A Framework for Interpreting Noisy, Two-dimensional Images, based on a Fuzzification of Programmed, Attributed Graph Grammars

THESIS
Submitted in fulfilment of the
requirements for the Degree of
Doctor of Philosophy
of Rhodes University

by

Gregory Shroll Watkins

February 1997

Abstract

This thesis investigates a fuzzy syntactic approach to the interpretation of noisy two-dimensional images. This approach is based on a modification of the attributed graph grammar formalism to utilise fuzzy membership functions in the applicability predicates. As far as we are aware, this represents the first such modification of graph grammars. Furthermore, we develop a method for programming the resultant fuzzy attributed graph grammars through the use of non-deterministic control diagrams. To do this, we modify the standard programming mechanism to allow it to cope with the fuzzy certainty values associated with productions in our grammar.

Our objective was to develop a flexible framework which can be used for the recognition of a wide variety of image classes, and which is adept at dealing with noise in these images. Programmed graph grammars are specifically chosen for the ease with which they allow one to specify a new two-dimensional image class.

We implement a prototype system for Optical Music Recognition using our framework. This system allows us to test the capabilities of the framework for coping with noise in the context of handwritten music score recognition. Preliminary results from the prototype system show that the framework copes well with noisy images.

Acknowledgements

During the course of this work, Alfredo Terzoli has been both a supervisor and a friend. He has given selflessly of his time and has always been at hand with good suggestions and advice when it was needed. His ability to see past details and to grasp the "bigger picture" has been particularly valuable. His numerous suggestions and meticulous proofreading of this document have greatly improved the final product. Thank you, Alfredo.

Peter Clayton must be thanked for his advice, which has improved this thesis significantly, for proofreading the final document, for his encouragement and for obtaining funding for this research.

I am also grateful to Shaun Bangay, James Gain, Gwyn Graham, Cindy Kulongowski, and my brother, Kevan, who all proofread this thesis. I am indeed fortunate to be surrounded by so many friends happy to give freely of their time.

The postgrads and staff of the Department of Computer Science at Rhodes University have provided a stimulating and enjoyable environment in which to work. They have always been willing to listen to my ideas and have provided much useful feedback.

Last, but not least, I thank my family and Gemma for their constant encouragement and support.

This work was funded mainly by the Foundation for Research Development.

Contents

Introduction **1**

I Developing the framework: modifying graph grammars 9

1 The problem **10**

1.1 Graph grammars and noise 11

1.2 Previous work 12

2 Graph grammars **18**

2.1 Definitions 18

3 Grammar programming **29**

3.1 Definitions 29

3.2 The generative power of programmed grammars 35

4 Fuzzifying graph grammars **37**

4.1 Coping with noise 37

4.2 Fuzzy string grammars 41

5 Programming fuzzy grammars: an informal introduction **44**

5.1 A method for programming fuzzy grammars 44

5.2 Augmenting the control diagram with output nodes 53

5.3 Noise Productions 57

5.4 Advantages of programming a fuzzy grammar 62

6 Programmed fuzzy attributed graph grammars 67

6.1 Definitions 67

6.2 Fuzzy certainty functions 72

6.3 Calculating the certainty of a parse path 73

7 Efficiency considerations 75

7.1 Pruning the search tree 76

7.2 Reducing the amount of the search tree examined. 80

7.3 Reducing the complexity of applying a production 86

II Using the framework: a test application 90

8 The Problem 91

8.1 Optical music recognition 91

8.2 A comparison of OMR with OCR 93

8.3 A comparison of handwritten music with printed music 94

8.4 Previous work 95

9 Preliminary processing 102

9.1 Image acquisition 102

9.2 Staff line removal 103

10 Selecting primitives (features) and their attributes 110

10.1 A classification of music scores 111

10.2 The primitives . . . 114

11 Extracting lines and curves 118

11.1 Line thinning . . . 119

11.2 Following curves . . . 122

11.3 Calculating a curve's attributes . . . 126

11.4 Removing curves from the image . . . 127

12 Extracting blobs 130

12.1 Blob extraction . . . 130

12.2 Calculating a blob's attributes . . . 132

13 Defining the grammar 135

13.1 Choosing an external representation for the productions . . . 135

13.2 Writing the productions . . . 138

13.3 Translating the productions into their internal representation . . 143

III Conclusions 145

14 Preliminary results 146

14.1 A simple case . . . 146

14.2 Coping with noise . . . 149

14.3 More noise . . . 153

15 Concluding remarks 159

15.1 What was achieved . . . 159

15.2 Future work . . . 162

15.3 In closing . . . 163

References **164**

Appendices 171

A Defining the grammar **172**

A.1 The grammar definition language 172

A.1.1 Lex rules 172

A.1.2 YACC rules 173

A.2 Minimal grammar to do a full parse, up to the bar level 176

B Defining the control diagram **193**

B.1 The control diagram definition language 193

B.1.1 Lex rules 193

B.1.2 YACC rules 194

B.2 Control diagram for applying productions in appendix B 197

C Internal representation of productions **198**

D Fuzzy parsing algorithm **200**

E State space search trees **206**

E.1 Search tree for the simple example (14.1) 207

E.2 Search tree showing ambiguity (14.2) 209

E.3 Search tree showing more ambiguity (14.3) 211

F Components of the prototype system **213**

F.1 Low-level components 213

F.2 Parsing components 215

G Systems and languages used **216**

List of Figures

1	An excerpt from a music score.	2
2	Problems caused by staff line removal.	5
3	Use of context to recognise ambiguous characters.	5
4	A difficult situation.	6
2.1	Initial graph for a bar of music.	21
2.2	Final graph for a bar of music.	22
2.3	A Sample Production.	25
3.1	A simple control diagram.	30
3.2	Control diagram, in Fahmy and Blostein format.	33
3.3	The control diagram of the previous figure in Bunke format.	34
4.1	A difficult situation.	39
4.2	Definition of *CLOSE*.	40
4.3	Comparison of crisp and fuzzy certainty functions.	42
5.1	A simple control diagram.	45
5.2	Search trees (A) without and (B) with backtracking.	47
5.3	Search tree for fuzzy grammar.	49
5.4	State space search tree for a simple example.	51

5.5 Control diagram for competing productions. 53
5.6 State space search tree for competing productions. 54
5.7 A problem with noise productions. 59
5.8 Primitives making up the image. 63
5.9 Context-sensitive grammar to generate the language $a^n b^n c^n d^n$. 63
5.10 Programmed context-free grammar to generate the language $a^n b^n c^n d^n$. 64
5.11 Programmed context-free grammar to recognise the language $a^n b^n c^n d^n$. 65
5.12 A possible version of the fuzzy membership function *up*. 66

7.1 Possibilities for pruning the search tree. 77
7.2 Pruning the search tree. 78
7.3 Competing noise productions. 80
7.4 Search strategy. 83
7.5 Example of the use of heuristics. 85

9.1 Detecting the staff lines. 105
9.2 Staff line removal. 107
9.3 Removing the staff lines. 107
9.4 A problem caused by staff line removal. 108
9.5 More problems caused by staff line removal. 109

10.1 Common music symbols and terms. 111
10.2 A simple class one score. 112
10.3 A printed class two score. 113
10.4 Examples of a chord. 115
10.5 Composition of primitives to form a music symbol. 116

11.1 4- and 8-connectivity. 120

11.2 A sample 3x3 mask. 121
11.3 Thinning an image. 121
11.4 Checking for a corner in the curve. 123
11.5 Checking for the next pixel of a curve. 123
11.6 Curves found in the image. 125
11.7 A problem case. 125
11.8 Intersection in a curve. 127
11.9 Calculating the width of a line. 128
11.10 Removing curves from the image. 129

12.1 Extracting a blob from the image. 132
12.2 Calculating the solidity of a blob. 133

14.1 A simple bar of music. 147
14.2 Image with staff lines removed. 147
14.3 Primitives extracted from a simple image and their attributes. 147
14.4 Simple search tree associated with the parse. 148
14.5 A handwritten example. 149
14.6 Image with staff lines removed. 150
14.7 Primitives extracted from the image and their attributes. 150
14.8 Search tree associated with the parse. 152
14.9 Another handwritten example. 153
14.10 Problems caused by the staff line removal process. 154
14.11 Spikes caused by line thinning (magnified hollow note-head). 154
14.12 Primitives extracted from the image. 155
14.13 Two productions for recognising hand-drawn note-heads. 157

14.14 The Trap function. 158

F.1 Relationship between the low-level programs. 214

Introduction

This work originated as an attempt to recognise (i.e., read) sheet music, both printed and especially handwritten. We soon realized that existing methods were not always well-suited to the problem. In general, we found a lack of methods for dealing with noisy, two-dimensional images that have inherent structure. This led to an investigation into the possibility of developing such methods.

Setting the scene

The field of image recognition and understanding is an interesting one. Human beings perform this task with ease every moment of their waking lives, yet in general it remains a problematic achievement using computers. Substantial research has been done on various aspects of image recognition and successful computer-based solutions have been found for some of them, from low-level image processing (e.g., noise filtering and template matching) to high-level image recognition (e.g., scene understanding).

A number of image classes contain a rich structure which encodes complex information, and it is these which are of interest here. Examples are flowcharts, electric circuit diagrams and music scores.

Figure 1: An excerpt from a music score.

To a trained musician, the image shown in Figure 1 contains all the information necessary to perform the piece. The information contained in the first two bars of the top line could be represented as follows.

```
TIME 3/4
BAR {G sharp,1.5; E sharp,0.5; SlurStart G sharp, 0.75; E sharp,0.25}
BAR {SlurEnd A sharp,2; E sharp,0.75; G sharp,0.25}
```

This is an arbitrary representation that we have chosen for this example. It is not used by musicians. Let us examine the image a little closer. The three sets of five equi-spaced horizontal lines are staves, and act as a reference for the vertical positions of the other symbols. The first note is formed from a combination of a blob (the note-head) and the vertical line below it (the stem). The first note has the pitch G sharp because of the combination of three factors: the position of its note-head relative to the staff lines, the presence of a bass clef at the left of those staff lines, and the presence of a sharp sign in the key signature between the same two staff lines as the note-head. The duration of this note is determined by the solidity of the note-head, the augmentation dot to the right of the note-head, the presence of a stem and the absence of beams attached to the stem. Although much information is represented by the position of the three primitive elements (i.e. the line, blob and small blob)

relative to other symbols in the image, this information observes a strict structure and can thus be extracted through a set of rules.

Because the inherent structure in the classes of images which interest us can be described by a set of rules, they are naturally suited to recognition by a syntactic approach. In this approach, a set of primitives (image elements) are selected and the higher-level symbols in the image are defined in terms of simpler symbols and the primitives. For example, a note in a music score consists of a note-head and possibly also a stem, tail and augmentation dot. The note-head is either a curve or a blob, the stem and tail are lines and the dot is a small blob. The notes can themselves be grouped together, along with other music symbols, to form a bar of music. Because the same primitives appear in different combinations and positions to form different symbols, a very large number of symbols can be described in terms of a few primitive elements, through a relatively small set of rules.

String grammars are most common in syntactic pattern recognition, as they can be computationally inexpensive to use. However, they cannot be easily used to describe two-dimensional images as they are inherently one-dimensional, limiting the ways of combining sub-part into parts to concatenation. Although string grammars can be augmented with operators to describe two-dimensional relations, this does not, in general, support a simple description of the image structure. Two-dimensional images can sometimes be coded as strings, for example by using Freeman's octal chain codes [Fu *et al.*, 1980]. In this system, the outline of an object in the image is coded as a string of the eight unit vectors ($\uparrow\nearrow\rightarrow\searrow\downarrow\swarrow\leftarrow\nwarrow$). If the image consists of more than one separate object, this is normally unsuitable, as a chain code cannot directly express the spatial relationship between separate objects.

Graph grammars are a natural extension of string grammars to two dimensions and so overcome the string grammar limitation just mentioned [Fahmy & Blostein, 1992b]. They have been used to describe a number of two-dimensional image classes including sheet music, flowcharts and electric circuit diagrams. (A number of other two-

dimensional grammar types exist, but we won't consider them in this work. A summary of many of them can be found in [Golin, 1991].) A drawback of graph grammars is the fact that they are computationally expensive.

The tokens (symbols) used in syntactic pattern recognition are most often unattributed (that is, they do not have any attributes associated with them), as this simplifies the parsing process. Consider, however, the tokens mentioned in the example above based on a music score: notes are attributed with their pitch and duration. Failure to include these attributes would require a very large number of separate note symbols to be defined, one for each pitch and duration. As the spatial relation between symbols is important, it is also necessary to attribute primitives such as lines and blobs with their position and size. This need for attributed symbols applies to two-dimensional images in general. Graph grammars have in fact been extended to cater for attributes in the graph they describe (attributed graph grammars).

While a number of researchers have worked on the computer recognition of structured two-dimensional images, we feel that not enough work has been done in coping with noise in these images. All images contain some degree of noise, and we consider it important that a general framework be developed which can cope with noise. Even if an image appears to be perfectly printed, low-level processing often introduces some noise. See, for instance, the example given in Figure 2. (This image is used to illustrate the same point in section 9.2.) Here the staff line removal process has caused a gap in the top of the bass clef and also at the bottom of the 8. Even more importantly, the variation present in many classes of images, such as the one of handwritten music scores, can be modelled as a superimposition of noise on the "ideal" image.

Figure 3 shows two examples where high-level information can be used to solve local ambiguities in the context of optical character recognition. In the first example, the words are recognised as 'HOT' and 'CAT' respectively. The use of a dictionary causes the letter enclosed in a dotted line to be recognised as an 'H' in the first case, but

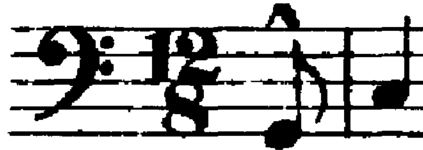


Figure 2: Problems caused by staff line removal.

as an 'A' in the second case, since the alternative words of 'AOT' and 'CHT' are not known. Notice that the same approach does not always enable us to choose between two possibilities. In the second example in the figure, the letter enclosed in a dotted line could be either an 'O' or a 'U'. The use of a dictionary does not assist us, as the resulting words 'SON' and 'SUN' are both possible. We need to examine the context of the word in the sentence to be able to decide that the letter is a 'U'. Of course, there will be cases where it is not possible to resolve the ambiguity, given a certain level of knowledge.

Example 1

HOT CHT

Example 2

THE EARTH ORBITS AROUND THE SON

Figure 3: Use of context to recognise ambiguous characters.

Figure 4 shows an example where high-level information can be used to solve local ambiguity in a music score. (This figure will be used again to illustrate the same point in Chapter 4.) When considered in isolation, it is not clear if the first note-head is solid or hollow. However, if we know that there are three beats in a bar, as is specified

by the $\frac{3}{4}$ time signature, then we can say that the first note must be a minim, and the blob is hollow.

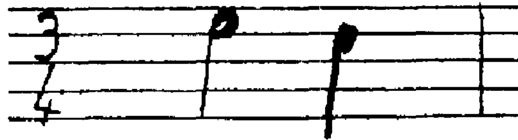


Figure 4: A difficult situation.

Fuzzy string grammars make it easy for high-level information to be used to solve ambiguities arising from noise in images by replacing the binary applicability predicates associated with the productions with continuous certainty functions. If we allow ambiguity to propagate to a higher level in a parse, non-local information can be used to make a better decision. To achieve this using crisp grammars, we need to make the applicability predicates associated with all productions very lenient. Although this allows ambiguity to propagate to a higher level in the parse, it does not offer us a way of directing the search when the parser is presented with a choice because of ambiguity. The fuzzy certainty functions associated with productions in a fuzzy grammar allow the parser to distinguish between more and less promising paths. This means that the parse can be directed by adopting a uniform, or better still, a best-first search stragety.

Fuzzy string grammars have thus proved effective in recognising noisy images [Pal & Dutta Majumder, 1986]. Other approaches to allow string grammars to deal with noise include stochastic grammars and error-correcting parsing [Fu *et al.*, 1980]. A number of arguments for and against fuzzy grammars as opposed to stochastic grammars can be found in the literature [Pal & Dutta Majumder, 1986]. Although fuzzy string grammars have proved effective in dealing with noisy images, they can (like any string grammar) only be used if the image can be described in terms of a string of its primitives. As we have said, this is not, in general, an easy way to describe two-dimensional images.

Since fuzzy string grammars overcome most of the problems that crisp string grammars have in dealing with noisy images, it seems reasonable to attempt to modify graph grammars in a similar way. This is the approach that is taken in this work. The resultant framework should enable one to develop a robust recognition system for a specific image class.

Structure of the thesis

Although this work originated as an attempt to recognise sheet music [Watkins, 1994], it soon evolved into the investigation of a method for interpreting noisy, two-dimensional images. What has emerged is a general framework for recognising noisy, two-dimensional images [Watkins, 1996]. We present our work in a way that reflects this new point of view.

The first part of this thesis (Chapters 1 to 7) describes the general framework, and the second (Chapters 8 to 13) describes the prototype of a music recognition system that uses the framework. These two sections are followed by a shorter section (Chapters 14 and 15) which presents preliminary results, lists possible future work and summarizes our achievements.

In the first part, Chapter 1 precisely defines the problem this thesis attempts to solve and presents previous work on graph grammars. Chapters 2 and 3 define standard (crisp) programmed attributed graph grammars. (Although this is a précis of standard work, we include it to make the thesis self-contained.) Chapter 4 discusses the problem of noisy images, and the inability of crisp programmed graph grammars to properly deal with it. It suggests developing a fuzzy version of the graph grammar, by replacing the binary applicability predicates with fuzzy certainty functions, as is done in fuzzy string grammars. The remainder of the first part discusses our changes to graph grammars to allow them to cope better with noise. Chapter 5 examines

the problems of programming fuzzy grammars and presents a further extension to the grammar mechanism required to deal with noise elements that are recognised as primitive elements. Chapter 6 gives a formal definition of the programmed fuzzy attributed graph grammar that was informally introduced in Chapter 5, and Chapter 7 briefly discusses the efficiency of the parser that we have developed.

In the second part, Chapter 8 introduces the problem of optical music recognition and presents previous work. Chapter 9 describes image acquisition and the preprocessing phase of removing the staff lines. Chapter 10 discusses selection of the primitives, and Chapters 11 and 12 describe the extraction of these primitives from the image. Chapter 13 describes the external and internal representations of productions used by the system, as well as the translation from external to internal representation. Most importantly, it discusses the process of writing the grammar for this application.

In the third part, Chapter 14 presents preliminary results obtained from the prototype. Finally, Chapter 15 presents our achievements and discusses future work.

Part I

Developing the framework: modifying graph grammars

Chapter 1

The problem

Programmed attributed graph grammars allow the intuitive design of a grammar for describing two-dimensional images. Therefore, they are a natural choice for driving the syntactic recognition of such images. The main limitation of graph grammars has been that they do not cope well with noise.

The objective of the work described in this thesis is to create a framework that can easily be used for the recognition of noisy, two-dimensional images.

We propose to develop our framework by modifying programmed attributed graph grammars to allow them to cope with noise in the input graph.

1.1 Graph grammars and noise

Syntactic pattern recognition is a natural and effective way of interpreting patterns which have structure. In this approach, a set of primitive elements which are used to compose the patterns is selected. A set of rules (i.e., the grammar) is written to describe the class of patterns in question in terms of its symbols; symbols in turn are made up of smaller symbols or primitive elements.

A two-dimensional pattern can be naturally described by using graph grammars. Examples of two-dimensional images with inherent structure (e.g., flowchart diagrams, electric circuit diagrams, architectural plans, engineering drawings and music scores) are numerous.

A graph grammar consists of a set of productions, each of which is a graph-rewrite rule (in other words, it replaces one subgraph with another). An example of a production in a graph grammar is given in Figure 2.3. The application of this particular production starts by searching the host graph for a subgraph consisting of a note-head and a stem. If such a subgraph is found, the applicability predicate is used to determine if the subgraph meets the specified criteria. In this case, the production specifies that the top of the stem must be close to the left-hand side of the note, or the bottom of the stem must be close to the right-hand side of the note. If the subgraph meets these criteria, then it is replaced with the right-hand subgraph specified in the production (a note in this case). The embedding transformation specifies how to connect the new subgraph to the host graph. The production shown in Figure 2.3 specifies, through the embedding transformation, that all edges which were connected to either the note-head or the stem must be connected to the note that replaces them. Lastly, the attribute transfer function calculates the attributes associated with the nodes (and possibly the edges) of the new subgraph.

Grammars used for pattern recognition are often programmed. This means that the order in which productions may be applied is specified by a control diagram. This has

the advantage of increased efficiency and expressiveness. Figure 3.1 shows a simple control diagram. Each node in the control diagram specifies which productions may be applied at that point. A Y-edge in the control diagram is followed if a production was successful, while an N-edge is followed if no production in the node could be applied. Thus, the control diagram in Figure 3.1 specifies that production 1 is applied first, then (if it is successful) either production 2 or production 3 may be applied. The parse ends successfully when the final node in the control diagram is reached. Notice that this control diagram does not specify which node to apply if production 1 is unsuccessful. This situation would thus result in an unsuccessful parse.

Graph grammars have two main shortcomings: they are not particularly efficient (even when programmed), and they cannot cope well with noise in the input graph. Our overall goal is to allow graph grammars to cope better with noise. Efficiency is discussed in Chapter 7.

How can we allow graph grammars to cope better with noise? Consider once more the applicability predicate associated with the production given in Figure 2.3. A natural approach for allowing a graph grammar to cope with noise is to replace the binary function that constitutes the applicability predicate with a fuzzy (continuous) certainty function. Although the idea is simple, it necessitates a number of changes to the associated parsing algorithm. For example, we need to decide how to let the control diagram deal with certainty values between 0 and 1.

In this thesis we develop a new parsing algorithm, needed as a result of fuzzifying the applicability predicate associated with each graph grammar production, and we show that the new algorithm is promising with regard to coping with noise.

1.2 Previous work

Graph grammars were first introduced in [Pfaltz & Rosenfeld, 1969] to solve picture processing problems. Since then, they have been applied to a number of other

problem domains, including graph theory, databases [Bartini, 1979; Furtado, 1979; Ehrig & Kreowski, 1980], formal semantics of programming languages [Göttler, 1979], translation of programming languages, incremental compilers, and two-dimensional programming [Denert *et al.*, 1975].

Graph rewriting has been successfully applied to various problems in software engineering, including use as a specification tool, for the study of concurrent and distributed systems, for the implementation of functional languages, and for data-structure manipulation [Blostein *et al.*, 1995]. A significant development in this regard is the programmed graph rewriting language PROGRES, designed to specify tools within the IPSEN (Integrated Project Support ENvironment) software engineering project. Within IPSEN, it has been used to specify the operational behaviour of syntax-directed editors, static analyzers, and incremental compilers and interpreters. Non-deterministic control diagrams are used to define the control structures in the language. These include both deterministic and non-deterministic versions of AND, OR and LOOP constructs [Zündorf & Schürr, 1991]. In the case of the non-deterministic control structures, backtracking is used if an "incorrect" choice is made. The system exhibits angelic non-determinism: if a sequence of choices exists which results in a successful path through the control diagram, it will be found.

Graph rewriting is also used to implement functional programming languages [Peyton Jones, 1987]. A program is converted into an abstract syntax tree, which becomes a DAG (directed acyclic graph) due to repeated expressions. Graph rewriting is used to reduce this DAG to a single node. Only a few rewrite rules are needed due to the limited number of constructs in a functional programming language.

[Ehrig & Kreowski, 1980] discuss the application of graph rewriting in the construction of database systems. They use the algebraic approach to graph rewriting. Since this has strong theoretical results, it is easy to show that proper synchronization is observed and that consistency is maintained in the database. They construct a small library database system as an example.

Parallel graph rewriting has been used to model cell division and organism development in biological modelling and visualization systems [Panel, 1991].

Graph rewriting has been used for diagram generation. ([Dolado & Torrealdea, 1988] used graph grammars to construct Forrester diagrams.) and it has also been used for the specification of diagram editors for a number of diagram types [Göttler *et al.*, 1991]. [Blostein *et al.*, 1995] gives a good revue of the use of graph rewriting systems.

Our concern concentrates on the use of graph grammars for image recognition, so the rest of this section describes previous work in that area.

Bunke modified graph grammars to form attributed programmed graph grammars, which he applied to the recognition of flowcharts and electric circuit diagrams [Bunke, 1982a]. In his work, the input image is represented by a graph, with edges denoting lines, and nodes denoting connections or end-points of lines. This is transformed, by means of the grammar, into an output graph with nodes denoting symbols and edges denoting connectivity between the symbols. The order in which productions are applied is determined by a control diagram. Although non-determinism is allowed in the control diagram, backtracking is not considered for the sake of efficiency. The system thus exhibits erratic non-determinism: if some sequence of choices leads to a successful path through the control diagram but other choices lead to a dead-end, a successful path may or may not be found.

Fahmy and Blostein have applied the programmed attributed graph grammar formalism developed by Bunke to the recognition of music scores [Fahmy & Blostein, 1991]. In their work, the input graph contains a set of disconnected nodes, where each node is a primitive or a music symbol, such as a note-head, stem, beam or clef. The graph grammar transforms this graph into a graph containing music symbols, such as notes and rests. Since input in their work was obtained from the music editor "Lime', the input graph was perfect and did not contain any noise. Many of the uses of graph grammars, for example for parsing visual programming languages, share this feature. In this case, normal programmed attributed graph grammars are well suited for the

task. In contrast, if the input graph is acquired by primitive extraction routines operating on a scanned image, the input graph will contain noise. Even if the image is perfectly printed, noise is normally introduced by primitive extraction routines. Furthermore, the extra variation present in hand-drawn diagrams can be modelled as noise superimposed on the exact definition of the image.

Some work has been done on trying to cope with noise. The work by Bunke uses error-correcting productions to cope with a limited number of expected types of noise, such as line breaks and spurious lines. Because the order of application of productions is controlled (by the control diagram), the error-correction productions can be applied before other productions.

The most recent attempt to allow a graph rewriting system to cope with noise is described in [Fahmy, 1995] and [Fahmy & Blostein, 1996]. An early version of this work was reported in [Fahmy & Blostein, 1992a]. They extended Bunke's programmed attributed graph grammar formalism in two ways. First, they extended the programming mechanism to allow the inclusion of non-programmed sub-grammars. Of more importance here, they developed a method to cope with uncertainty in the input graph.

Uncertainty about the label of a node (primitive) in the input graph, is represented by a connected subgraph containing a node for each possibility. These nodes are connected by "exclusion" edges; an exclusion edge signifies that the two nodes it connects are alternative interpretations of the same symbol. The possibility of a primitive being noise is noted by a "noise" node connected to the primitive node with an exclusion edge.

Fahmy and Blostein used four types of graph-rewrite rules to construct the neighbourhood model and to apply constraints to reduce ambiguity. The four types of rules are: *BUILD*, *CONSTRAIN*, *SPLIT* and *INCORPORATE*. The grammar is split into levels (three in the system that they implemented), each of which is further split into four sub-grammars, one for each of the rule types.

BUILD rules are the first to be applied in any level of the grammar. They are used to build potential associations between nodes that are represented by edges labelled "P.S.A" (Potential Semantic Association). These rules are built wherever an association is possible. If there is a possibility of an edge being formed, it is formed. This means that, when uncertainty is present, spurious edges will be formed. This removes an excessive dependency on thresholds for deciding whether to form an edge or not. The spurious edges are removed by the later application of *CONSTRAIN* rules. Each P.S.A edge has an "affinity" value associated with it. For example, the affinity of an edge between a note-head and a stem is inversely proportional to the Euclidean distance between them. Affinity values are used by the *CONSTRAIN* rules to make choices between competing edges.

After the *BUILD* rules have been applied, the *CONSTRAIN* rules are applied. These attempt to reduce ambiguity in the graph by deleting nodes and edges which do not satisfy certain constraints. One type of *CONSTRAIN* rule explicitly examines the affinity of competing P.S.A edges and deletes the edges with a weaker affinity. Both binary and multi-way constraints are supported. An example of a binary constraint, in a music score, is that there must be a clef to the left of a note. The multi-way constraint that they implement, again referring to a music score, is that the number of beats in a bar must be consistent with the time signature. If no interpretation of the input graph satisfies this constraint, all of the nodes will be deleted and the output graph will be empty.

The *SPLIT* rules split nodes to facilitate the construction of alternate higher-level interpretations. When a node is involved in k contradictory associations, and there is not enough information to judge which association is valid, it is split into k nodes so as to enumerate all possible associations. This stage also includes clean-up productions, which delete the redundant nodes that can be created by *SPLIT* rules. Two nodes are redundant if they have the same label, attributes, and the same connections to other nodes in the graph.

The last group of rules to be applied in each level of the grammar are the *INCORPORATE* rules. These are used to construct a higher-level interpretation of the image. When *INCORPORATE* rules are applied, the semantics of node associations are identified and recorded in the attributes and/or labels of nodes. For example, the pitch of a note can be calculated from its position and from the clef that is associated via a P.S.A edge with it.

Fahmy and Blostein's work improves over previous attempts to cope with noise in graph rewriting systems, but still has the following limitations. It assumes that no segmentation errors are made in the primitive extraction layer. Furthermore, alternate labels for symbols are not ranked, although primitive recognisers can often provide certainty measures. Because the symbols are not ranked, it is not possible for the framework to rank two different interpretations of an image that both satisfy the required constraints. They have begun some work on associating a "reality attribute" with the exclusion edge between a primitive and a noise node, to represent the "degree of existence" of the primitive, but this is still in early stages. There are also certain ambiguities that their system cannot resolve: one example is a primitive that could be either a bar line or a stem. The framework described in this thesis begins to address these limitations.

Both Bunke and Fahmy and Blostein have used deterministic control diagrams (or non-deterministic control diagrams without backtracking) for image recognition primarily for efficiency reasons. Non-deterministic control diagrams (with backtracking) should enable the framework to cope better with noise, at the cost of an increased computational complexity.

