A character-recognition system for Hangeul

Johan Sageryd
j@1616.se

18 September 2009

Master's dissertation
Language Technology

LUND UNIVERSITY
Centre for Language and Literature
Helgonabacken 12, SE-223 62 Lund, Sweden

Supervisor: Johan Frid
johan.frid@ling.lu.se

Abstract

This work presents a rule-based character-recognition system for the Korean script, Hangeul. An input raster image representing one Korean character (Hangeul syllable) is thinned down to a skeleton, and the individual lines extracted. The lines, along with information on how they are interconnected, are translated into a set of hierarchical graphs, which can be easily traversed and compared with a set of reference structures represented in the same way. Hangeul consists of consonant and vowel graphemes, which are combined into blocks representing syllables. Each reference structure describes one possible variant of such a grapheme. The reference structures that best match the structures found in the input are combined to form a full Hangeul syllable. Testing all of the 11 172 possible characters, each rendered as a 200-pixel-squared raster image using the gothic font AppleGothic Regular, had a recognition accuracy of 80.6 percent. No separation logic exists to be able to handle characters whose graphemes are overlapping or conjoined; with such characters removed from the set, thereby reducing the total number of characters to 9 352, an accuracy of 96.3 percent was reached. Hand-written characters were also recognised, to a certain degree. The work shows that it is possible to create a workable character-recognition system with reasonably simple means.

Contents

List of Figures

List of Tables

1 Introduction

Character recognition has proven useful in a wide range of applications. Handwriting input methods for mobile devices is one example, which belongs to a category referred to as *on-line* character recognition. In on-line recognition, the system has access to information about *how* the character was entered, such as the variations in writing speed, and the order and direction of strokes.

The other category is *off-line* character recognition, where the only input is a static image representation of the text. Examples include automatic processing of bank cheques or other forms, sorting of mail by address or postal code [20, 21], and reading vehicle licence plate numbers. Another use is in creating searchable or even editable copies of scanned documents or other printed matter.

The work of this dissertation falls into the latter category. References to the term *character recognition* implies *off-line character recognition* unless otherwise specified.

1.1 Previous work

The area of character recognition is widely explored. Highly accurate recognition of clear printed Latin script has been accomplished, and even though it is not perfect it does indeed give good enough results to be useful. Extensive research has also been made in hand-writing recognition, both on-line and off-line. [20]

Recognition of scripts containing several thousand characters such as Chinese, Japanese and Korean usually involves more difficulties due to, among other things, the complexity in each character and the high level of character similarity. [22] Despite this, a high level of accuracy has been reached in the recognition of both printed and hand-written text. [7] The use of language models for error correction has also been shown to be a good complement. [18] As for recognition of the Korean script Hangeul, several studies are based on neural nets [3, 4, 11, 17], hidden Markov models [13, 19], or variants thereof.

Lee et al. [17] have achieved very good results (99.65 percent) for off-line recognition of printed data using an MLP[1] classifier. The classification is divided into two stages. In the first stage, the character type of the input is determined (Figures 1(a) through 1(f) on page 9). The input image is then divided into sub-images based on the assumed locations of its graphemes (jamo), keeping much of the surrounding context in order to prevent ambiguities that would otherwise occur. In the second stage, a classifier used only for the determined character type is used to classify each of the extracted sub-images separately. A validation process evaluates the result of the type classifier by looking at the result of the grapheme classifier. A very strong argument for using a system based on a neural network or similar, is that it is often very easily extended to support additional character styles (i.e. additional fonts).

Although seemingly not as common, there are also examples of rule-based approaches with very good results. A method for off-line hand-writing recognition described by Kim and Kim [12] has some similarities with the system developed in this work. Kim and Kim [12] use a set of 28 pre-defined stroke types (cf. line types, Section 2.2.5 on page 16) in combination with 300 stroke separation rules to create a stroke sequence string representing the input, which in turn

[1]The multi-layer perceptron (MLP) is a feed-forward artificial neural network model.

is matched to a character using a recognition tree (cf. graph matching against structure definitions, Section 2.4 on page 21). The recognition rate achieved by this method was 94.43 percent. After noise-removal, the input image is thinned and feature points at line ends and crossings are extracted. A vertical vowel[2], if present, is first identified and removed to prevent it from interfering with stroke extraction and determination of stroke order. Using a set of 300 stroke separation rules, the thinned input is split up and stroke types are identified. The stroke sequence is determined by examining the vertical ordering and positional relationships between consecutive strokes. A recognition tree originally used for on-line recognition is extended to support additional strokes found in off-line characters, and used to match the extracted stroke sequences to a character.

1.2 Aim

The primary aim of this work is to, from the ground up, design and implement a simple character-recognition system for Hangeul. The intention is not to create a state-of-the-art character classifier, but rather to see how good a system can be created with the simple means presented.

A secondary aim is to give an introduction to the level of complexity involved in character recognition, yet still show that it is possible to create a working system with fairly uncomplicated methods. The use of a rule-based approach makes it easy to interpret the recognition results.

1.3 Scope

While the aim is of course to have the system yield as good results as possible, some restrictions must be applied to keep it from exceeding the bounds of this dissertation.

First and foremost it must be noted that this work does not in any way try to be a complete solution, it should merely be thought of as a test of concept.

The input will be assumed to be a completely noise-free, clean image of one character. Images containing more than one character will not be segmented. There are also characters whose elements (*jamo*, as described later) are conjoined. For example, 구 consists of the two conjoined elements ㄱ and ㅜ. 큘 (ㅋ + ㅠ + ㄹ) is another example. Characters of this kind will not be handled. Section 4 discusses difficulties in separating conjoined elements.

The system will also be limited to recognition of, primarily, a Gothic font. No efforts will be put into making it work with brush script or hand-writing, or even fonts with serifs or cursive style. Due to the fairly flexible structure however, the system will to some extent be able to classify clear hand-written characters.

1.4 Possible uses

To be useful in processing images that are not of optimal quality, as is often the case with for example scanned data, the system needs to be combined with other mechanisms that handle things such as character separation and noise. As a stand-alone recognition system the use is limited to situations where such pre-processing is not needed. It could for example be used, although perhaps

[2]The hand-writing examples in Figures 9 and 10 on page 29 all contain a vertical vowel, except 10(c), 10(d), and 10(f) which only have horizontal vowels.

not optimally (as the system does not use on-line, but off-line recognition), as an input method, recognising characters drawn by mouse, or a tablet pen. Such an input method could be used as it is, or it could be incorporated into other systems; for example, it could be used in an educational tool for learning how to read and write the Korean script, making it possible to easily look up characters by their appearance, and also making it possible to have the system check if a drawn character matches what is asked for, in for example a transcription exercise.

1.5 Conventions

Korean characters and words will be transcribed according to the character names defined in the Unicode Standard [25] unless there is an alternate more preferable spelling available.

The notion of a *line* throughout this dissertation is not restricted to its mathematical sense[3], but may refer to any kind of line or path.

Various synonyms to the word *simple* is used throughout this dissertation to describe the system developed. These refer not to the level of complexity in the system itself, but to its rather basic nature as compared to other more developed systems, such as those referred to in Section 1.1.

1.6 The Korean writing system

Korea has a population of over 70 million people, with almost 50 million in South Korea and just over 20 million in North Korea. [5] The one and same Korean language is used in both the North and the South with only minor dialectal differences. Korean minorities are also found in several countries, especially in China, North America, Japan, and the former Soviet Union. [16]

1.6.1 Background

The Korean writing system is structurally unlike any other writing system in the world. It has a solid history, starting in the year 1443, or the twenty-fifth year of the reign of King Sejong. Part of the entry dated 25-12-30[4] in volume 102 of the Annals of King Sejong (世宗實錄) reads as follows:

> 是月, 上親制諺文二十八字, 其字倣古篆, 分爲初中終聲, 合之然後乃成字, 凡干文字及本國俚語, 皆可得而書, 字雖簡要, 轉換無窮, 是謂《訓民正音》。 [1]
>
> ‘This month, His Highness personally created the Vernacular Script of 28 letters. They imitate the old seal script, and are divided into initial, medial, and terminal sounds. When combined they form characters with which any Chinese, as well as the rustic language of this country may be written. Although they are simple, the characters shift and change without end. They are called the “Correct Sounds for the Instruction of the People”.’

[3]“A line is a straight one-dimensional figure having no thickness and extending infinitely in both directions.” [27]

[4]Day 30 of the twelfth lunar month of the twenty-fifth year of King Sejong’s reign

This is the very first known record about the Korean script. It claims that the person who created the script was the reigning monarch at the time, King Sejong himself. One very widespread interpretation however, says that even though the entry attributes the invention to the king, the alphabet was actually created by a number of scholars who served as his advisers. It was the custom to ascribe accomplishments such as this to the monarch, even though the actual work was performed by other people. Recent studies however show the opposite — the script was in fact not a collaborative creation. No other accomplishments or inventions during King Sejong's reign were referred to in this way, as his 'personal creation'. This entry is unique in all the records for this period. King Sejong was a scholar of phonology and writing without parallel in Korean history, and the alphabet was indeed his personal invention. [16]

A little over two and a half years later in 1446, a document was published revealing the details of the new alphabet. The document was written by the king, and carried as its title the name of the letters themselves, *Hunminjeongeum* ("The Correct Sounds for the Instruction of the People", 訓民正音). The document itself was very short and contained not more than a few pages. [16] Below is the first few lines from it.

> 國之語音, 異乎中國, 與文字不相流通, 故愚民有所欲言, 而終不得伸其情者多矣。予爲此憫然, 新制二十八字, 欲使人人易習, 便於日用矣。 [14]
>
> 'The spoken language of this country differs from that of the Middle Kingdom, and it is incompatible with their written language. Therefore, even if the ignorant people want to communicate, many of them are unable express their thoughts. In answer to this piteous quandary, I have created 28 new characters; I wish for normal people to learn them with ease, that they might conveniently use them as a matter of course.' [2]

Attached to this work, was a much longer and detailed text called the *Hunminjeongeum haerye* ("Explanations and Examples of the Correct Sounds for the Instruction of the People", 訓民正音解例). This was however not written by the king, but by a group of scholars commissioned by him. [16] Together these documents constitute the very first detailed description of the Korean script, which in modern days is commonly referred to as Hangeul (한글). The transition to the new writing system did not happen over-night however, Chinese was still used in most government documents and professional writings during a long period even after the invention of the new alphabet. It was not until the beginning of the 20th century that it started to become more widely used, and although Hangeul is indeed the main script of Korea today, Chinese writing is still used to some degree in a variety of contexts including for example academic writing and names, and Chinese characters are still taught in school.

1.6.2 Structure

Hangeul is a syllabic script where each character is built up from a set of consonant and vowel segments, known as *jamo* (자모). These are grouped into three classes: *initial* (choseong/초성), *medial* (jungseong/중성), and *final* (jongseong/종성).[5]

[5]For readers who are familiar with Chinese, 'cho' (초), 'jung' (중), and 'jong' (종) correspond to 初, 中, and 終 respectively. 'seong' (성) is 聲, or 声 in its simplified version. Jamo (자모)

The jamo of each class are shown in Table 1. The medial jamo represents a vowel, and the initial and final jamo represent consonants. An exception however is the jamo ㅇ, which, when placed in the initial position, works as an empty place holder to indicate that there is no leading consonant. When ㅇ appears as a final jamo, it represents the consonant /ŋ/, transcribed 'ng'. [8] To exemplify, the character 강 (ㄱ + ㅏ + ㅇ) would be transcribed as 'gang', whereas the character 앙 with the same medial jamo ㅏ (a) would simply become 'ang'.

