial features. Once it is known which objects will be paired up, we need to know how well two of them match. This could be based on Euclidean-distance, local shape information, or any number of attributes.

3.6 LCS Distance

3.6.1 LCS Distance Motivation

The theory of string matching is very closely related to pattern matching in images, as images could be considered to be 2D strings. This is especially true if we look at the theory of approximate string matching, where we do not ask if two strings are identical, or if they have identical matches, but only if the have good approximate matches. In handwritten pattern

recognition, we are always trying to answer the question of whether or not something is a good *approximate* match.

Due to this realization, we wanted to try some of the most well-known approximate string matching algorithms in the context of 2D pattern recognition.

One algorithm that came to mind was the edit-distance problem. It answers the question "What is the cost of transforming string A into string B if the costs of insertion, deletion, and copying are specified?" Fortunately, edit distance has already been tried in image matching, but alas the running time was too long. For two strings of length n, the running time is $O(n^2)$, and for two $n \times n$ images, the running time is $O(n^4)$, which is too slow to be of practical use.

The other problem we considered was the longest common subsequence (LCS) problem. To explain what this is, consider the following two strings:

APPLES FOR BREAKFAST

POTATOES FOR DINNER

How similar are these strings? Well, the longest sequence of letters they have in common is

PESFORE

This sequence need not be consecutive, it just must appear as a subsequence (i.e. in that exact order) in both strings. Formally, given words $S = s_1,\ldots,s_k$, and $T = t_1,\ldots,t_l$, we seek to maximize n for subsequences of the indicies $i_1,\ldots,i_n$, where $1 \le i_1 < i_2 < \cdots < i_n \le k$ and $j_1, \ldots, j_n$, where $1 \le j_1 < j_2 < \cdots < j_n \le l$, so that the strings agree with each other on all the points along the subsequences: $s_{i_1} = t_{j_1}$, $s_{i_2} = t_{j_2}$, ..., and $s_{i_n} = t_{j_n}$.

Now, how do we convert an image into an integer sequence? One obvious way is to consider the images as a sequence of bits, with the successive rows of the image concatenated together. This essentially moves the problem back down to 1D string matching. However, the algorithm runs in $O(lk)$ where l and k are the string lengths, so this would lead to a running time of $O(m^2n^2)$ for two m by n images. This is the same as the running time for edit-distance, which is too slow.

To make the algorithm feasible, we would need to reduce the image size, or find a way to summarize the image information into a more compact form. We found a way to compress an image from $O(mn)$ to $O(m)$ by storing only one integer for each row; the number of background to foreground transitions encountered in a row scan. Of course this is not the only choice we have. It would be possible to count the number of foreground pixels, or any measure of each line for that matter. Another idea that we think has potential is to compare the Freeman chain code

for the boundary of the image. (The Freeman chain code stores the direction to the next pixel in an ordered traversal of a shape's boundary) Two similar shapes will likely have a long common subsequence of these boundary vectors.

Finally, any such simplified sequence can then be passed to the LCS algorithm.

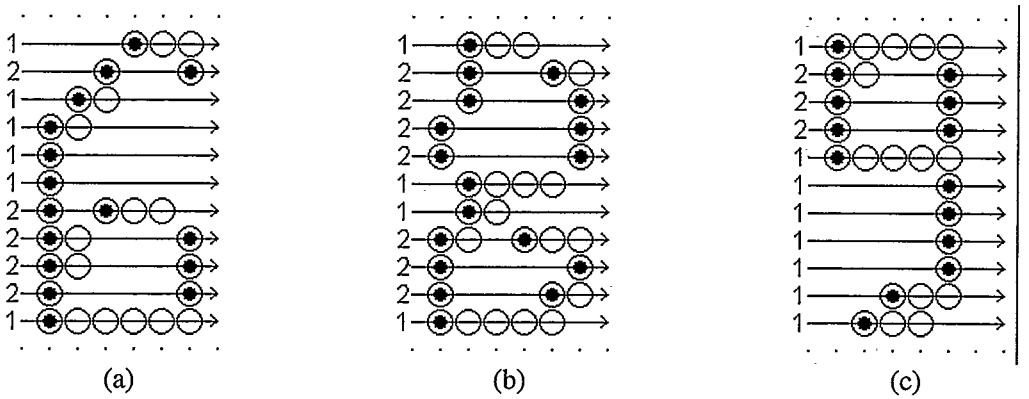


Figure 33 LCS Distance Calculation

The row sequence of image Figure 33 (a), above, is <1,2,1,1,1,1,2,2,2,2,1> and the row sequence of image (b) is <1,2,2,2,2,1,1,2,2,2,1>. The LCS of these two strings is <1,2,1,1,2,2,2,1>. In (c), the row sequence is <1,2,2,2,1,1,1,1,1,1,1>. Even though the row sequences of (a) and (c) have many members in common, they do not appear in the same order, and thus the LCS of the row sequence in (a) and (c) is shorter, consisting only of <1,2,1,1,1,1,1>

To normalize this distance value, we compute how long the common subsequence is in terms of the length of the original sequences. A value of 1.0 indicates the sequences are entirely identical, and a value of 0.0 indicates they are completely different. Being so easy to normalize, this metric is very scaleable and can give meaningful distances for arbitrary image sizes. In the example above, this ratio is 8/11, as 8 of the original 11 characters are matched, and this means that 73% of one image could be "explained" by the other image.

Slicing only in the row direction captures a limited amount of information about the image. A '9' and a '1' are different since the 9 will have many 2's at the top of the sequence, and then many 1's. A '9' and a '6' are also very different, since the six has the right proportion of 1's and 2's, but in the wrong order. Only one group (the 1's or the 2's) would be matched in an LCS calculation.

But a 'p' and a '9' are very similar since they will both have almost identical row sequences. To get around this problem, the image compression can be applied in more slicing directions. We

can slice the image horizontally, vertically, and both directions diagonally. Thus we can compute 4 sequences for each image, and then compare them, via an LCS calculation. By slicing in many directions, a much larger pattern space can be recognized. We present some results on how well this metric performs in Chapter 6. Although not better than the Inkwell Hausdorff distance, it will be shown this metric still possesses substantial discriminative power.

Chapter 4
Vector Templates

Vector templates introduce a new method of prototype representation in a template-based symbol recognizer. This new representation describes an adaptable, deformable, yet efficient template; one that takes the approach of "recognizing symbols by drawing them" [Parker2001a]. There are two extremes on the continuum of template matching algorithms for handwritten symbol recognition. On one end, we have very quick bitmap template matching algorithms, which produce low recognition rates [Vanderbrug1977]. On the other we have the modern, hi-tech world of deformable templates, with very high recognition rates, but very slow running times [Belongie1999], [Jain1997]. This thesis is an attempt to bridge the gap with the best from both worlds. We discuss how vector templates are much more flexible than standard templates in Sections 4.2 to 4.4, and then we discuss how vector templates are fast deformable templates in Section 4.5.

4.1 Introduction

Before getting started, we present an overview of how vector templates are used in the classification process. We hope this will help put each of the subsequent sections into a better context. When classifying an arbitrary symbol, we first compute the style features of the image. These primarily include size, orientation, and uniform line thickness. Next, for each template, (actually for a small set of linear transformations of each template) we render the template to match the aforementioned style parameters, thereby producing an image the same size as the input image. Finally, we measure the similarity between the two images using an Inkwell Hausdorff distance, effectively computing the cost of optimally deforming the template into the image and vice versa. Finally, the distances are passed to a 1-NN (nearest neighbour) classifier to determine the final classification.

4.1.1 Definition of a Vector Template

Stated simply, a vector template encodes the shape of a line-based symbol as a set of line segments. These segments can be thought of as the medial axes of the strokes (much like that of a skeleton) that make up an image of the symbol.

4.2 Vector Templates are Dynamic

Traditional bitmap template matching stores prototypes as raster images. This set of prototypes is then compared against a target image, and the prototype image that is most similar determines the image's class. This method has been demonstrated to work quite well on machine printed symbols, as all of these are a consistent, size, font face, and orientation. However, when moving to handwritten symbols, the variation in these parameters is significant, and the system has difficulties achieving good recognition rates. The difficulties stem from the fact that the templates are extremely inflexible. The templates are heavily scale dependent; if one wants to recognize a symbol that is much different in size from the templates, then they are out of luck. A new set of templates must be added to the library. Bitmaps, especially binary bitmaps at low resolution, generally look terrible when rotated. Thus, rotating the bitmaps in this traditional classifier is not really an option. Also, while scaling bitmaps down can produce reasonable looking results, scaling a bitmap up (to a larger resolution) generally creates a poor image. For all of these reasons, bitmap template matching is not very successful at handwritten symbol recognition.

We address these issues with a simple, scaleable prototype representation. We represent a template as a "vector template". Each template is stored as a set of line segments. We will explain how to create vector templates in Section 4.3 using a combination of thinning and vectorization. In Section 4.4, we will explain how to draw vector templates, which involves having the template adapt to match style parameters measured on the input image.

4.3 Creating Vector Templates

Vector templates can be created in many ways, depending on what one starts with. First off, a vector template can be created by hand by drawing a set of lines and recording their coordinates. This technique, although very primitive, can still produce good results [Parker1999], where an example was shown where one (hand-drawn) template was used to recognize 96% of all the instances of a '2' in a particular dataset. More generally though, a vector template can also be created from a binary image, by first thinning the image and then applying a vectorization process. Although there is extensive literature on thinning and vectorization, the technique is not sensitive to the exact choices made for these algorithms. This is due in part to the fact that many templates are chosen for each class, so glaring artifacts in one or two images will be overshadowed by the other (correct) templates in a class. The creation process could be made

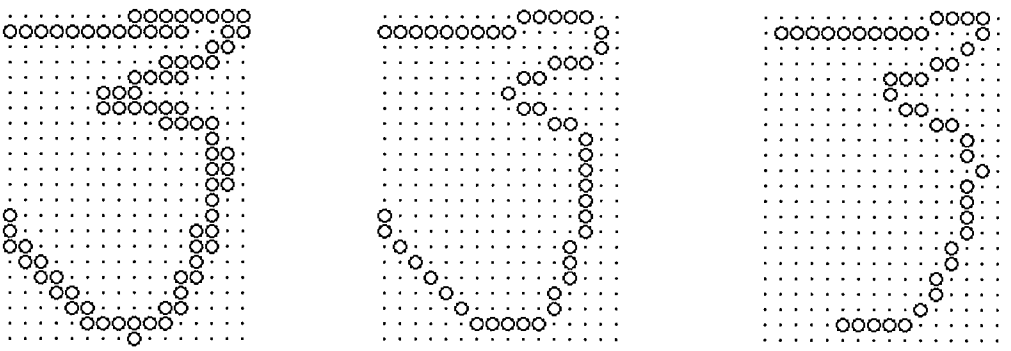
even less dependent on exact choices for these algorithms by using a small set of thinning algorithms and vectorization algorithms, and for each combination producing and storing a template.

4.3.1 Thinning

For thinning, we have used Holt's extension to the Zhang-Suen method described in [Zhang1984], [Holt1987]. Unfortunately, when using Holt's extension to Zhang-Suen, erroneous skeletons are sometimes produced[1]. However, this algorithm is not the only one to produce artifacts from thinning. In fact, many thinning algorithms do this.

> "There are thinning methods [that operate in different ways] but all will produce distorted skeletons in some instances. In thinning, as in many things, settling for good enough is the best that can be done, and for most applications, the Zhang-Suen method is good enough." [Parker1994].

However, to illustrate the types of problems one can encounter, consider the following results of thinning an image of the digit '3' using two thinning algorithms (Figure 34).



(a) Original Image (b) Zhang-Suen Skeleton (c) Zhang-Suen-Holt Skeleton

Figure 34 Demonstration of thinning algorithms at work

The original image is quite thin already, and this sometimes causes problems in thinning algorithms. In (b) we can see a reasonable skeleton, but in (c), a large portion of the '3' has been erroneously deleted.

---

[1] We have recently discovered corrections by Lu and Wang [Lu1986] to the Zhang-Suen algorithm. Implementing the corrections fixes the problems portrayed in Figure 34.

4.3.2 Vectorization

The primary difference between a skeleton and a vector template is that the vector template is further refined into only a set of line segments, as opposed to a skeleton which is only pixels. Vectorization is the process of transforming a skeleton into a set of vectors (or line segments). For the vectorization process, we have used the technique described in [Lam1988]. Briefly, this method identifies curves and then uses a divide-and-conquer strategy to split these curves into line segments.

We present a definition of a term that occurs often in the next section. The *crossing index* of a foreground pixel *a*, relative to image *A*, is the number of four-connected regions that remain in a 3x3 window of *A* centered on pixel *a*, after *a* has been removed.

Starting from the thinned image, we identify *curves;* a connected sequence of skeletal pixels. A curve can be one of two types: it can be open or closed. First, we repeatedly extract open curves from the skeleton by finding endpoint pixels (crossing index of 1) and then following the pixel trail until we find another endpoint or another pixel already encountered. After all such curves are extracted, there may be "loops" remaining in the resulting image. All pixels on a loop have a crossing index of 2, so we can no longer naively search for *endpoints*, as there are none. In this case, we start at an arbitrary point and then follow the pixel trail until we find a previously-encountered pixel or get back to where we started. Finally, for each such curve of either type, we apply a divide and conquer strategy to approximate it by line segments. Points along a curve segment are approximated by a line segment connecting the endpoints of the curve. If any of the points are too far away from this line segment, then the curve is split at a point that has a maximal distance from the line.

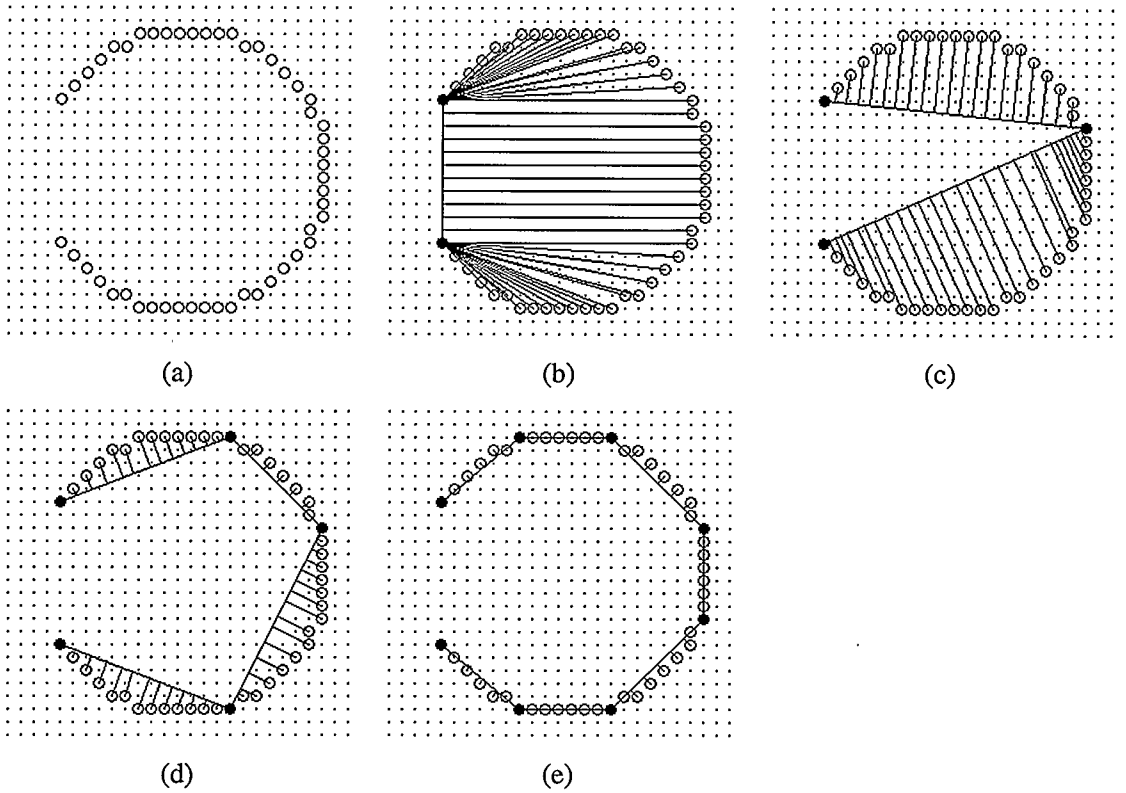
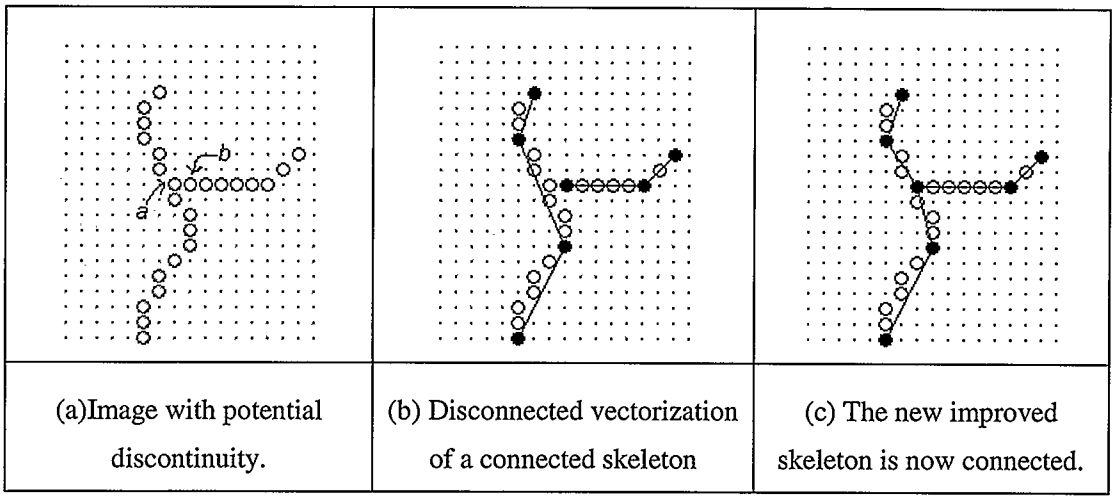