Chapter 2

Graph grammars

Before we describe our modifications to graph grammars, we define standard graph grammars. Definitions in this chapter follow [Bunke, 1982b; Bunke, 1982a], with some notational changes.

Whereas each production in a string grammar modifies a string by replacing a substring with another, each production in a graph grammar modifies a graph by replacing a subgraph with another subgraph. Graph grammars thus function in essentially the same way as string grammars, except that they transform graphs instead of strings.

2.1 Definitions

In all cases, the underlying graphs are finite, with nodes and edges labelled by the alphabets V and W respectively.

Definition 2.1 *An* **unattributed, directed graph** *(u-graph) over* (V, W) *is a 3-tuple* $g = (N, E, \lambda)$ *where*

1. N *is the finite set of nodes;*
2. $E = \{E_w \mid w \in W\}$ *is a set of relations* $E_w \subseteq N \times N$ *for each* $w \in W$*; and*
3. $\lambda : N \rightarrow V$ *is the node labeling function.*

Each element (n, n') in E_w is interpreted as an edge from node n to node n' labelled with the symbol $w \in W$. The label of a node $n \in N$ is given by $\lambda(n) \in V$. Let $\Gamma(V, W)$ denote the set of all unattributed graphs over (V, W).

The notation $N[g], E[g]$ or $\lambda[g]$ will be used, respectively, to denote the nodes, edges or labelling functions of a graph g. $E_w[g]$ will be used to denote all edges in g with the label w. An edge with the label w will be denoted as a w-edge for compactness.

An undirected unattributed graph is the same as the directed graph defined above, except that each element of E is an unordered pair which represents an undirected edge between two nodes. Such an undirected graph can be converted into a directed graph by replacing every element (n, n') in E_w with two tuples, (n, n') and (n', n), for each $w \in W$. Since directed graphs are thus more general than undirected graphs, the remaining definitions are built on the directed graph as in Definition 2.1.

Since both the primitives and higher-level symbols used to describe most two-dimensional images have attributes associated with them, some modification must be made to the graph to cater for attributes. Let A be the set of node attributes and B be the set of edge attributes.

Definition 2.2 *An* **attributed graph** *(a-graph) over* (V, W) *with attributes* (A, B) *is a 5-tuple* $g = (N, E, \lambda, \alpha, \beta)$ *where*

1. N, E, λ *are the same as in Definition 2.1;*

2. $\alpha : N \rightarrow 2^A$ *is a function which associates a set of node attributes with each node*[1]*; and*

3. $\beta = (\beta_w)_{w \in W}$ *is a tuple of functions* $\beta_w : E_w \rightarrow 2^B$ *associating a set of edge attributes with each* w*-edge, for each* $w \in W$.

Although this defines possible attributes for both nodes and edges, only the node attributes have been used in our work. This can, of course, be seen as associating an empty set of attributes with every edge. Let $\Gamma_{(A,B)}(V, W)$ denote the set of all attributed graphs over (V, W) with the attributes (A, B).

Consider the short music extract (with staff lines removed) shown in Figure 2.1. At the primitive level, it can be represented by the graph shown pictorially in the same figure. The attributes associated with these nodes are those shown in the lists at the end of Chapters 11 and 12. At a music symbol level, it can be represented by the graph shown in Figure 2.2. In this pictorial representation, each ellipse represents a node and each line represents an edge. Clearly, the purpose of the grammar is to yield a transformation from the graph containing primitives to the graph containing music symbols.

Definition 2.3 *Let* $g' \in \Gamma(V, W)$ *and* $g \in \Gamma(V, W) \bigcup \Gamma_{(A,B)}(V, W)$. *Then* g' *is a* **subgraph** *of* g *(denoted* $g' \subseteq g$*) iff*

1. $N[g'] \subseteq N[g]$

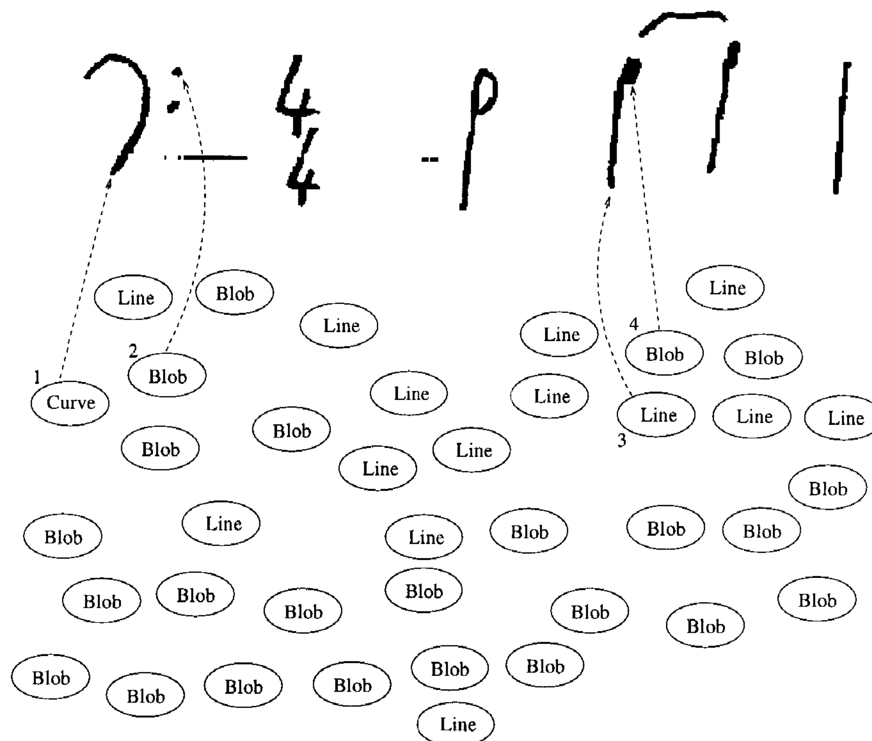
2. $E_w[g'] \subseteq E_w[g] \bigcap (N[g'] \times N[g']), \forall w \in W$

3. $\lambda[g'] = \lambda[g] \mid N[g']$

Note: $f \mid C$ denotes the restriction of the function $f : A \rightarrow B$ to the subset $C \subseteq A$.

Informally, this definition states that an u-graph g' is a subgraph of the (u- or a-)graph g (denoted $g' \subseteq g$) if all nodes and all edges of g' also belong to g. Additionally,

[1] 2^A denotes the set of all sets over the elements of A.



Attributes of sample nodes

1	2	3	4
x = 52	x = 93	x1 = 409	x = 415
y = 78	y = 48	y1 = 57	y = 47
intersections = 0	size = 0.089600	x2 = 404	size = 0.516800
aspect = 1.854167	aspect = 0.875000	y2 = 131	aspect = 1.117647
size = 6.835200	solidity = 1.000000	thickness = 0.280000	solidity = 0.990000
curvature = 3.127308		curvature = 0.221314	

Figure 2.1: Initial graph for a bar of music.

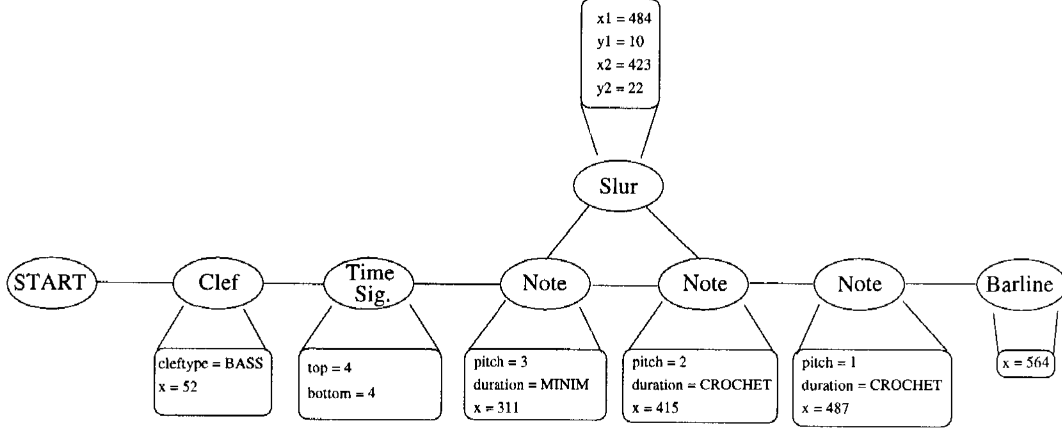


Figure 2.2: Final graph for a bar of music.

corresponding nodes and edges in g and g' must have identical labels. This definition does not require edge completeness of subgraphs.

Definition 2.4 *Let* $g, g' \in \Gamma_{(A,B)}(V, W)$ *and* $g' \subseteq g$. *The* **difference** $g - g'$ *is given by the graph* g'' *defined as follows:*

1. $N[g''] = N[g] - N[g']$
2. $E_w[g''] = E_w[g] \bigcap (N[g''] \times N[g'']), \forall w \in W$
3. $\lambda[g''] = \lambda[g] \mid N[g'']$
4. $\alpha[g''] = \alpha[g] \mid N[g'']$
5. $\beta[g''] = \beta[g] \mid N[g'']$

In other words, $g - g'$ denotes the graph which remains after removing g' from g.

Definition 2.5 *Let* $g, g' \in \Gamma(V, W) \bigcup \Gamma_{(A,B)}(V, W)$, $g' \subseteq g$ *and* $g'' = g - g'$. *We define*

1. $\mathrm{In}_w(g', g) \equiv E[g] \bigcap (N[g''] \times N[g'])$
2. $\mathrm{Out}_w(g', g) \equiv E[g] \bigcap (N[g'] \times N[g''])$

$\text{In}_w(g', g)$ denotes the set of all w-edges originating in $g - g'$ and terminating in g'. $\text{Out}_w(g', g)$ denotes the set of w-edges in the opposite direction. The union of $\text{In}_w(g', g)$ and $\text{Out}_w(g', g)$ for all $w \in W$, i.e., the set of all edges between the subgraph g' and the host graph $g - g'$, is called the embedding of g' in g, denoted by $\text{EMB}(g', g)$.

Definition 2.6 *A* **production in an attributed graph grammar** *over* (V, W) *with attributes* (A, B) *is a five-tuple* $p = (g_l, g_r, T, \pi, F)$ *where*

1. g_l *and* $g_r \in \Gamma(V, W)$ *are u-graphs, the left-hand and right-hand side, respectively;*
2. $T = (L_w, R_w)_{w \in W}$ *is the embedding transformation with the exact form of* L_w *and* R_w *depending on the class of embedding transformation being used;*
3. $\pi : \Gamma_{(A,B)}(V, W) \rightarrow \{TRUE, FALSE\}$ *is the applicability predicate; and*
4. F *is a finite set of partial functions* $f_a : N[g_r] \rightarrow D_a$ *and* $f_b : E[g_r] \bigcup \text{EMB}(g_r, g) \rightarrow D_b$ *with* $a \in A$, $b \in B$, *and* $g \in \Gamma_{(A,B)}(V, W)$. *The* f_a *are the node attribute and the* f_b *are the edge attribute transfer functions.*

In this definition, D_a and D_b denote the domains of the attributes a and b respectively. The left- and right-hand sides are straightforward extensions of the string grammar case. Productions are often written in a generative (top-down) fashion, so that when they are applied to recognise a language (bottom-up), the right-hand side is replaced by the left-hand side. Here, we have written productions in a bottom-up format, so that during a bottom-up parse they are applied as they are written, as is normally the case in programmed grammars (i.e., the left-hand side is replaced by the right-hand side).

The functions f_a and f_b are straightforward extensions of the way attributes are transferred in attributed string grammars. $f_a(n)$ specifies the value of attribute a at node n and $f_b(e)$ specifies the value of attribute b at edge e of the right-hand side.

The embedding transformation is needed to specify the embedding of the new (right-hand side) subgraph in terms of the embedding of the old (left-hand side) subgraph.

The class of embedding transformation used in this work is defined as follows: for each $w \in W$ we specify sets L_w and R_w consisting of pairs (n, n') of nodes, with $n \in N[g_l]$ and $n' \in N[g_r]$. A pair (n, n') causes an edge between node n on the left-hand side (g_l) and node n'' in the host graph to be transformed into an edge between node n' in the right-hand side (g_r) and node n''. L_w transforms $\text{In}_w(g', g)$ and R_w transforms $\text{Out}_w(g', g)$, where g is the host graph.

Since undirected graphs are used in this work and each edge in a undirected graph can be represented by two directed edges, one in each direction, it is always the case that $\text{In}_w(g', g) = \text{Out}_w(g', g), \forall w \in W$. This is the *analogous* class of embedding transformation, as defined by [Nagl, 1987].

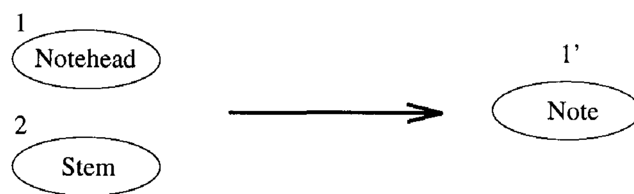
A sample production is shown in Figure 2.3. The same pictorial representation as before is used for the subgraphs g_l and g_r.

This production transforms a stem and a note-head into a note. The embedding transformation states that any edges which were connected to either the note-head or the stem should be connected to the new note. The attribute transfer function calls a function to determine the note's pitch from the note-head's vertical position relative to the stave, sets its duration to a crotchet, and sets its horizontal position to that of the note-head. The applicability predicate specifies that the production may only be applied if the note-head is close to an end of the stem and the note-head is hollow.

Definition 2.7 *A* **production is monotone** *iff* $|N[g_l]| \leq |N[g_r]|$. *It is* **context-free** *iff it is monotone and* $|N[g_l]| = 1$.

The above definition of a context-free graph grammar production is analogous to the definition of a context-free string grammar production.

The descriptive power of a graph grammar depends largely on the type of embedding transformation used. These range from the very powerful but highly complex *expression* embedding mechanism developed by [Nagl, 1979] to the simple *invariant* embedding mechanism. The power of an embedding mechanism depends on its ability



Embedding transformation =

{(Notehead,Note) , (Stem,Note)}

Attribute transfer function =

{Note.pitch = GetPitch (Centre(Notehead));
Note.duration = MINIM;
Note.x = Notehead.x;}

Applicability predicate =

{((Top(Stem) CLOSE Left (Notehead)) OR
(Bottom (Stem) CLOSE Right (Notehead)))
AND Hollow (Notehead)}

Figure 2.3: A Sample Production.

to specify the embedding in terms of node labels, edge labels, edge directions, and edge following. This observation is formalized by Nagl's classification of embedding mechanisms [Nagl, 1979; Nagl, 1987]. This classification is briefly summarized in [Fahmy & Blostein, 1992b]. The following classes are defined, from most complex to least complex. Let $g' \subset g$ be the subgraph that is to be replaced by g''. We refer to the edges forming the embedding of g' in g as pre-embedding edges, and to those forming the embedding of g'' in g (after the embedding transformation) as post-embedding edges.

Unrestricted These embeddings are expressed by Nagl's expression embedding mechanism. Starting at any node in g', a sequence of edges is followed (starting with a pre-embedding edge). The sequence is conditional on edge orientations and edge labels, and ends at a set of nodes in $g - g'$. A subset of these nodes can be chosen (based on their labels) as source or target nodes for post-embedding edges. The directions and edge-labels of the post-embedding edges are unrestricted.

Orientation and Label Preserving The same as *unrestricted*, except that each post-embedding edge must have the same label and direction as the corresponding pre-embedding edge.

Depth1 Only nodes in $g - g'$ which are direct neighbours of g' may act as source or target nodes of post-embedding edges. The orientation and labels of post-embedding edges may be conditional only on the labels of nodes that are direct neighbours of g', but are otherwise unrestricted.

Simple The same as depth1, but also orientation and label preserving.

Elementary The same as simple, with the additional restriction that source and target nodes must be selected independently of the labels of nodes directly neighbouring g'.

Analogous The same as elementary, with the embedding transformation independent of the orientations and labels of embedding edges.

Invariant Nodes of g' are mapped to nodes of g'' on a one-to-one basis. This type of embedding transfer process does not allow edge splitting or fusion.

If an expression embedding mechanism is used, a context-sensitive grammar is equivalent to a context-free grammar. This is because information about other nodes can be stored in the embedding transformation, rather than explicitly in the left-hand subgraph. Since elementary and simpler embedding mechanisms can not specify this information, context-sensitive and context-free grammars are not equivalent when one of these simpler embedding mechanisms is used.

We have chosen to use an analogous embedding transformation for our system as we have found it to be sufficient and necessary to specify the information required for describing music scores. Other classes of two-dimensional images might require the use of a more complex embedding mechanism.

Definition 2.8 *The* **direct derivation of a graph** g' **from a graph** g *by means of a production* p *(denoted* $g \xrightarrow{p} g'$*) is defined by the following procedure.*

1. *Check whether the left-hand side of the production occurs as a subgraph in* g *and check whether the applicability predicate is TRUE for this occurrence. If both conditions are fulfilled perform steps 2 to 4.*
2. *Replace the left-hand by the right-hand side.*
3. *Transform the embedding of the left-hand side in* g *into that of the right-hand side in* g'*.*
4. *Attach attributes to the inserted right-hand side according to the functions* f_a *and* f_b*.*

In a normal grammar, the order in which productions are applied is not explicitly controlled. It has been shown that *programming* a grammar, where the order in which

productions is applied is explicitly controlled through the use of a finite state automaton (called the control diagram), can increase the expressiveness of the grammar and can reduce the computational complexity of a parse using the grammar. These properties make programmed grammars well-suited to pattern recognition application, and they have been used for a number of applications.

Chapter 3

Grammar programming

In a normal grammar, the order in which productions are applied is not explicitly controlled. It has been shown that *programming* a grammar, where the order in which productions is applied is explicitly controlled through the use of a finite state automaton (called the control diagram), can increase the expressiveness of the grammar and can reduce the computational complexity of a parse using the grammar. These properties make programmed grammars well-suited to pattern recognition application, and they have been used for a number of applications. In this chapter we define programmed graph grammars.

3.1 Definitions

Definition 3.1 *Let P be the finite set of productions in some grammar. A* **control diagram** *over P is an unattributed graph with the set $P \cup \{I, F\}$ as node labels and the set $\{Y, N\}$ as edge labels. Furthermore, the following conditions hold true:*

1. *There exists exactly one initial node n_I labelled with I.*

2. There exists exactly one final node n_F labelled with F.

3. There exists no edge terminating in n_I.

4. There exists no edge originating from n_F.

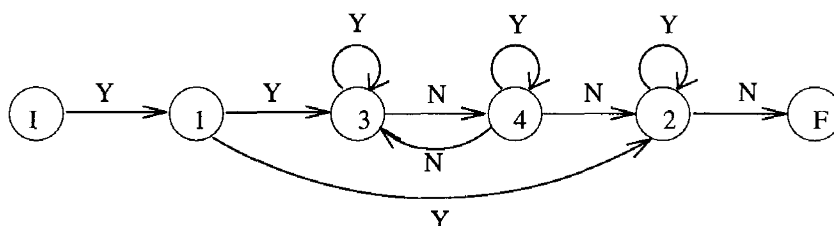


Figure 3.1: A simple control diagram.

An example of a control diagram is shown in Figure 3.1 (Bunke, 1982). All nodes, except the initial and final nodes, are labelled with productions. To apply productions according to the control diagram, we start with a production which is a direct successor of the initial node and try to apply it. After the successful application of a production, a Y-edge (yes) in the control diagram is followed, while an N-edge (no) is followed when a production fails. A derivation sequence is stopped when the final node is reached. A grammar is programmed if the order of application of its productions is controlled by a control diagram.

Example: Consider the following set of productions:

P1. $S \rightarrow A$

P2. $A \rightarrow a$

P3. $A \rightarrow BB$

P4. $B \rightarrow A$

The non-programmed grammar consisting of these productions generates the context-free language $L = a^n, n \geq 1$. If the order of application of these productions is

controlled by the control diagram given in Figure 3.1, the context-sensitive language $L = a^{2^n}, n \geq 0$ is generated. The use of the control diagram has thus increased the generative power of the grammar, allowing simple context-free productions to generate a context-sensitive language.

Note that the above definition of a control diagram applies to graph grammars as well as to string grammars.

Definition 3.2 *An* **attributed programmed graph grammar** *is a 7-tuple $G = (V, W, A, B, P, S, C)$ where*

1. *V and W are alphabets for labelling the nodes and edges, respectively;*
2. *A and B are finite sets of attributes for nodes and edges, respectively;*
3. *P is a finite set of productions;*
4. *S is a set of initial graphs; and*
5. *C is a control diagram over P.*

It should be noted that no distinction is made between terminal and non-terminal symbols in this definition. In the case of non-programmed grammars, this distinction is used to control the order of application of productions to some extent, including the ending of a derivation sequence. In the case of programmed grammars, the distinction is not needed, as the control diagram explicitly specifies the order of productions and the terminating conditions. Making a distinction between terminal and non-terminal labels adds no generative power to the programmed graph grammar defined above [Bunke, 1982b].

Definition 3.3 *Let G be an attributed programmed graph grammar. The* **language** *of G consists of all a-graphs which can be derived in the following way.*

1. *Start with an initial graph.*
2. *Apply productions in an order defined by the control diagram.*

3. *Stop the derivation sequence when the final node in the control diagram has been reached.*

Fahmy and Blostein use a modified form of grammar programming, which includes subgraphs and "non-programmed" grammar sections [Fahmy & Blostein, 1992a].

Definition 3.4 *Let P be the finite set of productions in some grammar. An* **extended Fahmy/Blostein control diagram** *over P is an unattributed graph with the set $2^P \cup \{I, F\}$ as node labels and the set $\{Y, N\}$ as edge labels. Furthermore, the conditions 1 to 4 of definition 3.1 hold.*

This differs from Bunke's programming in that nodes in the control diagram are labelled with sets of production, rather than individual productions. A Y-edge is followed if a production is successful, and an N-edge is followed if all of the productions in the set have failed. Given a control diagram C of this form, we can create a functionally equivalent control diagram C1 in the form used by Bunke, as follows.

1. Start with C1 containing only Initial and Final nodes, corresponding to the Initial and Final nodes in C.

2. Every other node in C is labelled with a set of productions. Add a node to C1 for each of these productions, labelled accordingly. For notational convenience, we denote the subgraph containing all nodes in C1 caused by node i in C as subgraph C1i.

3. For each Y-edge from node x to node y in C, create a Y-edge from every node in C1x to every node in C1y.

4. The N-edges are trickier. To describe an N-edge from node x to node y in C, a subgraph must be added for each node in C1x. Each of these subgraphs is created as follows. Add a node to C1 with the same label as each node in C1x except the current node. (Notice that, if C1x contains a single node, no extra

nodes will be added.) Connect each of these nodes to the next one with an N-edge, forming a "linked list" structure. Add a Y-edge from each of these nodes to each node in C1x. Then add an N-edge from the node at the end of the "linked list" to each node in C1y. This subgraph will ensure that an N-edge will only be followed when every production in a subset fails.

The above is illustrated by the following example. Figure 3.2 shows a simple control diagram in the format used by Fahmy and Blostein, and Figure 3.3 shows the construction of the same control diagram in the format used by Bunke. The steps 1-4 in Figure 3.3 correspond to the steps in the explanation above.

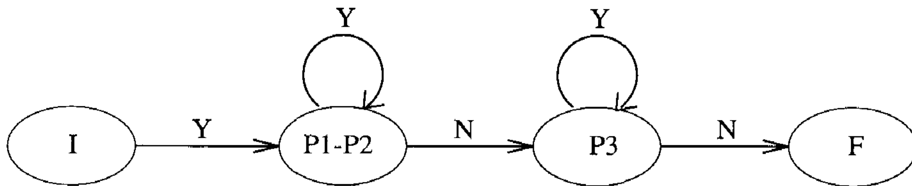


Figure 3.2: Control diagram, in Fahmy and Blostein format.

Even without the extension to allow more than one production per node, there are three possible sources of nondeterminism present in control diagrams. When a production is applied, the first step is to find a subgraph in the graph being modified which is isomorphic to the left-hand side of the production. In most cases there is more than one such subgraph, and the choice of one of them is, in theory, non-deterministic. If the application of a production in a node in the control diagram is successful and more than one Y-edge emanates from that node, the choice of which edge to follow is non-deterministic. The same applies if all productions in a node in the control diagram fail and more than one N-edge emanates from that node.

The first case is present in any use of a control diagram. The other cases are very often present. However, to reduce computational complexity, the possibility of backtracking is not normally catered for in programmed grammar systems. Zuñdorf and Schur̈r introduced the possibility of backtracking in designing the control structures for the

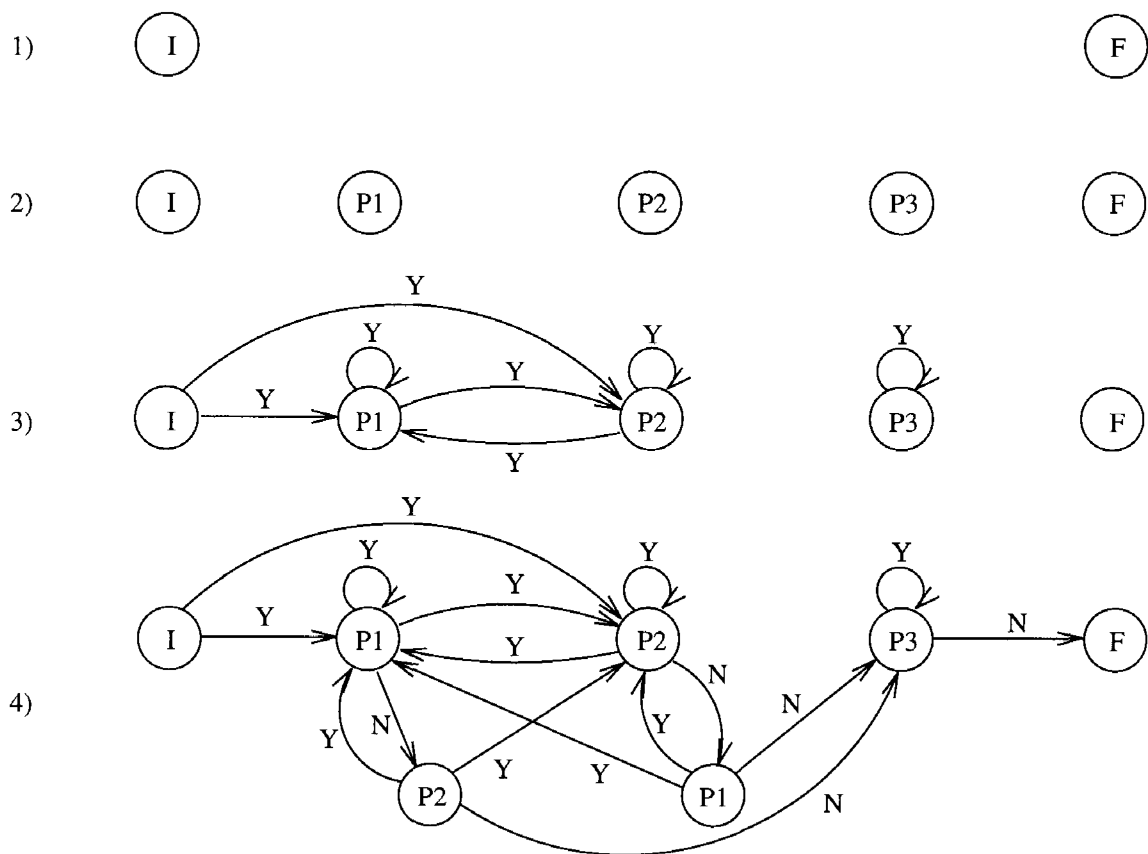


Figure 3.3: The control diagram of the previous figure in Bunke format.

high-level language PROGRESS [Zündorf & Schürr, 1991], allowing them to deal with the three cases of non-determinism described above.

3.2 The generative power of programmed grammars

In choosing a grammar type for any particular application, there is a play-off between the descriptive power of the grammar and the analysis efficiency of the grammar [Fu *et al.*, 1980]. Although many classes of patterns appear to be intuitively context-sensitive, context-sensitive grammars have rarely been used for pattern description, simply because of the computational complexity of parsing them. Context-free grammars are usually used instead.

It is in this trade-off that the advantage of programmed grammars lies. As seen from the example at the beginning of this section, a programmed context-free grammar can generate a context-sensitive language and thus has the descriptive power of a context-sensitive grammar. Consider the example just referred to. Although the productions are all context-free, the language $L = a^{2^n}, n \geq 0$ generated by the programmed grammar is context-sensitive. This makes it easier to write a grammar if a programmed grammar is used. Writing a grammar is also made easier by the fact that programming increases the modularity of a grammar. This means that the grammar writer does not have to consider all productions at once.

Programmed grammars also restrict the number of productions that can be applied at any one time (since only productions in the current node of the control diagram can be applied). This reduces the number of non-deterministic choices that have to be made during a parse. The overhead incurred by using a programmed grammar arises from the need to follow the edges in the control diagram, and is minimal. Thus, a parse using a programmed grammar normally has a lower computational complexity than the corresponding parse using a non-programmed grammar.

A detailed discussion of the generative power of programmed graph grammars can be found in [Bunke, 1982b].

Chapter 4

Fuzzifying graph grammars

Fuzzy string grammars have solved most of the problems that crisp string grammars have in coping with the all-pervasive problem, in image recognition, of noise. We therefore investigate the possibility of fuzzifying graph grammars so as to enable them to cope with noise.

4.1 Coping with noise

Almost all images contain some degree of noise. This can be caused by a variety of factors. In the case of black and white images, which we are considering, defects in the printing or scanning processes can cause either extra or missing pixels in the scanned image. These lead either to the deformation of symbols or to the presence of extra blobs in the image.

In the case of a hand-drawn image, a significant amount of variation is present in the symbols and in the spatial relations between them. This variation can be modelled as a superimposition of noise on an ideal (printed) version of the images. For example,

in handwritten music, stems are normally not vertical and they often do not touch the corresponding note-heads. Factors such as this obviously have an impact on the suitability of various methods for recognising handwritten music.