Modern Hangeul syllable blocks are expressed with either two or three jamo, either in the form initial + medial or in the form initial + medial + final. There are 19 initial, 21 medial, and 27 final jamo. $19 \times 21 = 399$ possible two-jamo syllable blocks together with $19 \times 21 \times 27 = 10\,773$ possible three-jamo syllable blocks gives a total of 11 172 Hangeul syllable blocks. [25] Out of these, around 3 000 characters are used daily. [12]

The arrangement of the jamo within a syllable is in essence controlled by the vowel. Vowels can be of horizontal or vertical type, or a combination of both arranged in something like a mirrored 'L' shape. The final jamo can be either a single jamo like ㄴ, or a combination of two like ㄴ + ㅎ → ㄶ. Figure 1 shows all possible arrangements of jamo in a syllable block. As a structural example, the word 'Hangeul', written as 한글, consists of two syllables. The first uses layout 1(d) with the three jamo ㅎ (h), ㅏ (a), and ㄴ (n); the second uses layout 1(e) with ㄱ (g), ㅡ (eu), and ㄹ (l). [9, 15]

The logical structure of Hangeul allows an elegant representation in Unicode. Its canonical nature enables dynamic composition and decomposition of syllable blocks. Unicode contains both the complete set of pre-composed modern Hangeul syllables and the set of conjoining Hangeul jamo. The code point of a pre-composed syllable can be algorithmically determined from the code points of its jamo, and vice versa. In other words, ㅎ, ㅏ, and ㄴ may be composed into the syllable block 한, which in turn may be decomposed to bring back those same jamo. Unicode uses the terms L (leading), V (vowel), and T (trailing) to refer to the initial, medial and final jamo. [6, 25]

is 字母.

Table 1: Jamo classes

(a) Initial jamo (L)

Jamo	Transcr.
ㄱ	G
ㄲ	GG
ㄴ	N
ㄷ	D
ㄸ	DD
ㄹ	R
ㅁ	M
ㅂ	B
ㅃ	BB
ㅅ	S
ㅆ	SS
ㅇ	(empty)
ㅈ	J
ㅉ	JJ
ㅊ	C
ㅋ	K
ㅌ	T
ㅍ	P
ㅎ	H

(b) Medial jamo (V)

Jamo	Transcr.
ㅏ	A
ㅐ	AE
ㅑ	YA
ㅒ	YAE
ㅓ	EO
ㅔ	E
ㅕ	YEO
ㅖ	YE
ㅗ	O
ㅘ	WA
ㅙ	WAE
ㅚ	OE
ㅛ	YO
ㅜ	U
ㅝ	WEO
ㅞ	WE
ㅟ	WI
ㅠ	YU
ㅡ	EU
ㅢ	YI
ㅣ	I

(c) Final jamo (T)

Jamo	Transcr.
ㄱ	G
ㄲ	GG
ㄳ	GS
ㄴ	N
ㄵ	NJ
ㄶ	NH
ㄷ	D
ㄹ	L
ㄺ	LG
ㄻ	LM
ㄼ	LB
ㄽ	LS
ㄾ	LT
ㄿ	LP
ㅀ	LH
ㅁ	M
ㅂ	B
ㅄ	BS
ㅅ	S
ㅆ	SS
ㅇ	NG
ㅈ	J
ㅊ	C
ㅋ	K
ㅌ	T
ㅍ	P
ㅎ	H

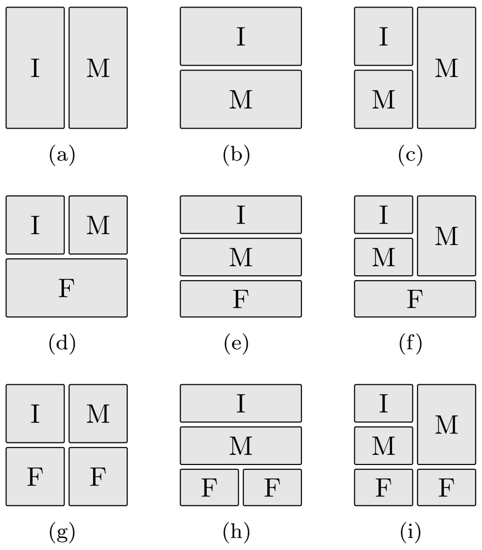


Figure 1: Jamo placement

2 Method

The initial idea was to create a system which would thin down the input image to a skeleton and perform a simple brute-force nearest-neighbour analysis, matching each point in the input to its corresponding closest point in a known reference character. While this method may have yielded decent results, it would have been limited to recognizing characters of almost the exact same appearance as the reference character, and would so be quite prone to error with even the slightest change of for example in-character element positioning. Also, even though there are just a handful of grapheme (jamo) types contained within a certain character, a system unable to identify these individually would have to, for each input character, search through a huge set of reference structures containing at least one definition for each of the 11 172 possible Hangeul syllables, which would be an unnecessarily resource-intensive task. With this in mind, a different approach was chosen which identifies and classifies the individual graphemes separately. Similarities can be found in the methods presented by Kim and Kim [12] and Kang and Kim [10].

This section contains a detailed description of the recognition process. There may be slight differences between the descriptions given here and the actual implementation. These differences are not relevant for the algorithms themselves, and are used only to simplify some of the descriptions.

2.1 Design overview

Simply described, the system works by matching each element found in the unknown image to an element in a known reference. The system must be able to correctly match two elements of the same type even if there are differences in size, proportion or position. This requires an image representation that allows comparison on a more abstract level, as opposed to blindly looking at the raw image data.

Generating such a representation is the task of the first of two main parts of the recognition system. All of the reference images (each corresponding to one jamo variant) are read, and a collection of hierarchical graphs is built from each image. The same is done for the input image to be classified. Section 2.2 describes the process of building graph structures from an image. Section 2.3 explains the layout of the jamo reference database.

In the second part, the representation of the unknown image is matched against the representations of the reference images to find the best possible match. The jamo associated with the matching reference images are then assembled to a full character. This is described in Section 2.4.

The system is implemented in Java using an object-oriented approach.

2.2 Image representation

To be able to successfully match two images, a representation that ignores insignificant variation between the two images is desired. In this work, an image is represented by a collection of groups of interconnected lines, where each group of lines and each line itself is represented by a graph. The following subsections describe what this means and how it is done.

The process of analysing an image can be outlined in six steps.

In the first step, as described in Sections 2.2.1 through 2.2.3, the input raster image is loaded and run through a graph thinning algorithm to create a skeleton representation. Figure 2(h) on page 12 shows the skeleton extracted from an input image that looks like Figure 2(a).

Step two, described in Section 2.2.4, extracts the individual lines (paths) from the skeleton, separating at line ends and crossings. Three lines are found in Figure 2(h), two vertical lines and one horizontal, meeting at a three-way crossing in the middle. Each line is represented as a separate graph of nodes connected by edges. Connecting lines share the node forming their connection, that is, the node in the crossing of 2(h) is part of all three lines. After line extraction the underlying skeleton representation (referred to as the *graph matrix* below) is no longer needed, so it may be discarded.

Step three analyses each line and assigns a line type (Section 2.2.5) based on its shape. The three lines extracted from 2(h) would each simply be assigned type 'vertical' or 'horizontal'. All possible line types can be seen in Figure 5 on page 16.

In step four (Section 2.2.6), the list of lines is traversed to find groups of adjacent lines. These groups of lines are too modelled as graphs, but with each node representing a line, and each edge representing a connection between two lines. The lines from 2(h) would so result in one group with three nodes, one per identified line, connected with a total of three edges.

Step five is merely an attempt to reduce possible artefacts introduced in the thinning process as a result of for example jagged edges in the input image. Every line found too short in relation to the other lines in its line group is removed. Section 2.2.7 explains this.

Step six adds additional meta data to the edges in the line group graph by assigning node ports, each corresponding to one of eight compass directions. An edge forms a link between two nodes, and the port information tells how the lines represented by these nodes connect. This is explained in Section 2.2.8.

The result is a collection of line groups, each group containing a graph describing the relation between the lines contained within it, and each line in turn represented by a graph modelling its shape. Meta data has been added showing the types of the lines contained within each group and how they connect to one another.

2.2.1 Reading the image

The colour values of the input raster image are sampled at points of a rectangular grid. Spacing between sample points is automatically adjusted in order to keep the number of samples fairly constant regardless of image dimensions. A large image will so be sampled at a lesser frequency than a small image, making it theoretically possible to load an image of any size[6] without much difference in processing speed (however with compromises in detail).

Each sampled colour is passed through a filter to determine if its value is positive or negative. For this application the filter simply compares the grey level to a threshold, yielding positive if the sample is below set threshold (colour is darker), and negative if above (colour is lighter). The resulting values are stored in a binary matrix, referred to as the *graph matrix*. Figure 2(b) shows

[6]Limited to available memory, as the entire image is stored on the heap

an example of values retrieved for the input image in Figure 2(a). White dots represent the positive values, negative values are not shown.

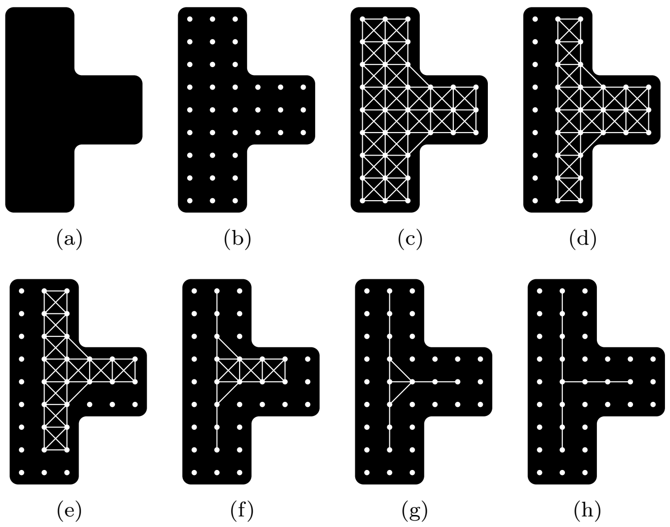
2.2.2 Graph matrix

The graph matrix represents a graph of 8-connected binary cells[7]. The sampled values retrieved in Section 2.2.1 are stored as the cells in this matrix, and at the same time each cell is connected to each of its neighbouring cells through an edge. One cell has a minimum of zero and a maximum of eight connecting edges. This creates a graph similar to Figure 2(c). Note that the illustrations in Figure 2 use a very low resolution (sample frequency) for demonstration purposes. In reality, the matrix may well be of the same resolution as the input image itself, with each cell corresponding to one pixel.

2.2.3 Thinning process

The thinning method used is a slightly modified and simplified version of the one described by Suzuki [23] and Suzuki and Ueda [24]. Figure 2 illustrates the thinning process step by step.

The algorithm traverses the graph matrix and removes cells that are part of the outer contour of the graph. Whether or not a cell is part of an outer contour is determined by its *connectivity number*, which is calculated based on the relation between the cell and its neighbours.