Figure 35 The Vectorization Process

The threshold for the vectorization process in Figure 35 was 1.5 pixels, meaning that an approximation to curve *C* by line segment *L* is deemed satisfactory provided that no point of *C* exceeds a distance of 1.5 pixels from *L*. In (e) of Figure 35, we see the final vector template what would be produced from the initial skeleton of an open circle in (a). There is a slight problem with this technique however. Consider Figure 36. If we follow curves only from endpoint to endpoint, then we sometimes pass over pixels, like point *a*, that have a crossing index of at least 3. These intersection points are important since if we split this 'twig' into two curves, the shaded pixels and the clear pixels, then there will be a discontinuity between pixels *a* and *b* (see Figure 36 (a)). Although this won't matter much for vector templates since the distance function is not too sensitive to boundaries / connectivity, it may be a desirably property of the vectorizer to produce a connected set of line segments for a connected image. To achieve this effect, we improved the original algorithm in [Lam1988], by redefining what a *curve* is. Originally, it was defined as a sequence of pixels from one end point to another, or a sequence of pixels forming a loop, i.e. starting and ending at the same pixel. We redefine a curve as a sequence of pixels from

one important point to another, important points being endpoints as well as intersections. For the special case of loops, a curve can also be a sequence of points starting and ending at the same pixel of crossing index 2. This improvement now ensures the preservation of connectedness between images and vector output. A demonstration of this improvement is shown in Figure 36.



(a)Image with potential discontinuity.	(b) Disconnected vectorization of a connected skeleton	(c) The new improved skeleton is now connected.

Figure 36 Preserving the Connectedness Property During Vectorization

4.4 Drawing Vector Templates

The process of drawing vector templates involves trying to render the template so it matches the *style* of the input image as best as possible. This is important because the vector template only captures the high level shape information of a pattern, and must be custom tailored to achieve a more precise fit. By being able to customize the way each template is drawn, this technique can be used to match a wider variety of patterns.

4.4.1 Applying Linear Transformations to Templates

Vector templates are easily scaled and rotated to fit arbitrarily sized and oriented patterns. To rotate or scale a vector template, the same linear transformation is applied to each point in the template. We have tried two types of linear transformations, rotations and shearing.

A rotation by an angle, θ, is defined by the transformation in (Eq 34).

$$R_\theta \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \qquad \textbf{(Eq 34)}$$

Shearing is accomplished only along the x-axis, as handwriting slant is usually in this direction. The shearing operation is defined in (Eq 35).

$$S_{s_x,s_y}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & s_x \\ s_y & 1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}. \qquad \textbf{(Eq 35)}$$

For our purposes, we have set s_y=0, and thus when we refer to *the shearing value*, we speak of s_x. In Figure 37, we see some examples of the linear transformations we applied to vector templates.

(a) (b) (c)

Figure 37 Linear Transformations applied to a Vector Template

In (a) we see the original vector template. In (b) and (c) we see a vector template under rotation and shearing. The rotation value was $\theta = -20$ degrees, and the shearing value was $s_x = +0.6$. In each case, after the transformation is applied, all templates are resized to fit the same size bounding box, so an additional side effect due to scaling is present.

4.4.2 Line Thickness in Images

When examining hand-drawn symbols, it usually appears that they are drawn with a uniform line thickness. However, this need not always be the case. The writing style, the instrument being used (a fountain pen can leave darker, wider areas), and other factors can all contribute to varying line width in an image. To try and match the style of these images, different strategies for drawing the lines of the vector template are needed. To illustrate this point, consider Figure 38, in which we see (a) a template and (b) an image. We would like to have our template match the style parameters of the image, and in this case that means mimic the line thickness more closely. To simplify the situation somewhat, we model each line in the vector template as having its own independent line width. The question we are asking in Figure 38 is "how thick do we make each line?" in the name of having the template match the style of the image as best as possible.

(a) (b)

Figure 38 Disparate line-thickness between (a) a template and (b) an image.

We discuss three ways in which to model line thickness in images, and possible ways to implement them. The first shows what happens when we ignore line thickness altogether.

4.4.2.1 Model 1 - Ignore Line Thickness

For some data sets, ignoring the line width outright, and plotting each line of the vector template as a pixel-wide line can sometimes give acceptable results. It would be statistically meaningless to cite an exact measure of improvement, as it would be sensitive to the exact nature of the dataset, and how many templates are being used. However, just as an example, we selected only 3 templates[2] at random for each digit class in the ETL dataset (see Chapter 6), and then compute a recognition rate on 2000 symbols. We compared many strategies of plotting thick lines due to stroke-width estimates on the images as well as the strategy of plotting thin lines (1 pixel wide). Thin lines produced a recognition rate of 88.15% and the best thick lines we had produced only about 92.45%. Again, these numbers do not show conclusive evidence that thick lines are better, but do show two things: (i) even thin lines can achieve some level of success, and (ii) plotting thick lines that match the image style are worthwhile.

It may be surprising that such a high recognition rate can be obtained even with thin lines. The reason for this is that the underlying image metric, the Inkwell Hausdorff distance, has a lot of discriminative power. Had we been using simply binary correlation to compare images, the thin lines would not perform nearly so well. Evidence of this will be presented in Chapter 6.

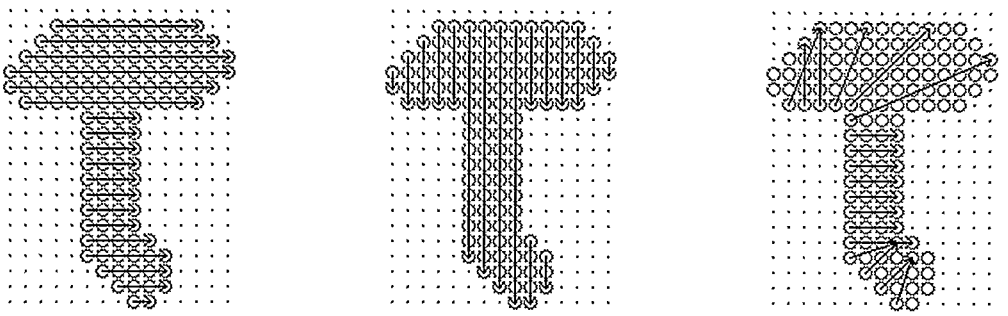
---

[2] This dataset is considered "easy" due to the very large, clear writing and flawless segmentation, so that is why so few templates per class are used.

4.4.2.2 Model 2 – Assume all lines exhibit uniform and identical stroke width

Under this model, we can compute an estimate of the global line thickness in an image. This is a reasonable model for most hand-drawn images. Given this assumption, we can compute this estimate in a number of ways. We will mention two broad classes of algorithms (i) sampling the stroke width directly by slicing and (ii) analytically from the measured perimeter and area.

There are two ways to perform slicing to sample the stroke width of the image. One way is to perform slices in the horizontal and vertical directions only. Another is to form *oriented* slices: we will perform slicing perpendicular to the tangents at the image boundary. We illustrate the difference in Figure 39.



(a) (b) (c)

Figure 39 Examples of Slicing

In (a) and (b) of Figure 39, we see examples of how to slice an image with horizontal and vertical slices. In (c) of this same image, we perform oriented slicing. At each boundary point in a direction perpendicular to the boundary tangent (based on a 3x3 neighbourhood). The lengths of these slices constitute a sample of the overall stroke width in the image, so we can look at properties of this distribution to determine our estimate. Hence, the average uniform stroke width can be estimated using the distributions of slice lengths from (a) and (b), or slice lengths from (c). A central statistical measure, such as the mean, will be sensitive to some of the longer slices, so taking the median often yields more accurate results.

The horizontal and vertical slicing technique works well if all of the strokes in the image are horizontal or vertical, (since then it is really equivalent to tangential slicing at the boundaries) but diagonal lines can cause an error as high as $\sqrt{2}$ times the correct width. Consider Figure 40, where we zoom in to a diagonal stroke in an image. Suppose would we like to estimate the uniform stroke width at a point x. For this point we take three slices, one horizontally, vertically,

and tangentially. The horizontal and vertical slices *xa* and *xb* are much longer than the true stroke width sample *xc*. This figure illustrates that to get an accurate stroke width estimate, oriented slicing is to be preferred over crude horizontal and vertical slicing.

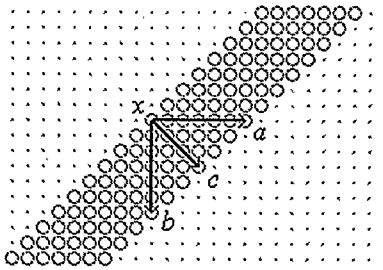


Figure 40 Width Estimation Errors From Slicing

In addition to slicing, there are some analytical techniques that base the answer upon the measured perimeter and area of the image. Since we assume the image is made up of stroke data, we can assume that if the image is "stretched out" while maintaining the same values of perimeter and area, it would appear as one large rectangle. Under this model, we proceed to solve the two equations in two unknowns.

$$A = lw \qquad \textbf{(Eq 36)}$$
$$P = 2l + 2w$$

The following method appears in [Hew1998], and we have made it slightly more accurate. Assuming again that the image perimeter and area are essentially that of a big rectangle, and that this rectangle is a long stroke of length l, we can assume that the perimeter is closely approximated by $P = 2l$. Using these values, the width is determined via $A = lw$, yielding an estimate of

$$w = \frac{2A}{P} \qquad \textbf{(Eq 37)}$$

We call the estimate of (Eq 37) the *linear predictor*. This approach works well if the width w is much smaller than the length l. Although the above estimate gives reasonable results, it is counter-intuitive to neglect the width contribution in one equation and then estimate it from another. Instead, we can estimate the width analytically by solving a quadratic formula:

$$l = A/w$$
$$P = 2\cdot(A/w) + 2\cdot w$$
$$\Rightarrow 2w^2 - Pw + 2A = 0$$
$$\Rightarrow w = \frac{P - \sqrt{P^2 - 16A}}{4} \quad \textbf{(Eq 38)}$$

We call the estimate in (Eq 38) the *quadratic predictor.* We present some images that show the vector template from Figure 38 (a), plotted with various uniform stroke width estimation techniques, estimating the width from Figure 38 (b). In Figure 41, we see estimates of the width using the following algorithms (a) horizontal and vertical slicing, (b) tangential slicing, (c) the linear predictor, in (d), the quadratic predictor.

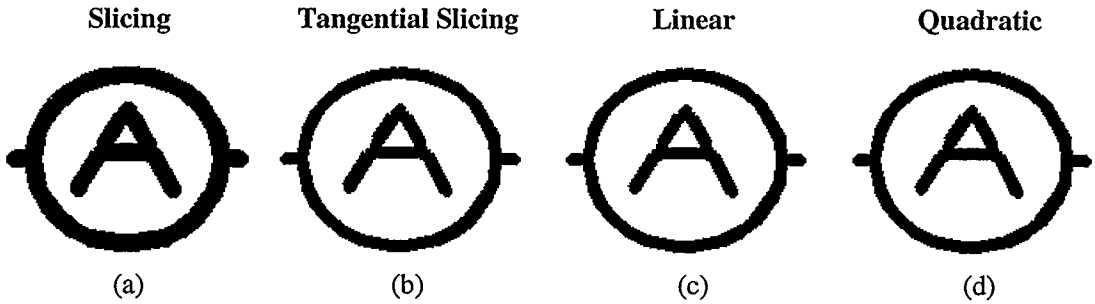


Figure 41 Visually Comparing Algorithms for Uniform Stroke Width Estimation

Slicing	Tangential Slicing	Linear	Quadratic
$w = 11.00$	$w = 7.62$	$w = 7.50$	$w = 7.58$

Table 1 Quantitatively Comparing Algorithms for Uniform Stroke Width Estimation

This concludes our survey of uniform stroke width estimation. First, we have shown how to estimate the uniform stroke width using information from the set of slice length samples. Second, we have shown, under the assumption that the image represents one large rectangle, how to derive a width estimate from formulas for area and perimeter.

4.4.2.3 Model 3 – Model the Uniform Line Width of Each Line Independently

Now we turn our attention to a more localized width estimation technique. Since we are looking at this from the perspective of vector templates, we would like a line estimate for each line segment in the template. It is true that even a line could exhibit non-uniform line thickness, but for now we restrict ourselves to this assumption. Thus, we model each line in a vector template as having an independent line width, and come up with an estimate for this value.

To solve this problem, we provide one technique that borrows from the slicing ideas of the previous section. For each line in the vector template, we perform perpendicular slices to the line, and measure local width estimates along this line. Again, this set of slices gives a reasonable estimate of the stroke width of the image in this section, provided the line overlaps with a part of the image. If it does not, we use the uniform line width estimate for the image when plotting this line. Figure 42 shows an example of this technique.

Figure 42 Localized Width Estimation

Although most of the time it appears the localized estimates performed quite well, in some cases a gross overestimate for the line width has been found. One way to solve this problem would be to produce a "width map" for the entire image, stating for each location what the estimate of the line width is. This function, really a 2D surface, could then be made smooth, with perhaps a bound enforced that no portion of the image be considered to have width greater than twice the uniform stroke width.

Before concluding this section, it may be interesting to see which style of drawing produced the template that is most similar to the original image. The comparisons in Table 2 show these results for the example image and template of Section 4.4.2.

Width Estimation Algorithm	Inkwell Hausdorff Distance $H(A,B)$	Image A	Image B
Slicing	$d = 11.05$	Figure 41(a)	Figure 38 (b)
Tangential Slicing	$d = 14.24$	Figure 41(b)	Figure 38 (b)
Linear	$d = 14.52$	Figure 41(c)	Figure 38 (b)
Quadratic	$d = 14.43$	Figure 41(d)	Figure 38 (b)
Localized Estimates	$d = 7.11$	Figure 42	Figure 38 (b)

Table 2 Inkwell Hausdorff Between Different Renditions of a Template, based on line thickness.

Thus the localized width estimates offer some improvement in reducing the distance between the image and template. More work would be necessary to better understand this relationship, especially when the template and the image do not represent the same symbol.