Additional noise is almost always introduced in the pre-processing and primitive extraction stages, so that the graph of primitives describing an image will normally contain some noise, even if the image was originally printed and scanned perfectly. For example, when staff lines are removed from a music score, the superimposed symbols will be affected to some extent (section 9.2).

Graph grammars assume implicitly, through the choice of a two-valued applicability predicate, a "crisp world", where each element of an image can be assigned with full certainty to one class or another. This limits their effectiveness in dealing with real images, where an element, considered locally, can belong to different classes and its final classification needs to be delayed until global information is available.

While backtracking can allow grammars to deal with ambiguity to some extent, it does not solve all problems. If a primitive element in an image is misclassified, backtracking through the grammar will not allow it to be reassigned to the correct class. Attempts to overcome this problem are: allowing primitives to be relabelled [Kato & Inokuchi, 1992], and allowing the parser to drive the segmentation and primitive extraction processes [Couasnon & Camillerapp, 1994]. If simple, attributed primitives are used, the primitive extraction does not have to make difficult classification decisions. Rather, the parsing layer can make these decisions based on the attributes of the primitives and, if necessary, on non-local information. This is the approach that we have used (see Chapter 10). Recall that an attributed, programmed graph grammar may have more than one input graph (Definition 3.2). This offers an alternative solution to the problem of errors caused by the primitive extraction layer, as a different input graph can be used for each possible set of primitives. This will allow the system to cope with segmentation errors as well as misclassification errors.

Another problem that cannot be solved by backtracking arises when two productions

attempt to convert the same symbol into different, higher-level symbols. Unless the applicability predicates for the productions overlap, only one of the two productions will succeed. Which production succeeds depends on the attributes of the symbol being converted. If, because of noise, the production which would have led to the correct interpretation of the image fails, backtracking will not allow the productions to be re-evaluated, since the symbol's attributes will not have changed.

Consider the example shown in Figure 4.1. It is obvious that a decision has to be made at some point as to whether the left-most note-head is hollow or solid. If this is done at the primitive extraction level or at an early stage in parsing, only local information can be utilised, limiting the level of intelligence which can be applied. If, on the other hand, the choice is delayed until global information is available, a better decision can be made. In the example below, a production that adds up the number of beats in the bar can be applied once all symbols have been identified. By comparing this with the actual number of beats in the bar (as specified by the time signature), the production can distinguish between correct and incorrect interpretations of that bar of the image.

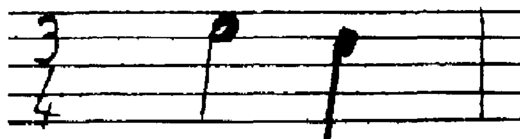


Figure 4.1: A difficult situation.

Assume that the number of beats in the bar of music contained in Figure 4.1 is known (it is in $\frac{3}{4}$ time). Once it is known that the only other symbol in the bar is a crotchet, it becomes obvious that the first note must be a minim, i.e., the blob is hollow.

To allow the formalism to deal with noise we have replaced the applicability predicate with a continuous certainty function (we say that we have "fuzzified" the graph grammar). This obviously eliminates the unnatural discontinuity inherent in a binary applicability predicate. More importantly, it allows final decisions to be delayed until

global information is available, so that the best (correct) interpretation of an image will emerge as the result of the application of the grammar.

Consider an applicability predicate of the form (*Top (Stem) CLOSE Left (Note-head)) OR (Bottom (Stem) CLOSE Right (Note-head))*. The binary function $CLOSE$ is very sensitive to noise: there is some state where moving one of the points by a miniscule amount will cause the output of $CLOSE$ to jump suddenly from true to false or vice-versa. A fuzzy function can give a much more realistic definition of $CLOSE$. The fuzzy functions used to define varying degrees of closeness in our prototype system are given in Figure 4.2.

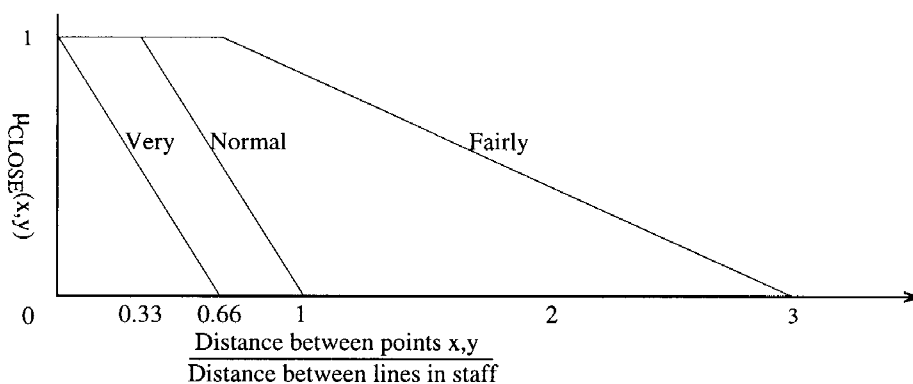


Figure 4.2: Definition of $CLOSE$.

This approach to finding a graph that is isomorphic to a crisp graph, and then determining the level of the match based on the value of a fuzzy certainty function over the attributes, is very similar to work done by K-P Chan and others. K-P Chan has described a method for learning these graph templates from fuzzy examples [Chan, 1996], which could possibly be used to give our system some degree of learning capabilities. The difference between this and other work is that we build a grammar on top of the matching mechanism. This allows our system to utilise non-local information (e.g., the number of beats in a bar) to decide which "template" is the best match for a group of nodes and edges, and thus to cope better with noise.

There is one type of noise which the mechanism developed cannot deal with. This is

the case where a primitive is simply noise, e.g., a smudge on the page. Such symbols cause a problem as they affect the certainty of all parse paths that attempt to classify them as a "normal" symbol. This problem is discussed in section 5.3.

4.2 Fuzzy string grammars

The fuzzy graph grammar formalism is a natural extension of fuzzy string grammars to two dimensions. We shall now briefly introduce fuzzy grammars. We do this for two reasons. Firstly, they serve as a good introduction to fuzzy graph grammars because of their lower complexity. Secondly, in example 5.4 we use fuzzy string grammars to examine the programming of fuzzy grammars, for the sake of simplicity.

Fuzzy string grammars have been used for a number of image recognition applications. A recent example is the FOHDEL (Fuzzy Online Handwriting DEscription Language) system, which uses fuzzy string grammars for online recognition of handwritten characters using a stylus pad[Malaviya *et al.*, 1996].

Figure 4.3 gives possible crisp and fuzzy certainty functions for the following two example productions.

P1. blob $\rightarrow$ note-head

P2. blob $\rightarrow$ augmentation dot

If V_T is a finite alphabet, then a **crisp language** L_c over V_T is defined as

$$L_c = \{x \mid x \in V_T^*\}.$$

A **fuzzy language** L_f over V_T is defined as

$$L_f = \{x, \mu(x) \mid x \in V_T^*, \mu(x) \in [0..1]\},$$

where $\mu(x)$ is the degree with which the string x belongs to the language.

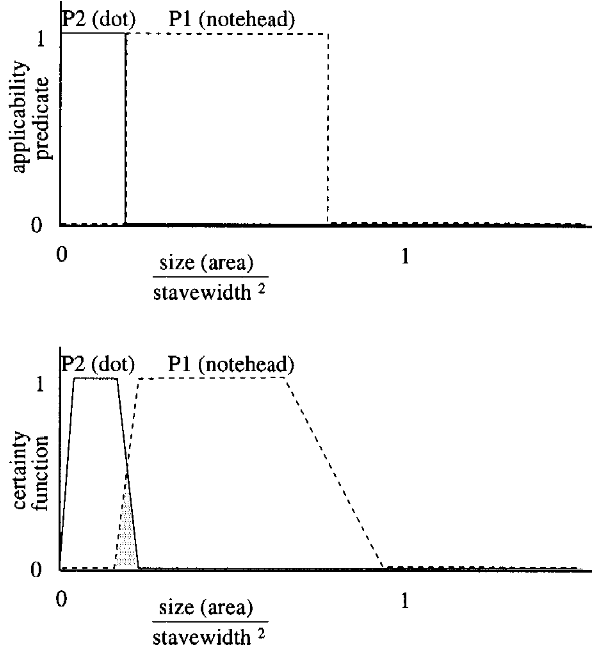


Figure 4.3: Comparison of crisp and fuzzy certainty functions.

In other words, each string belongs to a **fuzzy language** with a certainty in [0..1], as opposed to the crisp case, where each string is either in the language or not.

Definition 4.1 *A* **fuzzy string grammar** *is a quadruple* $G = (V_N, V_T, P, s)$ *where*

1. V_N *is a set of non-terminal symbols;*
2. V_T *is a set of terminal symbols (the alphabet);*
3. P *is a finite set of productions, the elements of which are expressions of the form* $\alpha \xrightarrow{\mu} \beta$ *,where* $\alpha, \beta \in (V_T \cup V_N)^\star$ *and* $\mu \in [0..1]$*; and*
4. $s \in V_N$ *is the starting symbol.*

A fuzzy string grammar is thus identical to a crisp string grammar, except for the certainty value which is associated with each production.

Definition 4.2 *Let* $\alpha_1, ..., \alpha_m$ *be strings in* $(V_N \cup V_T)^\star$ *and let* $\alpha_1 \xrightarrow{\mu_1} \alpha_2, \alpha_2 \xrightarrow{\mu_2} \alpha_3, ..., \alpha_{m-1} \xrightarrow{\mu_{m-1}} \alpha_m$ *be productions in a grammar* G*. Then* α_m *is said to be* **derivable**

from α_1 *in* G *(written* $\alpha_1 \stackrel{\mu}{\Longrightarrow} \alpha_m$*) with certainty* $\mu = \bigwedge_{i=1}^{m-1} \mu_i$

That is, the certainty of the derivation chain is the fuzzy conjunction of the certainties associated with productions in the chain. MIN (minimum) is normally used for the fuzzy conjunction in definition 4.2.

Definition 4.3 *The* **language generated by a fuzzy grammar** G *with starting symbol* s, $L(G) = \{x, \mu(x) \mid s \stackrel{\mu}{\Longrightarrow} x \ , \ \mu(x) = \bigvee_{i=1}^{n} \mu_i \ , \ \mu_1..\mu_n$ *are the certainties of the derivation chains* $s \stackrel{\mu_i}{\Longrightarrow} x\}$.

That is, the degree to which a string belongs to the language is the fuzzy disjunction of the certainties of its derivation chains. MAX (maximum) is normally used for the fuzzy disjunction in definition 4.3.

Chapter 5

Programming fuzzy grammars: an informal introduction

As far as we are aware, no previous work has been done on the field of programming fuzzy grammars. Since programming can be advantageous for crisp grammars, we felt it worthwhile to examine the possibility of programming a fuzzy grammar.

5.1 A method for programming fuzzy grammars

Consider the changes to a grammar that programming entails :

1. There is no distinction between terminal and non-terminal symbols.
2. Start by applying a production in a node at the end of a Y-edge from the initial node.

3. If a production is successful, follow a Y-edge in the control diagram to determine which production should be applied next, else follow an N-edge.

4. The derivation sequence is complete and the parse is successful if the final node in the control diagram is reached.

5. The parse is unsuccessful if the final node in the control diagram cannot be reached.

If we wish to apply this to a fuzzy grammar, 1 and 2 are not affected at all. Points 4 and 5 still hold as long as a suitable method is found for choosing which production to apply at any time, with the condition that the certainty of the derivation chain is determined as given in definition 4.2. However, point 3, which defines which production to apply at any time, must be modified. Since it relies on the success or failure of a production, it will be directly affected by the replacement of the binary applicability predicate with a fuzzy one.

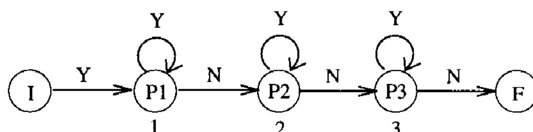


Figure 5.1: A simple control diagram.

Consider the very simple control diagram shown in Figure 5.1 for a grammar containing three productions, labelled P1, P2 and P3. A normal (crisp) graph grammar, programmed according to this control diagram, might yield the parse path shown in Figure 5.2(A), if production 1 can be applied twice (there are two subgraphs isomorphic to its left-hand side) and the others can each be applied exactly once. If backtracking is allowed, then the state space search tree will be that shown in Figure 5.2(B). The labels in the states (circles) represent the current position in the control diagram, and labels associated with edges represent the production applied and the edge followed. $P1^1$ and $P1^2$ represent two instances of production 1. Since it makes no difference in which order the two instances of production 1 are applied, there

is an opportunity for pruning this search tree. This will be discussed in Chapter 7, but we shall ignore it for now for the purpose of simplicity.

The state of the parse is fully determined by the current graph and the position in the control diagram. Of course, if we analyse it more closely, we can include other variables in the state, such as the current position in the code being executed. However, we know that when a production is applied it will either succeed (with a certainty less than or equal to one, in the fuzzy case) or fail. We can thus ignore the sub-steps of this process and consider the application of a production (whether successful or not) as an atomic operation. In the case where there is more than one production in a node, the failure of all productions in the node can be seen as an atomic operation, and the reasoning above still holds [Fahmy & Blostein, 1992b]. The graph under transformation is only changed if a production is successful, which means that any change to it will coincide with the following of a Y-edge in the control diagram. The position in the control diagram can obviously only change if either a Y- or N-edge is followed. The state thus changes if and only if an edge in the control diagram is followed. Each edge in Figure 5.2(A) corresponds to a change in the state of the parse and therefore to the following of an edge in the control diagram.

In the case of a fuzzy graph grammar, on the other hand, a problem arises from productions with a certainty strictly between 0 and 1, since the crisp control diagram relies on a production either succeeding or failing. Should a Y- or N-edge in the control diagram be followed after the application of such a production? A solution that immediately springs to mind is to add a branch to the state space search tree for both options. We label these branches 'Y' and 'n' respectively. (Notice that the second type of edges are labelled with a lower case 'n', to distinguish them from the N-edges.) A Y-edge can thus be interpreted as, "we consider the production instance to have succeeded" and an n-edge can be interpreted as, "we consider the production instance to have failed." N-edges have the same meaning as in the crisp case (no production in the current node of the control diagram can be applied with a non-zero certainty, so an N-edge in the control diagram must be followed), and thus remain

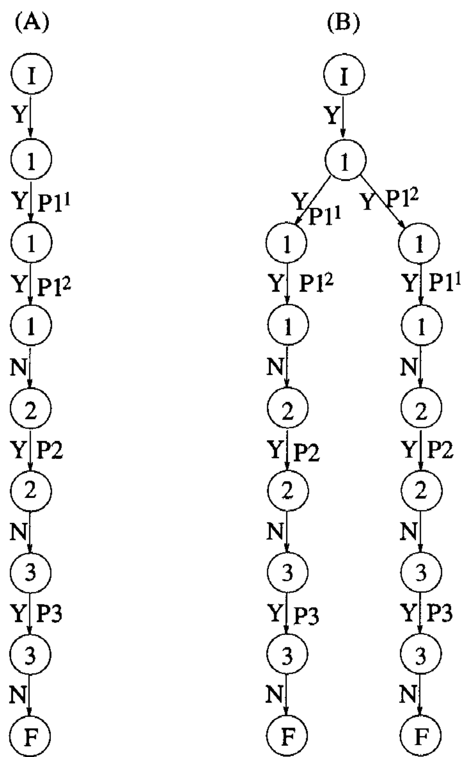


Figure 5.2: Search trees (A) without and (B) with backtracking.

unchanged.

Applying this to the grammar programmed by the control diagram in Figure 5.1 gives the new search tree shown in Figure 5.3. Since this figure depicts the full search tree (which corresponds to allowing backtracking), it can easily be derived from Figure 5.2(B) by replacing every Y-edge with two edges: one labelled 'Y' and one labelled 'n'. We are thus assuming that every production instance in this example has a certainty strictly between 0 and 1. If a production instance has a certainty of 1, the corresponding n-edge would fall away. A production instance with a certainty of exactly 0 cannot be applied and is thus ignored. If this production is the only potential production instance for the current node in the control diagram, an N-edge in the control diagram will be followed, otherwise another production instance will be applied.

The edges labelled 'n' thus correspond to the failure of an individual production instance. This does not cause an edge in the control diagram to be followed, since an N-edge in the control diagram is only followed if no successful production instance can be found for the productions in the current node of the control diagram. As before, the edges in Figure 5.3 labelled 'N' correspond to the following of an N-edge in the control diagram.

The approach above has two main problems. As can be seen in the figure, it leads to a drastic expansion of the search tree. The second problem is more important. If MIN is being used for the fuzzy AND, following Y-edges in the search tree (applying productions) will tend to decrease the certainty of the corresponding derivation path, while n-edges (considering productions to have failed) do not affect the certainty of the path. n-edges are thus favoured, which means that the "best" path will tend to be the one where the least productions are applied. This is clearly undesirable, and a solution must be found. This problem is illustrated by the following example.

Imagine a grammar consisting of the following three productions and programmed according to the control diagram given in Figure 5.1.

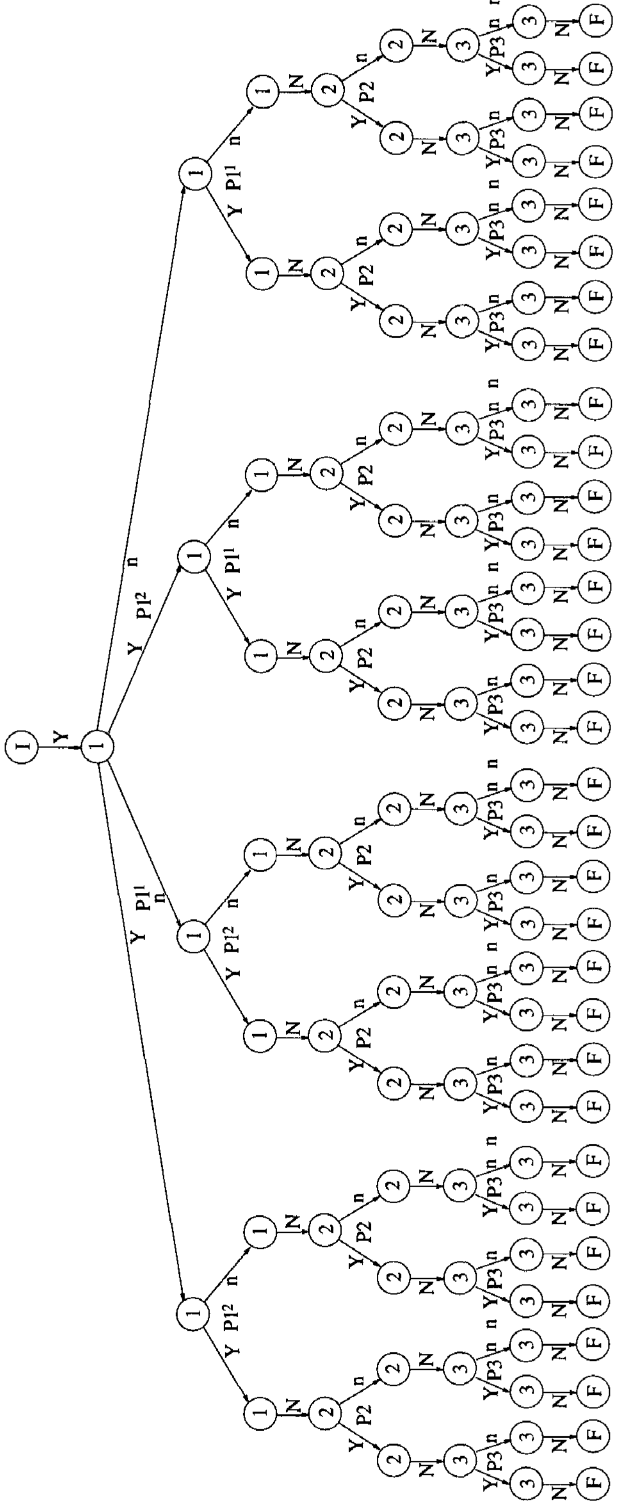


Figure 5.3: Search tree for fuzzy grammar.

P1. blob $\longrightarrow$ Note-head

P2. line $\longrightarrow$ Stem

P3. Note-head,Stem $\longrightarrow$ Note

An image consisting of a stem and a blob forming a note might lead to instances of the productions with certainties of 0.7, 0.8 and 0.9 respectively. The state space search tree would be that shown in Figure 5.4. The final graph and certainty associated with each parse path is as follows. Since we use the aggregate operator of MIN, these certainties are the minimum of the certainties associated with the production instances in each path. The final graph with the highest certainty is that which has not been changed at all, and still consists of a blob and a line. This is clearly undesirable.

Parse Path	**Certainty**	**Final Graph**
1	0.7	Note
2	0.7	Note-head Stem
3	0.8	Note-head line
4	0.7	blob Stem
5	1	blob line

A solution to this problem is to ignore all n-edges in the search tree. This approach, however, has the problem that it favours the productions applied first. Although this problem can occur in the case of crisp grammars, it occurs much more frequently in the case of fuzzy grammars.

For example, consider the two productions

P1. blob $\rightarrow$ note-head

P2. blob $\rightarrow$ augmentation dot

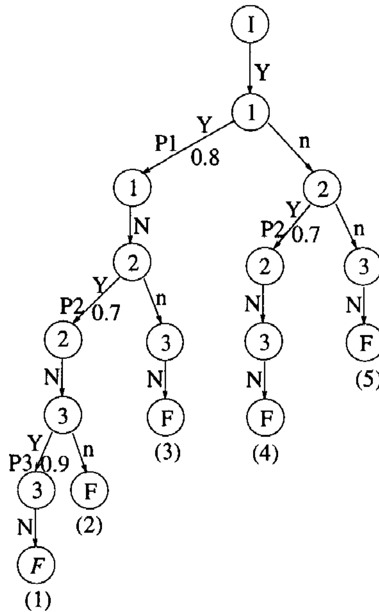


Figure 5.4: State space search tree for a simple example.

with the fuzzy certainty functions given in Figure 4.3. Imagine that, for a given blob, $P1$ has a certainty of 0.3, while $P2$ has a certainty of 0.7. If the control diagram specifies that $P1$ is applied before $P2$, then $P2$ cannot be applied, as the blob will already have been converted into a note-head by $P1$, even though $P2$ has a higher certainty value.

The problem arises because the two productions "compete" for the same symbol. We begin the discussion of a possible solution to this problem by defining the concept of competing productions and production instances.

Definition 5.1 *Two* **production instances compete** *if the subgraphs matching their left-hand sides both contain the same graph node (their redices are conjunct).*

The instances of productions $P1$ and $P2$, with certainties of 0.3 and 0.7, given above are examples of competing production instances.

Definition 5.2 *Two* **productions compete** *if there is a possibility of them having competing instances.*

So, two productions compete if their left-hand sides have at least one symbol type in common. Furthermore, it must be possible for both productions to have a non-zero certainty value when the common symbol types in their left-hand sides match the same graph node.

Crisp applicability predicates are normally defined in such a way that, when productions have a common symbol type in their left-hand sides, only one of them will succeed at a time. The crisp applicability predicates defined in Figure 4.3 are an example of this. In this case, the order in which productions are applied makes no difference. However, the fuzzy certainty functions of such productions will normally intersect, yielding a region of the input domain where more than one of the productions has a non-zero certainty. This region is shaded in Figure 4.3.

The crisp subsets for dots and note-heads of the set of blobs are disjoint (their intersection is the empty set), while the intersection of the corresponding fuzzy subsets is non-empty. This means that in the the crisp case the control diagram can specify the sequential application, in either order, of the productions, but in the fuzzy case they must be applied in parallel (yielding a branch in the state space search tree).

In a non-programmed grammar, competing productions can be applied in any order. This means that the state space search tree associated with a parse contains a different branch for each possible ordering. With the help of backtracking, a parse engine can find the best ordering of the productions. As was mentioned in Section 3.1, Fahmy and Blostein have introduced a modified form of grammar programming, which includes subgraphs and "non-programmed" grammar sections by allowing more than one production in a node in the control diagram.

We overcome the problem of competing productions by using this extension to group them in a single node in the control diagram. This means that the state space search tree associated with a parse will contain a branch for each different choice of the competing productions. As mentioned in Section 3.1, such a control diagram can be converted into one with only one production per node through the addition of extra

nodes and edges.

If we delete all n-edges from Figure 5.3 as well as the subtrees originating from the nodes at the end of these n-edges, we get a state space search tree identical to that shown in Figure 5.2(B), except that the productions and final graphs have certainties associated with them. In this example, the programmed fuzzy grammar thus has a state space search tree identical to that of the crisp case with backtracking allowed, except for the certainty values. Of course, if any of the productions in the example compete, the control diagram (shown in Figure 5.1) would have to be changed, as explained above, and the state space search tree would be different.

For example, imagine that productions $P2$ and $P3$ compete. They would then have to be grouped together in a single node of the control diagram. This control diagram is given in Figure 5.5. The resulting search tree is given in Figure 5.6.

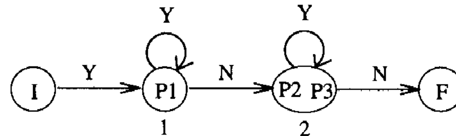


Figure 5.5: Control diagram for competing productions.

In general, fuzzy certainty functions will often evaluate to greater than zero in cases where the corresponding crisp functions evaluate to zero. In practice this normally leads to the search tree associated with the fuzzy grammar being substantially bigger than the one associated with the crisp grammar.

5.2 Augmenting the control diagram with output nodes

The mechanism described in the preceding section allows the realization of the aims mentioned in Chapter 4. Since all decision branches are developed, no final decisions are made until all information is available. Furthermore, we can force a condition

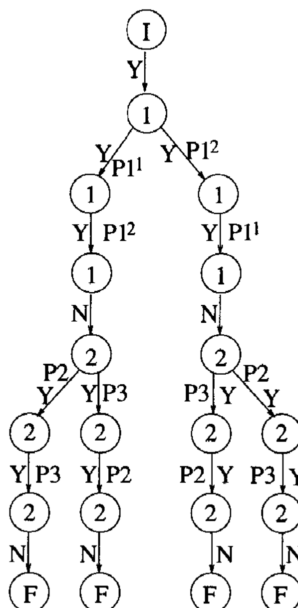


Figure 5.6: State space search tree for competing productions.

(e.g., a bar must contain the correct number of beats) to be satisfied. One way to do this is by writing a suitable crisp production that checks if the condition is satisfied, and by not having an N-edge emanating from its node in the control diagram. Since the sole purpose of these productions is to check if a constraint is satisfied, we refer to them as "check productions". There is, however, no real difference between them and other productions. The fact that the crisp production suggested has no corresponding N-edge means that if the graph does not satisfy the given condition, the production will fail (we will not be able to apply it). An N-edge in the control diagram must thus be followed, but since there is no such N-edge, that parse path reaches a "dead-end" and cannot be developed further. In other words, failure of the graph to satisfy the given condition causes that parse path to be abandoned and alternative parse paths to be explored.

This approach has a serious problem. If no parse path yields a final graph with the correct number of beats, the parse will simply fail. It would be preferable in such a situation if the parse could produce the interpretation (output graph) with the highest certainty. The user could then be alerted to the mistake and could correct

the symbol that was mis-classified.

This can be achieved in the fuzzy programmed grammar by setting the certainty of the production to return a very small value, but greater than zero, when the number of beats in the bar is incorrect. The resulting graph will have an associated certainty lower than the certainty of any final graph with a correct number of beats (if MIN is used for the fuzzy AND), but will provide the best possible output if there is no final graph with the correct number of beats. We say that a path is "successful" if it satisfies all check productions.

This feature cannot be obtained using a crisp grammar, as the parser will repeatedly be forced to backtrack until it fails, yielding no output graph.

Non-programmed grammars, when used as recognisers (bottom-up), reduce the input into a single "start" symbol if a parse is successful. We can define three subsets of the symbols used in a parse. The first consists of the terminal symbols. In the case of image recognition this corresponds to the primitives extracted from the image. The second consists of symbols that convey the high-level meaning required for image understanding. In a music score these would correspond to notes, rests, slurs, etc. The third subset contains only the "start" symbol. We refer to these three subsets as classes one, two and three.

The parse DAG (Directed Acyclic Graph) resulting from a successful parse of an input graph indicates the relationships between the symbols in these three subsets.

Programmed graph grammars normally only transform the input graph until all nodes (and edges) in the graph represent (are labelled with) symbols in the second subset defined above. For example, in [Bunke, 1982a] a programmed graph grammar is used to transform an input graph (representing primitives such as lines in a circuit diagram) into a graph containing nodes such as resistors and capacitors. The fact that such a parse is successful simply means that a path could be followed through the control diagram. This is perhaps too relaxed a constraint. Simply because lines could

be transformed into higher-level symbols does not mean that those symbols form a valid circuit diagram. If we wish to ensure that the symbols recognised conform to higher-level rules (by using the grammar mechanism), we need to define productions and a control diagram that will drive a full parse of the input graph, reducing it to a graph that contains the "start" symbol. This allows constraints to be defined as described above. Note that, unless explicitly specified, this final graph can contain other symbols as well as the "start" symbol; for example, primitives that were not mapped onto any symbol part with a certainty greater than zero.

As is the case with non-programmed grammars, the parse DAG resulting from a successful parse with a programmed grammar can be examined to yield the symbols from the second subset defined above and the relationships between them and the input symbols. In image understanding it is not normally necessary to know which primitives make up which higher-level symbol, rather it is necessary to find the higher-level symbols and the relations between them. For example, in the analysis of circuit diagrams it is not necessary to know which lines make up each resistor: we simply need to know which resistors, capacitors, etc. are present in the image and how they are interconnected.

Whenever this is the case, we can take advantage of the programming mechanism to simplify the process of extracting the symbols in our second subset from the parse DAG. If the grammar can be split into two sub-grammars, one that transforms the input graph into a graph containing only class two symbols and one transforming these into the "start" symbol, then we can mark the position in the control diagram which corresponds to the graph containing only class two symbols. If we store the graph associated with each parse path at that point, then there is no need to explicitly build up or analyse a parse DAG. As well as saving computing time, this greatly reduces memory requirements, since it obviates the need to store the full parse DAG for each parse path.