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

(e) (f) (g) (h)

Figure 2: Thinning process

Cell removal

The cell removal procedure is divided into four sub-cycles to prevent two-cell wide 'piers' (where all cells belong to the graph contour) such as can be seen

[7]The nodes in the graph matrix are referred to as cells to distinguish them from the nodes of graphs described later.

in Figures 2(d), 2(e), and 2(f) from vanishing. Each sub-cycle removes a group of cells on a single side of the graph. Cells and connecting edges on the left, bottom, right and top sides of the graph are subsequently deleted until there are no more cells that can validly be removed. The centre cell of any T-crossing, such as the one in Figure 2(g), is purposely left intact to prevent unwanted deformation. The extraneous diagonal lines that are left behind because of this are located and removed later.

Connectivity number

To determine if a cell is part of the graph contour, we look at its connectivity number. The connectivity number for a cell c is calculated by examining the presence of each of its eight neighbours n_0 through n_7. The neighbour positions are shown in Figure 3. We set $n_p = 1$ if a cell has a neighbour at position p, otherwise we set $n_p = 0$. To keep the calculation simple, we also define $n_8 = n_0$. All cells with a connectivity number of 1, except those with only one neighbour (i.e. line ends), will be removed.

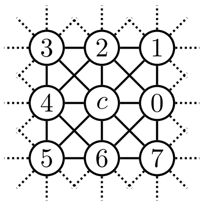


Figure 3: Cell neighbours

From this, we start by initialising the result value to zero. Then for each existing neighbour n_k of c, if there is no neighbour n_{k+1}, the result is incremented by 1. This will yield a value greater than zero for any cell part of the graph contour. However it will also yield a value greater than zero, namely 1, for a cell in a 90-degree concave corner. Given the structure of the graph, we can conclude that it is impossible for such a cell to be part of a concave contour line because it is always locked in by a hypotenuse running between its two perpendicular neighbours n_{2k} and n_{2k+2}.[8] Thus it should not be marked as a contour cell. Nevertheless, if this cell at the same time forms the opposite convex corner, then it is indeed part of the outer, convex contour. Since we still have the aforementioned hypotenuse on the concave side, it is safe to mark the convex corner cell for removal. The current value for a cell in this position will be greater than or equal to 2.

Both of these unwanted conditions are put into order by decrementing the value by 1 for every 90-degree concave corner the cell is part of (if the cell constitutes the centre of an intersection it can be thought of as part of multiple concave corners). In other words, if the cell has neighbours n_{2k} and n_{2k+2}, but not n_{2k+1}, we decrement the value by 1. This will give a value of zero for cells that are "concave corners" (thus not corners at all), and a value of 1 for cells that are convex corners.

The calculations so far can be expressed as shown in (1).

[8]Multiplying k by 2 ensures that we are dealing only with neighbours positioned at angles 0°, 90°, 180°, and 270°.

$$\sum_{k=0}^{7} n_k(1 - n_{k+1}) - \sum_{k=0}^{3} n_{2k}n_{2k+2}(1 - n_{2k+1}) \tag{1}$$

Furthermore, it is desirable to preserve the 90-degree angles in T-crossings. So far the cell in the intersection of such a crossing gets a connectivity number of 1, which means it will be removed, thus making the crossing slightly deformed. To prevent this from happening, the connectivity number is set to zero for any cell with exactly three edges (neighbours), arranged in the shape of a T at any angle. The state of a cell having edges that form one or more such T-shapes, can be expressed as shown in (2).

$$\sum_{k=0}^{7} n_k n_{(k+2) \bmod 8} n_{(k+4) \bmod 8} > 0 \tag{2}$$

To quickly summarise, the connectivity number for a cell is calculated using the formula shown in (1). If however the cell has exactly three edges, and condition (2) is true, the connectivity number is set to zero.

Extra edge and cell deletion

The last task in the thinning process is to eliminate the superfluous edges and cells that are left. Figure 4 shows the four situations in which these appear. The situation shown in Figure 4(a) also appears in Figure 2(g). The dashed lines represent edges and cells that will be deleted. The unlabelled cells indicate that there must be a cell connected for the particular situation to appear, the cell position (edge direction) however is not important.

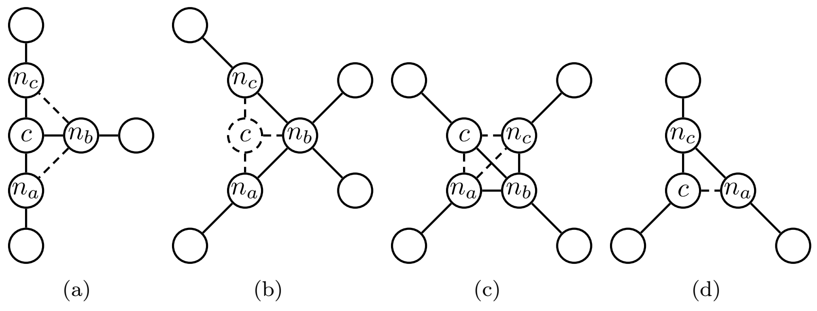


Figure 4: Extra edge and node deletion

The first two situations 4(a) and 4(b) would have already been taken care of in the preceding cell removal process if this was not blocked by condition (2) as mentioned previously. Cell c would have been deleted in both cases. While this is desirable in the latter case it would not be appropriate for the former, which is why both situations are handled here. Situations 4(c) and 4(d) appear naturally regardless of condition (2).

The implementation is fairly straightforward and works by simply checking the number of edges and their positions. To prevent clashes, situations 4(a) and 4(b) are exhaustively sought for and corrected before processing 4(c) and 4(d).

Following is a brief description of the search conditions. Cell reference numbers correspond to the illustrations.

Case 4(a) Find a T-crossing by checking the existence of $n_a = n_{2k}$, $n_b = n_{2k+2}$, and $n_c = n_{2k+4}$. If n_b has *four or less* connecting edges in total, delete edges $n_b \bullet\!\!-\!\!\bullet n_a$ and $n_b \bullet\!\!-\!\!\bullet n_c$.

Case 4(b) Find a T-crossing by checking the existence of $n_a = n_{2k}$, $n_b = n_{2k+2}$, and $n_c = n_{2k+4}$. If n_b has *more than four* connecting edges in total, delete c itself and all of its connecting edges.

Case 4(c) If c has neighbours $n_a = n_{2k}$, $n_b = n_{2k+1}$, and $n_c = n_{2k+2}$, and there is an edge $n_a \bullet\!\!-\!\!\bullet n_c$, delete edges $c \bullet\!\!-\!\!\bullet n_a$, $c \bullet\!\!-\!\!\bullet n_c$ and $n_a \bullet\!\!-\!\!\bullet n_c$.

Case 4(d) If c has neighbours $n_a = n_{2k}$, and $n_c = n_{2k+2}$ (but not $n_b = n_{2k+1}$), and there is an edge $n_a \bullet\!\!-\!\!\bullet n_c$, delete the edge $c \bullet\!\!-\!\!\bullet n_a$.

2.2.4 Line extraction

A collection of lines is created from the thinned graph. For this the graph matrix is first lifted over to an object-based graph representation for more convenient handling. Lines are then extracted by simply iterating through the edges in the graph assigning each to the line of its neighbouring edge, creating new lines as necessary. A more detailed description of the algorithm follows. The notion of E_{left} and E_{right} in the steps below is for illustrative purposes, in the implementation this is represented by directed edges, each edge having a to node and from node. Also, not visible in the description below is on-the-fly reversion of edges that are oriented against the stream, which is simply a matter of reversing the edge if it is oriented head-to-head or tail-to-tail to its neighbour. As the direction of edges or lines is not determined from the input image however, the direction is irrelevant, and is only maintained for ease of handling. A *valid* edge in the notion below is an edge that belongs to the same line as the edge E. This is found by reading the degree[9] of its to or from node (depending on direction of traversal); if the degree is 2, and $E \neq E_{start}$, then the edge is considered a valid edge.

The algorithm is as follows:

Step 1. If there is an edge E_{new} that does not belong to a line, set $E_{start} = E_{new}$, otherwise return the collection of lines and terminate the algorithm.

Step 2. Create a new line L and set $E = E_{start}$.

Step 3. Add E to the beginning of L.

Step 4. If there is a valid edge E_{left} to the left of E, set $E = E_{left}$ and go to step 3.

[9]Degree = number of connected edges

Step 5. If there is a valid edge E_{right} to the right of E_{start}, set $E = E_{right}$, otherwise go to step 8.

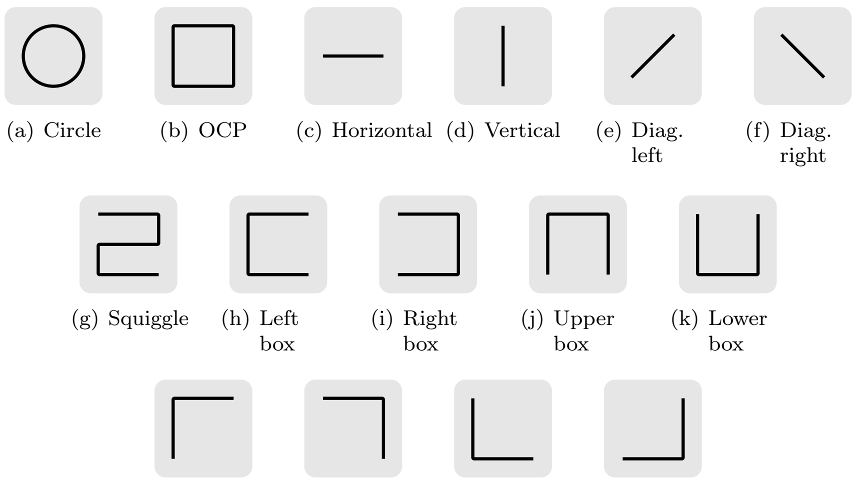
Step 6. Add E to the end of the line.

Step 7. If there is a valid edge E_{right} to the right of E, set $E = E_{right}$ and go to step 6.

Step 8. Add L to the collection of lines and go to step 1.

2.2.5 Line types

Each line found in the input is classified to be one of 16 different line types. Figure 5 shows all but the last type *unknown*, used for lines which cannot be classified as any of the first 15. Line type is determined by looking at a set of properties found by comparing measurements of the line with predefined threshold values. The ordering of the illustrations corresponds to the order in which each line type is tested for, if a match is found no further measurements will be made to the input line in question. The properties used to define each line type are listed in Table 2. Below is a description of each property.



(a) Circle (b) OCP (c) Horizontal (d) Vertical (e) Diag. left (f) Diag. right (g) Squiggle (h) Left box (i) Right box (j) Upper box (k) Lower box (l) Top left corner (m) Top right corner (n) Bottom left corner (o) Bottom right corner

Figure 5: Line types

Ellipses

We start by finding out if the line is a closed polygon by simply checking if its last node is a neighbour of its first node. If it is, the line is measured to see if it has an ellipse-like shape, line type 5(a). This is done through calculating the variance in distance from its centre point to each node, and then comparing this value to a set threshold.