This completes the section on localized width estimation, and thus the entire section on line thickness in images. We now move to discussing a few other problems encountered with vector templates. The first of these describes how vector templates may be *too* elastic.

4.4.3 The "Ones" Problem

When a vector template is plotted over top of an image, it is scaled to fit the dimensions (bounding box) of that image. If the image has a reasonable width and height, this strategy is fine. However, for very skinny objects like the digit '1', all templates would be scaled to fit into a tall narrow rectangle. If the line is only one pixel wide in extreme cases, all templates also scale to a one pixel wide line, and thus all templates look the same. In this sense, vector templates may be viewed as *too* elastic. To solve this problem, the notion of a *virtual extent* for each image is introduced. The bounding box of it is measured, and the aspect ratio, $\max\{w/l, l/w\}$, is calculated. If this value is too large or too small, then some whitespace is added to the image. This *virtual extent*, rather than the bounding box, is used to describe the size of the image, so that vector templates can be sized appropriately. For handprinted digits, the $\max\{w/l, l/w\}$ should be no larger than about 2.5. Thus a '1' pattern is centered in a slightly wider box, and scaling is always done relative to this. The same problem comes up when we have very skinny (degenerate) templates, and so vector templates are treated the same way. A bounding box for a vector template is computed, and if it is too thin, the virtual extent for that pattern is made larger. This prevents patterns from being stretched to degenerate proportions in either direction.

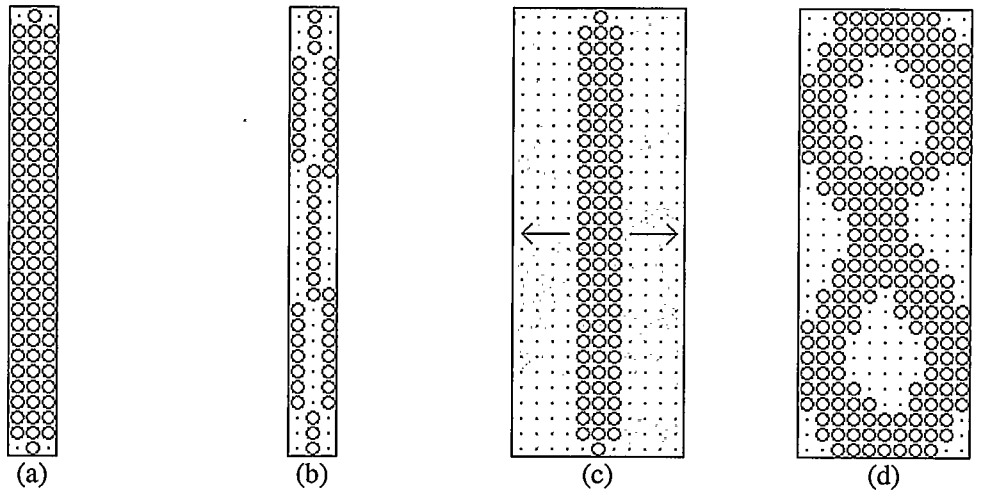


Figure 43 The "One's" Problem.

Figure 43 illustrates the "one's" problem. In (a), we have an image of a '1', which is very narrow. In (b), we see a vector template of an '8' plotted into the bounding box of the '1'. The bounding box is too narrow to see the '8' properly. In fact, this image would look even less recognizable if we had plotted the 8 with thick lines to match the measured stroke-width of the original '1'. For purposes of clarity, we have in fact rendered it with thin lines only. In (c), we see the same '1' with a virtual bounding box, enforcing a maximum aspect ratio of 2.5. The image is widened so that the height is not more than 2.5 the image width. When the template of the '8' is drawn in this more spacious bounding box, more of it's natural shape comes through. As a result, the distance values are quite large when comparing these images, i.e. (c) and (d), which is what we would like. The images in (a) and (b) generate an extremely small distance value from their comparison.

4.5 Vector Templates are Deformable

Now that we have seen how vector templates are an improvement over rigid bitmaps via their elastic nature, let us discuss how they are really deformable templates. This is achieved by the use of the Inkwell Hausdorff distance of Chapter 3. This fast (linear-time) metric effectively computes the cost of deforming one image directly into the other. Before discussing how vector templates are deformable, let us review two modern approaches to deformable templates.

In an attempt to improve the performance of template matching algorithms on handwritten symbol recognition, many authors have come up with ways to make templates less rigid. Templates can be rigid, globally deformable, or locally deformable. In [Jain1997], we see the

presentation of a locally deformable template that can bend itself to fit the contours of the unclassified image. The idea is to model templates in a flexible way, so that slight defects in a symbol could be accounted for. When measuring distance between a template and an image, two terms contribute to the overall distance penalty. One term explains how well the image of the deformed template (under the said deformation) and the target image match up. The other term assigns a cost to the magnitude of the deformation. As these two terms are combined via a weighted average, it is possible to adjust the "rubberiness" of the templates. The recognition rates obtained with this technique were in excess of 99% on the MNIST database, showing that deformable templates have a lot of promise. However, the running time for them is quite high. This is because the search for the optimal deformation, seems a bit unguided, and almost 'blind'. A lot of time is wasted looking at unlikely deformations. Their excessive running time makes them unsuitable 'for use in commercial systems' [Jain1997].

Recently, a new technique for template-based symbol recognition was introduced, showing how to achieve a deformable template via weighted bipartite matching [Belongie1999]. One key innovation is the idea of a 'shape context', a rich local descriptor of the image's shape, defined at each point in the image. The matching process is guided not by distance between points per se, but by how well the respective shape contexts of two points agrees. A recognition rate of 99.93% was published on the benchmark MNIST database. One of the obstacles encountered with this approach is that bipartite matching can be slow in the worst case. The running time to solve the weighted bipartite matching problem is $O(n^3)$. The fast algorithm in [Jonker1987] still has an asymptotic worst case running time of $O(n^3)$, but the performance it boasts is due to its expected running time. Although this was not clear from the text, it seems on the order of $O(n^2)$, or $O(n^2 \log n)$.[3] However, the running time for computing this matching is still quite cumbersome.

Vector templates are an innovation over deformable templates as they do not *search* for which deformation would take the template image into the target image. Since we *know* the desired end result – why not go there directly? Thus, we directly compute the optimal deformation from the template image to the target image, thereby saving an enormous amount of computation time looking for the "best" deformation. The magnitude of this deformation describes the distance to

---

[3] It is not uncommon for an algorithm to behave like this. 'Quicksort' has a worst case running time of $O(n^2)$, but runs so fast on average that it is competitive with or better than most $O(n \log n)$ algorithms in practice.

the template. Again, since the deformation calculation is carried out via the Inkwell Hausdorff distance, the magnitude means the sum of pixel displacements in a smooth morphing of one image into the other (see Figure 44).

(a) (b) (c) (d) (e) (f) (g)

Figure 44 Vector Templates as Deformable Templates: '3' to an '8'

In Figure 44 Vector Templates as Deformable Templates: '3' to an '8' we see the deformation calculation in progress. We measure how much work is required to transform, in this case, a '3' into an '8', by observing the migration of the individual pixels.

As another example, notice how we could recognize a 'pathological' instance of a '2' using a fairly regular template (Figure 45). Many traditional symbol recognizers would have a tough time with this case, especially those sensitive to boundary and connectivity information.

(a) (b) (c) (d) (e) (f) (g)

Figure 45 Similarity between a pathological '2' and a template of a 2 identified due to inexpensive deformation cost.

Really, how are these templates different from the two techniques mentioned briefly at the start of this section? Well, primarily in two ways: we model deformations more simply and we disregard the term describing the effect that the deformation induces on the plane.

We model a deformation in a way that allows us to efficiently determine the optimal deformation from one image to another. This is not so easy in Jain's deformation model. Our notion of a deformation is really a discrete vector field of an image. For each pixel location in the image, we consider a vector that translates this point to a new arbitrary location. Over the space of all such deformations, we efficiently find the vector field that minimizes the total length of all the displacement vectors. This is computed through the use of two distance transforms, and

allows us to efficiently determine how much effort is required to transform one image into another.

Secondly, and this is a potential weakness of our approach, we do not consider the effect that the deformation has on the plane. By disregarding this important term, we throw away potentially useful information. The vector field that is created is a byproduct of the Inkwell Hausdorff distance calculation, meaning that vector field records, for all points in A and B, the displacement they must undergo to find their corresponding nearest neighbour in the other image. This vector field between two images is defined in (Eq 39).

$$V_{A,B}(p) = \begin{cases} f_{ij} - p, & \text{if } p \in A, p \notin B \\ g_{ij} - p, & \text{if } p \notin A, p \in B \\ (0,0), & \text{otherwise} \end{cases} \quad \textbf{(Eq 39)}$$

where

f_{ij} is the feature transform for image A
g_{ij} is the feature transform for image B
$p = (i,j)$ is an arbitrary location in the image

This vector field captures the amount of work that has to be done to transform one image into the other. Notice that the Inkwell Hausdorff distance could be calculated directly from this vector field; it is the sum of the magnitudes of all the vectors (Eq 40).

$$H(A,B) = \sum_{p \in A \cup B} \left| V_{A,B}(p) \right| \quad \textbf{(Eq 40)}$$

Similarly, the regular Hausdorff distance could be determined from this vector field (Eq 41); it is just the maximum of the vector magnitudes.

$$h(A,B) = \max_{p \in A \cup B} \left| V_{A,B}(p) \right| \quad \textbf{(Eq 41)}$$

A sample vector field is shown in Figure 46.

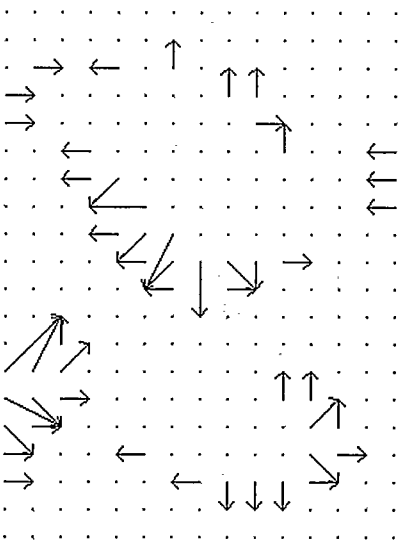


Figure 46 Vector field produced from a Hausdorff distance calculation between two images of an '8'.

We can imagine what effect this vector field induces on the plane. Whenever all the vectors in a small region point the same way, the effect on the plane is a smooth deformation, much like a bending or twisting. However, when vectors are pulling in opposite directions, like in the bottom left corner of Figure 46, this could be likened to a tearing or ripping of the plane. By disregarding these aspects of the deformation, we are potentially throwing away some useful information about the image similarity, i.e. if two images are similar, they should not induce these kinds of abrupt deformations to the plane. To model this term, we could look at each pixel location, and compare the displacement vectors in a gaussian window. A penalty term would be accumulated, and it could be proportional to the difference in angle of the displacement vectors as well as their magnitude. So, for example, two vectors of large magnitude pulling in opposite directions, would contribute a substantial penalty term to the overall cost of the deformation.

To see the effects this vector field induces on the plane, we have created a mesh where adjacent pixels in the image are connected by straight line segments. We translate each pixel according to the total force exerted on it by the vector field. The total force exerted on it is a weighted sum of all the vectors within a small radius.

$$F_{A,B}(p) = \frac{\sum_{q \in N(p)} w_q V_{A,B}(q)}{\sum w_q} \qquad \textbf{(Eq 42)}$$

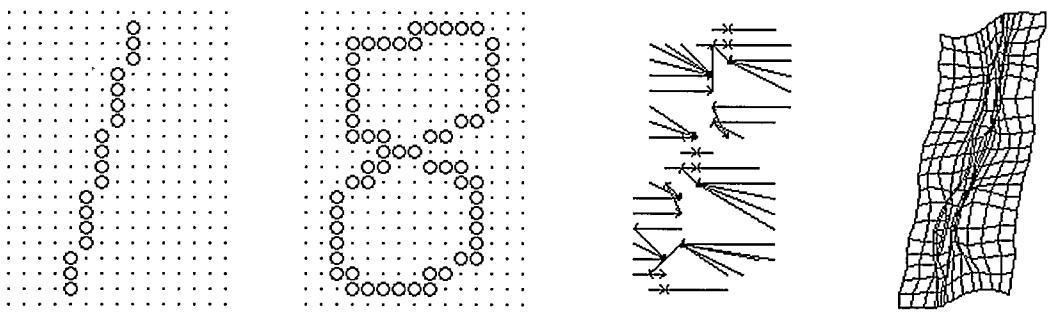
Where

$N(p) = \{q; |q - p| \leq r\}$, is the neighbourhood of p.

r is the radius of the neighbourhood

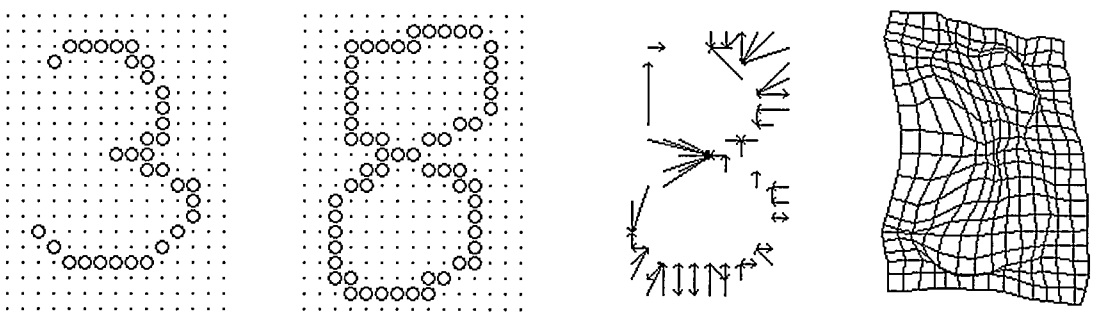
$w_q = 1/|p - q|^2$ is the weight of the vector at point q

The images in Figure 47, Figure 48, and Figure 49 illustrate these concepts, and show how the plane is affected using this deformation model. They are provided for clarity purposes only and were not used in the distance calculation. We mention them here since they help explain how distance is being measured, and they explain how vector templates are really deformable templates. In each of these figures (a) corresponds to image A, (b) corresponds to image B, (c) shows the vector field $V_{A,B}$, and (d) shows the mesh under the force function $F_{A,B}$.



(a) (b) (c) (d)

Figure 47 Deforming a '1' to an '8'.



(a) (b) (c) (d)

Figure 48 Deforming a '3' to an '8'

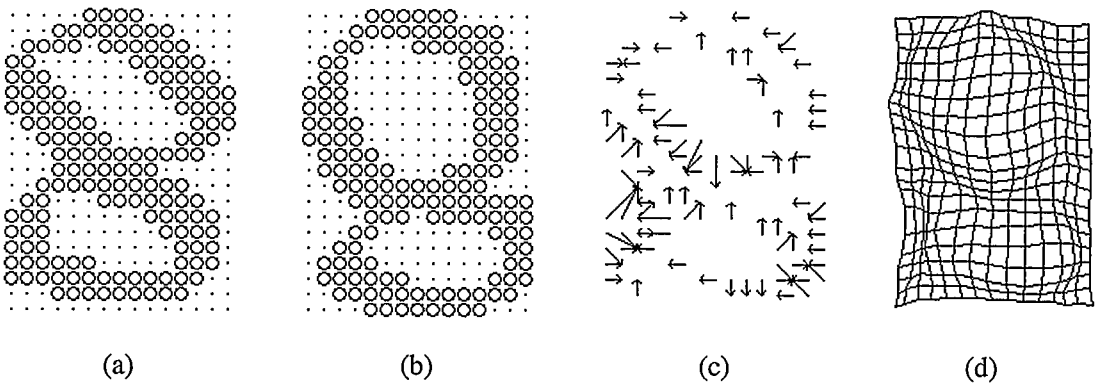


Figure 49 Two similar shapes, deforming an '8' to an '8'

In the final three chapters, we discuss design considerations of the vector template library that we developed from scratch, results from symbol recognition experiments, and finally some concluding remarks.