We have introduced this feature into the programming mechanism by adding a new

node label to the control diagram to represent an "output" node. When an output node in the control diagram is reached, the current graph associated with that path is stored and a Y-edge from the output node is followed. When the final node in the control diagram is reached, the stored graph is output rather than the final graph (which will contain only the "start" symbol). This addition to the control diagram and to the algorithm for parsing an input graph according to the control diagram is stated formally in section 6.1.

5.3 Noise productions

In the preceding sections we have developed techniques for dealing with malformed primitives and symbols, and with varying spatial relationships between symbols. We have not, however, dealt with the case where a primitive is actually noise (we call this a "noise primitive").

This can occur, for example, when a smudge present on the page is recognised as a blob or similar primitive. In the case of music recognition, spurious lines can be left from errors in the staff-line removal process, especially if a naïve algorithm is used. This can be seen in Figure 2.1: the horizontal line between the base clef and the time signature in this image is an example of such a spurious line.

It can be said that the problem is largely caused by our use of very generic primitives in this example. If the primitive extraction layer only recognised primitives expected in the image and ignored other components, most of the noise primitives would correctly be ignored (although the possibility of some being incorrectly recognised as image primitives cannot be completely discounted). There is, however, a trade-off here. If the primitive extraction layer is made stricter, malformed primitives that form part of valid symbols are more likely not to be recognised, causing errors of omission which cannot be corrected during the parsing phase. It is because of this that we have decided to deal with the problem of noise primitives within our grammar formalism.

Note that the problem is not limited to our system: in any recognition system there is a possibility that noise be incorrectly recognised as a primitive, although the probability of this happening may be low. These incorrect primitives, which are actually noise, are likely to confuse the recognition process. In the case of a syntactic approach, it would not be possible to reduce the input into a single "start" node, and a successful parse would not be found.

To remedy this problem we have introduced a class of productions called "noise productions". These productions delete symbols which are actually noise. That is, the left-hand side of these productions is a graph containing only the noise symbol and the right-hand side is the empty graph. The certainty associated with this production is the certainty that the symbol making up its left-hand side is noise.

Noise productions are different from all other productions because of the empty right-hand side and because of the different method of calculating their certainty values, which is needed because these values, at least in our approach, are dependent on the certainty values associated with other productions. The latter fact means that they need special treatment from the parsing algorithm (see Section 6).

There are two cases to consider: the noise symbol which is deleted by the noise production is either a primitive, or it is a class two symbol (see section 5.2 for a definition of class two symbols). We shall first consider the case where it is a primitive.

We have implicitly defined noise primitives as primitives that do not form part of a class two symbol defined in our grammar. The more certain we are that a primitive matches part of a symbol defined in our grammar, the more certain we are that it is not noise. Conversely, if a primitive does not match a symbol part, or only matches one very poorly, it is more likely to be a noise symbol. We thus define the certainty of a noise production instance to be the complement of the disjunction of the certainties associated with competing production instances, or 1 if there are no competing production instances (definition 5.1). That is,

Definition 5.3

$$\mu_{noise} = \begin{cases} \neg \bigvee_{i=1}^{n} \mu_i & n \geq 1 \\ 1 & n = 0 \end{cases}$$

where n is the number of competing applications of productions.

For example, if a blob in a music score can be mapped onto an augmentation dot with a certainty of 0.7 or onto a solid note-head with a certainty of 0.4, then the certainty of that blob being noise is $\neg(0.7 \vee 0.4) = 0.3$.

The other case that must be considered is where the noise deleted by the noise production is not a primitive but a class two symbol. For example, if a stem has no corresponding note-head, it can be classified as noise. The approach which immediately springs to mind is to use a noise production with the certainty function defined above for the case in which the noise is a primitive (definition 5.3). This can, however, give rise to some problems. Consider the example illustrated in Figure 5.7.

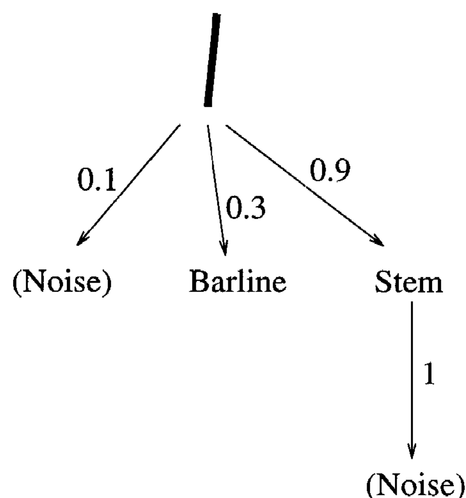


Figure 5.7: A problem with noise productions.

The line in this example can be mapped onto a stem with a certainty of 0.9 and a bar-line with a certainty of 0.3. This means that the certainty of the line being noise is 0.1. However, since there is no note-head present in this example, the certainty of

the stem symbol being noise is 1. The parse path *line* $\longrightarrow$ *stem* $\longrightarrow$ (*noise*) thus has a certainty of $0.9 \wedge 1 = 0.9$ (using minimum). This means that the line is transformed into noise (via a stem) with a certainty of 0.9. This is much higher than the certainty of it being a bar-line, which is undesirable.

The problem is effectively caused by an incorrect interpretation of the production *stem* $\longrightarrow$ (*noise*). In the example we have just described, it is effectively being interpreted as saying that the line primitive forming the stem is noise. Rather, it should be interpreted as saying, with hindsight, that the production mapping the line primitive onto a stem should not have been applied. We can force this interpretation by associating a very low (but non-zero) certainty with the noise production that deletes the stem. This will give that parse path a very low certainty, which means that the parse path containing the production that maps the line into a bar-line will have a higher certainty.

We can avoid the problem by associating a low certainty with noise productions that delete class two symbols. The certainty must be non-zero, as a production with a certainty of zero will simply not be applied, and the certainty of the corresponding parse path will remain unchanged. So, we choose a low certainty value (e.g., 0.001) and associate it with each of these productions.

We do not have a theory to prove that the certainty values we associate with noise productions are correct, although we have tried to find it. Our method is, at the moment, only supported by empirical evidence.

Notice that if a noise primitive cannot be mapped onto a class two symbol at all (no production transforming that noise primitive has a non-zero certainty), then it does not present much of a problem within our existing framework. At the end of a successful parse, the output (and the final) graph will contain the noise primitive as well as the recognised class two symbols. Since no productions were applied using the noise primitive, its presence will not have affected the certainty value of the parse path. An examination of the output graph can easily eliminate these unwanted,

"unused" primitives. This can be done with a production that deletes nodes of a certain primitive type, applied after no other productions transforming that type of primitive can be applied any more.

We abandoned this approach because of problems that arise when a noise primitive can be mapped onto a symbol, or part of a symbol, with a certainty greater than zero. For example, imagine that a line primitive can be mapped onto a stem with a low certainty, and that it cannot be mapped onto anything else. Then the production mapping the line onto a stem will be applied in every parse path, and if the standard aggregation function of minimum is used to calculate the certainty of parse paths, every path will have the low certainty of that production.

Similarly, if a noise primitive can be transformed (directly or indirectly) into one and only one class two symbol, then every final (and output) graph will contain that symbol. This could cause the check productions discussed in section 5.2 to fail. For example, if a noise primitive is mapped onto a breve (whole-note) in a bar of music, no parse path will have the correct number of beats. This is a problem irrespective of the aggregation function used for calculating the certainty of the parse paths, and justifies our decision to introduce noise productions.

As is the case with other productions, each noise production must be grouped in a node of the control diagram with the competing productions (see Chapter 13).

It should not be difficult to write a routine that "decorates" a grammar by adding noise productions to it. This would reduce the amount of work involved in writing the grammar, and would simplify the task, by removing the need for the grammar writer to consider noise productions.

5.4 Advantages of programming a fuzzy grammar

It is clear that some of the advantages of programming grammars are lost by allowing backtracking, and especially by allowing fuzzy certainty functions to be associated with productions. Most notable is the low computational complexity of parsing which "deterministic" control diagrams provide. The abilities of the mechanism described in the previous section could be provided by a non-programmed fuzzy grammar. It is thus necessary to examine whether the advantages claimed for the programming of a crisp grammar (section 3.2) are obtained from programming a fuzzy grammar.

In a non-programmed grammar, any production can, potentially, be applied at any time. Programming greatly reduces the number of productions that have to be tried in searching for possible redices. This is particularly important in the case of graph grammars, since finding an isomorphic (attributed) subgraph is an NP-complete problem.

Programming can increase the generative power of a fuzzy grammar in exactly the same way as it increases the generative power of crisp grammars. This has a favourable impact on the computational complexity of a parse, as a context-free grammar can often be used to define a context-sensitive language. This also eases the task of writing a grammar to describe a particular image class, as does the fact that the number of productions which have to be considered at any one time is greatly reduced in a programmed grammar.

The cost of programming a fuzzy grammar lies in the complexity of implementing the parser, as it introduces extra levels of non-determinism that must be dealt with.

We shall now develop a small example, as an illustration of the power of programming fuzzy grammars. For the sake of simplicity, the example utilises string grammars. The points made, however, are equally relevant to the use of graph grammars. We shall start by developing a crisp grammar and will then fuzzify it.

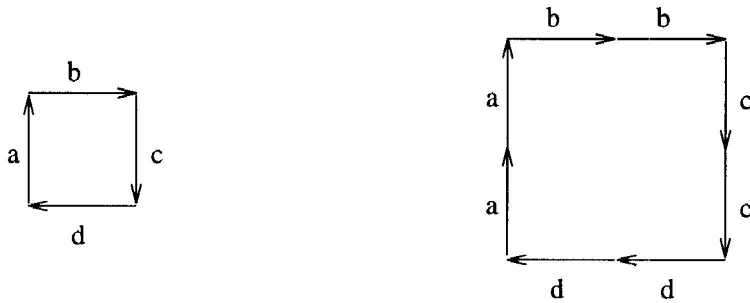


Figure 5.8: Primitives making up the image.

Imagine that a primitive extraction algorithm yields a string of the letters $a..d$ to denote line segments of unit length and with the directions shown in Figure 5.8. Then the language $L = \{a^n b^n c^n d^n \mid n \geq 1\}$ describes squares with sides of length $n \geq 1$. This is known to be a context-sensitive language, and can be generated either by a context-sensitive grammar (Figure 5.9) or by a programmed context-free grammar (shown in Figure 5.10). The programmed context-free grammar is, in our opinion, significantly easier to write and to understand (once, that is, one is acquainted with the use of control diagrams).

Context-sensitive grammar :
$G = \{V_N, V_T, P, S\}$
where
$V_N = \{S, A, B, C, D, X, Y\}$
$V_T = \{a, b, c, d\}$
$P = \{S \longrightarrow BXY$
$B \longrightarrow aBbC$
$B \longrightarrow abC$
$Cb \longrightarrow bC$
$CX \longrightarrow XcD$
$CX \longrightarrow cD$
$Dc \longrightarrow cD$
$DY \longrightarrow Yd$
$DY \longrightarrow d\ \}$

Figure 5.9: Context-sensitive grammar to generate the language $a^n b^n c^n d^n$.

We could use either of these grammars to drive the recognition of squares. The

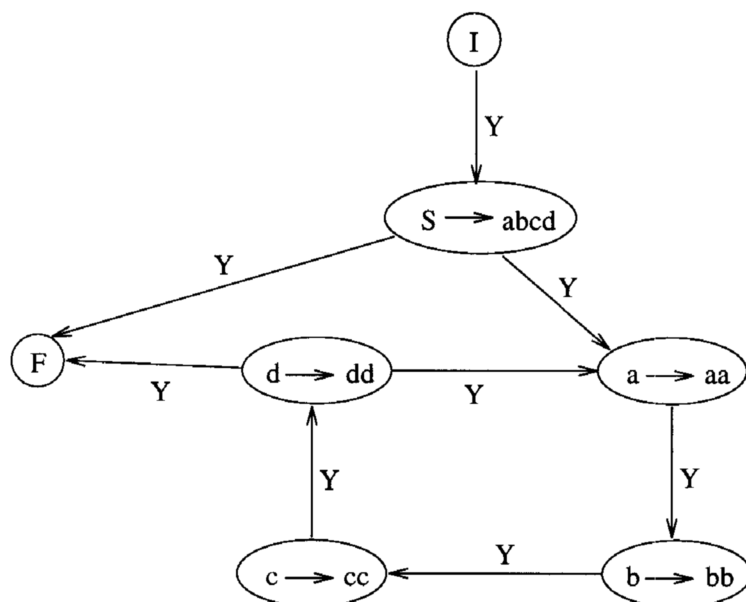


Figure 5.10: Programmed context-free grammar to generate the language $a^n b^n c^n d^n$.

programmed context-free grammar to recognise the language mentioned above can trivially be derived from the generating grammar, and is shown in Figure 5.11. Note that backtracking is needed to take care of the non-determinism present in the choice of Y-edges from the nodes I and 1, as an incorrect choice at these points will lead to a dead-end in the parsing process. If an N-edge is added from node 2 to node 3, this backtracking is no longer needed.

Note that productions in programmed grammars are normally written as transformations, i.e., they are applied as written. This is why the productions in the generator (figure 5.10) are written with a top-down ordering, while those in the recogniser (figure 5.11) are written with a bottom-up ordering.

Suppose, now, that we wish to cater for the possibility of lines not being perfectly straight, or perhaps not being perfectly connected. As was suggested in Chapter 4, we can achieve these aims by fuzzifying the grammar.

We can easily associate certainties with the productions in our context-free programmed grammar as follows.

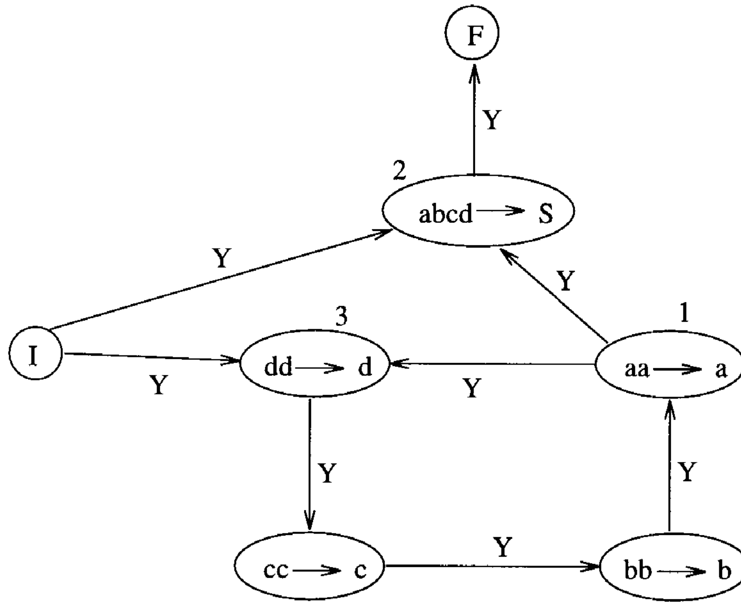


Figure 5.11: Programmed context-free grammar to recognise the language $a^n b^n c^n d^n$.

$abcd \xrightarrow{\mu_1} S$

where $\mu_1 = up(a) \wedge right(b) \wedge down(c) \wedge left(d) \wedge close(a,b) \wedge close(b,c) \wedge close(c,d) \wedge close(d,a)$

$a^1 a^2 \xrightarrow{\mu_2} a$

where $\mu_2 = close(a^1, a^2) \wedge collinear(a^1, a^2)$

$a.start = a^1.start$

$a.end = a^2.end$

and similarly for b, c and d.

start and *end* are attributes associated with the symbols a, b, c and d. *up*, *down*, *left*, *right*, *close* and *collinear* are fuzzy membership functions which use the *start* and *end* attributes of the given symbols to compute the appropriate certainty value in the range [0..1]. For example, *up* returns a value based on the angle between the symbol and the standard y axis, as shown in Figure 5.12. The productions $bb \longrightarrow b$, $cc \longrightarrow c$ and $dd \longrightarrow d$ can be given certainties in an analogous manner.

The use of a programmed fuzzy grammar has allowed us to describe the language and

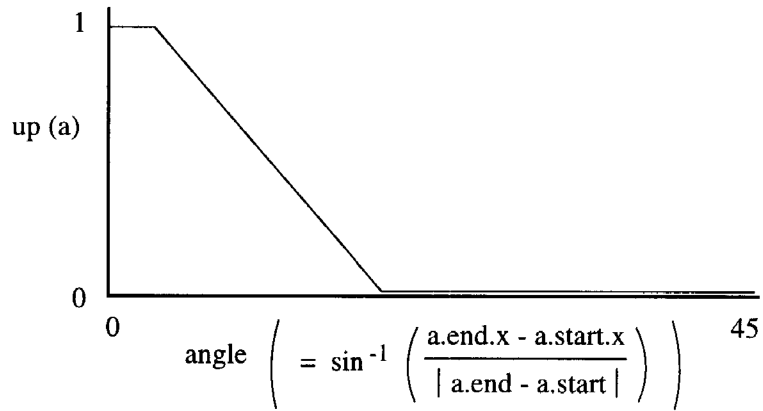


Figure 5.12: A possible version of the fuzzy membership function *up*.

to cater for noise in this example very easily. By contrast, how to go about fuzzifying the corresponding non-programmed context-sensitive grammar would not appear to be as intuitive.

Chapter 6

Programmed fuzzy attributed graph grammars

In the previous chapters we have sketched a mechanism for using programmed fuzzy attributed graph grammars for two-dimensional image recognition. In this chapter we formally define it. This chapter also includes a brief discussion of the type of fuzzy certainty functions used in the extended grammar.

6.1 Definitions

We build on the definitions given in Chapter 2. Definitions 2.1 to 2.5 apply unchanged.

Definition 6.1 *Let V and W be two alphabets for labeling the nodes and edges in a graph, respectively, and A and B be the sets of node and edge attributes, respectively.*

A **production in a fuzzy attributed graph grammar** *over (V, W) with attributes (A, B) is a five-tuple $p = (g_l, g_r, T, \mu, F)$ where*

1. *g_l and $g_r \in \Gamma(V, W)$ are u-graphs, the left-hand and right-hand side, respectively;*

2. *$T = (L_w, R_w)_{w \in W}$ is the embedding transformation with the exact form of L_w and R_w dependant on the class of embedding transformation being used;*

3. *$\mu : \Gamma_{(A,B)}(V, W) \rightarrow [0, 1]$ is the applicability predicate; and*

4. *F is a finite set of partial functions $f_a : N[g_r] \rightarrow D_a$ and $f_b : E[g_r] \bigcup \mathrm{EMB}(g_r, g) \rightarrow D_b$ with $a \in A$, $b \in B$, and $g \in \Gamma_{(A,B)}(V, W)$. f_a are the node attribute and f_b are the edge attribute transfer functions.*

The definition of monotone and context-free productions (definition 2.7) still holds.

Definition 6.2 *The* **direct derivation of a graph** g' **from a graph** g *by means of a fuzzy production p (denoted $g \overset{p,\mu}{\rightarrow} g'$) is defined by the following procedure.*

1. *Check whether the left-hand side of the production occurs as a subgraph in g and check whether the certainty function is non-zero (μ is > 0) for this occurrence. If both conditions are fulfilled, perform steps 2 to 4. We say then that the derivation (and the production) succeeds with a certainty of μ.*

2. *Replace the left-hand with the right-hand side.*

3. *Transform the embedding of the left-hand side in g into that of the right-hand side in g'.*

4. *Attach attributes to the inserted right-hand side according to the functions f_a and f_b.*

Definition 6.3 *Let P be the finite set of productions in a fuzzy attributed graph grammar. A* **control diagram over P** *is an unattributed graph with the set $2^P \cup \{I, F, O\}$ as node labels and the set $\{Y, N\}$ as edge labels. The following conditions hold true:*

1. *There exists exactly one initial node n_I labelled with I.*

2. *There exists exactly one final node n_F labelled with F.*

3. *There exists no edge terminating in n_I.*

4. *There exists no edge originating from* n_F.

5. *There exists at least one output node* n_O *labelled with* O *(see page 57).*

6. *There exists no path between two output nodes.*

7. *If two productions can be applied to transform the same symbol into differing symbols, they must be grouped in the same node. That is, they must be grouped in the label of a single node in the control diagram.*

We specify that there must not be a path between two output nodes because it does not make any sense for a parse path to have more than one output graph associated with it. The last condition solves the problem of competing productions discussed in the previous chapter.

Definition 6.4 *A* **programmed fuzzy attributed graph grammar** *is a 7-tuple* $G = (V, W, A, B, P, S, C)$ *where*

1. V *and* W *are alphabets for labelling the nodes and edges, respectively;*

2. A *and* B *are finite sets of attributes for nodes and edges respectively;*

3. P *is a finite set of fuzzy productions;*

4. S *is a set of initial graphs; and*

5. C *is a control diagram over* P.

Now that we have laid down the necessary supporting definitions, we are in a position to formalize the parsing algorithm.

Algorithm for applying productions in a programmed fuzzy attributed graph grammar:

1. **Choose** *a Y-edge emanating from the initial node in the control diagram. The node at which the edge terminates becomes the current node.*

2. *Find all subgraphs which are isomorphic to the left-hand sides of all productions in the current node which have a non-zero associated certainty value. Once all of these subgraphs (redices) have been found and the certainty of all productions except for noise productions are known, calculate the certainty of the noise productions, according to definition 5.3.*
   If there are no such subgraphs (redices) then

   > **Choose** *an N-edge emanating from the current node in the control diagram. The node at which the edge terminates becomes the current node.*

   else

   (a) **Choose** *one of the redices and apply the corresponding production instance to the current graph.*

   (b) **Choose** *a Y-edge emanating from the current node in the control diagram. The node at which this edge terminates becomes the current node.*

3. *If the new current node is an output node (it has the label "O") then*

   (a) *Store a copy of the current graph and denote it as the output graph associated with this parse path.*

   (b) **Choose** *a Y-edge emanating from the output node. The node at which this edge terminates becomes the current node.*

   If the new current node (possibly as a result of the application of the previous two steps) is the final node (it has the label "F"), then

   > *A parse path has been found. Calculate the certainty of the parse path. The output graph associated with the parse path is a possible interpretation of the input graph, with the certainty just calculated.*

   else

   > *Go to step 2.*

The word **choose**, used above, denotes possible non-determinacy. In general, this method thus generates more than one parse path (although it is possible for only one or zero paths to be generated). The parse path, and corresponding output graph, most likely to be correct is the one with the highest certainty. This means that, to be

guaranteed of finding the parse path with the highest certainty, we will, in general, be forced to find all parse paths and then make a selection. This is normally the way in which fuzzy grammars are used [Pal & Dutta Majumder, 1986].

Following the steps given in the algorithm it is possible to reach a situation where either a Y- or an N-edge must be followed, but the current node has no edge with the correct label (see page 12). Then that parse path is unsuccessful, as the final node cannot be reached. This implies that either an incorrect (non-deterministic) choice was made, or that there are no successful parse paths (the input graph cannot be parsed according to the given grammar). This situation can arise in the case of a programmed crisp grammar as well as in the case of a programmed fuzzy grammar.

In any parsing algorithm, the objective is to find a successful parse path if one exists. In other words, the algorithm must exhibit global angelic non-determinism. This means that, whenever a non-deterministic choice must be made, the branch which leads to successful termination of the program must be chosen. In general, this requires that the algorithm be allowed to backtrack to any level. If a parsing algorithm for a programmed grammar does not allow for full backtracking, it effectively exhibits erratic non-determinism. This means that non-deterministic choices are made without considering the consequences that they might have on successful termination of the parse. As was pointed out in the example on page 64, deterministic control diagrams can often be found. In this case, the only non-determinism present is in the choice of subgraphs isomorphic to the left-hand side of a specific production, but even this can lead to a dead-end in the control diagram.

If either the grammar being used or the graph being parsed is large, it is impractical to find all possible parse paths. Chapter 7 discusses methods of reducing the computational complexity of a parse.

6.2 Fuzzy certainty functions

We have not yet given much attention to the exact nature of the certainty functions that we associate with productions. Let G_{set} be the set of subgraphs that are isomorphic to the left-hand side of a production. The certainty function associated with the production is simply a fuzzy membership function defining the fuzzy subset of G_{set} with elements that match the specified attribute conditions.

This is, of course, the same as defining the subset over all graphs, and simply assigning a membership value of zero to any graph that is not isomorphic to the left-hand side. The advantage of this approach is that it could allow for the possibility of matching subgraphs which are not exactly isomorphic (e.g., an edge might be misplaced or missing). This sort of "deformable graph template" has been used for pattern recognition, but not in the context of a syntactic approach [Chan, 1996]. The possibility of incorporating this into our mechanism is a potential avenue for future research.

The reader will notice that all membership functions given as examples in this thesis are either triangular or trapezoidal (for example, figures 4.3 and 5.12). This is because these classes of membership functions are simple and are commonly used. S or Π functions may be a better choice in certain situations [Pal, 1992].

The choice of fuzzy operators is also not limited by our framework. In our prototype implementation we have used the standard operators of minimum and maximum for fuzzy conjunction and disjunction, respectively, but they can be replaced by other suitable operators such as, for example, the T-norm and T-conorm operators.

The suitability of various fuzzy operators to different applications has been discussed at length in the literature [Kim *et al.*, 1993], and is beyond the scope of this thesis. It suffices simply to state that the choice of membership functions and fuzzy operators is not controlled by the framework described in this thesis; rather, it must be driven by the properties of whatever image class the framework is being applied to. The framework is implemented in such a way that the user can easily use different fuzzy

operators.

6.3 Calculating the certainty of a parse path

The standard way of calculating the certainty of a parse path is to take the minimum of the certainties associated with the productions in the path. However, any aggregation operation can be used instead [Pal & Dutta Majumder, 1986]. There is scope for further work in investigating the performance of various aggregation operations within our framework.

The main disadvantage of using minimum is that the certainty of the whole path is affected by a single low value. For example, if one (correct) production has a low certainty μ_1, the system loses the ability to distinguish between two interpretations of another symbol when both interpretations have a certainty greater than μ_1. Thus, it has been argued that using a weighted average of the certainty values in the parse path is preferable [Stallings, 1977].

There are some advantages to using the aggregation function of minimum. At any point in the parse path, we know that the certainty of the path cannot increase. This means that we can greatly reduce search time by using a standard "minimum distance" algorithm to find the best parse path. This is discussed in Chapter 7. Using minimum also allows check productions, such as a production that checks the number of beats in a bar, to force a low certainty on the parse path if they fail. We thus know that a parse path which satisfies the condition specified by the check production will have a higher certainty than one which does not. This condition is not guaranteed if an aggregation function such as average is used, although the use of a weighted average could possibly solve the problem, as we can give the check production a high weight.

Consider the search tree given in Figure 14.8. Using an aggregation function of minimum makes the right-hand path take on the low certainty of the failed check

production (0.01). The certainty of the left-hand path is not affected by the successful check production.

If average is used to calculate the certainty of the parse paths in the example to which figure 14.8 refers, the left-hand path has a certainty of 0.898, while the right-hand path has a certainty of 0.878. Although the correct interpretation still has the highest certainty value in this case, the two values are very similar. In another case, the incorrect interpretation could have a higher certainty than the correct one.

Using a weighted average and giving the check production a large weight would improve the relative certainty of the path containing a successful check production.

The main objective of this work was to create a framework which can be specialised for recognising various images. We have designed and implemented the framework in such a way that specifics, such as aggregation functions, can easily be altered by the user of the framework. A full investigation of various aggregation functions is beyond the scope of this work. Readers who are interested in using the system to investigate such features are welcome to contact us.

Chapter 7

Efficiency considerations

It was mentioned in Chapter 5 that using a fuzzy grammar can drastically increase the size of the state space associated with a parse. This rapid expansion of the state space search tree is a major problem that must be overcome to some extent for the formalism to be of practical use. In this chapter we discuss the efficiency of the parsing algorithm, as given in section 6.1, and ways to improve it by reducing the amount of state space that has to be searched during a parse.

The total state space associated with any parse is composed of all the potential production instances. Even in the case of a crisp grammar this can be a very large space. Finding all possible parse paths involves applying all productions that can be applied at any time and fully developing the resulting branches in the state space search tree. If either the grammar being used or the graph being parsed are even moderately large, this is computationally very expensive and is thus impractical.

Efficient parsing algorithms are, of course, known for many classes of grammars. Even though the parsing algorithm we use is naïve and does not make any assumptions

about the class of grammar, there are a number of ways in which we can reduce the size of the search tree and thus the computational complexity. These are discussed in the remainder of this chapter.

7.1 Pruning the search tree

We can, in most cases, reduce the size of the state space search tree drastically without violating the criterion of finding the parse path with the highest certainty value.

This pruning relies on graphs being disjoint.

Definition 7.1 *Two graphs G and G' are* **disjoint** *iff $G.N \bigcap G'.N = \{\}$, where $G.N$ denotes the nodes of G.*

That is, two graphs are disjoint if they have no nodes in common. Two graphs can only have a common edge if they have at least two common nodes. Thus, the lack of any common nodes means that the graphs do not have any common edges.

Consider the portion of a search tree shown in Figure 7.1. If the two possible production instances (Px and Py) affect disjoint subgraphs, they do not affect each other, and Py can be applied after Px, or vice-versa. That is, whether the one production instance has been applied or not does not affect the application of the other production instance. Since the certainty values associated with the production instances are a function of the subgraph being transformed, the certainty value associated with each of these production instances is independent of whether or not the other has been applied.

We assume that the aggregation function used to calculate the certainty of a parse path is not affected by the order of the constituent certainty values. If a weighted average function is used, and the weights are associated with the productions and not with the positions in the parse path, the order in which the production instances are

applied still does not affect the aggregated certainty value associated with the parse path.