The polygon is first normalised. The centre point c of a (non-rotated) rectangle enclosing the polygon is found, and the distance from this point to each node in

Table 2: Line type properties

Circle	5(a)	Closed polygon; Elliptical
Other closed polygon	5(b)	Closed polygon; Not elliptical
Horizontal	5(c)	Straight; Average angle 0° or 180°
Vertical	5(d)	Straight; Average angle 90° or 270°
Left diagonal	5(e)	Straight; Average angle 45° or 225°
Right diagonal	5(f)	Straight; Average angle 135° or 315°
Squiggle	5(g)	Baseline intersection
Left box	5(h)	First and last fifths are horizontal; Left-balanced
Right box	5(i)	First and last fifths are horizontal; Right-balanced
Upper box	5(j)	First and last fifths are vertical; Top-balanced
Lower box	5(k)	First and last fifths are vertical; Bottom-balanced
Top-left corner	5(l)	End node location match; Upper fifth is horizontal
Top-right corner	5(m)	End node location match; Upper fifth is horizontal
Bottom-left corner	5(n)	End node location match; Lower fifth is horizontal
Bottom-right corner	5(o)	End node location match; Lower fifth is horizontal

the polygon is measured. The angle from c to its closest node is used to rotate the polygon so that this node faces north. The entire polygon is then squeezed together so that both sides of the enclosing rectangle are of equal length. This procedure is repeated 5 times to make the polygon as comparable to a circle as possible. The shape is then resized so that it fits perfectly inside a square with a side length of 100 units. Finally, the variance[10] in distance from the centre point to each node is calculated. If the variance is found to be less than or equal to a threshold of 20, then the polygon is defined as being ellipse-like, or a 'circle' as it is termed. The threshold value was found through trial and observation.

If the line is not found to have an ellipse-like shape but still is a closed polygon, it is defined as an *other closed polygon*, type 5(b). Note that the square shape of the line in 5(b) is arbitrarily chosen, it may have any shape as long as it is not ellipse-like.

Line straightness

The measurement of line straightness is used in every line type definition except 5(a), 5(b), and 5(g). The degree of straightness is determined by dividing the bird distance between the endpoints of the line with its actual length (sum of the length of all edges). The line is defined as being straight if this ratio is higher than 0.85. This number was also found empirically.

In cases 5(c) through 5(f) the line as a whole is measured. In the box-type lines 5(h) through 5(k), the first and last fifth are tested, what is in between is considered irrelevant. In corner-type lines 5(l) through 5(o), only the fifth at the uppermost or lowermost end of the line is tested, depending on line type.

[10]Variance is $\frac{\sum_{i=0}^{n-1}(x_i-\bar{x})^2}{n}$ where n is the total number of nodes, x_i is the distance between node i and the centre point, and $\bar{x}$ the distance mean.

Average angle

The average angle, or perhaps more appropriately the average direction of a line is calculated to determine the orientation of a (possibly slightly crooked) line. This is used to differentiate between horizontal, diagonal, and vertical lines or line fractions.

Expressing the angle θ of each edge as a vector $\vec{v} = \langle \cos\theta, \sin\theta \rangle$ makes it simple to calculate the average direction of all edges. Given a line with n edges, and θ_k being the angle of edge k, (3) shows how the average is calculated. The $\operatorname{atan2}(y, x)$ function is used in converting from rectangular (x, y) to polar (r, θ) coordinates. It is similar to calculating the arc tangent of y/x, but respects the quadrant of the result by looking at the signs of both arguments.

$$\operatorname{atan2}\left(\sum_{k=0}^{n-1} \sin\theta_k, \sum_{k=0}^{n-1} \cos\theta_k\right) \tag{3}$$

To define horizontal, vertical, left diagonal, and right diagonal, a slice resolution of precisely an eighth of a circle is needed. Thus an angle deviation of $\pm\frac{1}{8}\pi$ (a sixteenth of a circle) is accepted when comparing the angle of the measured line with the angle defined in the line type.

Baseline intersection

The baseline intersection check is only used for squiggle detection, line type 5(g). The use of the term *squiggle* to describe the look of a line may seem ambiguous; the reason is simply that the definition itself accepts a great variety of, well, squiggles. There is only one kind of squiggle-like line in (proper, non-sloppy) Hangeul, namely the one found in the jamo ㄹ (r/l) and its compounds. This allows for a very forgiving definition that covers most of its different styles. In some fonts, and especially in hand-writing, the line may become simplified to more or less resemble the shape of a 'z'. Slight rotation and variation in curvature and proportion also occurs.

A line that has not been identified as a closed polygon or straight line is checked for baseline intersection. The baseline is an imaginary straight line drawn between the two endpoints of the line. If any of the middle two fourths of the line intersects this baseline exactly once, the line is defined to be a squiggle. There are no other tests than this to define a squiggle. Worth noting however is that if the line intersects the baseline more than once, it is not defined as a squiggle simply because such lines do not exist in printed Hangeul.

Balance

The balance properties define the overall balance of the input line relative to its endpoints. This is used to differentiate between 5(h) and 5(i); and 5(j) and 5(k) respectively.

A straight line passing through the endpoints of the input line is used for reference, we call this the centre line. The nodes of the input line are traversed, and the position of each node is compared to its corresponding points in the centre line, one for each axis x and y. The differences along the x and y axes for all nodes are summed separately, each resulting in a number which may be negative, zero, or positive. If the x axis sum is negative, the line is said to be balanced to the left. If positive, the line is balanced to the right. If zero, the line is neither left nor right-balanced. Consequently for the y axis, the line is said to be bottom-balanced if the sum is negative and top-balanced if the sum is positive, neither nor if the sum is zero.

End node location

Lastly, to distinguish 5(l) from 5(m), and 5(n) from 5(o), the position of the topmost end node of the input line is compared to the centre point of the line's bounding rectangle. If the node is left of the centre point, the line type must be 5(m) or 5(n), if the node is right of the centre point, the line type must be 5(l) or 5(o). The orientation of the uppermost fifth (for 5(l) and 5(m)) or the lowermost fifth (for 5(n) and 5(o)) is then used to decide which of the two resulting types is the correct one.

2.2.6 Grouping of lines

The collection of lines extracted in Section 2.2.4 is searched to find lines that are adjacent. These are grouped into line groups. Two connecting lines always have a common head or tail node, making it easy to determine which lines are adjacent.

A simple stack-based approach is used:

Step 1. If there is a line that is not part of a line group, add it to the stack. Create a new line group and assign its (empty) graph to G. If there is no line not yet part of a line group, return the collection of line groups and terminate the algorithm.

Step 2. If the stack is empty, add the line group containing graph G to the collection, then go to step 1. If the stack is not empty, pop a line from it. If this line is not already part of a line group, create a node to represent it and add this node to G, otherwise repeat step 2.

Step 3. Find all neighbours of the line through examining its end nodes, and add them to the stack. Go to step 2.

The nodes of all line groups are then traversed, and nodes whose lines connect are linked with an edge.

2.2.7 Removal of short lines

As a simple measure to reduce some of the artefacts introduced by jagged edges or other noise, the groups of lines are traversed to find and eliminate comparatively short lines. Vertical lines are left intact because of the high possibility that these, although short, may actually be part of the jamo. Any other line shorter than 20 percent of the median line length in its group, is removed. The threshold percentage value was found by trial and observation.

2.2.8 Edge port assignment

Every edge in the graph of a line group, is assigned a head port and a tail port. These are used to define the relative locations of the lines in the graph. Possible ports are *centre*, *east*, *north-east*, *north*, *north-west*, *west*, *south-west*, *south*, and *south-east*. The port *centre* is used for every line type except the straight lines, 5(c) through 5(f), for which a direction is always specified. The assignment of a directed port is straightforward. The port is found by looking at the line type of each line in combination with what end nodes are shared between the two. For instance, if the rightmost end node of a horizontal line is the same node as the topmost end node of a vertical line, the edge will connect the east port of the horizontal line with the north port of the vertical line. Note that for this kind of connection to even exist, there must of course also be a third line connecting, otherwise the two lines would have been identified as one single curved line.

2.2.9 Representation examples

Figure 6(a) shows an example of an input image. This is Hangeul syllable 찮 (canh) rendered in the font AppleGothic. Thinning the image results in the skeleton seen in 6(b). Lines are extracted and grouped; there are 7 line groups in this image, and a total of 13 lines. Each line has an assigned line type (Figure 5 on page 16). In this image, line types *horizontal* (5(c)), *vertical* (5(d)), *diagonal left* (5(e)), *diagonal right* (5(f)), *bottom-left corner* (5(n)), and *circle* (5(a)), are found. Figure 6(c) illustrates the top-level graph representing the line groups and lines. The points, or nodes, in Figure 6(c) each represent a line, and the edges in the graph show how the lines are connected. Figure 7 shows the same information as Figure 6, but with the extracted data overlaid atop the input image for better clarity. Neither line types nor ports are shown in these figures; some more detail can be seen in the graphs in Figure 8 on page 24, which use the same format as Figure 6(c) but with meta data attached.

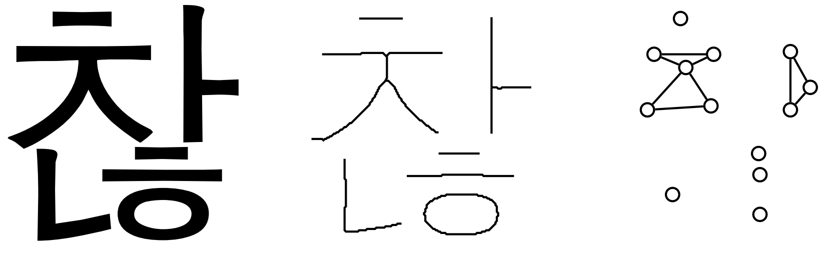


Figure 6: Image representation example

2.3 Reference data

A reference database holds information about every jamo in Table 1. It is used in mapping unknown elements in the input image to find out what jamo they resemble. Each jamo is stored with its Unicode character, its class (initial, medial,

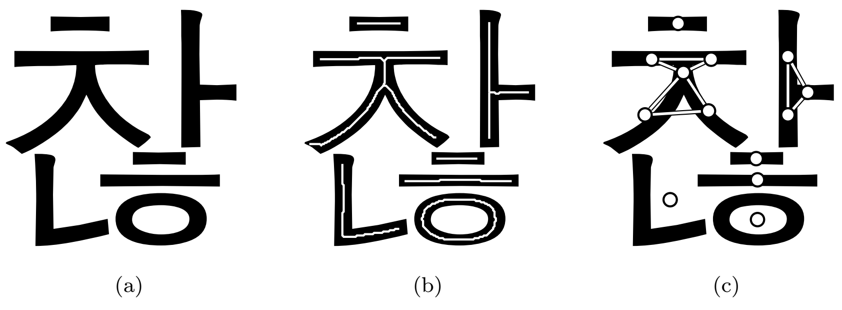


Figure 7: Image representation example with data overlaid input image

or final), and information about its different glyph variants. The glyph variants are kept as graph structures, as described throughout Section 2.2.

By using the jamo glyphs found in one or more font files the structure generation can be fully automated; glyphs can be rendered in different font sizes to create some variation. Using manually drawn glyphs however allows for more forgiving definitions (more variants) and avoids unnecessary redundancy, which is why this approach is preferred and chosen. Table 3 in Appendix A contains the original structure definitions, drawn with respect to the different line types. Rasterised versions of these are read and analysed by the system.