Chapter 5
System Design

This section describes some design decisions made while implementing the vector template library and the applications that use it. The system was designed in an object oriented fashion in C++, using Microsoft Developer Studio v6.0.

5.1 Modeling Vector Templates

It is important to model vector templates appropriately so they can be modified without affecting the rest of the system. An interface, called `IVectorTemplate`, was designed to expose only the essential properties of a vector template: the set of line segments that it constitutes, as well as identification of the class it models. To model affine transformations such as rotation, scaling, and shearing, special objects are created and treated as *decorators* on the `IVectorTemplate` interface. The entire system treats all modified templates through the common interface `IVectorTemplate` and doesn't need to see what is going on behind the scenes in terms of how the vector templates are customized via linear transformations, non-linear transformations, and others.

We now explain this interface and present some classes that implement it. But first, a brief word about naming conventions is in order. A class that provides no real implementation, and serves just like an interface (in the Java sense of the word), or as an abstract base class (in the C++ sense), is named with an "I" as a prefix. This includes interface that already start with an "I". For example the interface that represents an image metric is called an `IImageMetric`.

5.1.1 Interface: IVectorTemplate

```
Interface IVectorTemplate

IPointIterator *createPointIter() = 0;
ILineIterator *createLineIter() = 0;
Pattern *getPattern() = 0;
```

Figure 50 IVectorTemplate interface

This interface represents the essential concepts of a vector template. A vector template is really a set of line segments and represents a model of some *class*. The `createPointIter()`

function creates an iterator over all the points in the vector template, one for each endpoint of each line segment, and the `createLineIter()` produces a view of the vector template as a set of lines. The `getPattern()` function returns the identity information of the vector template, and answers questions like which symbol class it represents.

5.1.2 Classes that Implement IVectorTemplate

`class VTImpl`	A `VTImpl` stands for a 'vector template implementation'. It provides functionality to read and write templates to a file, and owns the point data for the template. The (x,y) points are in that template's native coordinate system, whatever that may be; it is not normalized for size, translation or any other quantity.
`class SimpleVT`	A `SimpleVT` is a minimal implementation of the `IVectorTemplate` interface; thus a `SimpleVT` is an `IVectorTemplate`. It contains an instance of `VTImpl`, which contains all of the vector template data.
`class TransVT`	A `TransVT` is an `IVectorTemplate`, but represents a kind of `IVectorTemplate` that has had a linear transformation applied to it. It is synthesized from an existing `IVectorTemplate` and a linear transformation object. Without having to modify the original `IVectorTemplate` object, various transformations can be applied, either all at once with a single linear transformation object, or by composing independent instances of `TransVT` objects. This is done behind the scenes by having it create smart iterators that apply these transformations 'on-the-fly', as they are requested.
`class DeformVT`	The `DeformVT` class models vector templates that have had some non-linear deformations applied to them. Since it is an `IVectorTemplate` it must still represent straight lines. Thus these deformations are applied to the endpoints of the vector template lines only, preserving the 'straightness' of the lines. A `DeformVT` object is created from an existing `IVectorTemplate` and an object that describes the non-uniform transformation.

This design, specifically factoring out the rather simple `IVectorTemplate` interface, makes many sections very easy to code, since any 'flavour' of a vector template could be treated

uniformly. This reduces coupling between many sections since they apply their own transformations to the vector templates. For example, it is useful to be able to apply non-linear transformations to "deform" a template, and then still be able to stretch the template to fit inside a certain area. This is accomplished easily, and the drawing routines for templates do not have to change at all.

To have this system work, appropriate interfaces for an `IPointIterator` and for an `ILineIterator` are defined. Subclasses are created when an iterator needs more information, like the transformation that needs to be applied to all the individual points or lines.

The above design is really a disguised version of the 'Factory' pattern from the Gang of Four [Gamma1995]. An `IVectorTemplate` interface really represents a factory that can produce two types of products, `IPointIterator` objects and `ILineIterator` objects. Concrete factories are defined (i.e. `TransVT`, `DeformVT`) that build concrete products, (i.e. the specialized line and point iterators). All of the products are of the same family, so one factory can be safely replaced by another. The new products, though different, behave just like the old ones.

5.2 Comparing Images and Templates

It is important to have a good design of classes that compare images and templates because this operation is so common. Two broad concepts are distinguished. We need to compare images with other images (`IImageMetric`) and we also need to compare images to templates (`ITemplateImageMetric`). It is important to model these separately. To make this distinction more clear, a function measures distance between a template and an image if it relies on some aspect of template representation. A template metric will often make use of an image metric.

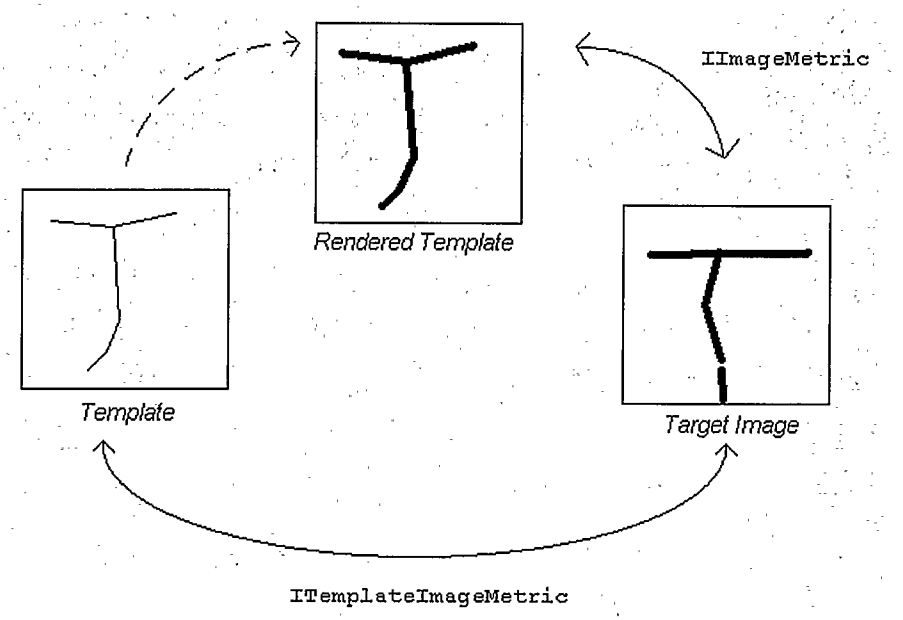


Figure 51 Differences between the IImageMetric and ITemplateImageMetric

5.2.1 Interface: IImageMetric

An `IImageMetric` interface represents the concept of a function that compares two images. If it models a function, why is it a class? Modeling functions with classes is not a new idea and it is often a good idea if one function needs to be easily replaced by another. Also, in the case that one function can be parameterized by many different settings, it is convenient to store these as state variables of a class and not pass them to every function call. This avoids bulky parameter lists and hence confusion. An additional benefit of this design pattern is that if a function needs to have 'memory', it can do so. This can be done with static variables in the C programming language, but this raises other issues. For example, you cannot have two instances of the function with each having its own independent memory.

Objects of the `IImageMetric` class need a memory for caching. It is often the case that a single image will be compared against a set of other images. One can often benefit from computing certain static features of the single image and caching them to avoid recomputing them in each comparison. For example, pre-computing a distance map can be a time-saving operation. Thus this interface exposes two different sets of functions for image comparison (see Figure 52).

```
Interface IImageMetric

// Scenario 1: unoptimized
 virtual double dist(BilevelImage &A, BilevelImage &B) = 0;

 // Scenario 2: optimized for 1-many comparisons
 virtual void setCache(BilevelImage &A) = 0;
 virtual double fastDist(BilevelImage &B) = 0;
 virtual void clearCache() = 0;
```

Figure 52 The IImageMetric Interface

The first scenario has the easier syntax, and allows a direct comparison of two images. Of course, no caching is performed in this case. In the second set (three functions), a user first identifies the image that will remain constant in future comparisons, then calls the `fastDist()` function for arbitrary images, and finally clears the cache. An `IImageMetric` object only compares binary images that are of the same size.

There are many descendants from this class that allow for different types of image comparisons. The most simple of these is `ImageMetricXOR` which performs a binary correlation. It superimposes the two images and counts how many pixels overlap. Next, there is a family of classes based on the Hausdorff distance. The classes `ImageMetricND8`, `ImageMetricND8_2`, `ImageMetricSND8`, `ImageMetricSND8_2` comprise the complete list. These functions all look at the sum of minimum distances to neighbouring pixels in the corresponding image, but change the way they define the distance measure between pixels. See Figure 53 for a summary.

`ImageMetricND8`	implements $\vec{H}(A,B)$ using L_∞.
`ImageMetricND8_2`	implements $\vec{H}(A,B)$ using L_∞^2.
`ImageMetricSND8`	implements $H(A,B)$ using L_∞.
`ImageMetricSND8_2`	implements $H(A,B)$ using L_∞^2.

Figure 53 Image Metrics in the software.

5.2.2 Interface: ITemplateImageMetric

Another interface, called `ITemplateImageMetric` represents the concept of a function that compares a static image and a dynamic template. This class might add too much

unnecessary complexity if it only plots the template on an empty canvas and then delegates the rest of the work to an `IImageMetric` object. But, even this is still useful, and the `GenericTIM` embodies this task. However, we often need to do quite a bit more when comparing an image and a template. We would like to consider different transformations of the template, and this is what the descendants of the `ITemplateImageMetric` class are responsible for. They try many different 'versions' of a template, and report only the distance to the best one.

An `ITemplateImageMetric` may also need to cache information about the image it is comparing. Often the same instance is used to compare a single image against many templates. In these cases, all features of the image itself can be precomputed to save time. The syntax for evaluating the distances with and without caching is very similar to that of the `IImageMetric` interface.

`class GATProcess` The `GATProcess` (standing for Global Affine Transformation) class handles global transformations to the template, such as rotation, shearing and scaling. When measuring distance, it can be customized with the range of rotation angles to try, and the angle increment that is used. It can be told whether or not to use shearing, and to what size to scale the vector template. It will try all parameters in this range on a given template, and give back the distance to the best transformed template. Again, for each transformation, the template is plotted onto a canvas, and the actual comparison is delegated to an `IImageMetric`. There is no cost attached to any of these transformations.

`class DefProcess` In the case of deformable templates, we need to comb a large parameter space to find an optimal deformation of a given template. Basis vector coefficients are sought so the resulting deformation minimizes the distance between the template and the image. Many image to image comparisons are needed during this search. Thus, a `DefProcess` object delegates this work to an `IImageMetric` object.

The suffix 'Process' is given to these metrics since we want to emphasize that the set of transformations of each template are examined.

5.3 Modeling Images

Images are modeled as 2D arrays of bytes (in the range 0 to 255). A common implementation is created in a class called `Image`, and then three classes are derived from this. The `BilevelImage` class is mainly for binary images, and primarily the values 0 and 1 are used. Other values are occasionally used to mark special pixels. The `GreyScaleImage` class models a grey-scale image, and can use the default implementation more or less as-is. Finally, the `ColorImage` class models color images. Unfortunately, 0-255 doesn't give much room for color information, so these are indicies into a 256-entry colormap with 24-bit color values, much like a GIF representation.

5.4 Image Transforms

Since distance transforms have become such an important part of the efficient computation of the Inkwell Hausdorff distance, we spent some time modeling these classes as well. A *feature transform* of an image is a matrix of point data, stating for each image location *where* the nearest foreground pixel is. Similarly, a *distance transform* is a matrix of scalars, and records, for each image location, the *distance to* the nearest foreground pixel. We have created a separate class for each of these, called `FeatureTransform` and `DistanceTransform` and then another class called the `EntireTransform` which is just a data structure housing both the distance transform and the feature transform.

We have created classes for computing the distance transform using any one of L_1, L_2, and L_∞ pixel metrics. These classes all implement a common interface called `IFTAlgorithm`, which represents an algorithm to build either a distance transform, a feature transform, or both. The added function to return both at once is useful because sometimes one type of transform is necessary to build the other. For example, building a distance transform for the L_2 metric involves building a feature transform anyway. Thus, to retain system efficiency, we provide a way to get both transforms at once.

```
Interface IFTTransform

// Methods
virtual DistanceTransform *createDXform(BilevelImage &img)=0;
virtual FeatureTransform *createFXform(BilevelImage &img)=0;
virtual EntireTransform *createEXform(BilevelImage &img)=0;
```

Figure 54 Interface for an IFTAlgorithm

The `FTAlgorithm_L1`, `FTAlgorithm_L2`, and `FTAlgorithm_L8` implement the `IFTTransform` interface, for the pixel metrics L_1, L_2, and L_∞, respectively. The algorithm ideas for `FTAlgorithm_L1` and `FTAlgorithm_L8` were taken from J.R. Parker [Parker1991] and the algorithm for `FTAlgorithm_L2` was taken from M. Gavrilova [Gavrilova2000]. All transforms can be computed in linear time for an image with n pixels, and require only two passes through the image. The Euclidean Distance transform is somewhat slower however, since it relies on floating point calculations, whereby the other transforms rely solely on integers.

5.5 Interactive Symbol Recognizer for Windows 98

We performed an interactive demo on symbol recognition at ICCV 2001 in Vancouver [Parker2001b]. The vector template code was used to build a system that allowed user input of arbitrary symbols with a pointing device: either a mouse or a WACOM Tablet, which is a touch sensitive pad with a stylus, and is a more natural interface for capturing handwritten symbols. After a symbol is drawn, the system cycles through all of the templates in its library, searching for the best match. It shows the current template in the top right corner, while the bottom row displays the top 3 templates in its search so far. It is possible to adjust the range of rotation angles and transformations in general that are applied to each template (see Figure 56). The line thickness can be set accordingly as well, to use either skinny templates, templates with uniformly thick lines, as well as variable width lines. It is also possible to try different techniques for measuring distance, and tune the parameters for them. These include distance using the LCS algorithm, as well as using bipartite matching, and of course the whole host of Hausdorff distance functions.

The demo application can also be used as a browser of template libraries. It shows a screen with the entire template library, as well as individualized screens for each template, emphasizing their underlying line segments.