The result is that states A and B in Figure 7.1 are identical, i.e., they have the same graph, certainty and position in the control diagram. The subtree emanating from A is thus identical to that emanating from B, and we can prune the tree by only considering one of them.

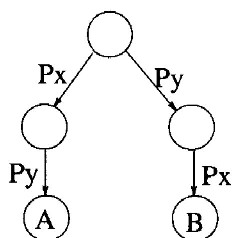


Figure 7.1: Possibilities for pruning the search tree.

For example, in the prototype system that we have implemented (described in part 2), the simple primitive extraction routines that we employed led to the recognition of a number of spurious "blobs" with an area of one or two pixels. To deal with this problem, we use a (crisp) production which deletes all very small blobs, and which is applied before any other productions. In effect, this production is a preprocessing filter. Imagine that there are 20 such blobs in an image being analysed. Then there are initially 20 potential production instances, one for each of these 20 blobs. For each one that we choose, there is a further choice between the 19 remaining instances which can still be applied, then 18 etc. The full state space search tree thus has $20! \approx 2.4 \times 10^{18}$ nodes, each one corresponding to a production instance. It is clearly not practically feasible to traverse a search tree of this size.

Since the subgraphs transformed by all of these production instances are disjoint (they are all different single nodes), the production instances are all independent, and the order in which they are applied makes no difference to the final graph or its certainty value. If we prune the search tree at each point by removing all branches except for one (which we can choose arbitrarily), we obtain a single search path containing 20

production instances.

In general, the situation is slightly more complex, as the subgraphs forming the left-hand sides of the productions can contain more than one node. In this case, when we are pruning a point in the search tree, we cannot ignore the branches corresponding to any production instance which is dependent directly or indirectly on the one we choose to examine.

Consider, for example, Figure 7.2. It shows a segment of music containing a crotchet and three semibreves. Assume that the primitives have already been transformed into a solid note-head, stem and three hollow note-heads. Using only local information, there is a possibility that the stem could be associated with either the solid or the first hollow note-head. If we choose to apply the production instance labelled 1, then we cannot ignore production instances 2 and 3. This is because the subgraph labelled 2 is conjunct with subgraph 1, and subgraph 3 is conjunct with subgraph 2, which in turn is conjunct with subgraph 1. Production instance 2 is thus directly dependent on production instance 1, while production instance 3 is indirectly dependent on production instance 1. However, production instances 4 and 5 are independent of production instance 1 and we can thus ignore them, thereby reducing the size of the state space search tree.

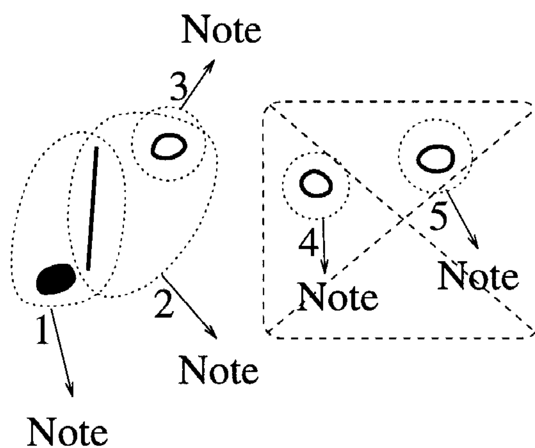


Figure 7.2: Pruning the search tree.

We can formalize this by replacing step 2 in the parsing algorithm (Section 6.1) with the following.

Improvements to the parsing algorithm

2. *Find all subgraphs which are isomorphic to the left-hand sides of all productions in the current node and which have a non-zero associated certainty value. If there are no such subgraphs (redices) then*

   Choose *an N-edge emanating from the current node in the control diagram. The node at which the edge terminates becomes the current node.*

   else

   (a) **Choose** *one of the redices, which we shall name R.*

   (b) Let $R_{set} = \{R\}$, where R_{set} is a set of redices.

   (c) Iteratively add to R_{set} every redex r which is conjunct with any redex in R_{set}. In other words, calculate the closure of R_{set} under conjunction.

   (d) Calculate the certainty of any "noise productions" in R_{set}

   (e) **Choose** *any redex in R_{set} and apply the corresponding production instance to the current graph.*

   (f) **Choose** *a Y-edge emanating from the current node in the control diagram. The node at which the edge terminates becomes the current node.*

The search through state space will be most efficient if the state with the highest certainty value is in R_{set}. We can guarantee this by choosing R to be the state with the highest certainty value in step 2(a).

Step (2d), calculating the noise productions, requires some further discussion. If the noise production deletes a non-primitive symbol, it will have a (fixed) low certainty, and will require no further calculation. If, however, the noise production deletes a primitive symbol, its certainty needs to be calculated according to definition 5.3. For each noise production of this type in R_{set}, we search through R_{set} to find all competing production instances. Since any production competing with the noise production has an associated subgraph that is conjoint with the noise production's

associated subgraph, and R_{set} is a closure under conjunction, R_{set} will contain all production instances that compete with the noise production.

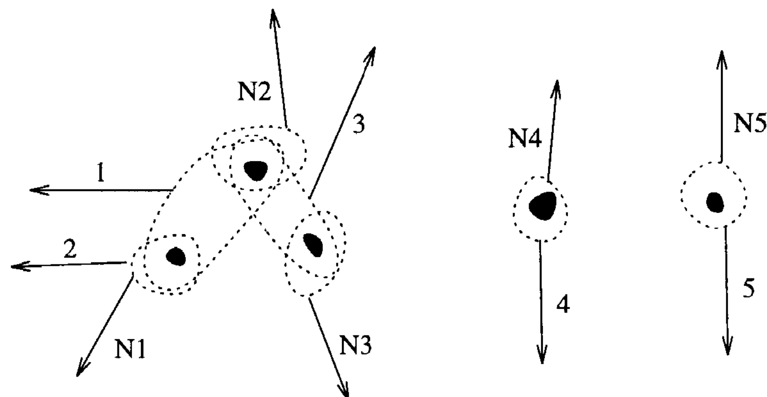


Figure 7.3: Competing noise productions.

Consider the hypothetical example given in Figure 7.3. Productions 1 to 5 map either one or two primitives to some higher-level symbol, with certainty μ_1 to μ_5. Each of the noise productions, N1 to N5, deletes one of the primitives. Imagine that production 1 is selected (i.e., $R = 1$ in step 2(b) of the algorithm). Then after step 2(c) of the algorithm has been completed, $R_{set} = \{1, 2, 3, N1, N2, N3\}$. We thus calculate the certainties of the noise productions N1 to N3 according to definition 5.3. We get

$\mu_{N1} = \neg(\mu_1 \vee \mu_2)$

$\mu_{N2} = \neg(\mu_1 \vee \mu_3)$

$\mu_{N3} = \neg(\mu_3)$

7.2 Reducing the amount of the search tree examined.

Although this pruning drastically reduces the size of the state space search tree, it can still be large. It can thus be impractical to examine the entire search tree. Fortunately, if we use MIN for the aggregation function to calculate the certainty of parse paths, we can use a uniform search strategy. If we use the certainty of the parse

paths as the basis for ordering, then this strategy is admissible (the first full parse path found is guaranteed to be the one with the highest certainty value). In general, it greatly reduces the amount of the search tree which has to be examined.

The basic strategy is simply to continue expanding the parse path which has the highest certainty value until its certainty becomes lower than that of a rival parse path. At that point, we expand the rival parse path until it no longer has the highest certainty. This requires that the certainty of the parse path be updated after each production is applied. Step 2(e) of the modified parsing algorithm given in the previous section becomes

2. (e) **Choose** *one of the redices and apply the corresponding production instance to the current graph. Calculate the new certainty of the parse path.*

We can express the algorithm as follows. (This search strategy is equivalent to a best-first search with no heuristic factor, and is essentially Dijkstra's shortest path algorithm.)

1. *Start at the root node of the state space search tree.*
2. *Repeat:*
   - *(a) Examine all the child nodes of the current search tree node. For each, calculate the state (graph, position in control diagram and certainty) which is the result of the parse path from the root node of the search tree to that child node.*
   - *(b) Insert these state values into the priority queue, ordered on descending certainty values, and such that new items are inserted in front of old items that have the same certainty value.*
   - *(c) Remove the item from the front of the priority queue. The corresponding node in the search tree becomes the current node.*
   - *(d) If the new current node is a goal node (the associated position in the control diagram is the final node, labelled F), then a full parse path has been found. If only the parse path with the highest certainty is required, the algorithm can terminate at this point.*

If, at any stage, a dead-end is reached (that is, an N-edge must be followed and there is no N-edge, or a Y-edge must be followed and there is no Y-edge), no new state value will be added to the priority queue, and the path will thus die.

We can apply this search strategy to the parse algorithm by adding the following two rules.

1. When we get to a **choose** command, examine all options and add the state resulting from each to the priority queue. In practice, we make a copy of the current state variables, apply the designated changes to the copy, and place the resulting state in the priority queue.

2. At the end of step 3, before going back to step 2, remove the item from the front of the priority queue. This becomes the current state.

To prove that this algorithm is admissible, we need to prove two things: firstly, that the first full parse path found will be one of those with the highest certainty value, and secondly, that if there is a parse path terminating at the final node in the control diagram, it will be found. This proof depends on the monotonicity of the aggregation function used to calculate the certainty of a search path.

Consider diagram 7.4. Assume that the portion of the search tree enclosed by the dotted line has been fully examined (the children of these nodes have been placed in the priority queue. YZ and $X_{i-1}X_i$ are candidate edges for following. That is, states Z and X_i are in the priority queue. Assume further that no node already visited is a goal node. Then we must show that, if we choose to follow the edge YZ and Z is a goal node, and X_n is also a goal node, then the parse path AZ has an equal or higher certainty than the parse path AX_n.

The algorithm chooses YZ such that $\mu(AZ) \geq \mu(AX_i)$.

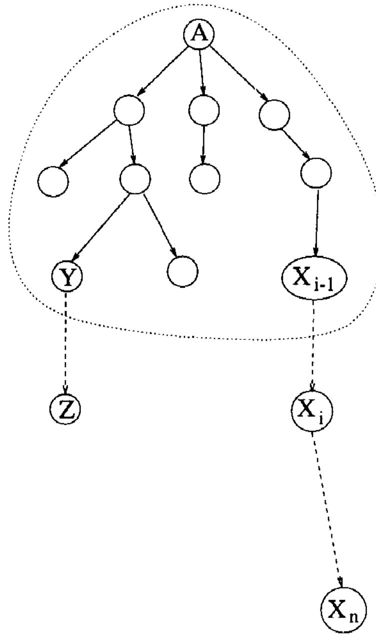


Figure 7.4: Search strategy.

$$
\begin{aligned}
\mu(AX_n) &= MIN(\mu(AX_i), \mu(X_iX_n)) \\
&\leq \mu(AX_i) \\
&\leq \mu(AZ)
\end{aligned}
$$

While no goal nodes are found, the algorithm continues to examine the search tree. Paths are only terminated if they are at a dead-end, which means that the entire tree is eventually traversed. Thus, if there is a goal node, it will be found.

If no successful parse paths are found, the algorithm will examine each parse path until it reaches a check production (normally at the end of the parse path). (See section 5.2.) Since this check production will fail, that parse path will get a very low certainty. Another path will thus have a higher certainty and will be examined. This will continue, and the algorithm will examine virtually the whole search tree before any goal node with a very low certainty (due to failed check productions) has the highest certainty and its corresponding parse path is presented as a solution. If

the search tree is large, this can be impractical. If this is the case, the amount of the search tree examined can be reduced by defining a cut-off value, and by ignoring parse paths with certainty values below that cut-off point. The parse engine is then recognising the cut-point language consisting of all graphs which can be parsed with a certainty greater than or equal to the cut-off value. The parsing process then stops when the certainty of the best path is less than the cut-off value. If no parse path with a certainty greater than the cut-off value is found, the output graphs associated with the goal nodes with low certainty values mentioned above represent the best interpretation of the image to be found.

We can add a heuristic factor to the uniform search strategy described above, and transform it into a best-first search. This means writing a function which estimates how high the certainty of the unseen part of a parse path will be, based on the current state. A simple example is to favour paths with an associated position in the control diagram close to the final node. We found that, in many test cases, this heuristic had a marked effect on the amount of the search tree examined, reducing the number of search tree nodes examined by an order of magnitude. We envisage that heuristics which make use of a priori knowledge about the class to which the image being recognised belongs should lead to further reductions in the amount of the search tree which has to be examined before the optimal solution is found. Figure 7.5 gives an example of the use of a heuristic function. Imagine that A an B are the two parse paths with the highest current certainties. Although B has a higher certainty (0.52) than A (0.51), the heuristic factors derived from the position in the control diagram mean that A has a higher priority (0.46) than B (0.42). Parse path A will thus be expanded first.

We can cast this search strategy into the form of the A algorithm. In this case we examine the path which maximizes the function $f(n) = g(n) - h(n)$, where $g(n)$ is the certainty of the known portion of the path, and $h(n)$ is the heuristic estimate of the cost of the rest of the path. We know that for the algorithm to be admissible, we must have $h(n) \leq h*(n)$, where $h*(n)$ is the optimal (lowest possible) cost of the

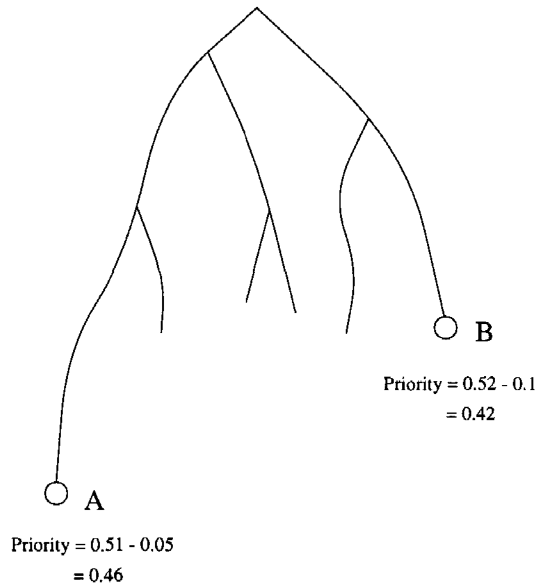


Figure 7.5: Example of the use of heuristics.

rest of the path.

In out prototype system, we implemented the heuristic by storing current search tree nodes in a priority queue. The priority of each node is the function f. This means that different heuristics can be used without having to make any changes, other than to the heuristic function.

Note that the simple heuristic suggested earlier, which is a factor of the distance to the final node in the control diagram, can lead to an inadmissible search path. If we use MIN to calculate the certainty of the parse paths, the certainty of a path might not decrease after a certain point, while the heuristic factor will decrease, but will still be non-zero.

It should be noted that the use of a non-monotonic aggregation function for calculating the certainty of a parse path would invalidate the result of admissibility for the search algorithm. An example of this sort of aggregation function is average, either a simple average or a weighted average (weights are associated with different productions): the certainty of the path can increase at any step, if the certainty value

associated with that production is higher than the current average. This fact, however, does not seem to render the scheme useless, but rather points to the fact that a search using such an aggregate function requires a heuristic term which is good enough to cancel out the non-monotonicity of the aggregation function.

The use of a function such as weighted average instead of MIN has some definite advantages, as mentioned in section 6.2, and thus deserves further work.

7.3 Reducing the complexity of applying a production

Although we have discussed ways of reducing the number of production instances that are applied before the best parse path is found, a lot of the complexity associated with the use of our system derives from the complexity of applying each production.

Let n be the number of nodes in the current graph (the graph to be reduced), and m be the number of nodes in the graph forming the left-hand side of a production. (We say that these graphs are of the order n and m respectively.) Let e be the maximum number of edges connected to a node in the graph to be reduced (e is the maximum degree of any node in g). The application of a production (definition 2.8) consists of the following five steps.

1. Find an isomorphic subgraph
2. Evaluate the certainty function
3. Make a copy of the current graph (as mentioned in the previous section)
4. Replace the left-hand subgraph with the right-hand subgraph and calculate the new embedding
5. Calculate the new attributes

The computational complexity of the certainty function (step 2) is normally low. Making a copy of the graph (step 3) has complexity in $O(n)$. The algorithm that we have implemented for replacing the old subgraph with the new one (step 4) has complexity in $O(me^2 + n)$. Since the number of edges connected to any node in the graph in not normally more than two in music scores, this cost is low. The number of edges will depend on the class of images being described. To calculate the new attributes we have to apply m attribute transfer functions. These functions tend to be very simple, and this is thus also not an expensive operation.

Finding an isomorphic subgraph, on the other hand, is an NP-complete problem, and is thus computationally very expensive. Finding a subgraph gl' in a graph g (order n) isomorphic to the graph gl (order m) can be divided into two parts; finding all subgraphs in g of order m, and checking them for isomorphism with gl. The worst case is encountered when all possible subgraphs have to be checked before a correct one is found.

Finding all possible subgraphs can, for the sake of analysis, be simplified to the problem of finding all lists of m elements that can be chosen from a list of n elements, which has complexity in $O(n^m)$. The worst-case cost of checking if a subgraph is identical to the required graph is the cost of checking the edges from every node. This cost is in $O(m^{el})$, where el is the maximum degree of any node in gl, thus $el \leq e$. Since each of the possible subgraphs has to be checked, the total computation complexity of finding an isomorphic subgraph is in $O(n^m m^e)$.

In our experience with writing a grammar to describe music scores, the subgraphs associated with productions did not have more than 4 nodes, i.e., $m \leq 4$. We also found that $e \leq 3$, since each symbol in a music score is connected to only a few other symbols. This places the worst-case complexity of the algorithm in $O(n^4)$, which is still quite high. Since a subgraph must be found every time a production is applied, it is important to improve the efficiency of the algorithm for the average case.

Consider a subgraph containing two nodes connected by an edge. Once a node N

has been chosen to correspond to the first node in the subgraph, the possible choices for the second node are limited to the set of nodes connected to the node N. An algorithm taking advantage of this fact (as is normally the case) will have a worst-case complexity in $O(ne^{m-1})$ if gl is a connected graph, where e is the maximum degree of any node in gl. This is easily derived from the fact that there are n possible choices for the first node and a maximum of e possible choices for each of the remaining $m-1$ nodes. This is obviously considerably better than n^m, since $e \ll n$. In fact, if e and m are bounded, this algorithm has a worst-case complexity in $O(n)$ when gl is a connected graph. If the nodes of gl are all unconnected, then the complexity reverts to $O(n^m m^e)$.

In the case of our grammar for recognising music, we have found that not as many of our productions have had connected graphs forming their left-hand sides as we had initially anticipated. Apart from writing productions with connected left-hand side graphs, a user of the system can keep efficiency reasonable by keeping m small, but this can have the effect of lengthening the parse paths. This means that more productions have to be applied, which results in an increased complexity. There is thus a trade-off between the complexity of the individual productions and the number of productions that have to be applied. If the high-level symbols in the image consist of large numbers of primitives which cannot be grouped to form intermediate (lower-level) symbols, then it will not be possible for the grammar writer to keep m small, and this method will be inefficient. The cost of isomorphism is also dependent on the number of edges connected to each node in the graph representing the image (e). If an image class has a high degree of interconnectedness between the symbols, then e will be large, and this method will be too inefficient.

There are thus image classes which will not be suited to recognition by this method, for efficiency reasons. However, a large number of image classes exist where symbols are built up from a small number of lower-level symbols, and are not highly-interconnected. Classical examples, other than music scores, are flowcharts, circuit diagrams and, in general, engineering drawings .

The size of the graph (n) will always be a major factor affecting efficiency. Fortunately, we feel that it is possible to partition many large images in such a way that the subparts can be recognised independently. If the nature of the image class does not allow this to be performed automatically, the user can segment the image before the recognition process begins. This should be quick to do, in which case the system will still speed up the process of image recognition. In the case of music, we can easily partition the input image by segmenting the music into bars before we parse the resulting input graph.

Of course, a problem could arise if we mistake some other symbol for a bar line, as the system would not be able to correct this mistake. To avoid this problem we only segment where we are absolutely sure that we have found a bar line. When the identification of a bar line is questionable, we do not segment the input graph, and leave the parsing engine to make the decision, as it is able to use more information to make the decision (as discussed in previous chapters). Some information may have to be carried over from one segment to the next. Examples in the case of music scores are the clef and time signature. The segments must, of course, be recombined once they have each been interpreted to yield the interpretation of the whole image. This segmentation of the input graph into bars keeps the graph relatively small and dramatically decreases the computational complexity of the parsing process.

Part II

Using the framework: a test application

Chapter 8

The Problem

In this part of the thesis we describe the prototype system that we have developed for optical music recognition (OMR). This illustrates how to apply the framework developed in the first part to the recognition of a specific image class. This chapter introduces the problem of optical music recognition and reviews previous work in this area.

8.1 Optical music recognition

With the decrease in the cost of computing power, computers have found widespread use in the field of music. This ranges from music performance, using for example the MIDI protocol, to music publishing and form analysis. As one result, many tasks which were previously performed by hand can now be automated. An example is the transposition of a music score. Of course, before such an operation can be performed by a computer, the score must be captured in a representation which allows the musical information to be manipulated: simply having a scanned image of a music score does not allow that score to be transposed.

Unfortunately, current methods for entering a music score into a computer system are still cumbersome. One method is to play the music on a keyboard connected to the computer. In general, it is difficult to generate automatically a correct score from performed music. Notes are not played with the perfect (notated) duration. The time and key signatures cannot always by deduced, and often have to be entered manually. Features such as slurs and expressive markings also have to be entered manually. The beaming of note groups normally has to be manually adjusted before a meaningful result is obtained. (See Figure 10.1 for an explanation of some common music symbols and terms.) A simpler and more precise method is to type the pitch and duration of each note into a text file. This is time-consuming and tedious. Newer graphical user interfaces do not alleviate the problem much: although they are typically easier to use, they are still very slow.

The aim of optical music recognition (OMR) is to automatically convert sheet music into a form which allows the music information to be directly manipulated by a computer, thereby speeding up the input process and drastically reducing the tedium associated with it.

This second part of the thesis describes the prototype system that we have developed to recognise handwritten music. This system should be seen as an example of the specialization of the framework described in the first part of this thesis. As such, we do not claim that the preprocessing steps or primitive extraction layers are optimal, nor do we claim that the system is complete. The lower-level parts of the system work well enough to test the parsing mechanism, and are described in this part of the thesis mainly to place the rest of the system in context.

As a means of estimating the complexity of the recognition of handwritten music, we compare OMR with optical character recognition (OCR), as well as handwritten music with printed music.

8.2 A comparison of OMR with OCR

The optical recognition of printed text is a problem which has received much attention over the years and for which a variety of commercial products exist. There are a few (superficial) similarities between OCR and OMR, which makes a comparison of these two problems a useful starting point from which to delve further into OMR. The most obvious similarity is the fact that the two problems both consist of the recognition of a set of characters from a two-dimensional image.

The first step in OCR is normally to isolate the individual lines of text by searching for the long, nearly horizontal white spaces between lines. Within each line, the characters are handled one at a time. The individual characters can easily be isolated from their surroundings as they are encompassed by white space. The limited number of cases where two adjacent characters touch each other (ligatures) are usually handled as special characters. Text in which many characters are connected due to bad printing is typically not handled well by current OCR products.

The number of different symbols present in text is not very great (typically the alphanumeric symbols and a few others, such as punctuation marks and possibly simple mathematical symbols). The various symbols are also fairly similar in size.

These two factors mean that each symbol can normally be detected as a single unit, often either through template matching or by a suitably trained neural network. Furthermore, the symbols can easily be grouped into words (by measuring the gaps between the symbols), which can be checked in a dictionary in order to detect errors or make intelligent choices concerning ambiguous characters.

On the other hand, the symbols on a staff of music are nearly all interconnected by the staff lines. The number of different symbols that can occur in a music score is exceedingly large, essentially because of the presence of beamed groups of notes. For example, if we consider only notes on the stave, there are over 200 million different possible groups of eight beamed demi-semi-quavers. This means that music symbols

cannot be recognised as single units, but must instead be dissected into smaller parts.

Furthermore, the rules defining the spatial relationships between symbols are much more complex than the simple concatenation used in text. This means that a more advanced approach is required to describe music images.

The combined effect of these complications is one of the reasons why commercial OMR systems have been slow to appear.

8.3 A comparison of handwritten music with printed music

A major objective of this work has been to develop a technique or system which is robust and impervious to slight changes in the input image. For example, it is not desirable to have to retrain a system before it can deal with a slightly different font. Satisfying these requirements will lead to a system which more closely mimics the performance of the human brain. The ultimate test for such a system is the recognition of handwritten music scores.

Many of the strict typesetting rules observed in printed scores are only approximately followed in handwritten scores. One example is note-stems, which are always straight and vertical in printed scores, but are only approximately straight and vertical in handwritten scores. Another example is the connection between stems and beams: while these are always well connected in printed scores, they often do not quite touch in handwritten scores. A further example is handwritten note-heads, which are sometimes simply an oblique line, rather than a well-formed ellipse.

A system designed to recognise handwritten scores must be able to deal with this greater variability. It should also be easily extensible, so that additional quirks found in different people's music handwriting can be catered for relatively easily. This means that such a system must be more general than a system designed to recognise only

printed scores.

For example, the specific cases of handwritten variability mentioned above, rule out the use of some approaches commonly used in OMR. In printed scores, a relatively simple algorithm can be applied to extract all the vertical lines. This set of vertical lines will include both bar-lines and stems. In handwritten scores, stems are often drawn at too great an angle from the vertical for such an algorithm to be useful.

While note-heads do vary from one publisher to the next, a much greater variance is introduced in handwritten scores. Any technique used for handwritten scores must be capable of dealing with this wide variation. Inflexible methods such as template matching clearly will not suffice.

8.4 Previous work

Optical music recognition is by no means a new research field – according to the literature the first attempts were made by Pruslin in 1966 [Pruslin, 1966] and a few years later by D. Prerau at MIT [Prerau, 1975]. Prerau used a flying-spot scanner[1] to capture images. After the staff lines were detected, the image was broken into fragments – each fragment being a group of connected pixels between the staff lines or directly above or below a staff. The fragments were then assembled into "components", which corresponded to the still unrecognised music symbols. Music symbols were identified based mainly on the size of their bounding boxes. When this led to ambiguities, the context of the unknown symbol was used to resolve the contention. Accidentals were treated as a special case as they have similar bounding boxes and occur in the same contexts.

The cumbersome nature of scanners available at the time, together with the lack of computing power, effectively halted further research for some ten years. The last ten

[1] A flying-spot scanner is an early type of scanner that uses a single moving scan head to scan an image pixel by pixel.

years, on the other hand, have seen an upsurge of interest in OMR and new research has been undertaken in various centres around the world. We shall now give a review of some of this research.

Work undertaken by Matsushima *et al.* at Waseda University in Japan led to the optical music recognition system of WABOT 2 (Waseda University Robot)[Matsushima, 1985]. This robot, which was demonstrated at the 1985 International Exposition held in Tsukuba, Japan, is described in [Roads, 1986]. Its OMR system used a high-definition CCD camera to scan an A4 sheet of music placed in front of it. The music symbols were printed in a specified font and size, which made recognition by template matching possible. This was performed by a hardware implementation of correlation. Another technique they used was slicing, a computationally inexpensive method in which the number of transitions from black to white, and vice-versa, in a slice of an object are counted. For example, a horizontal slice taken near the bottom of an accidental which is either a natural or a sharp sign will facilitate distinction between the two. Using these techniques, the robot was able to read a sheet of music in about ten seconds. The computing power required for the robot was provided by a total of seventeen 16-bit and fifty 8-bit computers, interconnected by fibre-optic cable. Later this OMR system was incorporated into a system which translated from Common Music Notation to Braille and back [Matsushima *et al.*, 1989]. A "musical syntax" is used to assemble the recognised note-heads, stems, beams, etc. into music symbols. This approach of recognising primitives and assembling them into symbols according to some rules allows the system to deal with complex symbols such as beamed groups.

At Osaka University, Katayose *et al.* developed an OMR system for printed piano music as part of an "Artificial Music Expert" designed to extract quasi-sentiment from music [Katayose *et al.*, 1989]. This system first located and removed staff and bar lines. A pattern recognition phase then synthesized music symbols as a combination of primitives extracted from the image. A semantic analysis phase encoded extracted music symbols into "playable music information". Two experiments yielded recognition rates of 89 and 94 percent accuracy, with a processing time of approximately 90

minutes per page. The information that we have on this project does not clarify how they dealt with beamed groups.

Another project, undertaken by Clarke *et al.* at the University of Wales, concentrated on producing an OMR system which was inexpensive both in terms of computational and memory requirements, so that it could be run on an IBM AT personal computer [Clarke *et al.*, 1988]. Staff lines were found by counting the number of black pixels in each row: lines containing more than a certain number of black pixels were regarded as staff lines. If the image was slanted, this algorithm would fail and would be re-applied to shorter sections of the image. Once the staff lines had been found, they were removed by tracking along them and following simple rules to attempt to delete only those pixels not forming part of a superimposed music symbol. The staff line detection and removal algorithms that we have used are similar to these (section 9.2).

Once the staff lines had been removed, symbols were identified by considering their bounding boxes and horizontal slices taken near the top, middle and bottom of each symbol. This slicing technique is slightly more elaborate than that used in the Waseda project since the number of black pixels in a slice is recorded as well as the number of transitions. Algorithms were later developed to deal with chords, using horizontal and vertical slices [Clarke *et al.*, 1989]. They wrote a separate algorithm specifically to deal with beamed groups. The algorithm finds the top and bottom pixel in each column of the beamed group's bounding box, so that each column is represented by a pair of numbers. These numbers are analysed to determine whether the symbol is a beamed group, and if so, the position and number of notes. The number of beams is calculated by examining the thickness of the beams, as well as the number of changes from black to white in particular columns.