Each resulting structure contains one or more line groups. There is no limit to how many line groups can be contained in a single structure. ㄴ (n) for example has a structure with only one line group (even though there is only one line contained, it is still referred to as a group), while a structure describing ㅞ (we) may consist of two or three line groups, depending on character style.

A lookup table connects each structure to its corresponding jamo. The structures are sorted in descending order primarily by the number of line groups contained, and secondarily by the total line count in each structure. This ordering removes the need of iterating through the entire list of structures if a match is found early.

2.4 Graph matching

The unknown input is matched against the reference structures (Section 2.3) and the retrieved jamo are combined to create a Hangeul syllable.

The sorted list of structures is traversed, and for each structure an attempt to find the best possible match between its line groups and line groups found in the unknown input is made. If a match is found, the structure is added to a list of candidates for further consideration. The candidates are compared to one other and a winner is chosen; the corresponding jamo of which is added to a result set. This continues until there are no more matches to be made. The resulting jamo are combined using simple rules to create a complete Hangeul syllable, which is then translated into Unicode. The following subsections describe the details.

2.4.1 Structure mapping

The structure mapping process runs on two levels. The first level deals with line group matching. That is, the task of finding out if the graph of one line group matches that of another line group, with regard to line types and edge ports. The attributed graphs are mapped using a somewhat altered version of an algorithm for determining sub-graph isomorphism described by Ullmann [26], the details of which are explained later. The result is something that can tell if two line groups represent roughly the same thing.

A line group is, to remind, a collection of lines which are physically connected in the image. The jamo ㅓ (eo) contains one line group, ㅔ (e) contains two. The input, resembling (hopefully) a full Hangeul syllable can contain a lot of groups; 챦 (canh) for example might have seven groups, although the number of groups of course depends on the character style. If each jamo consisted only of a single line group, finding them would not need much more than the above. It is often the case though that they do contain more than one group; handling this is the task of the second level.

The second level uses the same algorithm (Ullmann [26]) again, to find all possible ways that the line groups in a structure can be mapped to line groups in the input. All line groups in the structure and the input respectively are, for the algorithm, said to be adjacent to each other, and if the graphs of two groups are similar enough they are set up as a possible match. The list of candidates returned by the algorithm is filtered to remove mappings considered improper, through examining a set of characteristics as described below.

The relative positions of all the line groups are compared; for each line group in a set, the angle to every other line group in the same set is found. This is done for both the structure and the input line groups. The angle of each line group pair is compared to the angle of the corresponding mapped line groups, and if there is a difference of more than set threshold $\pi/6$ between any two pairs of groups, the candidate is discarded. This is to prevent multiple occurrences of simpler jamo from being swallowed by compounds, such as the ㅎ (h) and ㄴ (n) in 한 (han) being incorrectly classified as the final jamo ㄶ (nh).

The input line groups are also checked to see if there are any unrelated lines blocking the space in between them. A web of lines is drawn, connecting the centre points of all line groups in the input that are mapped to the structure. The candidate is discarded if anything not mapped to the structure touches any of these lines. This check is needed to be able to correctly classify characters such as 층 (ceung), which would otherwise be incorrectly classified as ㅎ (h) + ㅈ (j), instead of the proper ㅊ (c) + ㅡ (eu) + ㅇ (ng).

Next, for each candidate the distances between all of its input line groups are summed up to get a value indicating group proximity. Any candidates with a lower group proximity than that of the candidate with the most compact set of line groups, are removed. This will probably remove all but one candidate, but in case there are more candidates left with the exact same proximity value, a final filter is used to remove all but one last candidate. It calculates a line length ratio for each line in every group. The length of each line is divided by the total length of all lines in its line group. The length ratio sum for each line group is subtracted from that of its mapped line group and the absolute value of this difference is accumulated for the entire mapping. The candidate with the lowest sum is the winner.

The winning candidate is added to a list of possible structure matches and the process is repeated until all of the defined structures have been tested. A second possible match will however only be considered if its mapping has the same number of line groups as the first match. This is to get a match that contains as many line groups as possible. Without this limit an occurrence of for example ㅔ (e) would indeed be matched by the structure ㅔ, but also by ㅓ (eo) and ㅣ (i), which is not desirable.

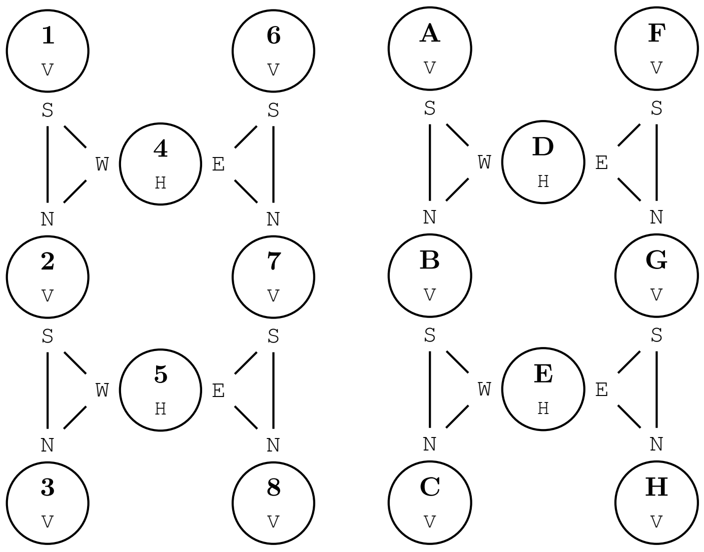
The last step is to select a single best match from the list of possible matches. The same line length ratio difference sum as described above is used, but it is modified to include the entire structure and its entire set of mapped input line groups.

The jamo associated with the single matching structure that is left is added to the set of matched jamo. The corresponding line groups in the input are erased. The entire process is then repeated from the beginning, until there are no more line groups left in the input that can be mapped to a structure.

2.4.2 The graph mapping algorithm

Graphs are mapped using a fairly simple but effective algorithm for finding (sub-)graph isomorphisms, described by Ullmann [26]. Given two graphs G_α and G_β, the algorithm will find all valid mappings between the former and the latter with respect to node compatibility and adjacency. The graphs in Figure 8 will be used as an example. Figure 8(a) shows the graph of the jamo ㅒ (yae); 8(b) shows the graph of an arbitrary line group found in the input that happens to be very similar to the graph in 8(a). The nodes in the graphs represent lines, markings `V` and `H` indicate line types vertical and horizontal. The edges attach to node ports, each marked with its location relative to the node using compass directions. This meta data helps to determine if two nodes are compatible. When mapping line groups, each node (representing a line) has to match by line type, and also by the head and tail port of all connecting edges. When mapping sets of line groups (that is, structures describing a full jamo, with each node representing a group), compatibility is simply determined by asking if one line group can be mapped to another.

M is a $p_\alpha \times p_\beta$ matrix describing which nodes in G_α (rows) may be mapped to which nodes in G_β (columns). In line group mapping, as in the example, the graphs are always of equal size, meaning $p_\alpha = p_\beta$. When mapping line group sets, $p_\alpha \leq p_\beta$, because a structure (jamo) may of course have fewer line groups than the total number of groups found in the input. The nodes in the example graphs are very distinctive as they are, there is for instance no other node in 8(a) than node 1 that has two connecting edges, both of which connect from its south port, one connecting to a north port, the other to a west port. The only nodes in 8(a) with identical meta data are nodes 4 and 5, making M, as seen in (4), very restrictive and uncomplicated.



(a) G_α (b) G_β

Figure 8: Graph mapping example

$$M = \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \end{pmatrix} \tag{4}$$

Based on and at the same dimensions as M with its elements m_{ij}, all possible matrices M' are generated, such that for each and every element m'_{ij} of M', $(m'_{ij} = 1) \Rightarrow (m_{ij} = 1)$, and where the elements of M' are 1s and 0s such that each row contains exactly one 1 and no column contains more than one 1. [26] Each M' represents one mapping of G_α to G_β. In this particular case there are only two possibilities, as seen in (5) and (6).

$$M^1 = \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \end{pmatrix} \tag{5}$$

$$M^2 = \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \end{pmatrix} \tag{6}$$

Generating these is in essence a depth-first tree search. It is implemented recursively as seen below. m_{ij} and m'_{ij}, as hinted above, represent the values of the elements at row (depth) i and column j of matrices M and M' respectively. M and M' both contain $p_\alpha \times p_\beta$ elements with top left entries at $i = 0$, $j = 0$. Each element in M' is initialised to zero. The depth d is also initialised to zero.

Step 1. Get the next column c of M. If this is the first iteration in the current call, get the leftmost column. If there is no next column, go to step 8.

Step 2. Go to step 1 if $m_{dc} \neq 1$.

Step 3. Go to step 1 if $m'_{ic} = 1$ for any value of i such that $0 \leq i < d$.

Step 4. Set m'_{di} to 0 for any value of i such that $0 \leq i < p_\beta$ and $i \neq c$. Set m'_{dc} to 1.

Step 5. If $d < p_\alpha - 1$, recurse, passing the matrix M' and a depth value (next d) of $d + 1$. A list of one or more matrices will be returned; add these matrices to the list that is to be returned by this call.

Step 6. If maximum depth has been reached ($d = p_\alpha - 1$), use condition (9) described below to see if M' is a valid isomorphism, and if so, add M' to the list of matrices to be returned.

Step 7. Go to step 1.

Step 8. Return the list of matrices.

Condition (9) is used to determine whether or not a certain matrix M' represents a mapping such that if two nodes in G_α are adjacent, the corresponding nodes in G_β are also adjacent. Adjacency matrices A and B respectively describe which nodes are adjacent in G_α, and which nodes are adjacent in G_β. All graphs are undirected, so the adjacency matrices will always be symmetric. Notations a_{ij} and b_{ij} are used to reference the elements in each matrix. The adjacency matrices (7) and (8) of the graphs in the example happen to be identical, since both graphs use the same node order.

$$A = \begin{pmatrix} 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 \\ 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 \end{pmatrix} \tag{7}$$

$$B = \begin{pmatrix} 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 \\ 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 \end{pmatrix} \tag{8}$$

It has to be determined if A can be matched to B under the current mapping M'. M' is used as a permutation matrix to rearrange the rows and columns of B, creating another matrix C. C, with element reference c_{ij}, is defined as $C = M'(M'B)^T$, where T denotes transposition. A can then be directly compared to C. If condition (9) is true, then M' specifies an isomorphism between G_α and G_β. [26]

$$\underset{\substack{0 \le i < p_\alpha \\ 0 \le j < p_\alpha}}{(\forall i \forall j)} (a_{ij} = 1) \Rightarrow (c_{ij} = 1) \tag{9}$$

Worth noting is that Ullmann [26] also describes a *refinement procedure* that removes some of the 1s from M beforehand, to reduce the amount of computation required. With optimisations being of secondary priority, this was left unimplemented.