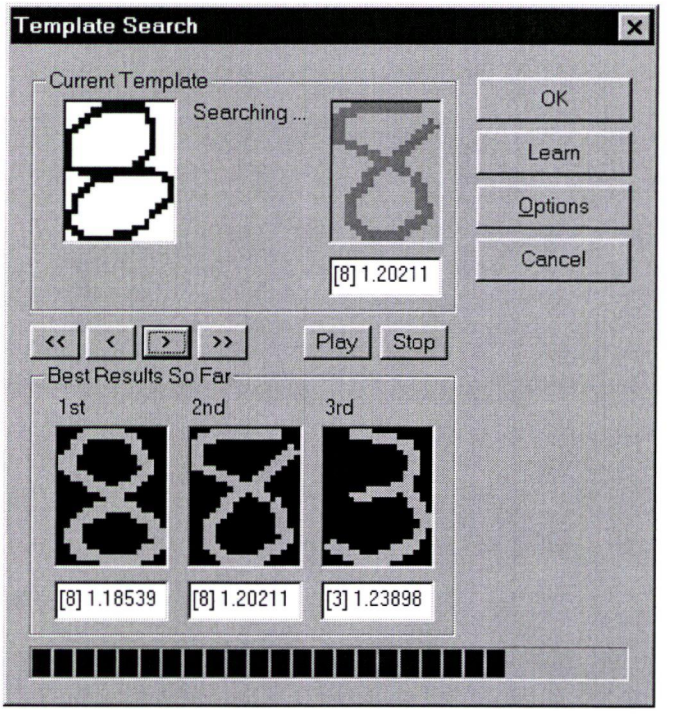


Figure 55 Screen Shot of the Interactive Symbol Recognizer

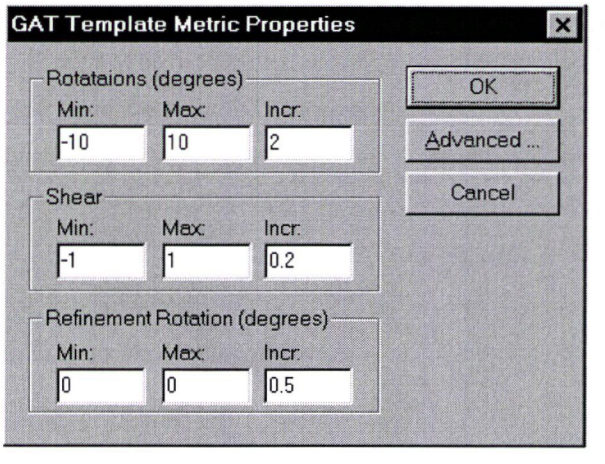


Figure 56 Dialog to Change Template Paremeters

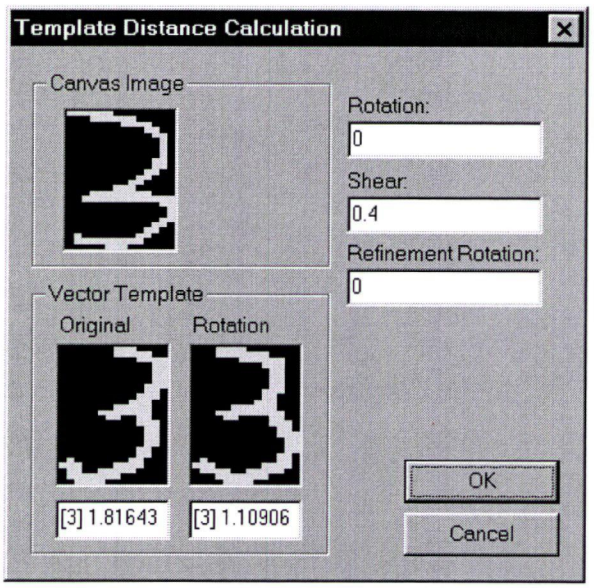


Figure 57 Transformation of Template improves Distance Calculation

In Figure 57, we see that applying linear transformations to the template can improve the distance value. In the two images at the bottom the text "[3] 1.81643" indicates the template represents a symbol class of '3', and the distance value was 1.81643. This distance goes down considerably as a shearing operation is performed on the template.

5.6 Conclusions

We have implemented an object oriented vision library for symbol recognition systems using vector templates. Some foresight went into making the separate algorithms interchangeable through the use of the 'Strategy' pattern [Gamma1995]. This frees us to use different image metrics or template metrics in the system, thus reducing the dependency of the overall technique upon any one specific algorithm choice and/or implementation. This facilitates making comparisons of individual components, as well as providing a fertile test bed in which to more readily test out new ideas.

Chapter 6
Results and Experiments

6.1 Overview of Chapter

In order to test out the general applicability of vector templates, we used them in many symbol recognition experiments. Experiments were performed on hand-written digit databases, as well as on machine-printed electrical engineering symbols, and symbols from a chess openings book. This chapter is organized into 3 main sections, one for each dataset.

The first dataset consists of isolated hand-drawn digits. We had access to four different databases, some publicly available and others purchased privately. A brief description of each set is provided along with some sample images that show the quality of the data. In this section on digits, we also explore different parameters of the Hausdorff distance function for comparing images, and provide evidence as to why the final function chosen works the best. We discuss the different image alignment techniques that were tried, and how they compared against each other. We present our recognition rates on the four aforementioned databases, and where possible, compare them to the best published results to date. Finally we describe the template selection strategies that were tested, and which one was used for our final published results.

The second dataset consists of noisy, machine drawn symbols from electrical engineering. This dataset was provided as part of a symbol recognition contest for ICPR 2000 (International Conference on Pattern Recognition), in which our algorithm won first place [Ye1999], [Askoy2000], [Parker2000]. We examine the question of how to measure the line thickness of an image, and present and compare three algorithms. Noise removal strategies are also discussed.

In the final dataset, we consider an application of machine-drawn symbol recognition from a chess openings book. This has practical value as most opening books are available only in printed form. One can make use of a digital chess openings book in chess software, and in tuning static evaluation functions during the early stages of a chess game.

6.2 Handwritten-Digit Databases

We had access to four different handwritten digit databases. These consist of the MNIST database, the CENPARMI database, the USPS database, and the ETL database.[4] Each is organized quite differently; the images are stored in various formats and some datasets have an official partitioning into training data and test data, while others do not. We put them all under a common organizational umbrella to make conducting experiments and comparisons easier. All the datasets luckily fit onto a single CD, and we chose to store one image per file for easy random access. Although this is somewhat wasteful, as there are 100,000 files and each file header wastes some space, it makes the dataset very easy to access. Also, we chose the uncompressed file formats PBM and PGM for their simplicity in file I/O, as we felt this outweighed the storage penalties. The CD also has a short HTML documentation section that discusses the technical description of the datasets, and is included here.

	Number of Images		
Dataset	**Entire Database**	**Training Data**	**Testing Data**
MNIST	70000	60000	10000
SUEN	13982	TrainA 1578	12404
		TrainB 1610	12372
USPS	9298	7291	2007
JAPAN	2000	1200	800

Table 3 A Size Comparison of Handwritten Digit Databases

[4] We had named two of our datasets initially with informal names, and as a result some of them have two names. The CENPARMI dataset is informally called the "SUEN" dataset on the CD, and the ETL dataset is informally called the "JAPAN" dataset on the CD.

Technical Description of Database				
Dataset	**Data Type**	**File Format**	**Image Resolution**	**Pattern Resolution**
MNIST	Greyscale	*.PGM	28x28	approx 20x20
SUEN	Bilevel	*.PBM	approx 20x20	approx 12x15
USPS	Greyscale	*.PGM	16x16	approx 12x16
JAPAN	Bilevel	*.PBM	72x76	approx 36x38

Table 4 Description of Image Content in Databases

The CENPARMI Dataset

The CENPARMI dataset consisted of 13,982 isolated hand-printed digits. It has two training sets of approximately 2,000 digits each.

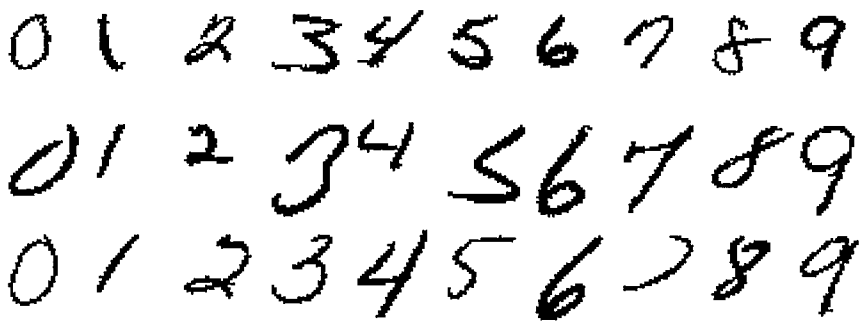


Figure 58 Sample Digits from the CENPARMI Dataset

Most of our effort was spent on this database, and for a number of reasons: (i) this dataset had characters that are quite difficult compared to the neat characters in the ETL dataset, (ii) its size is manageable compared to the large MNIST database, and finally (iii) the segmentation seems cleaner than for the USPS database, where occasionally fragments of other digits appear on some test cases. Also, this dataset is in black and white, which is easier to work with than greyscale, the format used in the MNIST and USPS datasets.

The CENPARMI dataset comes with two partitions of the data into training and testing data. Unfortunately, even though both official training sets have 2000 images, there are many duplicates in both of them. After removing duplicates, there remained 1578 and 1610 distinct images in '*traina*' and '*trainb*' training sets, respectively.

Most of our experiments were conducted using the CENPARMI dataset, using the *traina* training set with the duplicates removed.

The ETL Dataset

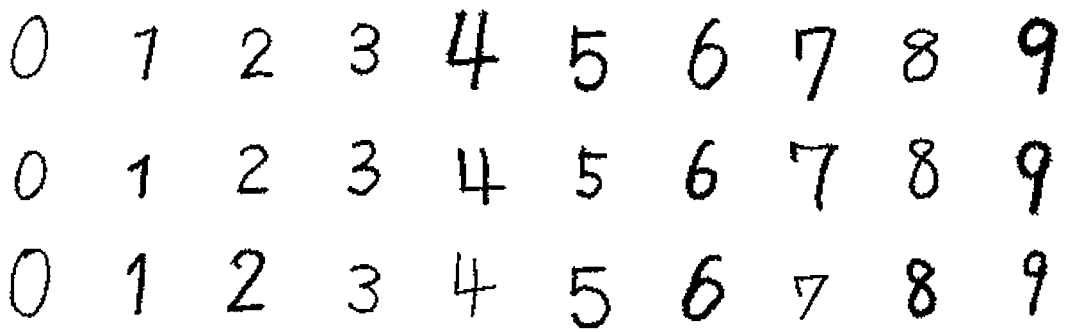


Figure 59 Sample Digits from the ETL Database

The ETL database was collected by the Japanese Technical Committee, and consists of digits drawn by Japanese students. The digits are subjectively very neat and consistent. The size of the dataset is quite small, consisting of only 2000 images in total.

The MNIST Dataset

Due to its large size of 70,000 images, this dataset is one of *the* standards for symbol recognition benchmarking. A standard such as this is necessary to provide meaningful comparisons between symbol recognition algorithms. Many different algorithms have been tried on this database, and their recognition rates have been tabulted by the dataset's maintainer, Yann LeCun of AT&T Labs-Research [LeCun1995]. The dataset contains digits from high school students and census employees in the United States.

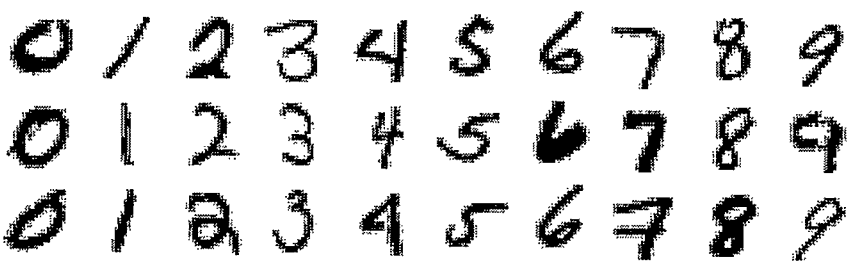


Figure 60 Sample Digits from the MNIST Dataset

The USPS Dataset

This dataset was collected from ZIP codes on United States Postal Code envelopes.

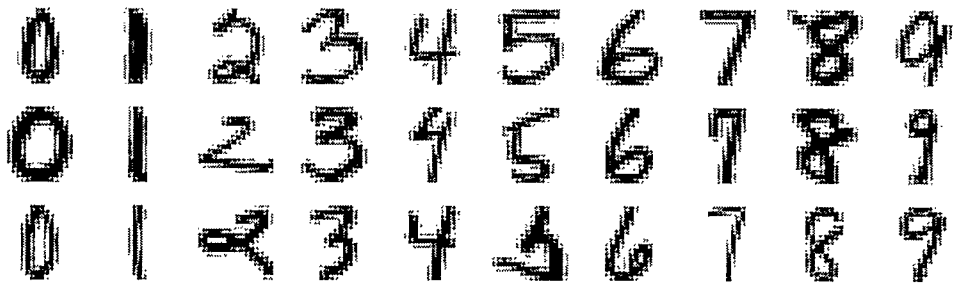


Figure 61 Sample Digits from the USPS Dataset

The dataset we have, of 9,000+ images, is actually only a (free) sampler of the true database, which is much larger.

6.2.1 Image Metric Comparisons

Before getting to the recognition rates we achieved on these datasets, we would like to discuss how we arrived at our choice for an image metric, the important function that compares two images. Due to its popularity and demonstrated success, we have focused on the Hausdorff distance and our improvements to it, namely the creation of the new Inkwell Hausdorff distance. In fact, we compared many different types of Hausdorff distances, as well as other distance metrics. To arrive at a fair comparison, we repeated the same symbol recognition experiment, using vector templates, while varying only the underlying distance metric.

There are many different ways to take advantage of the Hausdorff distance. Since we want to demonstrate an improvement over the traditional Hausdorff distance, we wanted to make sure to explore a good sample of the possibilities. Table 5 contains a review of the definitions for Hausdorff distance and a standard pixel metric.

Definitions of Hausdorff Distance, Pixel Norms	
Directed Hausdorff Distance	$\vec{h}(A,B) = \max_{a \in A} \min_{b \in B} d(a,b)$
Directed Inkwell Hausdorff Distance	$\vec{H}(A,B) = \sum_{a \in A} \min_{b \in B} d(a,b)$
Symmetrical Hausdorff Distance	$h(A,B) = \max(\vec{h}(A,B), \vec{h}(B,A))$
Symmetrical Inkwell Hausdorff Distance	$H(A,B) = \vec{H}(A,B) + \vec{H}(B,A)$
Pixel Norm	$L_n((x_1,y_1),(x_2,y_2)) = \sqrt[n]{(dx^n + dy^n)}$, where $dx = \lvert x_1 - x_2 \rvert$, and $dy = \lvert y_1 - y_2 \rvert$

Table 5 Review of Definitions of Hausdorff distance, Pixel metrics

First off, we computed the directed (one-way) Hausdorff distances and Inkwell Hausdorff distances. We want to compare how various underlying pixel metrics affect these distances. To perform the comparison, 200 random images from the *traina* training set of the SUEN dataset were chosen with 20 templates for each digit class. For test data, we selected 200 images from the testing dataset, again choosing 20 images for each digit class. We used vector templates with angles, in degrees, at {-10,-5,0,+5,+10} and shearing constants at {-0.4,-0.2,0,+0.2,0.4}. Thus, we performed a total of 200 images × 200 templates × 10 orientations/template = 400,000 image comparisons to arrive at a recognition rate for each combination of a Hausdorff distance variant with a pixel metric.

Recognition Rates for One Way Hausdorff Distance		
Pixel Metric	$\vec{h}(A,B)$	$\vec{H}(A,B)$
L_0	10.00%	59.00%
L_1	52.50%	62.50%
L_1^2	52.50%	60.50%
L_2	53.50%	61.00%
L_2^2	53.50%	60.50%
L_∞	45.00%	59.50%
L_∞^2	45.00%	59.50%

Table 6 One-way Hausdorff distance (left) compared with one-way Inkwell Hausdorff distance

The one-way Hausdorff distance performed quite poorly, as is evidenced in Table 6. This is mainly due to the fact that larger images are preferred by a one-way Hausdorff distance. We noticed that regions with a large area can be easier to match (at least in one direction). For example, consider images A and B where B has substantially more area. Then, when iterating over pixels in A to find their nearest neighbours in B, most distances will be small since B is large – it is like the difference between landing a plane on a runway and on an aircraft carrier. Matching one image to another image with a large area is like landing a plane on a spacious runway.

To compensate for this fact, we tried measuring distance using both the symmetrical Hausdorff distance and the symmetrical Inkwell Hausdorff distance, again under varying pixel metrics. By

measuring distance from A to B as well as B to A, we neutralize the one-way bias a large image can cause in the overall distance.

We conducted the same experiment as in Table 6, but using the symmetrical Hausdorff distances. Table 7 presents our results. In the first column, we see the performance of the (symmetrical or two-way) Hausdorff distance, and in the second column we see how the Inkwell Hausdorff distance outperforms it in every case. Recall that the metric L_0 corresponds to binary correlation, and counts a penalty for each foreground pixel that doesn't overlap with a corresponding foreground pixel in the other image. Thus it is expected to perform poorly, especially when used with $h(A,B)$, as all mismatched pixels are said to be distance one away from a nearest neighbour. It is included for the sake of completeness only. We would like to note as well that these recognition rates are not maximal. The experiments were run solely to better understand the role of pixel metrics in the Hausdorff distance, and to compare the Hausdorff distance with the Inkwell Hausdorff distance. Once we have acquired this understanding, we can focus on achieving higher recognition rates by tuning the operations we perform on vector templates.