Fujinaga *et al.* began an OMR project at McGill University, Canada, in 1988 [Fujinaga *et al.*, 1989a; Fujinaga *et al.*, 1989b; Pennycook, 1990; Fujinaga *et al.*, 1991; Fujinaga *et al.*, 1992]. The staff lines were found using projections (in a similar way to [Clarke *et al.*, 1988]). Everything other than the staff lines was removed and the

resulting image was XORed with the original image to delete the staff lines. Musical symbols were separated by using horizontal and vertical projections. Initially these projections are taken over the whole image. If an element in the vertical histogram has a value of zero, the image is split into two rectangles, along the blank vertical line. Similarly, if an element in the horizontal histogram has a value of zero, the image is split into two rectangles along that horizontal line. Projections are then taken over each of the new rectangles, and the process is repeated. In this way, the image is segmented into smaller and smaller rectangles until they can no longer be segmented by this method.

The resulting rectangle was sent to the feature extractor and a feature vector was calculated for the symbol. A k-nearest-neighbour (k-NN) classification scheme was used to identify each symbol. In this scheme, the distances[2] between the unknown symbol's feature vector and the feature vectors of known symbols are calculated. If the majority of k ($k \geq 1$) closest samples to the unknown symbol belong to class ω, then the unknown symbol also belongs to the class ω.

The system included two special predefined symbols, **split_x** and **split_y**. When found, they direct the recogniser to further segment the given symbol either horizontally or vertically, and to repeat the classification process. They allow the system to split up symbols such as notes and beamed groups into smaller symbols such as note-heads and stems, and also allow touching symbols to be classified separately. A special routine to specifically locate stems was added to speed up the splitting process. The papers that we have available do not specify how the recognised primitives are re-combined to form the music symbol that they are part of.

A learning module used human corrections to update the database and to recalculate the weights required for best results. To determine the weights which would result in the most accurate classification, the weights were altered and the distances between symbols were recalculated. This is a time-consuming task, since the best set of weights

[2]The distance used is a weighted sum of the differences between the features of the two symbols.

can only be obtained by examining all possible combinations.

The OMR system developed by Carter *et al.* at the University of Surrey [Carter & Bacon, 1991] first found all pixel runs[3] in the image. The pixel runs which formed part of staff lines were detected and removed, and those remaining were combined to form groups of connected pixels. Each group was processed separately. Fairly complex algorithms were needed to deal with large groups such as beamed semiquavers. The algorithm for beamed groups locates vertical lines (stems) in the connected group and searches on either side of these stems to find the note-heads. The beaming complex (the beams which connect the notes in the beamed group and determine their duration values) is isolated and its thickness is measured at the end of each stem to ascertain the duration of each note.

Neural networks have also been used for OMR. Although the mechanism is different, there are similarities between the nearest-neighbour approach used by Fujinaga and neural networks. Both approaches use weighted feature vectors to perform clustering, and incorporate a learning phase to adjust the weights. [Martin, 1989; Martin & Bellissant, 1991] used a multilayer perceptron to erase the staff lines. They dealt with note groups by finding the stems in a thinned version of the image and matching ellipses to the image at the end of these stems to find the note-heads (the process of thinning images is explained in section 11.1). Another multilayer perceptron was used to deal with rejected symbols. Its input was based on the skeleton graph of the symbol (produced by a thinning algorithm).

Neural networks were also used by [Yadid-Pecht *et al.*, 1992]. Their system first located the staff lines, then rotated the page if necessary and removed the staff lines. It then searched across the staff for symbols and sent them to the neural network one at a time to be classified. As far as we are aware, they did not split symbols before sending them to the neural network, and did not test the system on any complex symbols.

[3]A pixel run is a horizontal row of adjacent black pixels.

Grammars have been used, at various levels, in a number of OMR projects. Fahmy and Blostein have used graph grammars to describe music structure in terms of primitives such as note-heads and accidentals [Fahmy & Blostein, 1991; Fahmy & Blostein, 1992a] (graph grammars are described in detail in Chapter 2). An OMR system has recently been developed at IRISA (Institut de recherche en informatique et systèmes aléatoires) using a string grammar augmented with positional symbols (e.g., ABOVE) [Couasnon & Camillerapp, 1994]. As these systems describe complex symbols (such as beamed groups) in terms of their constituent primitives, they do not require special algorithms for dealing with them.

Very little work has been done on recognising handwritten music notation. Bulis *et al.* attempted to identify symbols by comparing the horizontal and vertical projection histograms[4] of the unknown symbol with a library of projections of known symbols [Bulis *et al.*, 1992]. The distance measure used was the sum of the squares of the differences between elements in the histograms of the known and the unknown symbols. Facilities were provided to allow the user to add new symbols to the library or to modify histograms in the library. The system was limited to symbols which were not connected graphically to each other, and could not deal with complex symbols such as beamed groups. Although this has the advantage of being a very simple method, it is not nearly powerful enough to deal with general handwritten music.

The basic design of a music notation system using pen-based input of a music score was described in [Leroy *et al.*, 1994]. The use of a pen-based system obviates the need for staff line removal. This, together with the possibility of using timing information associated with the pen-strokes, greatly simplifies the segmentation process. They propose the use of a time delay neural network (TDNN) to provide a list of labels for each stroke. These labels can be shapes (e.g., vertical lines) or symbols (e.g., sharps). They propose to "group the strokes into elementary music symbols using a music symbol graphic grammar" and then to use a symbol classifier such as a TDNN to validate these symbols. Finally, all possible music "sentences" will be constructed

[4]Projections are explained in Section 9.2.

and their associated probabilities computed. We are not aware of further reports on this work.

One of the problems peculiar to handwritten music is that symbols are often broken. A common example is a note-head which is not quite connected to its stem. [Wolman & Yaeger, 1994] discuss the modifications that need to be made to a system that recognises printed music scores if it is to cope with this problem.

It is worth noting that of the systems described which can deal with complex symbols such as beamed groups, almost all recognise the individual primitives (such as note-heads and stems) which make up the symbols and then combine them according to some rules (either implicit or explicit). Exceptions are Carter and Clarke, who developed specialised algorithms for dealing with these symbols. An advantage of a syntactic approach such as that described in the first part of this thesis is that it allows the system developer to cope with these complex symbols within the general framework.

Although the first commercial OMR system was released in mid-1994 by Musitek, the problem has by no means been completely solved. High error rates necessitate many corrections, so that it is still currently faster for an experienced operator to capture a score manually than by using an existing recognition system [Lindstrom, 1994].

Furthermore, no successful system has yet been developed that can recognise hand-written music scores.

Chapter 9

Preliminary processing

In OMR, as in any document analysis work, the first step is of course to scan the sheet of paper and convert it into a computer-legible format. Since TIFF (tag-based image file format) is the format most commonly used by scanner software, it was chosen as the format for image files in this project. One of the difficulties of optical music recognition is the presence of staff lines superimposed on the symbols: the second section of this chapter describes the method that we use to remove the staff lines from the image.

9.1 Image acquisition

Sheet music is a black-and-white notation, so images have been scanned and stored in black-and-white, i.e., one bit per pixel. This enables eight pixels to be stored in a single byte, which greatly reduces the storage space required for a given image. As an example, an image stored with 256 levels of grey-scale would require eight times as much storage space.

It has been claimed that the direction of pen-strokes can be deduced from an image scanned in grey-scale, as each line starts dark and ends lighter [Wolman & Yaeger, 1994]. This information is useful, as it helps both in segmenting the image (extracting features) and in recognising the music symbols. However, this is limited mainly to pencil writing, as many pens tend to produce lines of consistent darkness and it is also not present in printed music. This technique would thus appear to restrict the input domain of programs that use it.

The software written to process scanned music assumes that the image is orientated fairly correctly, i.e., that the staff lines are nearly horizontal. Although it was considered an unnecessary complication to deal with very skew images, it is practically impossible to place a sheet exactly straight in a scanner, and some degree of rotation must thus be allowed.

Although the techniques developed here are independent of scanner resolution, it was found that scanning a music sheet at a resolution lower than 300dpi can led to the loss of information, while a higher resolution did not typically improve the quality of the scan, but simply increased the storage size of the image. This "optimal" scanning resolution is, of course, dependent on the size of the music symbols on the sheet being scanned. The pixels in the scanned image must be small relative to the symbols in the image so that information in not lost. We found this to correspond to staff lines being roughly two to four pixels wide.

Routines from the TIFF library written by Sam Leffler at Silicon Graphics were used for the input and output of TIFF files.

9.2 Staff line removal

As stated in the previous chapter, one of the difficulties of optical music recognition is the presence of staff lines superimposed on the symbols. Although not all researchers have felt it necessary to remove the staff lines as a first step, the vast majority have

done so. The reason for removing the staff lines first is to separate the other symbols from each other as much as possible. The presence of staff lines connecting these symbols to each other makes the task of recognising those symbols more difficult: "Although the original design of the OMR system did not remove the staff lines, it became evident, as the system started to deal with more complex scores, that the staff lines must be removed. [Fujinaga *et al.*, 1991]"

In our system this step is essential, given the level of primitives that we have chosen. The blob primitive, for example, is obviously not amenable to recognition by template matching, or by any other technique which could be easily applied with the staff lines still in place.

Each staff consists of five lines which are approximately straight, horizontal and equispaced. Staves are often grouped together to form a staff system, which is identified by bar-lines that span the whole group.

There are a few factors that complicate the removal of staff lines. The main one is that the music symbols superimposed on the staff lines must be altered as little as possible. Any alteration of the symbols could introduce ambiguity or uncertainty into the image. Even taking all precautions, the staff line removal process does normally deform the symbols to some extent. This situation can be modelled as the introduction of extra noise (see Chapter 4).

Other difficulties are caused by defects, either in printing or scanning. For example, the lines will normally not be perfectly horizontal, as the page normally has to be manually aligned in the scanner. It can, however, be assumed that the lines are fairly close to horizontal: the deviation can reliably be expected to be less than two or three degrees.

Similarly, scanning a page from a thick book can lead to lines being slightly bent, as the page is not always flat against the scanning surface. Finally, lines in the scanned image can have several pixels missing.

Before the staff lines can be removed they must, of course, be located. This is achieved by means of projections. Projecting a two-dimensional image with one bit of data per pixel onto one of its axes reduces the image to a histogram along that axis. The value at any point in the histogram is the number of set bits in the corresponding row or column of the input image. Thus, if vertical strips of the image are projected onto the y-axis, any horizontal lines in those strips will cause corresponding peaks in the resulting histogram. Provided that the strips are not too wide, each staff will cause five equi-spaced peaks, even if the staff lines are not quite either horizontal or straight. By locating these peaks in the projections, the position of the staff lines can be easily determined. If projections are taken at a few points along the staff, bowed staff lines are approximated by straight line segments, and do not present any special problems. Taking projections at various points on the staff also provides some redundant information, which can be used to improve the robustness of the detection algorithm.

Once the vertical position of a staff line is known, it can be followed to either side to locate the horizontal positions of its end-points.

The general process is illustrated in Figure 9.1, where A and B represent projections taken at the left and right ends of the staff line, respectively.

Figure 9.1: Detecting the staff lines.

Once the positions of the staff lines are known, they can be removed. This is done one line at a time.

Condition	Action
Nothing above or below the line	delete the line
1 pixel directly above XOR 1 pixel directly below the line	delete the line if the column on neither side stayed, then also delete the extra pixel
1 pixel directly above AND 1 pixel directly below the line	if the column on either side stayed, then this one also stays, else it is deleted
2 or more pixels directly above XOR 2 or more directly below the line	delete the line but leave the rest
(2 or more pixels above AND below) OR (1 above AND 2 or more below) OR (2 or more above AND 1 below)	all stays

The basic method is to track along the line and delete it, leaving only those sections which are part of a superimposed symbol. This is achieved by examining points above and below each "column" of the line, where a column of the line is simply all pixels on the line with a specific x-coordinate. Since the algorithm operates on the pixel level, it requires that the image be scanned in at a reasonable resolution (section 9.1).

The following rules control the process for each column of the line. In this table, "line" refers to the current column of the staff line.

This is illustrated in Figure 9.2, and the result of this process is shown in Figure 9.3.

Unfortunately, this method is not perfect, and there are cases in which part of a symbol superimposed on a staff line will accidentally be erased. All of the examples of staff-line removal algorithms which we have seen in the literature suffer from some problems. This indicates that these problems are at least very hard (if not impossible) to solve, and that the resulting errors must be dealt with at some later stage.

Examples of this are shown in Figures 9.4 and 9.5. In Figure 9.4 the stem of the right-most note no longer touches the note-head after the staff lines have been removed. This is not a new problem in handwritten music scores, since notes are often drawn with a gap between the stem and the note-head. The first (left-most) four notes

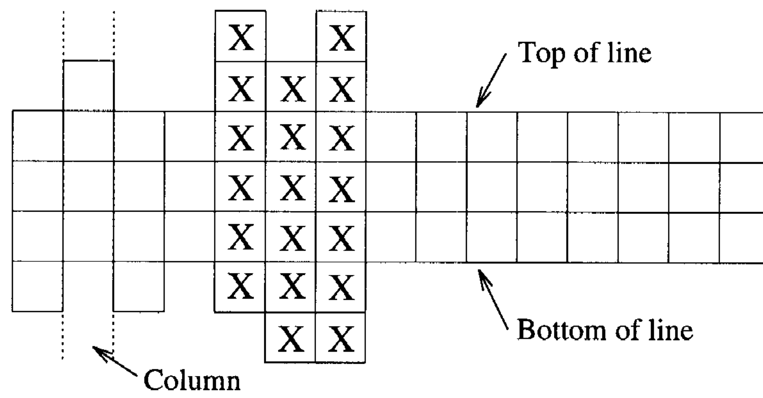


Figure 9.2: Staff line removal.

Figure 9.3: Removing the staff lines.

Figure 9.4: A problem caused by staff line removal.

in Figure 9.4 are examples of this. Fortunately, this problem can easily be solved by our grammar. The production which maps a stem and a note-head into a note states that they must be close together, rather than stating that they must touch (see Figure 2.3).

The errors in Figure 9.5 are more problematic. Note, in particular, the gap created in the top of the bass clef and at the bottom of the 8. The grammar will only be able to cope with these errors if extra productions are written to describe these specific cases.

This imperfection in the staff-line removal process is one of the many examples which we feel substantiates the general philosophy we have adopted in this work: the low-level image processing functions are error-prone and this must be dealt with at a higher level (later stage). Chapters 5 to 7 (in the first part of the thesis) discuss our methods for dealing with these errors. Preliminary results showing how the system copes with noise are given in Chapter 14.

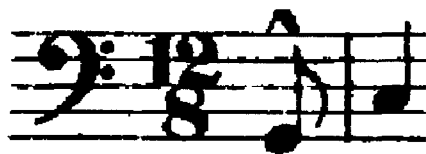


Figure 9.5: More problems caused by staff line removal.

Chapter 10

Selecting primitives (features) and their attributes

The first step in developing a syntactic solution to a pattern recognition problem is to select the primitive elements in terms of which the image will be described (by the grammar). This selection is, in general, the result of a compromise between the complexity of the grammar and the complexity of the primitive extraction layer.

There is an extremely large number of different levels at which primitives can be chosen, ranging (at least in theory) from the pixel to the complete symbol, where each symbol is extracted as a single unit. Simple primitives have the advantage that they can be easily extracted from the image. However, the use of simpler primitives adds complexity to the syntactic level. In contrast, a choice of complex primitives leads to a simple syntactic level, but places a greater demand on the primitive extraction layers.

We have chosen to use a simple set of primitives in this work. The main reason for this is that even some features which can be extracted relatively easily from

printed scores vary greatly in handwritten scores. Accidentals, for example, can be recognised fairly reliably in printed scores (using templates), but their recognition in handwritten scores presents a significant challenge. We decided to deal with this problem by selecting simpler primitives, at the expense of complicating the syntactic level. This should lead to a system which is better equipped to deal with noise in general.

Although the staff lines are obviously a feature of music scores, they are removed from the image before the primitive extraction stage, and are thus not relevant to this chapter.

10.1 A classification of music scores

Before selecting primitives, we must decide what class of music notation we wish to be able to recognise (Figure 10.1 shows some of the common symbols and terms used in music notation).

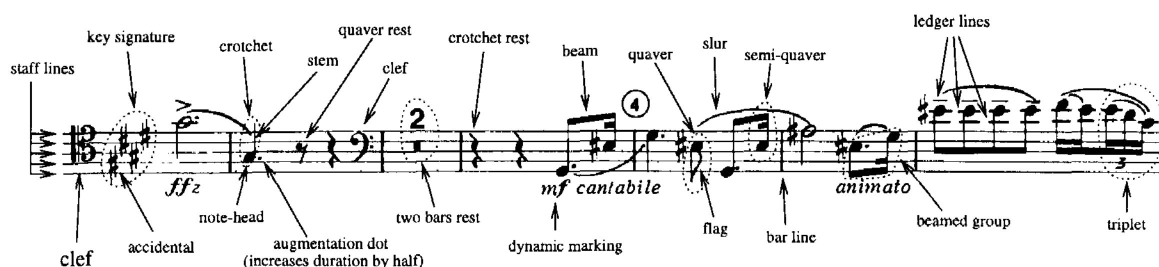


Figure 10.1: Common music symbols and terms.

An initial investigation of CMN (Common Music Notation) scores, with the aim of dividing them into different classes for OMR, yielded a division of scores into three basic classes, ranging from simple to complex. Note that the division is made with respect to the difficulty of OMR, and not to the complexity of the music itself. Although the two are related, OMR is affected by factors such as noise in the image, which is often completely independent of the complexity of the music.

Class one - Simple scores

This is the simplest of our three classes, but includes a large enough subset of music scores to be useful in practice.

Most music symbols are allowed in scores belonging to this class (C1 scores), although only one part is allowed per stave. Triplets and similar special symbols (e.g., duplets) are also excluded from this class, as are time signatures. (Although these symbols are present in even the simplest scores, the fact that they contain digits makes their recognition fairly complex.)

Adjacent symbols may not overlap in C1 scores. Symbols must also have approximately the correct shape. For example, a thin line is not allowed to represent a note-head (as is sometimes seen in handwritten music).

However, the common problem of deformed symbols caused by missing ink (due to bad printing) is allowed in this class. Equivalent deformities are often created during staff line removal and therefore this problem can occur in even the most well printed scores, so that it clearly has to be dealt with in even the simplest class of scores. (The deformation should not, however, cause any symbols to become disconnected. This case is covered by the extended classes, discussed at the end of this section.)

Figure 10.2 gives an example of a simple class one score.

Figure 10.2: A simple class one score.

Class two - Intermediate scores

Class two is a superset of class one. The main extension is that multiple parts per stave are allowed, including note-clusters (chords in which two or more note-heads touch each other). Triplets, duplets, etc. are also allowed in C2 scores.

Overlapping symbols are also permitted. A typical example of this is an accidental touching a ledger line. Class two is thus general enough to include most printed scores.

Symbols in class two can have the "incorrect" shapes that are often seen in handwritten music: a short, slanted line can, for example, represent a note-head. Notice that, since these "incorrect" symbols can still be described in terms of the three chosen primitives, their inclusion does not affect the primitive extraction level. It must be dealt with at the syntactic level.

Figure 10.3 gives an example of a class two score.

Figure 10.3: A printed class two score.

Class three - Complex scores

Class three includes all scores consisting of standard music symbols. Some of the extra symbols that were not included in the previous class are grace notes and cues,

as well as expression markings (for example tempo indications).

The complexity of understanding a music score is influenced by another factor that we have not yet discussed: the sub-parts of individual symbols are disconnected. Although this occurs mainly in handwritten scores, it can be considered as an extension to each class discussed above, since it does not affect the primitives used to describe the score. For example, the class containing all C1 scores and also allowing "disconnected" class one symbols would be known as "Extended Class One" (EC1). Extended classes two and three are defined in a similar way.

Since the primitives are unchanged, the primitive extraction level of an extended class will be identical to that of its base class.

10.2 The primitives

The primitives that were selected are lines, curves and blobs. Lines and curves may not contain sharp corners or undergo a sudden change in width. Each connected group of pixels remaining after all lines and curves have been removed from the image is classified as a blob. This means that the blob class effectively acts as a catch-all. (Features classified as blobs range from note-heads to noise.) The reason for proceeding this way, rather than finding first the blobs and then the lines, is that the shape of the blobs might vary considerably. This makes it difficult to write low-level routines to extract the blobs.

For class two and three scores to be successfully recognised, we will have to cope with the problem of touching note-heads (e.g., in chords). This can either be done at the primitive extraction layer or by means of productions to split "composite" blobs in two.

Consider the two chords in Figure 10.4. The parts enclosed in the dotted lines consist

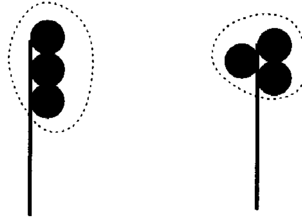


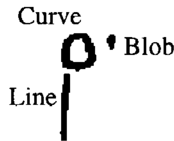
Figure 10.4: Examples of a chord.

of three touching note-heads. The system needs to separate each of these "composite" blobs into its three component note-heads. There is a possibility of doing this at the primitive extraction phase. The blob extraction algorithm could be modified to search for indentations in large blobs. The composite blob could then be split at these points, and the resulting blobs processed separately. Of course, the divisions will not always be easy to make, and may be too difficult for the primitive extraction layer to cope with. By moving the splitting up to the syntactic level, we allow for the possibility of using contextual information to aid the process. It may be possible to make such a split using only the bounding box of the blob and the relative position of nearby stems and staff lines. However, this information may be inadequate, in which case a new attribute will have to be added to the blob primitive.

We feel that the primitives selected are sufficient for describing all three image classes mentioned above (a few extra attributes may possibly need to be added). They are also likely to enable us to describe new symbols that a musician might define.

A number of example productions have already been given (e.g., Figures 2.3 and 4.3). Figure 10.5 shows another example of the way in which these primitives can combine to form a music symbol, in this case a dotted minim. The attributes associated with these primitives are given below the symbol. The meaning of the the attributes is given at the end of this chapter. (The extraction of these primitives from the image is discussed in subsequent chapters.)

An advantage of this choice of primitives is that they can be used to describe a much larger class of images than just music scores. (This obviously increases the generality



curve: 73 33 0 0.904762 1.380623 6.456662 6.456662
line: 65 84 66 46 0.352941 -0.355636
blob: 97 27 0.186851 1.500000 0.777778

Figure 10.5: Composition of primitives to form a music symbol.

of the system.)

It may seem inefficient to have chosen lines and blobs as primitives rather than stems, beams, flags, note-heads and dots. However, especially in the case of handwritten scores, it is not always possible to distinguish between, for example, a flag and a beam, with only local knowledge at one's disposal.

Note-heads also appear to be simple elements to recognise. There are, however, some factors which complicate their recognition. Hollow note-heads which are situated in the space between two staff lines are often deformed during the staff line removal process, resulting in two disconnected lines. As previously pointed out, an oblique line can also represent a note-head in handwritten scores (section 14.3). It is not, in general, possible to distinguish these lines from other short lines such as flags without some information about the surrounding symbols.

By keeping the primitive extraction layer simple and only making these decisions at the syntactic level, it is possible to use knowledge of surrounding symbols to make a correct choice.

Since the primitives we have chosen are ubiquitous and occur in many different forms and contexts, it is essential to associate a list of attributes with each one to describe its salient aspects. If these attributes were not calculated and stored, information would obviously be lost.

To make the recognition process independent of the size of the score and the scanning resolution, all distance measures are recorded in terms of the vertical distance between two adjacent staff lines (hereafter referred to as the staff width). For example, the length of a line is recorded as: (length of line in pixels)/(distance between two adjacent staff lines in pixels).

A list of the attributes of each primitive follows.

blob	x	x-coordinate of centre of blob
	y	y-coordinate of centre of blob
	size	area of bounding box
	aspect	aspect ratio of bounding box
	solidity	fraction ranging from 0 (completely hollow) to 1 (completely solid)
line	x1	x-coordinate of first end-point of line
	y1	y-coordinate of first end-point of line
	x2	x-coordinate of second end-point of line
	y2	y-coordinate of second end-point of line
	thickness	
	curvature	
curve	x	x-coordinate of centre of bounding box
	y	y-coordinate of centre of bounding box
	intersections	number of intersections
	aspect	aspect ration of bounding box
	size	area of bounding box
	curvature	
	total absolute curvature	

Chapter 11

Extracting lines and curves

As was mentioned in the previous chapter, the line and curve primitives are extracted from the image first. They are then removed, to simplify the extraction of blobs. Our initial approach was to extract first the blobs and then the lines. However, the large amount of variation that occurs in blobs made this difficult, and we settled on the current order.

Since line primitives in this system are not required to be perfectly straight (in the prototype system a line is permitted a maximum curvature of 120 degrees) and may have any orientation, the only differences between lines and curves are the maximum curvature allowed and the fact that curves may intersect themselves. The latter factor does not make curve following any more complex than line following, since lines may intersect other lines or curves. The reason for distinguishing between lines and curves is that different attributes need to be stored for each. Consider Figure 10.5. The most intuitive representation of the line is to store its endpoints, while it is more intuitive to store the centre and bounding box of the curve forming the note-head.

If the distinction is not made between lines and curves, more data must be stored

with each primitive. Nevertheless, this division into lines and curves is not necessarily optimal. More work can be done in comparing these two separate classes with a single unified class.

Since the requirements of an algorithm for detecting and following line primitives are a substantial subset of those for detecting and following curves, it makes sense to develop an algorithm which can detect and follow general curves and to use it for following both curves and lines.

This algorithm converts each curve (or line) into a list of pixel coordinates. Only when this list is analysed, is a distinction made between lines and curves.

11.1 Line thinning

The process of following curves is complicated by the fact that they are, in general, more than one pixel thick. This prevents the use of an algorithm which simply finds the next neighbouring pixel, as it would meander around in the thick line. One possible solution to this problem is to thin the image first. If the image can be thinned in such a way that all lines are one pixel thick, and are either four- or eight-connected, the task of following these curves is simplified. (Four- and eight-connectivity are illustrated in Figure 11.1: the pixel labelled 'X' is four-connected to any pixel marked with a '4', and eight-connected to any pixel marked with either a '4' or an '8'.)

Fortunately, simple algorithms exist for thinning an image in this way [Sonka *et al.*, 1993]. The algorithm used requires no a priori knowledge about the image, and proceeds by matching a mask to each point in the image. Each entry in a mask is set to either 0, 1 or 2 (denoting a wild-card). If the mask matches the underlying segment of the image at a particular point, the point in the image corresponding to the centre of the mask is deleted (set to 0). In our specific case 3x3 masks were used, as results did not improve when using 5x5 masks, which are computationally more

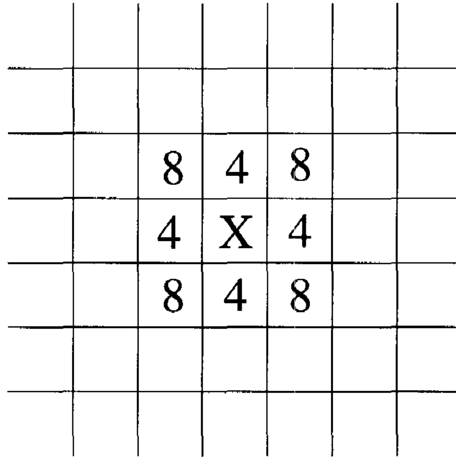


Figure 11.1: 4- and 8-connectivity.

expensive).

More specifically, the basic algorithm is as follows.

```
repeat as many times as is necessary
   make a copy of the image
   for all masks
      for each point in the original image
         if the current mask matches then
           delete the pixel corresponding to the centre of the mask
           from the copy of the image
      replace the original image with the altered copy
```

If suitable masks are used, each iteration removes one pixel from the borders of objects, until they are reduced to four-connected lines one pixel wide. For example, the 3x3 mask shown in Figure 11.2 will remove the top row of pixels from a line that is roughly horizontal. This mask is rotated about its centre to obtain another three masks. Figure 11.3 shows a scanned image and the same image after four iterations of the thinning process, using eight masks (derived by rotating two different masks).

0	0	0
2	1	2
1	1	1

Figure 11.2: A sample 3x3 mask.

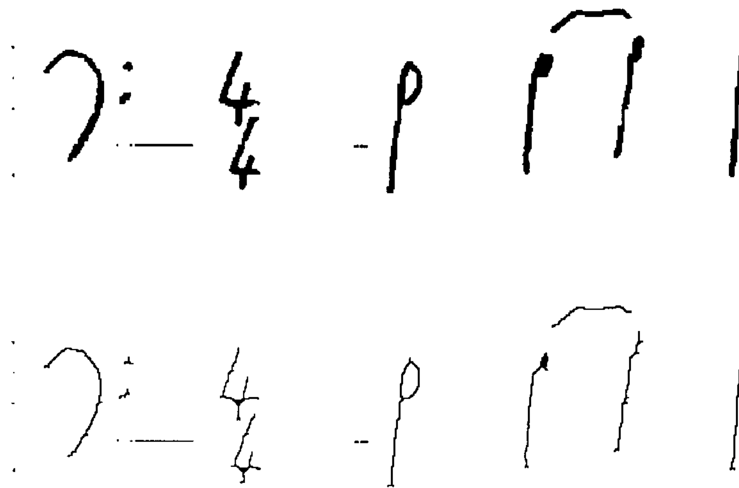


Figure 11.3: Thinning an image.

11.2 Following curves

As described in the previous section, thinning the image reduces the task of following curves to that of following (tracking) a four-connected curve one pixel wide.

The thinned image is searched systematically, and the region around each black pixel is searched to determine whether it is part of a curve or not. The thinning process used has the side-effect that any point on a curve can have at most three neighbours (when it intersects with another line). This means that any pixel with four neighbours signals the end of the curve (another feature that can signal the end of the curve is a sharp angle, as discussed below).

If the pixel being examined is part of a curve in the thinned image, the curve is tracked, first to one end and then to the other. During this process, the coordinates of each point found on the curve are stored in a list. The corresponding pixels are also marked in a separate array, which is initially blank. Whenever a pixel in the thinned image is examined, the corresponding position in this "marker" array is checked. This prevents the same curve from being processed many times. It also prevents the routine from following cyclic curves more than once.