2.4.3 Hangeul composition

The identified jamo are combined to form a complete Hangeul syllable. They are first sorted according to the vertical positions of the structures in the input image to which they are mapped, topmost first. Then follows a procedure to correct jamo that have been assigned an incompatible class. As mentioned in the introduction and as seen in Table 1 on page 8, some jamo appear in both initial and final position. Jamo of this kind exist at two code points in Unicode; one represents the initial variant, the other represents the final variant. For a concrete example, ㄱ is found at both U+1100 (*Hangul choseong kieok*) and U+11A8 (*Hangul jongseong kieok*).[11] The system as it is implemented will have classified the input to either one of these with equal probability, since they are

[11] ㄱ is also found among the *Hangul Compatibility Jamo* at U+3131. Unicode provides this set for compatibility reasons only. [25] It is not used in this work.

identical in appearance and thus share the same structure definition. It is likely that one or more jamo have been assigned the wrong class, i.e. initial instead of final or vice versa. This is put right by swapping any affected jamo with its counterpart. The jamo to be used in the syllable are also picked out and put into proper order.

Three slots are prepared to respectively hold one initial, one medial, and one final jamo. Once a slot has been filled it may not be changed. This means that the first (topmost) proper set of jamo found is what will be used to build the syllable, and any other jamo will be ignored. Changing the class of an initial or final jamo is done through simply replacing its character with its equivalent in the other class. The sorted list of jamo is traversed and as each class is encountered its corresponding slot is filled, in accordance with the steps below.

Step 1. Get the next jamo in the list. If there are no more jamo or all slots have been filled, exit.

Step 2. If the medial slot is free and the jamo is of class «medial», fill the slot and go to step 1.

Step 3. If the initial slot is free and the jamo is of class «initial», fill the slot and go to step 1.

Step 4. If the initial slot is free and the jamo is of class «final», see if the class can be changed to «initial», and if so, fill the slot and go to step 1.

Step 5. If the final slot is free and the jamo is of class «final», fill the slot and go to step 1.

Step 6. If the final slot is free and the jamo is of class «initial», see if the class can be changed to «final», and if so, fill the slot and go to step 1.

If, after traversal, at least the initial and medial slots are filled, a Hangeul syllable is composed by systematically adding the individual jamo code points together as demonstrated and explained by Davis and Dürst [6] and The Unicode Consortium [25].

3 Results

3.1 Printed input

The system was fed the entire set of 11 172 characters, each rendered as a 200-pixel-squared raster image using the gothic font AppleGothic Regular. 9 004 characters out of the 11 172 where correctly classified, giving an accuracy of 80.6 percent. 1 820 characters out of the 11 172 were found to contain conjoined jamo, which, as mentioned earlier, the system was not designed to handle. With these removed from the set, thereby reducing the total number of characters to 9 352, an accuracy of $9\,004/9\,352 = 96.3$ percent is reached.

A very common source of error among the remaining 348 characters is that sharp curves, such as the one in ㄱ (g), sometimes cause an additional line to be generated. This is to some degree prevented by the removal of short lines as described in Section 2.2.7. Another occurring error is the opposite, where lines are too short to be recognised as lines. An example is the jamo ㅅ (s), which when it appears at the top or bottom of a syllable such as in 수 (su) or 흣 (heus), is often so flattened out that its vertical top line is lost.

The system was found unable to classify the two jamo ㄹ (r) and ㅒ (yae) when found together in a syllable, because they very much resemble ㄼ (lb) as it is defined. There are 28 characters containing this combination.

Two other fonts were also tested, but due to the large amount of conjoined jamo the results are not comparable to the above. The results were 63.7 percent for the gothic font Dotum, and 54.5 percent for the rounded font Gulim. Gulim more or less eliminated the problem with extraneous line generation; instead it sometimes introduced new problems because of its rounding, giving line types different from those in the reference jamo structures.

Glyphs for the separate jamo (Table 1 on page 8) exist in all three fonts. Testing these individually gave an accuracy of 100 percent; all jamo were correctly identified, confirming that the misclassified characters in the above tests are due only to the jamo being combined into a syllable block, and to how they are then rendered slightly differently in shape and structure as compared with the isolated forms.

As also mentioned in Section 1.3 the system was not designed to handle images containing any kind of noise, and there is no value in conducting tests with such images; the accuracy would be zero percent. Section 4 discusses this further. Relevant however, is the tolerance against distortion. Heavily dependent upon accurate recognition of the individual line types (Figure 5 on page 16), the system fails to correctly identify any image with as little as one incorrectly classified line. On the positive side, as long as the correct line types are found and the structures they form together match those defined, the system is very tolerant to variance in element proportion and position; the individual elements may be of any proportion to each other, and the positions need only be such that the syllable composition can be logically determined by the vertical order of the jamo contained.

3.2 Hand-written input

Printed input being the primary focus, no exhaustive testing has been conducted for hand-written characters. A set of 52 hand-written characters was acquired from an unrelated external source. The characters were digitised and any noise was removed before feeding them to the system. 14 out of the 52 were correctly classified, giving an accuracy of 26.9 percent. The 14 correctly classified cases are all well-formed characters; the structures are defined in the reference and there are no structures that conjoin or overlap. Figure 9 shows some of the characters that were correctly classified.

Figure 10 shows examples of characters that were incorrectly classified or not recognised at all. Figures 10(a) through 10(c) contain jamo structures that are not found in the reference. The jamo in 10(d) through 10(f) are conjoined and can therefore not be identified at all.

Further examples of non-printed input can be seen in the screenshots of the running application, found in Appendix B.

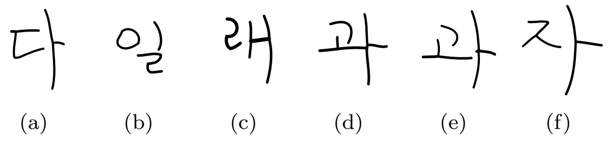


Figure 9: Examples of correctly classified hand-writing

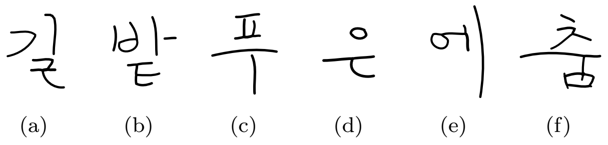


Figure 10: Examples of incorrectly classified hand-writing

4 Discussion

The character-recognition system presented uses very direct methods to analyse the input image. A skeleton representation of the input is created, and the features of this is used to find individual lines. The lines are then further classified to a set of pre-defined line types. With line types primarily defined by the angle of strokes, the system becomes very sensitive to rotation, unless rotated variants of the jamo structures are also defined. The system contains no learning mechanism, and the structure definitions (Appendix A) are constant. Characters containing undefined structures will so never be recognised without manual additions. An example is the character in Figure 10(c) in which the upper part (ㅍ) is recognised, but not its lower part (ㅜ) because there is no definition for this jamo that allows it to have separated lines. The use of this rather naive approach, where the system assumes the input to be a neatly drawn character conforming to the shapes defined, can be seen as a major drawback of this system. Furthermore, the lack of methods to handle noise, distortion, and conjoined jamo is of course another disadvantage of the system in general; even though it was not designed to handle such things.

In the real world, the current system is thus limited to applications where the aforementioned does not necessarily pose a problem. Used as part of an educational tool for the purpose of learning how to read and write the Korean script is one example of such an application, as also mentioned in Section 1.4.

An advantage of this system as compared to many other off-line character-recognition systems however, is its tolerance to variance in jamo proportion and position. The size of the individual jamo is not important; there are theoretically no limits to how small or how large a jamo may be relative to its neighbours, as long as it contains enough pixel data to be correctly recognised. Jamo positioning is also very free; a jamo may be positioned anywhere but in-between separate lines or line groups of another jamo.

4.1 Possible improvements

Within its limited scope the method works well. The recognition accuracy of 100 percent for the individual jamo in all three fonts that were tested indicate that for this type of data, the basic idea is sound. Yet there are of course many improvements and additions that can be made; a couple of examples are listed as follows.

Further modularisation where structures can be nested inside other structures would significantly reduce the number of reference images needed, and could possibly also increase performance. For example, ㄶ (nh) could be described as ㄴ (n) + ㅎ (h), and ㅙ (wae) could be described as ㅗ (o) + ㅐ (ae). ㅐ (ae) could in turn be described as ㅏ (a) + ㅣ (i).

Conjoined jamo are found in many characters and it seems that separating them is not a trivial task. First and foremost, there must exist logic to determine if an unknown structure is indeed a combination of two (or more) jamo. This could be partly solved by using sub graph mapping to determine if a certain known structure is part of the unknown structure. Problems with this occur when there are ambiguities, each case has to be clearly defined, and there are many cases that need to be handled. Jamo forming T-crossings such as seen in for example 니 (ㄴ + ㅣ) is possibly one of the easier cases. A solution could be to, after determining that the structure consists of two or more known sub-structures, allow it to be split only where it is obvious that a line connects to another. In other words, allowing disconnection of a line only if it is more or less perpendicular to the lines it is connected with. Figure 11 illustrates such a rule.

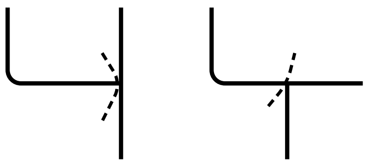

(a) Legal (b) Illegal

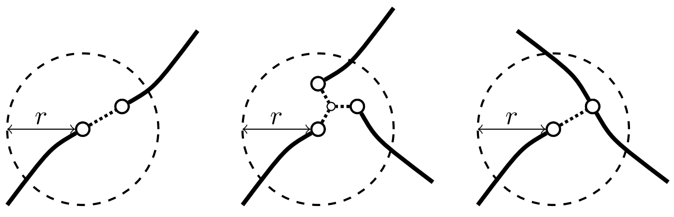
Figure 11: Line disconnection

On the other hand, characters such as the one mentioned in the introduction, 큘 (ㅋ + ㅠ + ㄹ), give rise to other complications. Even if the final ㄹ is dropped giving 큐, it is not perfectly clear how it should be split. Notice for example that the ㅋ is positioned above the ㅠ in such a way that the lower tip of ㅋ is almost precisely aligned with the rightmost vertical line in ㅠ, possibly forming a cross-like shape when thinned down to a skeleton. There are then theoretically plenty of matching structures; ㅣ, ㅓ, ㅏ, ㅗ, ㅜ to list a few. In practise the possible matches are not as many since there are certain limits to how the jamo may be connected with one another. This again however calls for another set of rules to determine when it is safe to ignore a possible match.

Another example is 표, where it is not entirely obvious, especially if all lines are conjoined, that the jamo contained are ㅍ + ㅛ and not ㅠ + ㅍ.

There are many other examples of the complications in separating conjoined jamo, but the above should give at least some insight.

Broken structures are another potential problem if for example the input is of bad quality and the lines are discontinuous, or if lines that are supposed to be connected in a structure are slightly detached. One possible solution to mend these lines or structures after they have been converted to a graph representation (Section 2.2) could be to examine the locations of the end points of every line, and link them to points which are within a set proximity. This may be best explained by an illustration.

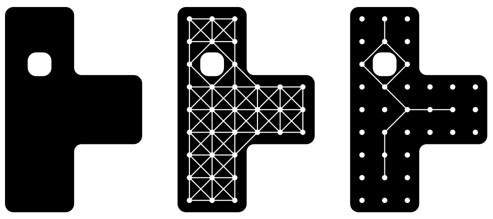


(a) Two nodes (b) Multiple nodes (c) In-line node

Figure 12: Proximity linking

Figure 12(a) shows linking the end points of two lines together to create a single whole line. r indicates the radius of the area to be sought for other nodes. Figure 12(b) shows linking multiple end points together, creating an additional node in the middle. Figure 12(c) shows how an end point of a line could be connected to a node in another (whole) line within reach.