The third column of this table shows what happens when we perform the Inkwell Hausdorff distance computation on the negatives of the images as well. The motivation to try this idea came from some special cases. If we are computing $H(A,B)$ and A is empty, it will have a small distance to *any* image B. To avoid this situation, we also tried inverting the images (complementing each bit) and combining these results with the Inkwell Hausdorff distance on the plain images. However, there is no noticeable improvement in the recognition rate offered by this extra term, so it was discarded.

Pixel Metric	$h(A,B)$	$H(A,B)$	$H(A,B)+H(\overline{A},\overline{B})$
L_0	10.00%	67.50%	67.50%
L_1	78.50%	90.50%	90.00%
L_1^2	78.50%	88.50%	88.50%
L_2	83.50%	89.50%	~H(A,B)[5]
L_2^2	83.50%	89.50%	~H(A,B)^2
L_∞	77.50%	89.50%	88.00%
L_∞^2	77.50%	89.50%	90.00%

Table 7 Hausdorff distance (left) compared with Inkwell Hausdorff distance (right).

It is interesting to note how much better the Inkwell Hausdorff distance performs over the regular Hausdorff distance.

In Table 8 we provide some comparisons of different variations of the Hausdorff distance that we tried, as we were zeroing in to the best fit of the Hausdorff distance. Also, we tried to run a more substantial experiment to ensure we had significant recognition rates. The table shows results from an experiment run with 150 templates, 15 for each digit, randomly selected from the *traina* subset of the CENPARMI dataset. With this template set, we recognized 1000 symbols chosen randomly from the test data. In addition to resizing all templates, each underwent a small set of linear transformations to aid in the search for the best fitting template.[6] The data shows the recognition rates for the given Inkwell Hausdorff function.

---

[5] The running time was too slow, so the run was aborted midway. However, the results appeared consistent with $H(A,B)$ when image negatives were not used.

[6] All templates were rotated from –10 degrees to 10 degrees at 2 degree increments. Shearing was also performed with a shear constant ranging from –0.4 to +0.4.

Inkwell Hausdorff Function	Recognition Rate	
	using L_∞	using L_∞^2
$\vec{H}(A,B)$	80.80%	80.35%
$\vec{H}(B,A)$	87.39%	83.90%
$H(A,B)$	88.59%	**90.24%**
$\min(\vec{H}(A,B),\vec{H}(B,A))$	82.07%	80.86%

Table 8 Combinations of the Hausdorff distance using different pixel metric functions

From this data, we have decided to use the (undirected) Inkwell Hausdorff distance in combination with the metric L_∞^2. From this point forward, $H(A,B)$ will mean this distance. We have not mentioned the LCS distance metric yet, and we have tested this metric in an experiment identical to that of Table 8. It's recognition rate was 88.50%, but this number was not significantly higher than the (more efficient) Inkwell Hausdorff distance functions. Even though it looks at quite different features of the images when comparing them, we could not find a good way to make use of this metric. More work would need to be done to possibly increase its efficiency or improve its rates.

6.2.2 Performance Comparisons on Handwritten Digit Databases

We now present our results on the four handwritten digit databases. Before proceeding however, a word on recognition rates is in order. What recognition rate are we trying to achieve? Well, one might think 100% but it turns out that even human experts would have a hard time achieving that. The reason is that we are given a set of digits, and their classes, but the class information was *determined by the person that drew the digit.* The best recognition rate we can get is dependent upon what the digit looks like, not what it was intended to be. We only have the latter information, and determine what it looks like on our own. The exact recognition rates for these databases (that is, as determined by a panel of human experts) is not known; as to our knowledge this experiment has not yet been done. We would estimate at least 1% to 2% of the digits could have controversial identities according to a small panel of people. Conservatively, on MNIST, at least 0.7% of the images are misclassified by the world's top programs.

We have conducted these experiments differently from the preliminary experiments to measure Hausdorff distance. To achieve as high a recognition rate as possible, we have increased the number of transformations of each template that we consider at each comparison. We now try a total of 18 different orientations of each template, (as opposed to about 10 for the Hausdorff distance tests). These orientations are made up of rotation angles, in degrees, of (-10, -8, -6, ..., 6, 8, 10) and of shear values (-0.6, -0.4, -0.2, ...,+0.2,+0.4,+0.6). A set of 200 templates was chosen from the training set in each of the databases, and then a small test set of 200 images was chosen from the test data.

Recognition Rates for various Databases				
Database	20 templates per class, alignment by bounding box, 200 test images	20 templates per class, alignment by centroid, 200 test images	40 templates per class, alignment by centroid, 600 test images	40 templates per class, alignment by centroid, 600 test images, LCS
MNIST	91.00%	91.50%	93.50%	92.83%
SUEN	91.50%	93.00%	94.50%[7]	94.83%
USPS	88.50%	88.00%	89.33%	90.67%
JAPAN	98.50%	98.50%	99.00%	98.67%

Table 9 Algorithm Performance across different databases

From the data in Table 9, we see that the algorithm is capable of extremely high recognition rates. True, the JAPAN dataset is perhaps easier than the others because (visually) the handwriting is very neat, the images are large, and very cleanly segmented, but the effectiveness of the technique is still demonstrated. The first column in the table shows some preliminary results when using a smaller dataset for both testing and training. Only 200 images were used for training, and only 200 images were used for testing. In the next column, we have demonstrated the effects of improved image alignment. In the first column, images were aligned by their bounding boxes only. In the second column, we have used the bounding boxes to guide us in how large to make the template, but then have aligned the images properly by their centroids. A slight improvement in recognition rates was observed in nearly all cases. From this point, we increased the size of the training and testing data, to 400 templates and 600 images for testing. These experiments ran for approximately 1 hour per dataset on a Celeron PC. Again, a slight improvement in the recognition rates was observed. After looking at the errors that were made,

[7] On a more substantial test, using the leave-one-out approach on 1578 symbols, we scored 96.39%

we thought of trying an optimization with another classifier. We used the Inkwell Hausdorff distance only to select the 5 best templates, and each template at it's best orientation. Looking at only these 5 images (of drawn templates), we asked the LCS Image Metric function to decide the closest match. Although this fresh perspective helped in some cases, it decreased the performance in others. Thus we ultimately left this metric out.

It seemed like the system under the Inkwell Hausdorff distance was making mistakes on pretty easy test cases, so we introduced a slight optimization, that of getting a second opinion. We use the LCS distance metric to re-sort the top 5 candidates as determined by the Inkwell Hausdorff distance. The Inkwell Hausdorff distance compares images only spatially and doesn't care about what structure an image has; are certain loops closed, do certain lines touch? The LCS metric, on the other hand, disregards the spatial similarity, and looks only at the structure of an image. It seemed like a reasonable idea to try: to have the Inkwell Hausdorff distance first find the best candidates for best matching template, according to spatial reasoning, and then have a second opinion from a source that doesn't look at spatial features. The attempt we made was somewhat ad-hoc but it does demonstrate (1) that the LCS metric and the Inkwell Hausdorff distance are looking at quite different features of the images and (2) that it is possible to combine them to achieve higher recognition rates. Further work would have to be done to combine them in an "optimal sense", and this falls more under the heading of strategies for combining multiple classifiers.

6.2.3 Errors

We now explain the types of errors that were made for each database. We have chosen four errors that represent the mistakes made in that database. For each error, we will show three images, to show not only the image that was misclassified, but also the best matching template. The first of these three images is the image we are trying to classify. The second and third both represent the best matching template. The second image shows the template in it's original form and the third image shows the rendition of the template we have produced after creating a vector template, re-sizing to fit the original's bounding box, applying linear transformations, and finally matching the line width of the first image.

Again, the three-part figures represent Image || Template || Rendered Vector Template.

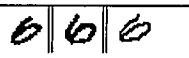
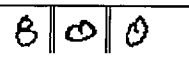
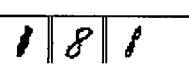
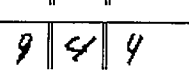
(a) 0 as a 6		(c) 8 as a 0	
(b) 1 as an 8		(d) 9 as a 4	

Figure 62 Selected Misclassifications from the MNIST database

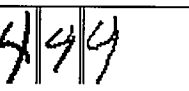
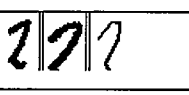
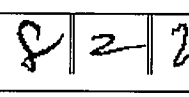
(a) 1 as a 7		(c) 4 as a 9	
(b) 2 as a 7		(d) 8 as a 2	

Figure 63 Selected Misclassifications from the SUEN database

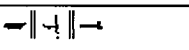
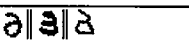
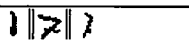
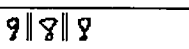
(a) 0 as a 4		(c) 2 as a 3	
(b) 1 as a 7		(d) 9 as an 8	

Figure 64 Selected Misclassifications from the USPS database

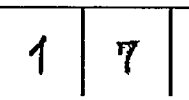
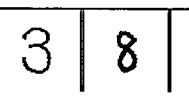
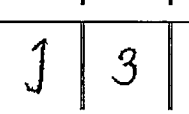
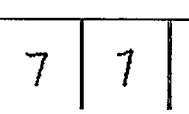
(a) 1 as a 7		(c) 3 as an 8	
(b) 1 as a 3		(d) 7 as a 1	

Figure 65 Selected Misclassifications from the JAPAN database

These mistakes show two things. First, the problem of symbol recognition is very difficult. Achieving 100% recognition is unreasonable as some data is entirely unrecognizable even to a human (see Figure 62 (a)). At other times, the data can be very ambiguous (see Figure 65 (d)). Second, sophisticated as it is, the algorithm we are using can still produce fairly naive mistakes, such as Figure 65 (c), where even though the '3' is drawn with the loop sections almost closed, there should still be no mistaking it for an '8'. We think part of this is due to the fact that we still suffer from making templates "too" elastic. Very thin regions should not cause templates to be deformed into such a degenerate case. Although this does not happen when the images are oriented with parallel the coordinate axes, it still happens with thin, slanted figures.

The examples of errors in Figures Figure 62 through Figure 65 were drawn from an experiment with a training set size of 400 and test set size of 600. The actual recognition rates for these experiments were published in the second last column of Table 9.

6.2.4 Comparison to Other Works

We compared our algorithm against two of the top published results on this database. Unfortunately, we did not have the time to do an exhaustive test, using the enormous amount of training data, 60,000 images coupled with a test set of 10,000 images. Thus, the resulting comparisons may not be on a level playing field.

Performance Comparisons on MNIST Database			
Algorithm	**Recognition Rate**	**Experiment Size**	**Running Time**
K-NN, shape context matching [Belongie1999]	99.93%	20,000 Training 10,000 Testing	63 comparisons/ minute[8]
Boosted LeNet-4, distoritions [LeCun1998]	99.30%	60,000 Training 10,000 Testing	extremely fast
Deformable Templates [Jain1997]	99.25%	2000 Training 2000 Testing	26 comparisons/ minute
Vector Templates[9]	93.5%	400 Training 600 Testing	4000 comparisons/ minute
Vector Templates	95.0%	2000 Training 2000 Testing	4000 comparisons/ minute

Table 10 Comparison on MNIST Database

The numbers in Table 10 show that our technique is very fast but about 4 percent less accurate than existing techniques. We have done some other preliminary experiments with our technique that would give us recognition rates in the ballpark of 95.7% using a few simple extensions to the Inkwell Hausdorff distance (covered more in the Future Extensions section of Chapter 7). Thus,

---

[8] When I asked the authors how long it took to run this experiment, they said it was 6 computer years.

[9] Time and CPU resources were not available to use all 60,000 images as templates, and then to conduct an experiment on 10,000 images of test data.

with a little more algorithmic refinement, this technique could be used as an efficient alternative to expensive variations of deformable template matching. We present the results for running times in "comparisons per minute", which means how many templates an image could be compared against in each minute on a modern machine. These units were chosen since the number of training images varies in many of the above experiments, and thus quoting a number of symbols per second would be not as meaningful.

However, to give an estimate of running times required to achieve the published recognition rates, we will consider two examples. In our case, a running time of 4000 comparisons per minute means that if the library size is 2000 templates, we can recognize one symbol every 30 seconds. To use the shape context technique and achieve a 99.93% recognition rate, 20,000 comparisons must be made at 63 comparisons per minute, which yields one symbol every 5.3 hours.

We now move to a comparison on another database, the JAPAN database. Besides the published rates from previous work of J. R. Parker, we did not find other published rates on this database. However, we were able to demonstrate an improvement over these previous results.

Performance Comparisons on JAPAN Database			
Algorithm	**Recognition Rate**	**Experiment Size**	**Running Time**
Vector Templates [Parker1999]	94.3%	25 prototypes (hand-drawn) 2000 testing	4.4 sec / symbol
Vector Templates, with improvements	99.0%	400 training 600 testing	~ 1 minute / symbol

Table 11 Comparison on JAPAN Database

The principal differences in these experiments were in template generation and use of specialized code. The work done in [Parker1999] used hand-drawn templates after a human looked at the training data. In our work, we automatically generated templates by selecting them at random from the training data. Also, in the first work specialized code was used to handle the '1' digits, and in our case we had a more generalized technique that handled this problem. We enforced a minimum aspect ratio to avoid deforming images into a degenerate 1D object such as a

line. If an image (or template) had a bounding box with an aspect ratio that was too "thin", white space was added until a reasonably-dimensioned image was produced.

As for the CENPARMI database and the USPS database, it was difficult to find published results of recognition rates on these sets. We now present the full confusion matrices for the recognition rates in the third column of Table 9.

Confusion Matrix on MNIST Database (Recognition Rates in Percent)										
Actual	Recognized As									
	0	1	2	3	4	5	6	7	8	9
0	98.33	-	-	-	-	-	1.67	-	-	-
1	-	85.00	-	5.00	-	-	-	5.00	3.33	1.67
2	-	-	98.33	-	1.67	-	-	-	-	-
3	-	-	1.67	93.33	-	1.67	-	-	1.67	1.67
4	-	-	-	-	96.67	-	1.67	1.67	-	-
5	-	-	-	3.33	-	95.00	1.67	-	-	-
6	-	-	-	-	-	-	100.00	-	-	-
7	-	-	1.67	-	3.33	-	-	91.67	1.67	1.67
8	1.67	-	1.67	-	1.67	1.67	-	-	91.67	1.67
9	-	-	1.67	-	5.00	-	-	5.00	3.33	85.00

Table 12 MNIST Confusion Matrix, overall recognition rate of 93.50%

Confusion Matrix on SUEN Database (Recognition Rates in Percent)										
Actual	Recognized As									
	0	1	2	3	4	5	6	7	8	9
0	100.00	-	-	-	-	-	-	-	-	-
1	-	91.67	-	-	-	-	-	3.33	3.33	1.67
2	3.33	-	93.33	-	-	-	-	3.33	-	-
3	1.67	-	5.00	83.33	-	5.00	-	-	5.00	-
4	-	-	-	-	95.00	-	-	1.67	-	3.33
5	1.67	-	-	-	-	93.33	5.00	-	-	-
6	-	-	-	-	-	-	100.00	-	-	-
7	-	-	1.67	-	-	-	-	96.67	-	1.67
8	-	-	3.33	-	1.67	-	-	-	93.33	1.67
9	-	-	-	-	-	-	-	1.67	-	98.33

Table 13 SUEN Confusion Matrix, overall recognition rate of 94.50%

Confusion Matrix on USPS Database (Recognition Rates in Percent)										
Actual	Recognized As									
	0	1	2	3	4	5	6	7	8	9
0	93.33	-	-	1.67	1.67	-	-	1.67	1.67	-
1	1.67	81.67	5.00	1.67	-	1.67	3.33	5.00	-	-
2	-	-	90.00	5.00	1.67	-	-	-	3.33	-
3	-	-	1.67	88.33	-	3.33	-	1.67	3.33	1.67
4	-	1.67	-	-	86.67	1.67	3.33	1.67	-	5.00
5	3.33	-	-	3.33	-	81.67	3.33	-	1.67	6.67
6	-	-	-	-	-	-	100.00	-	-	-
7	-	-	-	-	3.33	-	-	96.67	-	-
8	1.67	-	1.67	-	3.33	1.67	-	3.33	86.67	1.67
9	-	-	-	1.67	3.33	-	-	5.00	1.67	88.33

Table 14 USPS Confusion Matrix, overall recognition rate of 89.33%

Confusion Matrix on JAPAN Database (Recognition Rates in Percent)										
Actual	Recognized As									
	0	1	2	3	4	5	6	7	8	9
0	100.00	-	-	-	-	-	-	-	-	-
1	-	96.67	-	1.67	-	-	-	1.67	-	-
2	-	-	100.00	-	-	-	-	-	-	-
3	-	-	-	98.33	-	-	-	-	1.67	-
4	-	-	-	-	100.00	-	-	-	-	-
5	-	-	-	-	-	100.00	-	-	-	-
6	-	-	-	-	-	-	100.00	-	-	-
7	-	5.00	-	-	-	-	-	95.00	-	-
8	-	-	-	-	-	-	-	-	100.00	-
9	-	-	-	-	-	-	-	-	-	100.00

Table 15 JAPAN Confusion Matrix, overall recognition rate of 99.00%

Finally, we performed one substantial test on the CENPARMI dataset, to see what the recognition rate was using a leave-one-out test. In this test, we looked at 1578 images in the *traina* dataset. For each image, we built vector templates from the 1577 other images to see if we could recognize it correctly. We achieved a recognition rate of 96.40% using this testing method, and the running time was still approximately 60 sec / symbol.