While a curve is being tracked, the current direction (heading) is continually calculated. This is complicated slightly by the fact that the image space is not continuous, but is divided into square pixels. In practice, the current direction is taken to be the average direction (orientation) of the line over the last few pixels (in our prototype we used a length of ten pixels). The orientation of the line segment of equal length preceding those pixels is also calculated. If the angle between these two directions is greater than a threshold value (fifty degrees in the current prototype system) then a corner (and thus the end of the curve) has been found. This is illustrated by Figure 11.4. The choice of fifty degrees was made after experimental tests on a few sample images. The presence of this artificial threshold makes this not a particularly good method for detecting corners. The algorithm could be improved by using the

previous curvature as well as the current angle. This would allow it to distinguish between tight curves with a uniform curvature (such as small circles) and genuine corners.

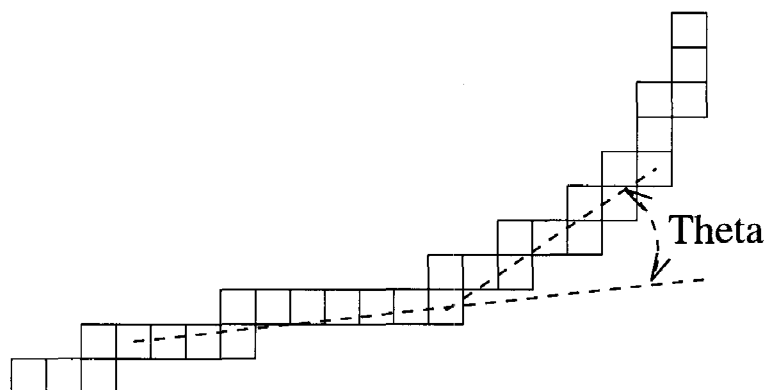


Figure 11.4: Checking for a corner in the curve.

One difficulty in curve tracking is dealing with branches and intersections. While this is relatively easy to handle in the case of straight lines, it can be a problem when dealing with general curves, as it is not always obvious which branch to follow. The approach taken is to attempt to continue in the current direction. When looking for the next pixel in the line, the four neighbours of the current pixel are ordered in terms of their deviation from this ideal direction (from best to worst), and are tested in that order. This ordering is shown in Figure 11.5.

Since this is a local (short-sighted) technique, it can be fooled by small anomalies at intersections. Of course, if the branch taken veers off at a sharp angle, a corner will be detected and the curve will end at the intersection point. The thinning process

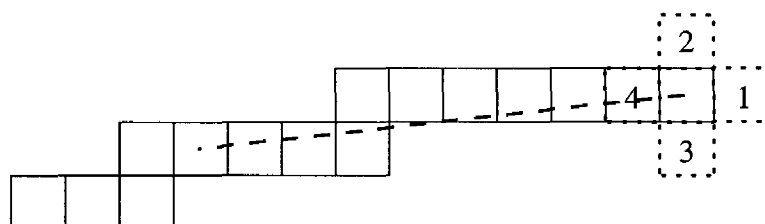


Figure 11.5: Checking for the next pixel of a curve.

can introduce sharp corners in thick curves at intersection points. In this case the curve will be segmented at the intersection point to form two curves. The algorithm could cope better with a branch in the curve if it followed a short section of both branches before deciding which one to follow.

We stated in Chapter 10 that a curve ends if it suddenly changes width. To check for this, the width of the line in the original (non-thinned) image is calculated at each point, and compared with the previous width. If the width has changed by more than a certain number of pixels (three in our prototype system), the end of the curve has been found.

The width of a line at any point is the number of pixels in a direction perpendicular to the direction of the line at that point. To simplify matters, the method actually used was to measure the number of pixels in either the horizontal or vertical direction, depending on which was closer to being perpendicular to the line at that point. This simplification has not had any adverse effect on the accuracy of the curve-following algorithm. The reason for this is that when the line is at approximately forty five degrees from the vertical, at which point the algorithm might switch from using the "vertical width" of the line to the "horizontal width", there is not much difference between the two measures.

Once again, the presence of branches and intersections complicates matters, as they obviously cause a sudden change in the width of the curve. To deal with this problem, the curve is permitted to change width over a short length, provided that it returns to its original width afterwards (in our prototype we used a length of five pixels).

Figure 11.6 shows the curves found in the image in Figure 11.3. The short line to the left of the minim is recognised as a blob (because of its short length), as is the bottom part of the lower 4. The latter shows up a difficulty that the system has with intersecting lines. The minim is recognised as two lines, because the line thinning algorithm left a short "spike" at the top. One could argue that it should have been recognised as a single curve. This is shown in Figure 11.7. Fortunately, we can write

a production to cope with this case (see section 13.2). If the curve was not split into two, it would have been recognised as a single complex curve.

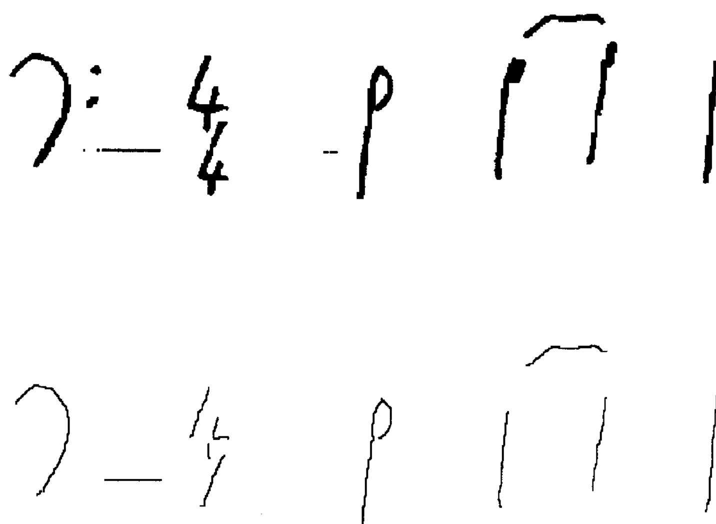


Figure 11.6: Curves found in the image.

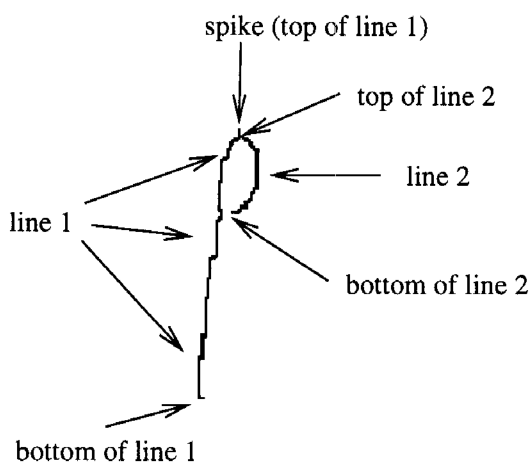


Figure 11.7: A problem case.

11.3 Calculating a curve's attributes

Once a curve has been followed, it is stored as a list of coordinates and a width (in terms of pixels). While this contains all the information needed for the line, it is certainly not in an easily accessible form. The attributes mentioned in Chapter 3 must be calculated from the list of coordinates.

The total curvature of a curve is calculated by approximating the curve with straight-line segments and adding all the angles between adjacent segments. It is thus a summational measure, and is not normalized on line length. The curvature is used to decide whether the curve being analysed is a line primitive or a curve primitive. Subsequent processing of the coordinate data is, of course, dependent on this decision.

If the curve is a line primitive, the only other calculation that needs to be made is to divide its width (in terms of pixels) by the staff width, to obtain the width as a fraction of the staff width. The end-points of the line are simply the first and last entries in its list of coordinates.

A few more calculations are necessary for a curve primitive. The number of intersections are counted by comparing each (unordered) pair of coordinates for equality. If two coordinates are equal, an intersection has been found. One complication is caused by the fact that an intersection can occupy two adjacent pixels, instead of just one (Figure 11.8). This is dealt with by analysing the list of intersections and ignoring all but one intersection in each adjacent group (cluster).

The aspect ratio and size of a curve primitive are determined from the bounding box. This is calculated by finding the top, bottom, left and right extremities of the curve (in the coordinate list) and then adjusting these values to allow for the curve's width. The aspect ratio of the curve is the box's height divided by its width, while the size is simply its area.

In the prototype, the appropriate attributes are calculated and written to a text file,

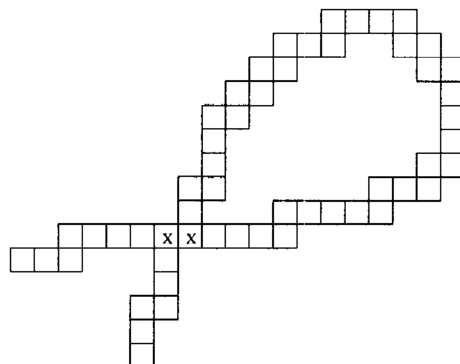


Figure 11.8: Intersection in a curve.

which serves as input for the syntactic layer. A list of the curve and line primitives extracted from the image shown in Figure 11.3 follows.

	x	**y**	**no.intersections**	**aspect ratio**	**size**	**curvature**
curve:	52	78	0	1.854167	6.835200	2.761086

	x1	**y1**	**x2**	**y2**	**thickness**	**curvature**
line:	484	10	423	22	0.280000	1.190290
line:	165	77	179	38	0.280000	0.159913
line:	560	133	569	41	0.240000	0.134321
line:	310	77	312	46	0.440000	1.570796
line:	312	50	299	146	0.240000	0.588003
line:	484	46	474	118	0.240000	0.000000
line:	409	57	404	131	0.280000	0.221314
line:	183	78	186	61	0.240000	-0.134321
line:	194	78	183	78	0.240000	0.000000
line:	179	82	180	94	0.240000	0.000000
line:	173	130	191	90	0.280000	0.183111
line:	143	110	99	110	0.120000	0.000000
line:	51	119	47	122	0.120000	0.000000

11.4 Removing curves from the image

The method used to extract blob primitives dictates that curves be removed from the image before the blobs can be detected.

The method used to delete a given curve from the image is relatively simple. A blank

mask (a small array with every element set to zero) is centred on each coordinate in the list describing the curve, and is used to erase the underlying pixels (through a bitwise AND). The mask is square, with its sides slightly longer than the width of the curve. The width of the curve is calculated by taking the width every few pixels, and filtering out values which are much larger than the surrounding values and are thus obviously caused by a branch or intersection (in our prototype we measured the width every 5 pixels). The width is then taken to be the maximum of the remaining values. This calculation is illustrated in Figure 11.9. Given the values shown in the figure, the widths of eight and sixteen pixels would be filtered out. These widths are both caused by what the algorithm perceives as branches (in the case of the width of 8 pixels, it is a branch at the end of the line). The width of the line would thus be calculated to be five pixels.

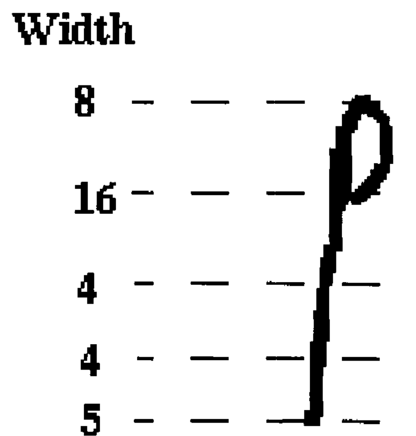


Figure 11.9: Calculating the width of a line.

The reason for the sides of the mask being slightly longer than the curve's width is that the thinning process may not leave the thinned curve exactly in the centre of the original curve. Recall that the coordinates in the list describing the curve are those of the thinned curve. A decision was made to err on the side of deleting a little too much rather than delete too little. If a few pixels are accidentally erased from a blob, the consequences will not be serious, but if part of a curve is left in the image, it will

confuse the blob detection algorithm.

Figure 11.10 shows the result of removing the curves from an image.

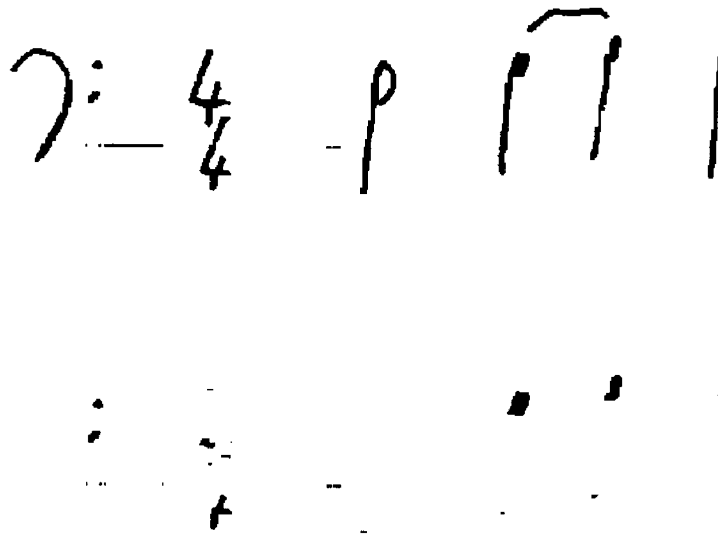


Figure 11.10: Removing curves from the image.

Chapter 12

Extracting blobs

Each group of four-connected pixels left in the image after the staff lines and curves have been removed is considered to be a blob primitive. This chapter discusses the extraction of these primitives from the image.

12.1 Blob extraction

The groups of four-connected pixels are found using a two-pass method. First, all horizontal pixel runs are found, and then these are grouped into blobs. (A pixel run is a line of pixels not separated by any white space.) Finding all horizontal pixel runs in the image is a straightforward task. They are stored using an array, each element of which is a list of the pixel runs in the corresponding row of the image. This simplifies the functions that extract blobs, and occupies considerably less space (typically about fifteen percent of that required for the bitmap representation). The latter aspect was important in an early version of the prototype, which ran under DOS.

Grouping the pixel runs into blobs is not quite as simple. Any unmarked pixel run is chosen, marked and copied to a blank "pixel runs" data structure. All other pixel runs that are connected to this are also marked and copied to the new data structure. This is achieved in an iterative fashion through successive passes up and down the blob as it forms. When passing down through the image, each unmarked pixel run on the row below is checked to see if it overlaps (horizontally) with any pixel run on the current row. If it does, it is marked and copied. The marking prevents any pixel runs from being selected more than once.

When no unmarked, overlapping pixel runs are found on a row, the process is reversed, passing back up through the image. The direction is reversed once more when no unmarked, overlapping pixel run is found in a row. This is repeated until an entire pass occurs without any pixel runs being added to the new blob's data structure, at which stage every pixel run in the blob has been marked and copied. Once the whole blob has been identified, its attributes are calculated as described in the next section. In the prototype, they are written to the text file which already contains the attributes of all lines and curves.

Figure 12.1 shows a sample blob and the four passes needed to extract it from the image. Step 1 shows the initial pixel run, before the examination begins. The first pass is done in a downward direction. The result of this is shown in step 2. The pixel runs constituting part A of the image are missed, as they do not overlap with the preceding pixel runs. When the bottom of the blob is reached, the direction is reversed, and an upward pass begins. The pixel runs in part B of the blob are missed, as they do not overlap with the directly preceding pixel runs (step 3). The next downward pass identifies these pixel runs as being part of the blob (step 4). The next upward, pass does not find any new pixel runs, and the algorithm thus terminates.

This process is repeated for every blob in the image. When all pixel runs in the image have been marked, all blobs have been processed and the primitive extraction stage is complete.

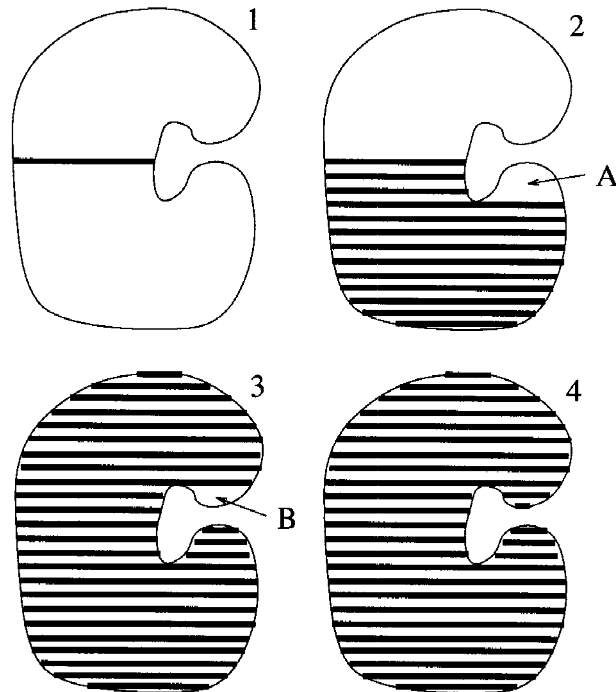


Figure 12.1: Extracting a blob from the image.

12.2 Calculating a blob's attributes

Once a blob has been extracted as a list of pixel runs, the attributes mentioned in Chapter 3 can be calculated. The centre of the blob is simply taken as the centre of the blob's bounding box. The bounding box is found by locating the top, bottom, left and rightmost pixel runs in the blob.

The size of the blob is taken as the area of the bounding box, normalized to the square of the staff space width. The blob's aspect ratio is calculated by dividing the bounding box's height by its width.

The blob's solidity is calculated by analysing the blob one row at a time. The area between the extremities of each row is divided into three regions, the first quarter, middle half and last quarter. The first and last quarters are labelled as the "outside" of the blob, and the middle half is labelled as the "inside" of the blob. This is shown

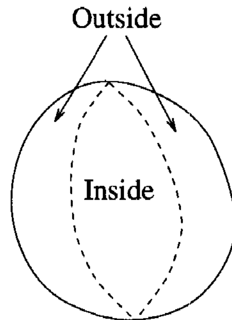


Figure 12.2: Calculating the solidity of a blob.

in Figure 12.2. The number of pixels in each of these regions is calculated, and the values are summed for all rows of the blob. The solidity of the blob is the number of pixels in the outside divided by the number in the inside. This specific solidity measure was developed with note-heads in mind, so that a roughly scribbled note head can be seen as solid, provided that the degree of fill is consistent throughout. A list of the blob primitives and their associated attributes extracted from the image follows.

	x	y	size	aspect ratio	solidity
blob:	487	34	0.416000	1.538462	1.014286
blob:	4	33	0.006400	1.000000	0.000000
blob:	179	37	0.008000	0.200000	1.000000
blob:	415	47	0.516800	1.117647	0.990000
blob:	569	40	0.008000	0.200000	1.000000
blob:	93	448	0.089600	0.875000	1.000000
blob:	5	57	0.001600	1.000000	0.000000
blob:	91	73	0.129600	1.000000	1.086957
blob:	176	78	0.134400	0.583333	1.000000
blob:	195	77	0.001600	1.000000	0.000000
blob:	4	81	0.003200	0.500000	0.000000
blob:	191	88	0.022400	0.285714	1.000000
blob:	179	94	0.009600	0.666667	1.000000
blob:	85	109	0.003200	2.000000	0.000000
blob:	95	109	0.006400	1.000000	0.000000
blob:	99	109	0.003200	2.000000	0.000000
blob:	143	109	0.003200	2.000000	0.000000
blob:	270	110	0.012800	0.500000	2.000000
blob:	276	110	0.019200	0.333333	0.666667
blob:	473	118	0.024000	0.600000	1.250000
blob:	187	131	0.691200	1.687500	0.921053
blob:	403	131	0.004800	0.333333	1.000000
blob:	4	133	0.009600	1.500000	0.000000
blob:	297	146	0.008000	0.200000	1.000000

Chapter 13

Defining the grammar

Once all the routines have been written to extract primitives from the image, the remaining task is to write the productions forming the grammar according to which the primitives are parsed.

We found that the easiest way to write the grammar in this particular case was to use a bottom-up approach to transform primitive elements into music (class two) symbols, and a top-down approach to define a bar of music in terms of music (class two) symbols.

Of course, we needed to select a representation for the productions, so that they could be entered into the computer.

13.1 Choosing an external representation for the productions

The algorithm to parse an input graph according to a grammar obviously needs the grammar to be represented somehow. We refer to this as the internal representation,

and give its details, as C code, in appendix C.

It is important that the productions in the graph grammar be easily modifiable, both for experimentation and to cope with a larger subset of music symbols. We therefore need a simple external representation for the productions, so we designed a simple language in which graph grammar productions can be easily defined.

In this language, the production shown in Figure 2.3 would be represented as follows:

```
Production 9

# form note from hollow note-head and stem

GL:
  Nodes = {(1,notehead), (2,stem)}
  Edges = {}

GR:
  Nodes = {(1,note)}
  Edges = {}

Embed = {(1,1), (2,1)}

Attrib:
  {_1.pitch =GetPitch (Centre(1));
   _1.duration = MINIM;
   _1.x =_1.x;}


Apply = {(Close (Top(2), Left(1)) ||
          Close (Bot(2), Right(1))) &&
          Hollow (1)}
```

One aspect that is not as intuitive as it could be, is that the subgraphs represented pictorially in Figure 2.3 must be entered as a list of text statements. This simplifies

the job of parsing the production definition language, which is defined through a list of LEX and YACC rules. It would not be difficult to build a system which allowed the user to draw the left- and right-hand subgraphs associated with each production (using some standard drawing package) and then converted the diagrams into the required textual format.

In the textual representation that we chose for our prototype system, each subgraph is represented by a list of nodes and a list of edges. Each node is represented by a (*node number, node label*) pair, where the label is the name of one of the token types. Each edge is represented by a pair of node numbers. Since undirected edges are used, the edge (N, N') is equivalent to the edge (N', N).

In the grammar description language, any line in which the first non-blank character is a # is recognised as a comment and is ignored. Node numbers in the attribute transfer function are preceded by an underscore (_) to simplify the parsing.

The full LEX and YACC specification of the production definition language is given in appendix A.1.

A simple music recognition grammar with productions written in this format is given in appendix A.2 as an example. This is the grammar that was used for the tests described in Chapter 14.

A similar textual representation is used for the control diagram, which consists of a list of nodes and a list of edges. Each node is given in the form *node number : list of production labels*. Each edge is given in the format $(n, n', label)$, which defines an edge from node n to node n'. The edge label is either Y or N.

For example, the control diagram given in Figure 3.2 would be represented as follows:

```
# Text format of a control diagram

Nodes:
1 : 1,2
2 : 3

Edges
(I,1,Y), (1,1,Y),(1,2,N), (2,2,Y),(2,F,N)
```

The LEX and YACC rules defining this representation of the control diagram are given in appendix B.1. The control diagram for the grammar given in appendix A.2 is given in appendix B.2. As is the case with the subgraphs that form part of the productions, it would not be difficult to write a system that allowed the user to draw the control diagram (using some standard drawing package) and then converted the diagram into the required textual format.

13.2 Writing the productions

We shall now examine a few productions as an illustration of the type of productions we have used and to extract some general rules for defining them. Efficiency issues which must be born in mind when writing productions are discussed in section 7.3.

A grammar is typically written in an iterative manner. The first stage normally involves writing a grammar which describes the image class according to primitives that are extracted in the way that we expect. The production given in section 13.1 illustrates this stage. The fuzzy certainty function requires some additional care Chapter 4). For example, the function "Close", which, in this case, is used to test the distance between the end of the stem and the note-head, can only be defined after examination of a number of images. These fuzzy functions will invariably have to be improved after experimental results are available.

The next production was written to add edges that will associate a slur with the two

notes close to its end-points. We include it here as an example of a production that adds only edges.

```
Production 7

# form edges between a slur
# and the corresponding notes

GL:
  Nodes = {(1,slur), (2,notehead), (3,notehead)}
  Edges = {}

GR:
  Nodes = {(1,slur), (2,notehead), (3,notehead)}
  Edges = {(1,2), (1,3)}

Embed = {(1,1), (2,2), (3,3)}

Attrib:
  { /* just copy across all the attributes */
/* slur   (1) */
   _1.x1 =_1.x1;
   _1.y1 =_1.y1;
   _1.x2 =_1.x2;
   _1.y2 =_1.y2;
/* notehead (2) */
   _2.x =_2.x;
   _2.y =_2.y;
   _2.solid =_2.solid;
/* notehead (3) */
   _3.x =_3.x;
   _3.y =_3.y;
   _3.solid =_3.solid;}

Apply = {(Close (Fairly, Top(2), Left(1)) ||
          Close (Fairly, Bot(2), Left(1))) &&
```

```
(Close (Fairly, Top(3), Right(1))||
 Close (Fairly, Bot(3), Right(1))) &&
 !Edge(1,2) && !Edge(1,3)}
```

Note that the attributes of the nodes in the right-hand side graph are copied directly from those in the left-hand side graph. A construct of the form _1.all = _1.all would be useful in situations like this. The function `Close` used in the certainty function is given in Figure 4.2.

The second iteration of writing the grammar typically begins once experimental results are available from the primitive extraction routines. In our case, after the primitive extraction routines had been implemented, we found that the list of extracted primitives included a number of very small blobs (errant pixels) which increased the complexity of a parse. Since we felt that it was safe to delete these small blobs (as they could not be anything other than noise in our application), we wrote the following production with a crisp certainty function to delete them. Efficiency considerations arising from this production are discussed in section 7.1.

```
Production 1

# remove blobs that are too small
# just to filter out a lot of extra noise

GL:
  Nodes = {(1,blob)}
  Edges = {}

GR:
  Nodes = {}
  Edges = {}

Embed = {}

Attrib:
```

```
    {}

Apply = {_1.size < 0.07}
```

The next production describes the altered minim in figure 11.7. A minim normally consists of a hollow blob and a stem, as defined by the first production given in this chapter. In the case of Figure 11.6, the primitive extraction routines did not behave exactly as we had expected, as explained in section 11.2. To cope with this case, we wrote Production 5.

```
Production 5

# map line and broken notehead onto
# stem and hollow notehead

GL:
  Nodes = {(1,line), (2,line)}
  Edges = {}

GR:
  Nodes = {(1,stem), (2,notehead)}
  Edges = {}

Embed = {(1,1), (2,2)}

Attrib:
  {/* stem */
   _1.x1 =_1.x1;
   _1.y1 =_1.y1;
   _1.x2 =_1.x2;
   _1.y2 =_1.y2;

   /* notehead */
   _2.x =MidPoint(2)->x;
   _2.y =MidPoint(2)->y;
```

```
  _2.solid =0.0;}

Apply = {Match (Length(1)/4.0, 1.0) &&
  Match (Length(2), 1.0) &&
  Close (Very, Top(1), Top(2))
```

The function `Match` simply implements a fuzzy version of equality. Its value depends on the difference between the two arguments. The certainty function thus specifies that one line must be approximately the length of an average stem, while the other should have a length equal to the width of the gap between two staff lines. Furthermore, the tops of the two lines must be close together. We could have specified additional constraints, such as the fact that the line representing the stem must be roughly vertical, but this was not necessary, as the certainty function given was found to be sufficient to isolate this case.

We imagine that a grammar to describe class three music would probably contain between two and three hundred productions. Some of the additions are not difficult. Although they were not implemented, we have written productions to cope with rests, accidentals, ledger lines and beams. Other symbols present more difficulty. Further work is needed to enable the system to cope with chords. One possible approach is to leave the primitive extraction layer as is, and add a production to split a blob which is actually a cluster of note-heads into separate blobs, one for each note-head (see Chapter 10. Another source of potential problems is the small numbers denoting duplets, triplets, etc. Further investigation of the curve primitive (Chapter 11) is needed to determine if it is suited to describing, among other symbols, these small numbers.

13.3 Translating the productions into their internal representation

Before the productions can be applied to parse the input graph, the file containing their definition must itself be parsed and the productions translated into the form used by the fuzzy parser (we have referred to this as the internal representation). For each production, the left- and right-hand subgraphs are stored as lists of nodes and edges. The embedding transformation is stored as a list of tuples. The attribute transfer function and the applicability predicate are a bit more difficult to deal with. Since they are defined as pseudo-C functions, the easiest way to deal with them is to translate them into true C functions, and to use an existing C compiler to compile them and produce executable versions. This is clearly easier than writing an interpreter or compiler for these functions ourselves.

This strategy leads to a two-pass approach to parsing the productions. On the first pass, the attribute transfer function and applicability predicate are translated into C functions which are written to a file. The system also generates code to assign pointers, from the data structure that will contain the productions, to these functions.

Once the C functions written in the first pass have been compiled, they are linked in with code for the fuzzy parser. The initialization code for the fuzzy parser translates the productions into their internal representation, as explained above. In order to set up the attribute transfer functions and applicability predicates, it simply executes the code (written by the first pass) that places the addresses of these functions in the data structure containing the productions.

After this stage, the productions are in a form in which they can be applied to parse the input graph.

We wrote all the code to parse the productions and to apply them to parse the input graph in C and C++. The graph-rewriting tool PROGRES has since been released, and should make the task significantly easier for future developers of graph-rewriting

systems. Further information on PROGRES can be found at the URL http://www-i3.informatik.rwth-aachen.de/research/progres/index.html.

Part III

Conclusions

Chapter 14

Preliminary results

We now present some preliminary results. This allows us to show the complete working of the prototype system, and thus also of the framework. It will also shed some light on the advantages and limitations of both.

The prototype for recognising music scores described in the second part of this thesis shows how the framework we have developed can be used for practical applications.

We shall now examine three examples. The first shows the basic working of the prototype (and thus also of the framework), while the others show how the system copes with certain types of noise.

14.1 A simple case

Consider Figure 14.1. This shows a bar of a very simple music score, in $\frac{4}{4}$ time.

The staff lines are detected by using vertical projections and are erased according to the simple algorithm given in section 9.2. Once the staff lines have been removed we

Figure 14.1: A simple bar of music.

get the image shown in Figure 14.2. Note that, even in this simple case, the staff line removal process introduces some noise by thinning the note-heads a little, and introduces a gap between the first note-head and its stem.