Noise is another thing that can cause problems, especially so in this particular system. The system can without trouble deal with artefacts that are not in contact with the significant parts of the input, and which cannot be mistaken for a jamo or part of one. These artefacts are simply ignored because they cannot be mapped to any known structures. Any other noise or artefact would alter the structure of the character in the input image and would most likely prevent it from being properly recognised.



(a) (b) (c)

Figure 13: Thinning with noise

Figure 13(a) shows the same image as Figure 2(a) on page 12, but with noise introduced. For illustration purposes the white area representing the noise is very large, just as the nodes are very few because of the low sample frequency (resolution). Normally each node would represent one pixel of the input image, which also means that this piece of noise would only have to be one pixel in size. Noise that appears in the black areas like this is interpreted like any other hollow area, and the thinning algorithm will simply create a path around it. Given the graph in 13(b), the result of the thinning process will be like seen in 13(c). Needless to say, the resulting structure is quite dissimilar to its noiseless counterpart in Figure 2(h) on page 12.

Thinning an image containing a lot of noise would result in an entire mesh of loops like the one in 13(c), which would be very hard to handle. The preferable solution to this problem would thus likely be to filter the input image to remove all noise beforehand. No research has been made to find what algorithm(s) would be most suitable; it would of course depend on the kind of input. Additionally, although not related to noise, evening out sharp corners to prevent them from being mistaken for short lines would clearly improve the results.

To further improve the results, the thinned structure could be searched to find and eliminate loops or lines that are smaller than a set threshold, relative to for example the character size. The system does this to a small extent, as explained in Section 2.2.7.

Returning multiple matches ordered by the level of accuracy instead of returning only the single best match would make a slight improvement in combination with context analysis. The cases where ㄹ and ㅐ are together classified as ㄼ as mentioned earlier could be eliminated, because they all result in vowel-less characters which are not valid. Used as a sub-component when processing multiple characters, it would be advantageous if the system could return several candidates for each character, so that the best match may be chosen based on context.

5 Conclusion

This work has presented a method to extract and classify a Hangeul syllable from an input raster image. A graph thinning algorithm was used to thin down the image to a skeleton. A rule-based approach was then used to extract the individual lines and assign meta data, describing the lines themselves and how they connect to one another. The same procedure was used to scan a collection of reference images, each resembling a possible variant of one jamo. An algorithm for graph and sub-graph mapping was used in combination with a set of tests to retrieve the best matches from among the reference structures, and the corresponding jamo were assembled to form a full Hangeul syllable.

The first aim of this work was to create a working character-recognition system for the Korean script, Hangeul. While the usage area of the system is limited, it does perform beyond expectations for the kind of input data it was designed for (see Section 1.3 for details on scope).

The second aim was to illustrate the complexity involved in character recognition, while still trying to keep the system reasonably simple. Although the system as a whole may be perceived as slightly tangled, the methods employed are not very complex by themselves and should be easy to understand without much prior knowledge in the area. The design of the system allows it to be very *open*, in that every step of the recognition process is observable — rules and structures are clearly exposed, and the source of a recognition error can be pin-pointed exactly. With a full understanding of how the system works, it is also in most cases very easy to determine beforehand whether or not a certain image will be correctly recognised, and the reasons why.

Finally, this work has helped me the author achieve a much greater understanding of the field of character recognition with its many different approaches, and also of the world of research in general.

References

[1] The Annals of King Sejong (世宗實錄), volume 102, Sejong 25-12-30 (1443/1444). `http://sillok.history.go.kr/inspection/insp_king.jsp?id=wda_12512030_002`, accessed 2009-05-19.

[2] Cho, E.-S. Tokyo University of Foreign Studies, Kunminseion kairei (訓民正音解例), 2009. `http://www.tufs.ac.jp/ts/personal/choes/korean/middle/text/kairei1.html`, accessed 2009-05-19.

[3] Cho, S.-B. and Kim, J. H. Hierarchially structured neural networks for printed hangul recognition. *Neural Networks*, 1:265–270, 1990.

[4] Cho, S.-J. and Kim, J. H. Bayesian network modeling of hangul characters for on-line handwriting recognition. *Document Analysis and Recognition*, 1:207–211, August 2003.

[5] CIA. The World Factbook, 2009. `https://www.cia.gov/library/publications/the-world-factbook/fields/2119.html`, accessed 2009-05-19.

[6] Davis, M. and Dürst, M. Unicode standard annex #15: Unicode normalization forms (rev. 29), March 2008. `http://www.unicode.org/reports/tr15/tr15-29.html#Hangul`, accessed 2009-05-05.

[7] Hildebrandt, T. H. and Liu, W. Optical recognition of handwritten chinese characters: Advances since 1980. *Pattern Recognition*, 26(2):205–225, 1993. ISSN 00313203.

[8] International Phonetic Association. *Handbook of the International Phonetic Association.* Cambridge University Press, 1999. ISBN 978-0-521-63751-0.

[9] Irisa, N. and Mun, H.-j. *Yoku wakaru kankokugo, step 1.* Hakuteisha, 6th edition, 2005. ISBN 4-89174-587-8.

[10] Kang, K.-W. and Kim, J. H. Handwritten hangul character recognition with hierarchial stochastic character representation. *Document Analysis and Recognition*, 1:212–216, 2003.

[11] Kim, E. and Lee, Y. Handwritten hangul recognition using a modified neocognitron. *Neural Networks*, 4(6):743–750, 1991. ISSN 08936080.

[12] Kim, P. K. and Kim, H. J. Off-line handwritten korean character recognition based on stroke extraction and representation. *Pattern Recognition Letters*, 15 (12):1245–1253, 1994. ISSN 01678655.

[13] Kim, W. S. and Park, R.-H. Off-line recognition of handwritten korean and alphanumeric characters using hidden Markov models. *Pattern Recognition*, 29(5): 845–858, 1996. ISSN 00313203.

[14] King Sejong. Hunminjeongeum (訓民正音), 1446. `http://www.cha.go.kr/korea/heritage/search/Culresult_Db_View.jsp?VdkVgwKey=11,00700000,11`, accessed 2009-05-19.

[15] Language Research Associates. *Sugu ni tsukaeru kankokugokaiwa.* Unicom Inc., 4th edition, 2005. ISBN 978-4-89689-436-3.

[16] Lee, I. and Ramsey, S. R. *The Korean Language*. SUNY Press, January 2001. ISBN 978-0-7914-4831-1.

[17] Lee, J.-S., Kwon, O.-J., and Bang, S.-Y. Highly accurate recognition of printed korean characters through an improved two-stage classification method. *Pattern Recognition*, 32(12):1935–1945, 1999. ISSN 00313203.

[18] Nagata, M. Japanese OCR error correction using character shape similarity and statistical language model. In *COLING-ACL*, pages 922–928, 1998.

[19] Park, H.-S. and Lee, S.-W. Off-line recognition of large-set handwritten characters with multiple hidden Markov models. *Pattern Recognition*, 29(2): 231–244, 1996. ISSN 00313203.

[20] Plamondon, R. and Srihari, S. N. On-line and off-line handwriting recognition: A comprehensive survey. *Pattern Analysis and Machine Intelligence*, 22(1):63–84, January 2000. ISSN 01628828.

[21] Srihari, S. N. High-performance reading machines. *Proceedings of the IEEE*, 80 (7):1120–1132, July 1992. ISSN 00189219.

[22] Srihari, S. N., Hong, T., and Srikantan, G. Machine-printed japanese document recognition. *Pattern Recognition*, 30(8):1301–1313, 1997. ISSN 00313203.

[23] Suzuki, S. Graph-based vectorization method for line patterns. *Computer Vision and Pattern Recognition*, pages 616–621, June 1988.

[24] Suzuki, S. and Ueda, N. Robust vectorization using graph-based thinning and reliability-based line approximation. *Computer Vision and Pattern Recognition*, pages 494–500, 1991.

[25] The Unicode Consortium. *The Unicode Standard, Version 5.0*. Addison-Wesley Professional, 5th edition, November 2006. ISBN 0321480910.

[26] Ullmann, J. R. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, January 1976. ISSN 00045411.

[27] Weisstein, E. W. “Line” from MathWorld — A Wolfram Web Resource, 2009. `http://mathworld.wolfram.com/Line.html`, accessed 2009-05-14.

A Jamo structure definitions

Table 3: Jamo structure definitions

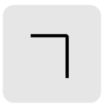
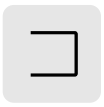
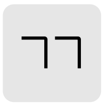
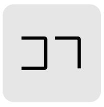
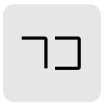
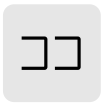
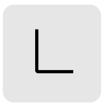
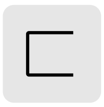
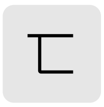
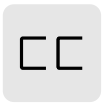
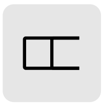
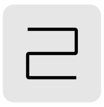
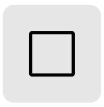
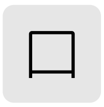
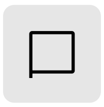
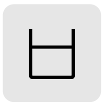
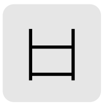
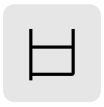
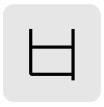
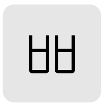
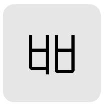
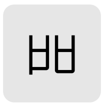
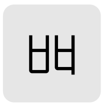
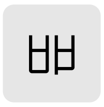
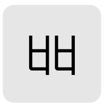
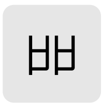
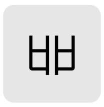
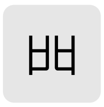
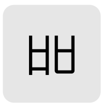
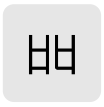
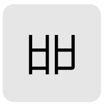
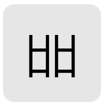
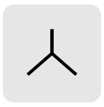
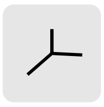
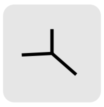
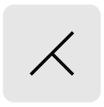
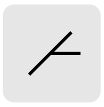
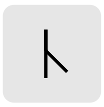
ㄱ (g)					
ㄲ (gg)					
ㄴ (n)					
ㄷ (d)					
ㄸ (dd)					
ㄹ (r)					
ㅁ (m)					
ㅂ (b)					
ㅃ (bb)					
					
					
ㅅ (s)					
					

Table 3: (continued)