Confusion Matrix on SUEN Database, Leave one out (Recognition Rates in Percent)										
Actual	Recognized As									
	0	1	2	3	4	5	6	7	8	9
0	96.77	-	-	1.61	-	1.61	-	-	-	-
1	-	96.00	-	0.67	-	0.67	-	2.67	-	-
2	0.64	-	98.09	0.64	-	-	-	-	0.64	-
3	-	-	1.13	94.35	-	3.39	-	0.56	0.56	-
4	-	-	-	-	97.79	-	-	0.74	-	1.47
5	1.69	-	-	0.56	-	96.07	-	-	1.12	0.56
6	0.70	-	-	-	-	2.11	97.18	-	-	-
7	-	-	0.74	-	-	-	-	97.78	-	1.48
8	0.53	3.21	0.53	-	0.53	-	-	-	95.19	-
9	-	0.52	1.04	1.04	-	0.52	-	-	1.04	95.83

Table 16 SUEN Confusion Matrix, overall recognition rate of 96.39%

6.2.5 The Template Selection Problem

One of the problems with template based symbol recognition systems, is that it can be time consuming to compare an image against so many templates. One way around this is to reduce the number of templates. This is not a trivial task by any means. In his paper on Deformable Templates, [Jain1997], Anil K. Jain describes a technique he used. His approach clusters the patterns for each class, and selects a representative from each cluster. This was implemented using a complete-link hierarchical clustering on the patterns of each class, independently. All of the data for one digit class was cut into p clusters, and a representative was chosen for each class. The representative was chosen to minimize its total distance to the other templates in that cluster. In this way p prototype images were chosen for each digit class. [Jain1988].

Here are the recognition rates based on the number of templates

Recognition Rates using Complete Link Hierarchical Clustering (on Deformable Templates)	
Number of Templates per class	Recognition Rate
5	73.60%
10	77.40%
20	84.90%
30	84.30%
200 (full database)	92.50%

Table 17 Improvements from Template Selection

Again, Table 17 is the table reported in [Jain1997]. A good template selection strategy however, should pick prototypes that are similar to others in their own class, but also as different as possible from other classes. We tried to make use of this idea when designing our own prototype selection strategy.

In fact, a few techniques were mentioned in [Belongie1999], in which he termed this problem *nearest neighbour editing methods*. He cites [Ripley1996] and [Dasarathy1991] as good references on algorithms for this problem. The approach also computes a matrix of pairwise similarities between all possible prototypes. From that point on, the individual classes are split into clusters, just as before, and then a representative is chosen. They come up with some rules for choosing how many representatives are needed, which is an improvement over the simplistic assumption that the same number of templates should be used for each class.

We first built a distance matrix, a square matrix that represents the distances between all pairs of prototypes. D_{ij} = distance when treating image i as the test image, and image j as a template. Thus one row represents a view of a prototype as an image, and one column represents it as a template. The templates are sorted by their classes, so we can even look at the block diagonal portions of this distance matrix, where there are 10 "blocks", one for each digit class. It was very useful to have this matrix stored once since we could then try different template selection strategies on it, without having to re-compute distances between all the prototypes. This was a substantial saving as it took at least 12 hours to build it initially. The distance matrix we built was for the CENPARMI dataset, on the *traina* training data file. This yielded a 1578×1578 matrix of distance values.

We chose templates from this matrix by a *popularity* criterion. We define a *radius* of distance values for each template as follows:

$$r(T_j) = \min_{\forall i | c_i \neq c_j} d(A_i, T_j)) \qquad \textbf{(Eq 43)}$$

where A_i is the i^{th} image, T_j is the j^{th} template, and c_i is the classification of pattern i. That is, the radius is the distance to the nearest out-of-class image from this template. The neighbourhood of a template T_j is the set of all images that are within a distance $r(T_j)$. By definition, only images of the same class as the template will be in its neighbourhood. A large neighbourhood indicates that the template is similar to many other images of the same class, and is not easily confused with images of other classes.

After determining the radius and neighbourhood for each template, we selected the *best* template for each class, the one with the largest number of points in its neighbourhood. Since this template has the largest neighbourhood, we call it the most popular template (for that class, for that iteration). After identifying these templates, we mark off all images that are in one of their neighbourhoods. We then repeat the experiment, with all unmarked images in the training set. Thus from the set of images we start with, we whittle it down at each step by picking out the best templates from each class, and discarding all images that fall into a neighbourhood of a chosen template.

To do this test, we partitioned *traina* into testing and training data. We chose 500 images of the 1578 for training, and the remaining 1078 for testing.

The results show what one would expect. Template selection helps if we are only allowed to use a small number of templates per class. Picking k templates randomly, when k is small, is significantly worse and much less consistent (higher standard deviation on recognition rates) than choosing them carefully. However, in the limit, as k gets large we see that the recognition rate from the randomized strategy gets closer and closer to that of the recognition rate of the popularity strategy. Thus, if the number of templates that we are allowed is not important, selecting templates randomly works just fine. If an application is very sensitive to the amount of work being performed, then a template selection strategy improves things. Also, it seems that with twice as many templates per class chosen randomly as opposed to selected carefully, we get approximately the same recognition rate. So for example, if three digits per class are chosen via the algorithm, we can match this recognition rate with approximately six digits per class chosen randomly. This is not 100% accurate, but could be used as a good rule of thumb.

Comparing Template Selection Strategies				
Number of Templates per Class	Randomized Template Selection		Template Selection Strategy by Popularity	
	Rate	Stdev	Rate	Stdev
1	64.06%	6.9%	73.77%	2.1%
2	72.70%	3.1%	80.34%	2.1%
3	77.94%	2.4%	83.96%	1.4%
4	79.97%	2.5%	85.58%	1.5%
5	83.69%	1.3%	86.91%	1.1%
10	88.19%	1.2%	89.17%	1.2%
15	90.07%	0.6%	90.70%	1.2%
20	91.39%	1.1%	91.78%	0.9%
30	92.93%	0.8%	92.70%	0.7%

Table 18 Comparing Template Selection Strategies

6.3 The Electrical Engineering Dataset

This dataset was designed by the Intelligent Systems Laboratory at the University of Washington [Ye1999]. This same dataset was used as the dataset for the Symbol Recognition Contest for ICPR 2000, called the "Algorithm Performance Contest".

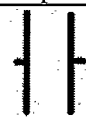
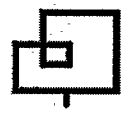
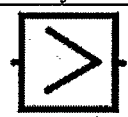
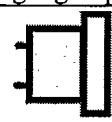
 Ammeter	 Amplifier	 Annunciator	 Battery	 Buzzer
 Buzzer2	 Capacitor	 Ctrletrode	 Current Regulator	 DC Converter
 Diode	 Direct Coupler	 FRT Beadring	 Fuse	 Ground
 Keyctrl	 Lamp	 Lgongloop2	 Loud Speaker	 Manual Ctrl
 One way RPT	 Resistor	 Telrecv	 Touch Sensor	 Voltmeter

Table 19 The electrical symbol library

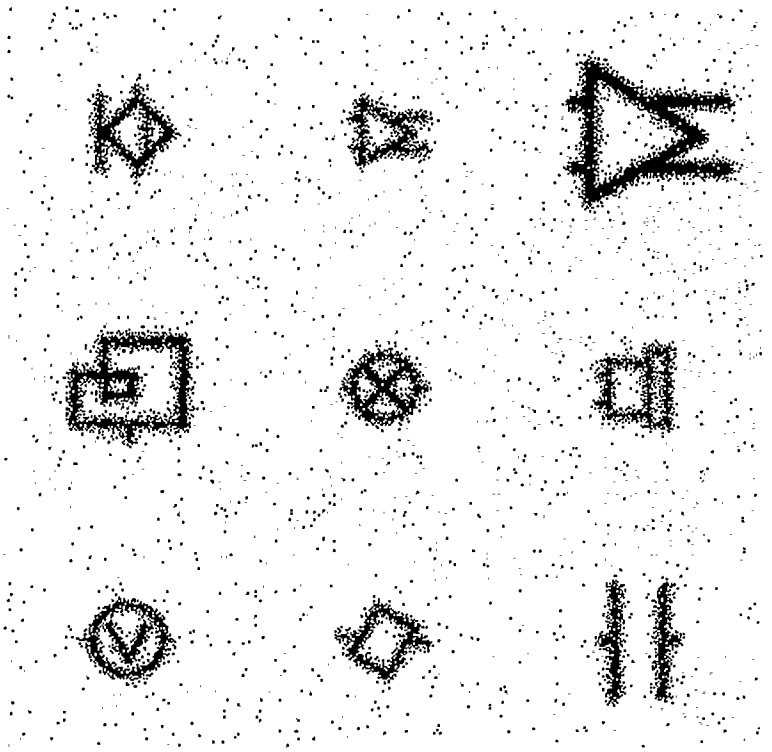


Figure 66 Sample Symbols from the EE Dataset

The symbol library consists of 25 electronic symbols, shown in Table 19, and the software package was able to produce images with a varying number of symbols on them and at different scales and orientations. The official dataset consists of 600 images of symbols, with 25 symbols per image. The 600 symbols are partitioned into 12 classes, crossing three scale levels (50x50, 75x75, and random scale) with four different levels of noise (no noise, level 1 noise, level 2 noise, and level 3 noise). The noise model was introduced to simulate two effects that are often produced by photocopiers. A photocopier may sometimes blur the edges of the text, and can also leave little pieces of "dirt" on the white areas. The latter kind of noise is called *salt and pepper noise*, and is modeled with a random variable for each pixel in the image. With probability p, the pixel is flipped. The other kind of noise, called *edge blurring noise*, was also imposed. To achieve the edge-blurring effect, a decision is made whether or not to toggle a pixel based on how far away it was from an edge in the image. Pixels closer to object boundaries have a higher chance of being toggled than pixels that are farther away. In all three increasing levels of noise, both salt and pepper noise and edge blurring noise were present.

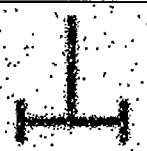
Electrical symbols under varying levels of noise.				
	No Noise	Noise1	Noise2	Noise3
Scale: 50x50				
Scale: 75x75				
Scale: random				

Table 20 Organization of the EE Dataset

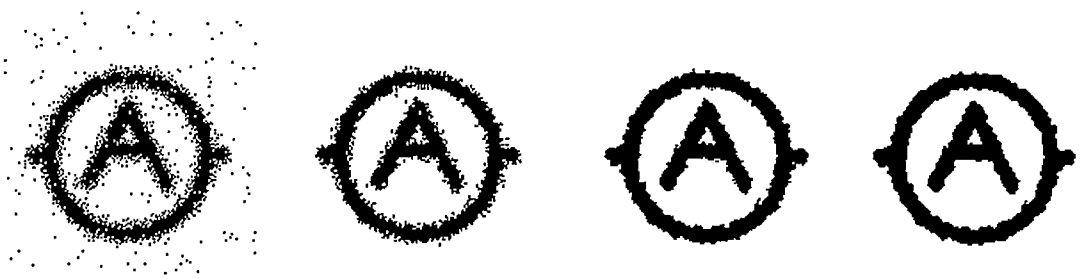
Each of the 12 cells in Table 20 corresponds to a test set of 50 images, each containing a grid of 25 randomly-selected symbols matching the noise levels indicated. Thus we attempted to recognize 1,250 symbols in each category for a total of 10,000 symbols overall.

Table 21 contains a summary of our algorithm performance using a simplified noise removal algorithm. Small isolated black regions were removed from the background and small white regions from the foreground. The recognition rates were sensitive to how noise was removed from the images. As a result, we spent some time improving the noise removal algorithm.

Algorithm Performance				
	No Noise	**Noise1**	**Noise2**	**Noise3**
50x50	100%	100%	99.92% (1 error)	95.04% (62 errors)
75x75	100%	100%	100%	97.44% (32 errors)
rand	100%	100%	100%	99.28% 9 errors

Table 21 Recognition Rates on noisy machine-drawn images, using naive noise removal strategy.

Below is a table of our algorithm performance using a noise removal filter developed by D. Royko [Parker2000] which uses a *median filter* and then an *edge filter*. The median filter counts the number of background and foreground pixels in a 3×3 region around each pixel. It replaces the center pixel with whatever type is most prominent. This filter provides a first phase of "cleanup" to clarify the image around the edges, but the edges are still quite jagged. The edge filter is then applied to smooth them out. Each background pixel that looks like it is on an edge (its crossing index is 1) is toggled into a foreground pixel. Also, a final noise removal pass is made to trim bounding boxes, as the algorithm relies on an accurate bounding box for image alignment. In Figure 67, we see an illustration of the noise removal algorithm. In (a) we see an example of an ammeter under level 3 noise. In (b) we have removed most of the salt and pepper noise, by toggling isolated pixels. In (c) we have applied the median filter and finally in (d) we have applied the edge filter.



(a) (b) (c) (d)

Figure 67 An illustration of the noise removal algorithm

Nearly identical results were obtained using both handdrawn and machine-generated vector templates.

Algorithm Performance				
	No Noise	**Noise1**	**Noise2**	**Noise3**
50x50	100%	100%	99.6% (5 errors)	100%
75x75	100%	100%	100%	100%
rand	100%	100%	100%	100%

Table 22 Recognition Rates on Machine Drawn Electrical Symbols

The images in the test set were slightly rotated, from –3 degrees to +3 degrees. We tried rotations of our templates at degree increments in this interval. After finding the best orientation, we further rotated a half degree in either direction.

6.3.1 Matching the Line Width

The test images have symbols with thick lines, and we try to match this as closely as possible when drawing vector templates. The vector templates are created from large bitmap templates of the symbols drawn with a pen width of 30 pixels. One easy technique for estimating the correct line width is to first resize the vector template to match the image, and then compare its total line length to that of the original image it was created from. As a ratio, this number gives us the ratio of the stroke width we should be using. For example, if the total length of the line segments in the original template (drawn with line width 30) is 1000 pixels, then if we resize a template that has a new length of only 100 pixels, then we could try drawing it with a line width of 3 pixels. This gave very good estimates of stroke width, not surprisingly, as both the length and the stroke width are parameters measured in the same "space". In addition to measuring the stroke width in this way, we also estimate the stroke width with two other algorithms, described in Chapter 4. We use the scale method described above to compare how well our own estimates of stroke width measure up.

If we define E_1 to be this true width, and E_2 to be the estimated width, with $E_2 = 2A/P$, then we can observe the distribution of the error variable $E_1 - E_2$ in Table 23.

Stroke Width Error Rate			
Data	**Mean (pixels)**	**Standard Deviation (pixels)**	**Min (pixels)**
Clean	0.37	0.66	-.75
Noisy	0.51	0.61	-.57

Table 23 Statistical Distribution of error in stroke width estimation

From this it can be seen that the estimated stroke width is within half a pixel of the correct width, and because the mean is positive, it means that the width is almost always underestimated. This could be corrected once a great deal more data has been collected for various symbol types. As a final note on stroke width, we emphasize again that these estimates are of *uniform* stroke width and that a different technique would be necessary if the stroke width varied locally. We refer the reader to Chapter 4 for more details on variable stroke width estimation.

6.3.2 Skeletal Heuristic

To improve the performance of the algorithm, we came up with an easily-implementable heuristic. Especially for machine drawn symbols, one would expect that a vector template plotted with thin lines would have a high level of overlap with an image that is a good match. Thus, when comparing a template and image, we first plot the template using a 1-pixel wide line, and count which percentage of the pixels overlap with the image. If we have 75% or more of the pixels hitting the target, then we perform the full match. The full match entails plotting the vector template with thick lines that try to duplicate the style of the original image. This heuristic gives us about a 35% performance increase. In handwritten digit recognition, we cannot demand such a high level of correspondence between templates and images, and thus this heuristic is not used.

This heuristic is similar to the more detailed work done in [Vanderbrug1977], where a search for an optimal 'sub-template' size was carried out.

6.4 The NCO Dataset

We obtained scanned images of pages from "Nunn's Chess Openings" [Nunn1999] to assess the feasibility of recognizing all of the symbols using a combination of the vector templates and the image metrics described in this thesis.

13 ♘xd4 ♕f6 =; 7 ♕c2 ♘d7 8 e4 dxe4 9 ♕xe4 g6 10 ♗d3 ♗g7 11 0-0 0-0 12 ♖fe1 c5 13 d5 ♘b6 =; 7 e4 dxe4 8 ♘xe4 ♗b4+ 9 ♔e2 ♕f4 10 ♕c2 ♗e7 =

Figure 68 Sample Text and Chess Piece Symbols from NCO

Although we did not have time to complete this project, we were able to identify all the chess symbols on the page with 100% accuracy. The algorithm made mistakes with one's and lowercase L's but this was expected as they appear almost indistinguishable. The running time was very fast because, rather than recognizing each symbol as we encountered it, we made of list of the distinct symbols on the page, and then performed a full classification only on this list. The list was constructed by using raw bitmaps as templates in conjunction with the Inkwell Hausdorff distance. This saved a lot of time because when comparing a vector template and an image, we have to scale, transform, and plot the vector template, and we do this for a few orientations. Rather than going through all this work at each step, we just compared a particular symbol with all of the symbols we had seen to date (often much smaller than the full library), to see if it was a new one. If it's minimal distance to this growing library exceeded a threshold, we could assume it represented a new symbol and added it to the library. By using such simple templates, we could quickly identify the distinct symbols and then proceed to run a thorough symbol recognizer on them, using vector templates.

This experiment helps to show the more general applicability of using vector templates, even for machine printed symbols, and helps re-affirm the superiority of the of the Inkwell Hausdorff distance to standard bitmap template matching.

Chapter 7
Conclusions

7.1 Summary

Symbol recognition is a complicated problem and many different approaches have been tried to build accurate, efficient symbol recognizers. The approaches can be broken down into two broad categories, prototype-based and prototype-free. These two categories take fundamentally different approaches to solving the problem, and each has its strengths and limitations. Under the umbrella of prototype-based symbol recognizers, there is a spectrum of template models, ranging from rigid to deformable. However, the former is usually fast but not so accurate and the latter are very accurate but also very slow. We consider merging the best of both worlds into a new prototype-based classifier, one that is fast and robust.

We have tried to solve this problem from two angles; choosing an appropriate prototype representation coupled with a sophisticated image metric. We represent prototypes as vector templates, which are lightweight, flexible and can be quickly normalized to match the style of an arbitrary image. We measure distance between images using the Inkwell Hausdorff distance, which, improving over other techniques, is tolerant to local misalignments. Our technique is 30 to 70 times faster than current techniques but only provides a recognition rate of 95% compared to 98% or 99% reported by these same algorithms. However, the approach shows promise as an efficient alternative to expensive template matching.

What sets our technique apart from true deformable templates? The primary difference is that a deformable template should smoothly deform into a new alignment. Our templates are allowed to "rip" or break in half, and undergo other non-continuous transformations while deforming to match the target image. We model a term for how much work is required to do the deformation, but without analyzing how natural or how likely this transition might be. Since this term is lacking in our model, it allows us to have a very fast technique which is essentially an approximately deformable template. But there is hope to retain efficiency while introducing terms that evaluate the likelihood of a certain deformation. This leads us into some possible future extensions.

7.2 Future Extensions

While studying the Inkwell Hausdorff distance, we have come to see how to derive the mutual proximity vector field. This vector field is really a combination of feature transforms for the two images, and records for each point in one image where it must move to find its nearest neighbour in the other image. (See Chapter 3). By visualizing this vector field, we have come across some key observations. By smoothing the vectors and applying them to a mesh, we can see the effect the vector field has on the plane, and this gives us a measure (at least visually) of how costly the implied deformation would be. Also, by examining a quantity very similar the divergence of a vector field, we can identify areas where the vectors are pulling on a point from opposite directions, as in Figure 69.

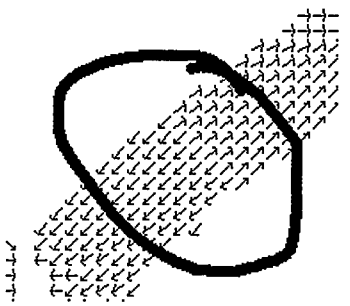


Figure 69 Points of high divergence in a vector field.

By identifying these locations, we can then proceed to use them as a weight on the Inkwell Hausdorff distance contributions. In fact, we have implemented an experiment to test this idea, and it does appear to improve our results on a 25 hour (somewhat substantial) experiment involving 2000 templates . However, more testing would be necessary to understand the exact relationship between the divergence of the vector field and how to combine it with the Inkwell Hausdorff distance.

Another idea to try is to reverse the roles of the templates and the images we are recognizing. That is, we could treat the image we're recognizing as the template, and all the templates as if they are images. It would be interesting to see how the recognition rates compare. We have gone with the principle of trying to avoid any alterations to the input image, for fear of changing it into something it is not. How important is this really? Also, trying this idea would give us an understanding of how robust the vector template generation software is. By that, we mean that the software relies on thinning to create a vector template from an image, which is a process that

can create artifacts. By converting many exemplars into vector templates, we minimize the impact of creating an non-representative template from an exemplar. However, if we only have one opportunity to convert an image into a vector template, and we produce a poor result, the flaws, if significant, in the vector template generation would be accentuated.

As an interesting side effect of the mutual proximity vector field mentioned earlier, we have found a way to quickly produce a morphing animation between two black and white images. The algorithm produces visually convincing animations in many cases, but more work is necessary to understand its strengths and limitations.

Bibliography

[Aksoy2000] S. Aksoy, M. Ye, L. Schauf, M. Song, Y. Wang, R.M. Haralick, J.R. Parker, J. Pivovarov, D. Royko, C. Sun, G. Farneback, "Algorithm Performance Contest", *International Conference on Pattern Recognition*, 2000.

[Belongie1999] S. Belongie, J. Malik and J. Puzicha. "Shape Context: A new descriptor for shape matching and object recognition" *Technical Report*, University of California at Berkeley, Jan 2001 EMAIL: {sjb,malik,puzicha}@cs.berkeley.edu

[Bookstein1989] F. L. Bookstein "Principal Warps: thin-plate splines and decomposition of deformations." *IEEE Transactions on Pattern Analysis and Machine Intelligience, 11(6):567-585*, June 1989.

[Brooks1981] R. A. Brooks, "Model-based Computer Vision", *A revision of author's thesis*, Stanford University, 1981.

[Casey1970] R. G. Casey "Moment Normalization of Handprinted Characters", *IBM J. RES. DEVELOP*, September 1970.

[Cohen1999] S. Cohen. "Computing the Earth Mover's Distance under Transformations". *Master's Degree Thesis*, Stanford University, 1999.

[CENPARMI] C. Y. Suen "CENPARMI Handwritten Digit Database", *Concordia University*, Montreal.

[Dasarathy1991] V.B. Dasarathy, editor. "Nearest Neighbour (NN) norms: Pattern Classification Techniques", *IEEE Computer Society*, 1991

[ETL1993] Japanese Technical Committee for Optical Character Recognition, "ETL Character Database", ICDAR, Tsukuba, Japan, October 1993.

[Foley1990] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes, "Computer Graphics: Principles and Practice", *Addison-Wesley*, Reading Mass., 1990.

[Gamma1995] E. Gamma et al. "Design Patterns", *Addison Wesley*, 1995

[Gavrilova2000] M. Gavrilova and M. Alsuwaiyel, "Two Algorithms for Computing the Euclidean Distance Transform", *University of Calgary Department of Computer Science Research Report #2000/667/13.*, 2000.

[Hew1998] P. C. Hew and M. D. Alder, "The lim-adler algorithm for extended edge extraction,", *Technical Report TR98-02, Centre for Intelligent Information Processing Systems*, The University of Western Australia, November, 1998. (http://ciipse.ee.uwa.au/Papers)

[Holt1987] C.M. Holt et al., "An Improved Parallel Thinning Algorithm", *Communications of the ACM Vol 30, No. 2, p156-160*, 1987.

[HuttenLocher1991] D.P. Huttenlocher et al. "Comparing Images using the Hausdorff Distance", *Technical Report CUCS TR 91-1211* (revised), Cornell University NY 1991.

[Huttenlocher1987] D.P. Huttenlocher, S. Ullman "Object Recognition Using Alignment", *ICCV*, 1987.

[Jain1998] A.K. Jain and R.C. Dubes, "Algorithms for Clustering Data", *Englewood Cliffs, N.J.: Prentice Hall*, 1988.

[Jain1997] A.K. Jain and D. Zongker. "Representation and Recognition of Handwritten Digits Using Deformable Templates", *IEEE Transactions on Pattern Analysis and Machine Intelligience, Vol 19, No 12* December 1997.

[Jain1996] A..K. Jain, et al. "Object Matching Using Deformable Templates", *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 18, No. 3*, March 1996.

[Jain1988] A..K. Jain, and R.C. Dubes, "Algorithms for Clustering Data." *Englewood Cliffs,* N.J.: Prentice hall, 1988.

[Jonker1987] R. Jonker and A. Volgenant. "A shortest augmenting path algorithm for dense and sparse linear assignment problems." *Computing, 38:325-340*, 1987.

[Kuhn1955] H. W. Kuhn, "The Hungarian method for the assignment problem", *Naval Research Logistics Quarterly*, 2 , pp. 83-97, 1955.

[Lam88] Lam, L. and Suen, C.Y., Structural Classification and Relaxation Matching of Totally Unconstrained Handwritten Zip-Code Numbers, *Pattern Recognition, Vol. 21 No. 1. pp. 19-31*, 1988.

[LeCun1998] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE*, 86(11):2278-2324, November 1998.

[LeCun1995] Y. LeCun, et al. "Learning Algorithms for Classification: A Comparison on Handwritten Digit Recognition", http://www.research.att.com/~yann/exdb/mnist/index.html, *AT&T Bell Laboratories*, 1995.

[Lu1986] H.E. Lu and P.S.P. Wang "A Comment On 'A Fast Parallel Algorithm For Thinning Digital Patterns' ", Communications of the ACM (29), 1986, pp. 239-242.

[Nunn1999] J. Nunn, G. Burgess, J. Emms, and J. Gallagher, "Nunn's Chess Openings", *Everyman Chess*, 1999.

[Papadimitriou1982] C. Papadimitriou, K. Steiglitz "Combinatorial Optimization", *Dover*, 1998.

[Parker2001a] J. R. Parker, J. Pivovarov, "Recognizing Symbols by Drawing Them", *International Journal of Images and Graphics,* Vol. 1, No. 4 2001.

[Parker2001b] J. R. Parker, J. Pivovarov "A Demonstration of Handprinted Symbol Recognition", *Proceedings of the International Conference on Computer Vision*, Vol 2, p740. 2001

[Parker2000] J. R. Parker, J. Pivovarov , D. Royko "Vector Templates for Symbol Recognition", *IEEE International Conference on Pattern Recognition*, Vol 2, pp602-605. 2000.

[Parker1999] J.R. Parker, "Vector Templates for Handprinted Symbol Recognition", *3rd International Conference on Image Analysis and Graphics*, 1999.

[Parker1997] J. R. Parker, "Algorithms for Image Processing and Computer Vision", *John Wiley & Sons, Inc.* 1997.

[Parker1995] J. R. Parker, "Vector Templates and Handprinted Symbol Recognition" *The University of Calgary, Department of Computer Science*, Technical Report 95/559/11, 1995

[Parker1994] J. R. Parker, "Practical Computer Vision Using C", *John Wiley & Sons, Inc.* 1994.

[Parker1991] J. R. Parker, "A System for Fast Erosion and Dilation of Bi-Level Images", *Journal of Scientific Computing, Vol. 5, No. 3, p.187* 1991.

[Rauber1994] T. W. Rauber, "Two-Dimensional Shape Description", *Technical Report GR UNINOVA-RT-10-94,* Universidade Nova de Lisboa, and UNINOVA – Intelligent Robotics Center, 1994.

[Ripley1996] B.D. Ripley, "Pattern Recognition and Neural Networks", *Cambridge University Press*, 1996

[Roth1994] G. Roth and M. D. Levine, "Gemetric Primitive Extraction Using a Genetic Algorithm", *IEEE Transactions on Pattern Analysis and Machine Intelligience Vol. 16. 9:901-905*, 1994.

[Rubner1998] Y. Rubner, C. Tomasi, L. Guibas. "A Metric for Distributions with Applications to Image Databases". *Proceedings of the 1998 IEEE International Conference on Computer Vision*, January 1998 (http://robotics.stanford.edu/~rubner/publications.html).

[Suen1992] C. Y. Suen, C. Nadal, R. Legault, T. A. Mai, and L. Lam, "Computer recognition of unconstrained handwritten numerals.", *Proceedings of the IEEE, vol. 80, no. 7, pp. 1162--1180*, July 1992.

[Tchoukanov1992] I. Tchoukanov, R. Safaee-Rad, K.C. Smith and B. Benhabib. "The Angle-of-Sight Signature for Two-Dimensional Shape Analysis of Manufactured Objects" *Pattern Recognition, Vol. 25, No. 11, pp 1289-1305*, 1992.

[Wahba1990] G. Wahba. "Spline Models for Observational Data", *SIAM*, 1990

[Wakahara1994] T. Wakahara, "Shape Matching Using LAT and its Application to Handwritten Numeral Recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligience, Vol. 16, No 6*, June 1994.

[USPS] "Isolated Digits from ZIP Codes", *United States Postal Service* (ftp://ftp.kyb.tuebingen.mpg.de/pub/bs/data/)

[Valveny2000] E. Valveny, E. Marti. "Hand-drawn symbol recognition in graphic documents using deformable template matching and a bayesian framework": *IEEE International Conference on Pattern Recognition*, Vol 2, pp239-242. 2000.

[VanderBrug1977] G.J. VanderBrug and A. Rosenfeld, "Two-Stage Template matchings", *IEEE Transactions on Computers Vol. C-26, No. 4, pp. 384-394* 1977.

[Veltkamp1999] R.C. Veltkamp and M. Hagedoorn "State-of-the-Art in Shape Matching", *Technical Report UU-CS-1999-27*, Utrect University, the Netherlands, 1999, http://citeseer.nj.nec.com/veltkamp99stateart.html.

[Ye1999] M. Ye, "Symbol Recognition Package", *Intelligent Systems Laboratory, University of Washington,* Ver. 1.0, 1999. (http://isl.ee.washington.edu/IAPR/)

[Zhang84] Y.Y. Zhang and C.Y. Suen, "A Fast Parallel Algorithm for Thinning Digital Patterns" *Communications of the ACM, Vol. 27 No. 3, 1984. pp 236-239*, 1984