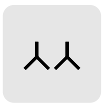
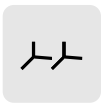
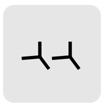
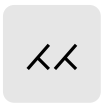
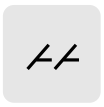
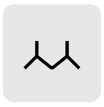
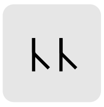
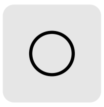
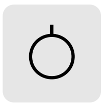
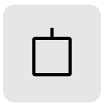
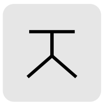
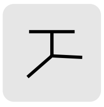
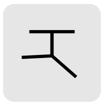
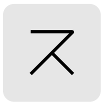
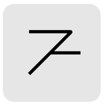
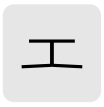
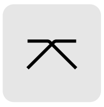
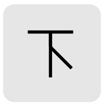
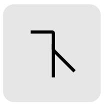
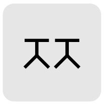
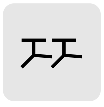
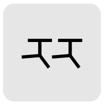
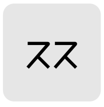
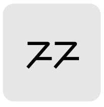
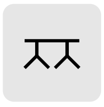
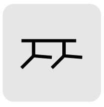
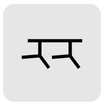
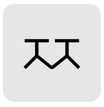
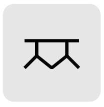
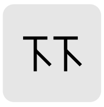
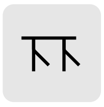
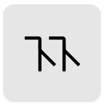
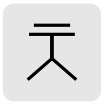
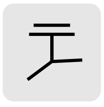
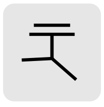
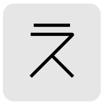
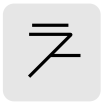
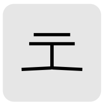
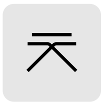
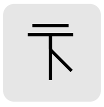
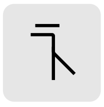
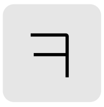
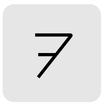
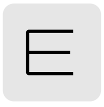
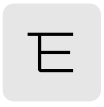
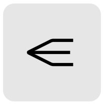
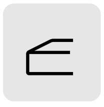
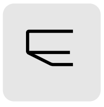
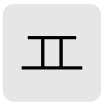
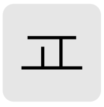
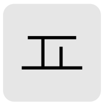
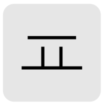
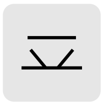
ㅆ (ss)					
					
ㅇ (ng)					
ㅈ (j)					
					
ㅉ (jj)					
					
					
ㅊ (c)					
					
ㅋ (k)					
ㅌ (t)					
ㅍ (p)					

Table 3: (continued)

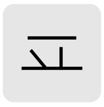
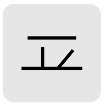
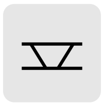
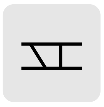
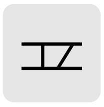
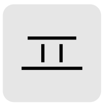
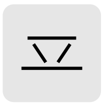
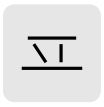
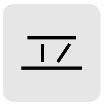
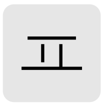
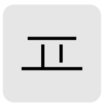
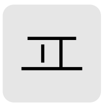
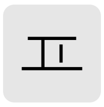
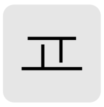
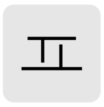
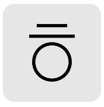
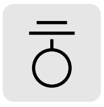
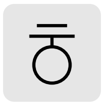
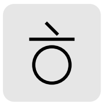
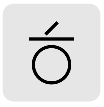
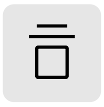
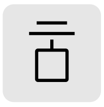
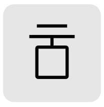
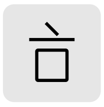
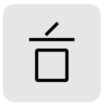
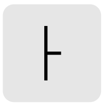
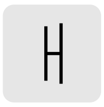
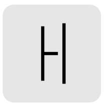
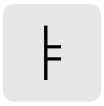
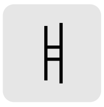
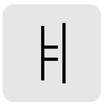
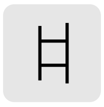
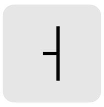
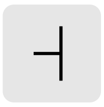
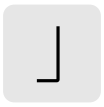
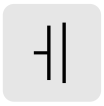
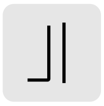
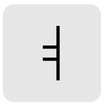
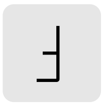
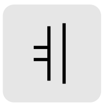
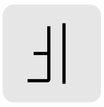
					
					
					
ㅎ (h)					
					
ㅏ (a)					
ㅐ (ae)					
ㅑ (ya)					
ㅒ (yae)					
ㅓ (eo)					
ㅔ (e)					
ㅕ (yeo)					
ㅖ (ye)					

Table 3: (continued)

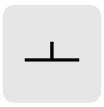
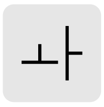
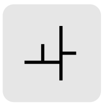
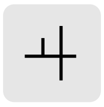
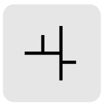
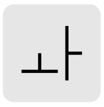
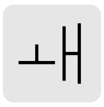
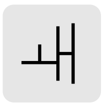
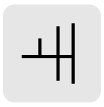
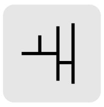
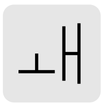
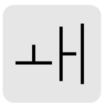
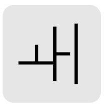
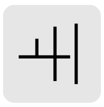
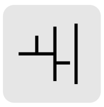
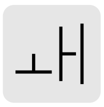
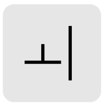
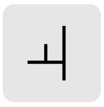
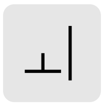
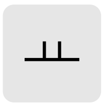
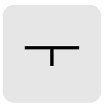
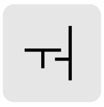
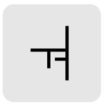
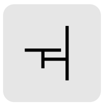
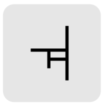
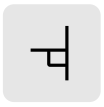
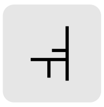
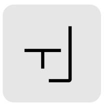
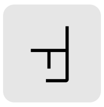
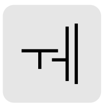
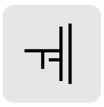
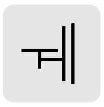
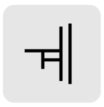
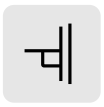
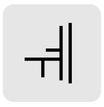
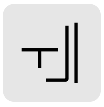
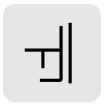
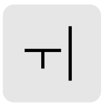
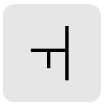
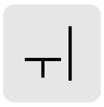
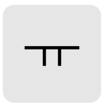
ㅗ (o)					
ㅘ (wa)					
ㅙ (wae)					
					
ㅚ (oe)					
ㅛ (yo)					
ㅜ (u)					
ㅝ (weo)					
					
ㅞ (we)					
					
ㅟ (wi))					
ㅠ (yu)					

Table 3: (continued)

ㅡ (eu)	ㅡ				
ㅢ (yi)	ㅢ	ㅢ			
ㅣ (i)	ㅣ				
ㄳ (gs)	ㄳ	ㄳ	ㄳ	ㄳ	ㄳ
	ㄳ	ㄳ	ㄳ	ㄳ	ㄳ
	ㄳ	ㄳ			
ㄵ (nj)	ㄵ	ㄵ	ㄵ	ㄵ	ㄵ
	ㄵ	ㄵ			
ㄶ (nh)	ㄶ	ㄶ	ㄶ	ㄶ	ㄶ
	ㄶ	ㄶ	ㄶ	ㄶ	ㄶ
ㄺ (lg)	ㄺ				
ㄻ (lm)	ㄻ	ㄻ	ㄻ		
ㄼ (lb)	ㄼ	ㄼ	ㄼ	ㄼ	

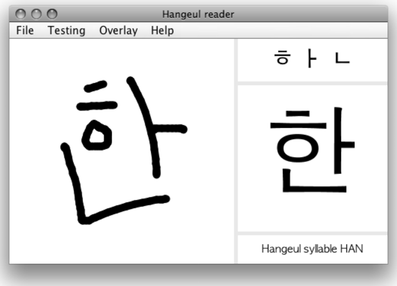
Table 3: (continued)

ㄽ (ls)	ㄽ	ㄽ	ㄽ	ㄽ	ㄽ
	ㄽ				
ㄾ (lt)	ㄾ	ㄾ			
ㄿ (lp)	ㄿ	ㄿ	ㄿ	ㄿ	ㄿ
	ㄿ	ㄿ	ㄿ	ㄿ	ㄿ
	ㄿ				
ㅀ (lh)	ㅀ	ㅀ	ㅀ	ㅀ	ㅀ
	ㅀ	ㅀ	ㅀ	ㅀ	ㅀ
ㅄ (bs)	ㅄ	ㅄ	ㅄ	ㅄ	ㅄ
	ㅄ	ㅄ	ㅄ	ㅄ	ㅄ
	ㅄ	ㅄ	ㅄ	ㅄ	ㅄ
	ㅄ	ㅄ	ㅄ	ㅄ	ㅄ
	ㅄ	ㅄ	ㅄ	ㅄ	

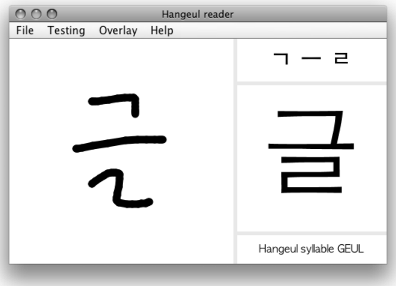
This page is intentionally left blank

B Recognition examples

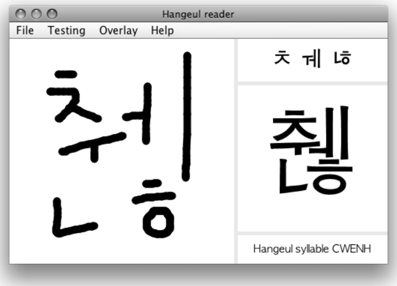
Below are screenshots of the running application. The left hand side of the application shows the input image. The right hand side is split into three fields, showing the individual jamo found, the composed Hangeul syllable, and the name (transcription) of the syllable. If a valid syllable cannot be composed, the first jamo found is shown in the middle field. Figure 14(e) shows an example of this, where ㄹ in combination with ㅒ is mistakenly recognised as ㄼ, as mentioned in Section 3. The other five figures show examples of correctly classified images. Figures 14(a) and 14(c) are intentionally drawn in a somewhat unnatural way to exemplify the method's tolerance to variance in element positioning and size. 14(f) shows how 14(e) can be drawn to avoid incorrect classification; separating the ㄹ and the ㅒ will force them to be classified individually.



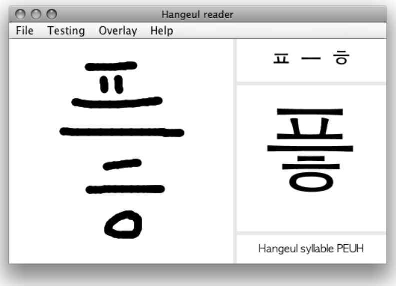
(a) Hangeul syllable HAN



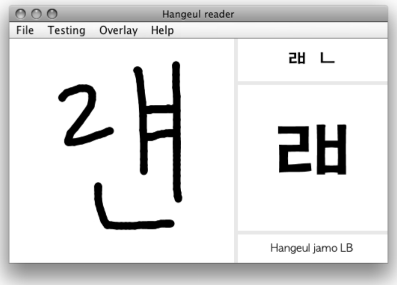
(b) Hangeul syllable GEUL



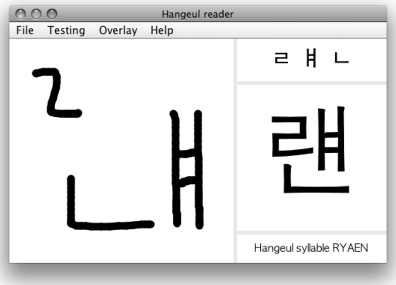
(c) Hangeul syllable CWENH



(d) Hangeul syllable PEUH



(e) Hangeul syllable RYAEN (→ jamo LB)



(f) Hangeul syllable RYAEN

Figure 14: Recognition examples