Figure 14.2: Image with staff lines removed.

```
Nodes:
line: 61 64  61 12  0.294118  0.110657
line: 345 63  346 15  0.294118  0.000000
line: 250 83  250 34  0.235294  0.000000
line: 151 97  151 50  0.235294  0.000000
blob: 335 71  1.019377 0.652174 1.020000
blob: 50 74  1.017301 0.666667 1.000000
blob: 239 90  1.003356 0.695652 1.050000
blob: 140 107  1.001280 0.913043 0.978102
```

Figure 14.3: Primitives extracted from a simple image and their attributes.

The next step is to extract the primitives from this image, using the methods discussed in Chapters 13 and 14. First the image is thinned to ease the task of extracting the curves (section 11.1), then the curves are found by following each one in turn (section 11.2). Next, the attributes of the curves are calculated (section 11.3), and then the curves are removed from the image (section 11.4). After this, the blobs are extracted (section 12.1) and their attributes are calculated (section 12.2). The

primitives extracted from the image are given in Figure 14.3, together with their attribute values.

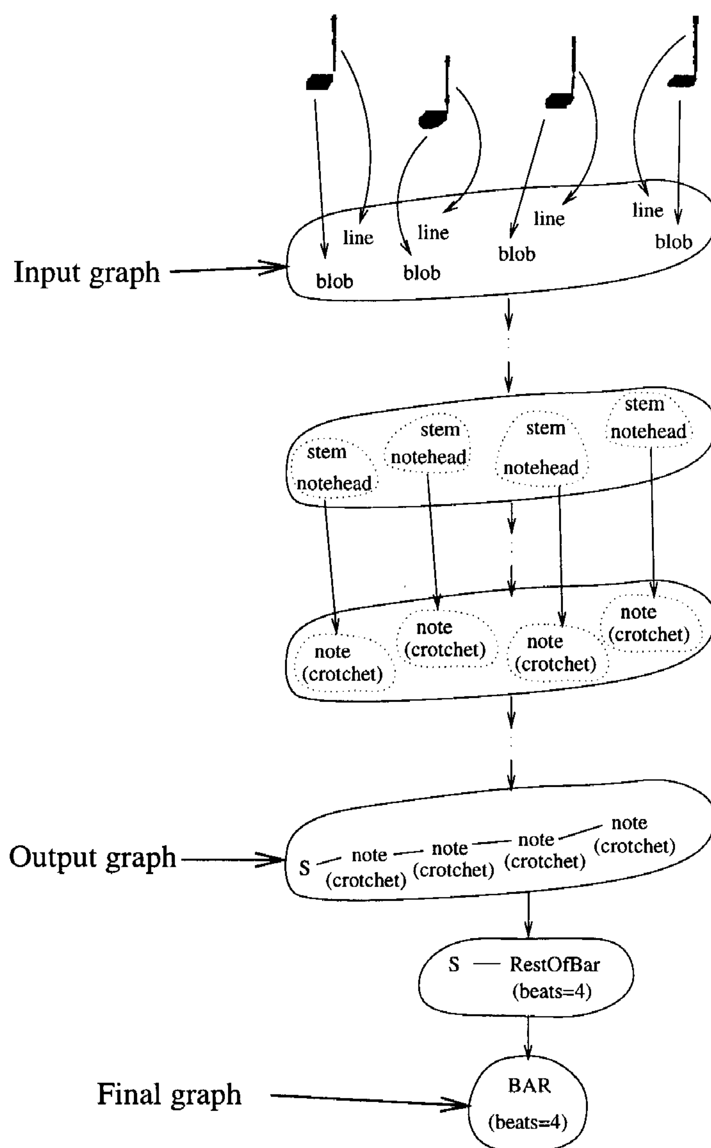


Figure 14.4: Simple search tree associated with the parse.

The input graph containing these primitives was parsed using the parser formally defined in section 6.1, the implementation (in C) of which is given in appendix D. our programmed fuzzy graph grammar mechanism, according to the grammar given in appendix A.2 and the control diagram given in appendix B.2. A condensed version

of the resulting search tree is given in Figure 14.4. The full part of the search tree examined is given in appendix E.1. Notice that certainty values greater than or equal to 2 in appendices E, F and G denote a noise production that deletes a primitive. The actual certainty value is calculated after the node has been written to the daVinci graph.

Since there is no ambiguity and little noise in this example, the correct interpretation would be found if a crisp grammar were used instead of the fuzzy grammar.

14.2 Coping with noise

The main objective of the work described in this thesis was to develop a framework that can cope well with noisy images. The framework that we have developed can correctly interpret noisy images that are incorrectly interpreted using a crisp version of the grammar. This is illustrated by the following example.

Consider Figure 14.5. There is ambiguity about the duration of the first note, which could be either a minim with extra ink or a crotchet with some ink missing. Although it appears to be more solid than hollow, if we know (from a previous bar) that the bar of music shown in the figure is in $\frac{3}{4}$ time, then we can establish that the note is a minim, i.e., the note-head is hollow. This causes difficulty for a system using a crisp grammar (and for systems that do not make use of higher-level information), as such a system is unable to resolve the ambiguity (see Chapter 4).

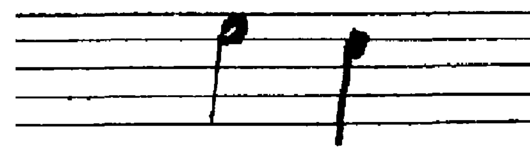


Figure 14.5: A handwritten example.

The staff lines are detected and erased as in the previous example. Once the staff lines have been removed we get the image shown in Figure 14.6.

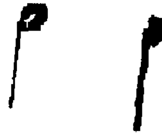


Figure 14.6: Image with staff lines removed.

The next step is to extract the primitives from this image. The primitives extracted from the image are given in Figure 14.7. The two blobs marked with asterisks correspond to the small dots in the image to the left of the notes. The first production in the grammar deletes these blobs before any other production is applied.

```
Nodes:
line: 230 42  219 126  0.200000  0.000000
line: 369 62  362 151  0.300000  0.221314
blob: 243 27  1.171111 0.911765 0.783920
blob: 378 42  1.110000 1.370370 0.959248
blob: 65 101  0.008889 0.500000 2.000000   *
blob: 86 101  0.006667 0.666667 1.000000   *
```

Figure 14.7: Primitives extracted from the image and their attributes.

The input graph containing these primitives was parsed using the parser defined in section 6.1, according to the grammar given in appendix A.2 and the control diagram given in appendix B.2. A condensed version of the resulting search tree is given in Figure 14.8. A full version of the part of the search tree that was examined is given in appendix E.2. If a crisp version of the grammar is used, the certainty values given are collapsed to 0 or 1, depending on whether they are less than 0.5 or not. This means that the production leading to the subtree enclosed by the dashed line has a certainty of 0. That subtree is thus not examined, and the interpretation found is a bar containing two crotchets. Notice that, although we could fix the problem in this specific case by changing the cut-off point, this would not work in general.

If a fuzzy grammar is used, the first note can be either a crotchet or a minim. If the uniform search strategy described in Chapter 7 is applied, the subtree in which the

first note is interpreted as a crotchet is examined first, as it has the higher certainty (0.79 compared to 0.21). However, when it is found that the interpretation in this subtree contains an incorrect number of beats (two instead of three), its certainty becomes lower than that of the unexplored subtree. That subtree (in which the first note is interpreted as a minim) is then explored, and the correct interpretation of the image is found. Since the interpretation associated with this parse path has the correct number of beats in the bar, it has a higher final certainty than the incorrect subtree examined first.

The use of our fuzzy graph grammar mechanism thus allowed the system to use global knowledge (the number of beats in the bar) to solve a local ambiguity (the solidity of the first note-head).

The advantage gained from programming the grammar can be seen in the first (pre-processing) production. If a non-programmed grammar is used, the first production can be applied any time that there is a match for its left-hand side subgraph. Because we use a programmed grammar, we can specify that the first production is applied completely before any other productions are applied, and never again after that. This increases both the intuitiveness of the grammar and the efficiency of the parse.

The pruning of the search tree explained in section 7.1 reduced the size of the search tree in this example by a factor of approximately 12 (to 707 nodes). A depth-first search would have to examine anywhere from 53 to all 707 nodes before finding the correct interpretation. A breath-first search would have to search nearly all of the nodes. Although the wrong subtree was examined first by the uniform search strategy explained in section 7.2 (because, using only local information, the first note-head appeared to be more solid than hollow), the correct final graph was found after 99 nodes out of a total of 707 in the state space search tree had been examined. None of the subtrees resulting from "noise" productions were examined, as they had low certainties.

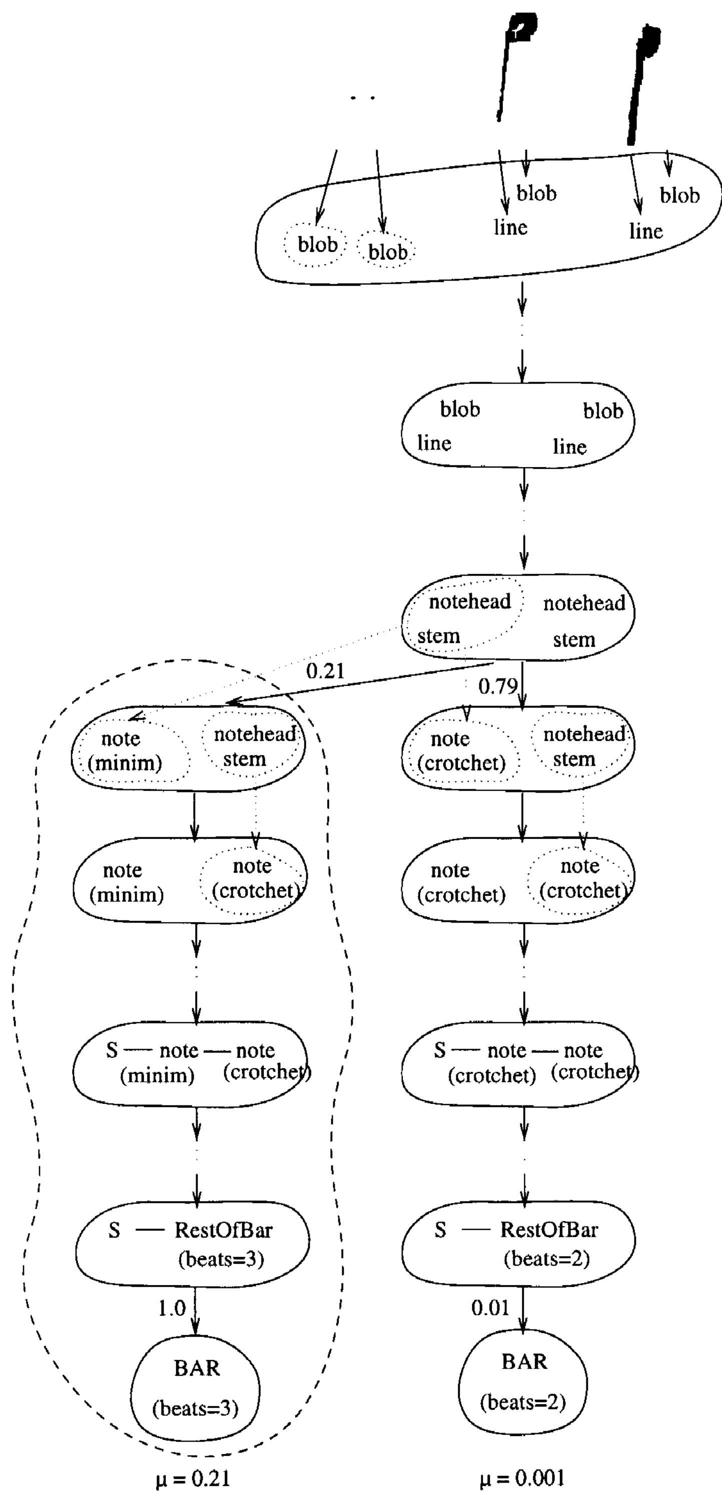


Figure 14.8: Search tree associated with the parse.

14.3 More noise

The example arising from the image shown in Figure 14.9 is a good illustration of some of the strengths and weaknesses of the system. The music score in this example illustrates some typical features of hand-drawn scores. The note-heads do not touch the stems, and are not well-formed as is the case with printed scores. The relatively thin lines representing the note-heads result in the first one being represented by a complex curve (as opposed to the hollow blob normally extracted from printed scores), while the others are represented by short lines (instead of solid blobs).

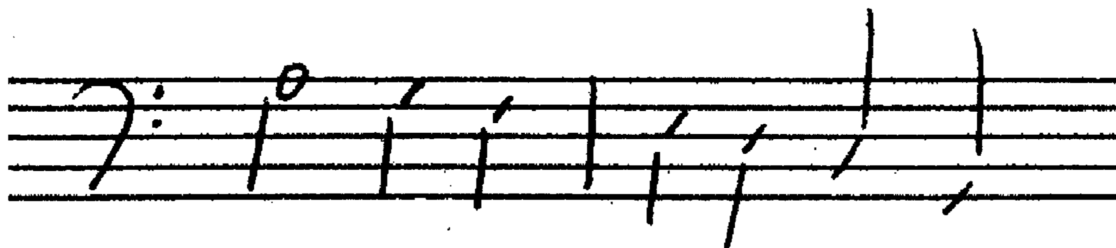


Figure 14.9: Another handwritten example.

Our simple staff line removal process can be confused by the thin lines (not much thicker than the staff lines) crossing the staff lines at an angle. This leads to gaps being formed in some of the lines representing note-heads. These were filled in manually before further processing. (We are not too concerned here about flaws in our low-level routines. Notice that the problem could be fixed in the parsing stage by a production that re-assembles collinear line fragments situated on either side of a staff line.) The staff line removal process also left a large number of small dots in the image. These, together with some small dots present in the input image, resulted in a total of 67 very small blobs being extracted from the image. The result of the staff line removal process (before the manual improvement) is shown in Figure 14.10.

The line thinning algorithm left a number of short "spikes" attached to the lines in

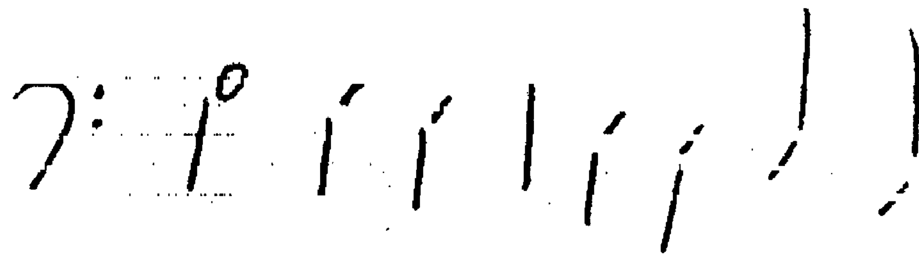


Figure 14.10: Problems caused by the staff line removal process.

the thinned image. A magnified example of this in the hollow note-head is shown in Figure 14.11. Because of the naïve manner in which the curve following algorithm chooses which branch to follow when it finds a fork in the curve, the curve representing the hollow note-head was split into a number of shorter curves. An improvement to the curve following algorithm to overcome this problem is suggested in section 11.2. Another approach to the problem would have been to write a production to re-assemble line segments that, when combined, form a uniform curve. However, we feel that this particular problem can be accurately dealt with at the primitive extraction layer, using the improvement suggested. Since the primitive extraction layer is not the focus of this thesis, we fixed this problem manually by editing the file containing the list of extracted primitives. The resulting primitives are given in Figure 14.12. The small blobs deleted by the first production have been omitted for the sake of brevity.

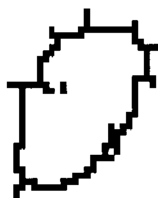


Figure 14.11: Spikes caused by line thinning (magnified hollow note-head).

```
Nodes:
line: 732 37  728 136  0.250000  -0.110657
line: 824 159  819 52  0.250000  0.134321
curve: 246 97 0 0.785714 1.069444 6.400000 6.200000
curve: 87 142 0 1.672414 9.767361 2.761086 4.254542
line: 357 99  338 116  0.458333  0.577902
line: 500 100  495 188  0.291667  -0.110657
line: 418 131  430 109  0.416667  -0.159913
line: 210 189  224 110  0.416667  0.244979
line: 576 123  567 140  0.416667  0.000000
line: 328 129  319 192  0.333333  -0.404892
line: 626 157  644 133  0.416667  0.000000
line: 401 204  406 136  0.291667  0.110657
line: 723 147  701 179  0.500000  -0.197396
line: 552 157  548 218  0.291667  -0.221314
line: 625 169  627 172  0.083333  0.000000
line: 625 166  611 241  0.250000  -0.134321
line: 794 209  816 181  0.458333  0.183111
blob: 134 105  0.187500 1.333333 1.066667
blob: 132 132  0.111111 1.000000 1.166667
```

Figure 14.12: Primitives extracted from the image.

The grammar given in appendix A.2 does not include productions for the types of note-heads found in this example. We thus wrote two additional productions: one to map suitable complex curves into hollow note-heads and one to map short, slanted lines into solid note-heads. This is very easy to do, and demonstrates the flexibility and extensibility of a system that uses our framework. The two production are shown in Figure 14.13.The `Trap` function used in the applicability predicates is the standard trapezoidal fuzzy certainty function, shown in figure 14.14. The function is parameterized, as shown in Figure 14.14.

No extra work is needed to deal with the small blobs left after the staff line removal process, since the first production removes them before the rest of the graph is parsed, as in the second example.

Although we have written a production to recognise bar lines, we have not written productions to split the graph into bars for the purpose of checking the number of beats. Recall that in Chapter 7 we said that, although we segment the image before parsing when we are absolutely certain about the identity of a bar line, the grammar must be able to perform this segmentation into bars when the bar line is not perfect. Since we have not yet written the required productions, we manually split the file containing the list of primitives into two files: the first containing all primitives to the left of (and including) the line representing the bar line, and the second containing all primitives to the right of the bar line.

The line primitive representing the bar line is an example of local ambiguity, since it can be mapped onto either a stem or a bar line. Although there is a high certainty of it being a stem, there is no note-head nearby which could combine with the stem to form a note. This branch of the search tree thus gets a very low certainty value, and is effectively killed. The branch in which the line is interpreted as a bar line thus has a much higher certainty value. Previous attempts to allow graph rewriting systems to cope with noise cannot solve this problem [Fahmy, 1995; Fahmy & Blostein, 1996]. The search tree associated with the parse of the first bar is given in appendix E.3.

```
Production 30
 # map a line to a solid notehead

GL:
  Nodes = {(1,line)}
  Edges = {}

GR:
  Nodes = {(1,notehead)}
  Edges = {}

Embed = {(1,1)}

Attrib:
  {_1.x = (MidPoint (1))->x;
   _1.y = (MidPoint (1))->y;
   _1.solid = 1.0;}

Apply = {NoteHeadLineSlope(1) && NoteHeadLineLength(1)}

#-----------------

Production 31
 # map a complex curve into a hollow note-head

GL:
  Nodes = {(1,curve)}
  Edges = {}

GR:
  Nodes = {(1,notehead)}
  Edges = {}

Embed = {(1,1)}

Attrib:
  {_1.x = (Centre (1))->x;
   _1.y = (Centre (1))->y;
   _1.solid = 0.0;}

#  Trap(x , x1, x2,y2, x3, x4)
Apply = {Trap(Size(1), 0.3, 0.6,1.0, 1.0, 1.4) &&
 Trap(_1.abscurve, 5.0, 5.9,1.0,6.5, 7.0) }
```

Figure 14.13: Two productions for recognising hand-drawn note-heads.

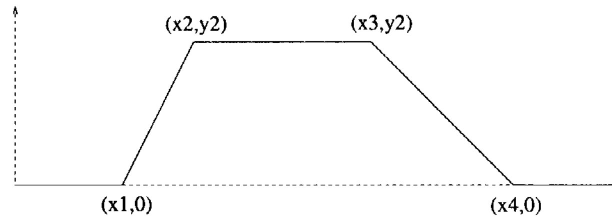


Figure 14.14: The Trap function.

(The part of the search tree corresponding to the deletion of the small blobs is omitted for the sake of simplicity.)

The system correctly recognised all symbols in the image. In both bars the correct interpretation, which had the highest certainty value, was the first interpretation found. The only error made was in determining the pitch of the second last note. The system interpreted it as being on the fourth staff line, while we intended it to be in the space above the line.

The system was further tested on a limited number of images containing the restricted music notation defined in the grammar presented in appendix A.2 and they were all correctly recognised. None of the examples were large enough to get significant execution time information, as the processing time was too short.

Chapter 15

Concluding remarks

We conclude with a brief summary of the more important aspects of this thesis, and present the advantages and limitations of the framework and prototype system developed. In the course of this thesis, we have mentioned a number of aspects that would benefit from future work. We review these here.

15.1 What was achieved

The work described in this thesis began as an attempt to recognise handwritten music scores. A syntactic approach appealed to us, as it can be used to describe the structure inherent in music scores in a natural manner. Since music scores are two-dimensional, graph grammars are particularly well suited to describing them, and because music symbols have attributes associated with them (e.g., pitch and duration), the symbols used by the graph grammar must be attributed. (Failure to do so would result in an explosion of the number of symbol types required.) Efficiency is a problem in most large parsing applications. This is particularly the case with graph grammars, as the

application of each production relies on finding an isomorphic subgraph, which is, in general, an NP-complete problem. One way to improve the efficiency and generative power of a grammar is to control, to some extent, the order in which productions can be applied. This order is specified by a control diagram. If the order in which the productions in a grammar can be applied is controlled in this way, we say that the grammar is programmed.

Graph grammars have two main shortcomings: even when programmed they can be inefficient, and they cannot cope well with noise in the input graph. We did not concern ourselves too much with efficiency in this work. Rather, our aim was to allow graph grammars to cope better with noise.

Consider once more the applicability predicate associated with the production given in Figure 2.3. We chose a natural approach to allowing graph grammars to cope with noise by replacing this binary function with a fuzzy (continuous) certainty function. This transformation is the core of the work described in this thesis.

We have defined a new class of programmed attributed graph grammars, known as programmed fuzzy attributed graph grammars. The productions in this class of grammar have a fuzzy certainty function instead of the usual binary applicability predicate. We have developed a parsing mechanism for the programmed fuzzy attributed graph grammars.

The main change in the parsing algorithm is needed to allow the programming mechanism to deal with certainty values between 0 and 1. As far as we are aware, this represents the first application of programming to a fuzzy grammar. The resulting mechanism maintains the advantages of crisp programming, namely increased efficiency of parsing and increased expressiveness of the grammar (the increase in efficiency will, however, be lower than in the crisp case). At the same time, it can cope better with noise. The fuzzy certainty values associated with production instances allow competing interpretations of symbols in the image to propagate to a higher level in the parsing, where non-local information can be used to make a better decision.

The parser uses the certainty values associated with production instances to compute a certainty value for each possible parse path. This allows parse paths (interpretations of the image) to be ordered if more than one valid interpretation is possible. The parsing engine also uses the certainties associated with parse paths to direct its search through state space (using either a uniform or a best-first search). The efficiency of the parsing process has been analysed, and a method for automatically pruning the state space search tree has been built into the parser.

We introduced a particular class of productions, called noise productions, that allow primitives that do not map onto higher-level symbols to be interpreted as noise and deleted. We derived a method for calculating the certainty values associated with these noise productions. In the case where a noise production deletes a primitive element, its certainty value depends on the certainties associated with the competing productions (see Definition 5.3). In the case where a noise production deletes a class two symbol, it is given a fixed, low certainty value.

The parser has been fully implemented, including the search tree pruning, various search strategies and support for the noise productions mentioned above. The implementation allows the user to easily define the grammar and control diagram in text files. For the sake of testing, it also allows the user to visualize the state space search tree associated with a parse.

In order to show how the framework can be applied to a real problem, we have developed and implemented a prototype system for recognising music scores. Although the low-level aspects of this system are limited, they are sufficient to demonstrate the use of the framework. Preliminary results obtained from this prototype have indicated that the framework developed is promising for dealing with noisy images.

15.2 Future work

The most important thing left to do is to test the framework more extensively, using, as a starting point, the system described in the second part of this thesis. Of course, a more complete set of productions will have to be developed, to allow the system to cope with a wider range of symbols. Ideally, one should upgrade the prototype described in the second part of this thesis into a full system. This would make it easier to test the system extensively and would, at the same time, test the scalability of the system.

An obvious future activity would be to develop systems that recognise other image classes. Although we have tried to consider general image classes in developing our framework, the work originated from a desire to recognise handwritten music scores. The resulting framework may thus be expected to be biased towards this application. Using the framework to develop recognisers for other classes of images will expose the extent of such a bias.

In the course of this thesis we have mentioned a number of specific areas that would benefit from further work. We summarize them here, beginning with the areas relating to the general framework, and then moving on to those pertaining to the specific application. Of course, there has been interaction between our development of the framework and of the application, so the distinction is not always clear-cut.

As was mentioned in Chapter 6, we have used the standard (simple) fuzzy operators and fuzzy aggregation function. The system's performance should benefit from experimentation with some of the many alternative fuzzy operators and aggregation functions.

The possibility of using deformable graph templates instead of normal graphs for the left-hand side of productions (section 6.2) is also a candidate for further exploration.

In section 7.2 we mentioned the use of heuristics to improve the uniform search

strategy and thus to reduce the search time. These have shown promise in initial tests and deserve further investigation. The possible heuristic mentioned, which favours nodes in the search tree that are closer to the final node in the control diagram, should apply to any system using this framework. Heuristics are often suited to a specific application domain, and the framework that has been implemented allows a user to specify a heuristic very easily.

The primitives used in the prototype for recognising music scores, in particular the line and curve primitives, are not necessarily optimal. More work can be done in investigating alternative primitives and attributes. The current routines for following curves are not robust, and should be improved.

Finally, we have not investigated the possibility of incorporating learning into our framework. This is a large area that would benefit from future work. K-P Chan has described a method for learning deformable graph templates from fuzzy examples [Chan, 1996], and the system developed by Malaviya for online handwriting recognition learns its rule-base from examples [Malaviya, 1996]. Such methods will probably best be used to allow a system to improve an initial grammar (especially the certainty functions associated with the productions), as it learns from more examples.

15.3 In closing

The set of two-dimensional images with meaningful structure is a large one, and contains many image classes which would benefit from automatic recognition by computer. Graph rewriting systems are an important method used for the recognition of these images. A limitation of this method is that it does not cope well with noise. This thesis addresses this limitation by developing a new type of graph grammars, called programmed fuzzy attributed graph grammars. The resulting framework for image recognition using this class of grammars offers a contribution towards coping with noise in two-dimensional, structured images.

References

Bartini, C. 1979. Rewriting systems as a tool for relational database design. *Lecture Notes in Computer Science*, **73**, 139–154.

Blostein, D., Fahmy, H., & Grbavec, A. 1995. *Practical Use of Graph Rewriting.* Tech. rept. 95-373. Queen's University, Kingston, Ontario.

Bulis, A., Almog, R., Gerner, M., & Shimony, U. 1992. Computerized recognition of hand-written musical notes. *Pages 110–112 of: Proceedings of the International Computer Music Conference.* San Jose.

Bunke, H. 1982a. Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **4**(Nov.), 574–582.

Bunke, H. 1982b. On the Generative Power of Sequential and Parallel Programmed Graph Grammars. *Computing*, **29**(Apr.), 89–112.

Carter, N.P., & Bacon, R.A. 1991. *Automatic Recognition of Printed Music.* Dept. of Physics, University of Surrey, GB. Preprint of an article.

Chan, K-P. 1996. Learning Templates from Fuzzy Examples in Stuctural Pattern Recognition. *IEEE Transactions on Systems, Man and Cybernetics*, **26**(1), 118–123.

Clarke, A.T., Brown, B.M., & Thorne, M.P. 1988. Using a Micro to Automate Data Acquisition in Music Publishing. *Microprocessing and Microprogramming*, **24**, 549–554.

Clarke, A.T., Brown, B.M., & Thorne, M.P. 1989. Coping with some really rotten problems in Automatic Music Recognition. *Microprocessing and Microprogramming*, **27**, 547–550.

Couasnon, B., & Camillerapp, J. 1994 (Oct.). Using grammars to segment and recognize music scores. *Pages 15–27 of: International Association for Pattern Recognition Workshop on Document Analysis Systems.* Kaiserslautern.

Denert, E., Franck, R., & Streng, W. 1975. PLAN2D - Toward a twodimensional programming language. *Lecture Notes in Computer Science*, **26**, 202–213.

Dolado, J., & Torrealdea, F. 1988. Formal Manipulation of Forrester Diagrams by Graph Grammars. *IEEE transactions on Systems, Man and Cybernetics*, **18.6**(November), 981–996.

Ehrig, H., & Kreowski, H. 1980. Applications of Graph Grammar Theory to Consistency, Synchronization, and Scheduling in Data Base Systems. *Information Systems*, 225–238.

Fahmy, H. 1995. *Reasoning in the Presence of Uncertainty via Graph Rewriting.* Ph.D. thesis, Department of Computing and Information Science, Queen's University.

Fahmy, H., & Blostein, D. 1991. A Graph Grammar for High-level Recognition of Music Notation. *Pages 70–78 of: Proceedings of First International Conference on Document Analysis*, vol. 1. Saint-Malo, France.

Fahmy, H., & Blostein, D. 1992a. Graph Grammar Processing of Uncertain Data. *Pages 373–382 of:* Bunke, H. (ed), *Advances in Structural and Syntactic Pattern Recognition (Proceedings of International Workshop on Structural and Syntactic Pattern Recognition, Bern, CH).* Series in Machine Perception and Artificial Intelligence, vol. 5. World Scientific.

Fahmy, H., & Blostein, D. 1992b (Sept.). A Survey of Graph Grammars: Theory and Applications. *Pages 294–298 of: Proceedings of 11th IAPR International Conference on Pattern Recognition*, vol. 2.

Fahmy, H., & Blostein, D. 1996 (July). *A Graph-rewriting Paradigm for Discrete Relaxation: Application to Sheet-music Recognition.* Submitted to the International Journal of Pattern Recognition and Artificial Intelligence.

Fu, K.S., Rosenfeld, A., & Wolf, J.J. 1980. Recent Developments in Digital Pattern Recognition. *In:* Fu, K.S. (ed), *Digital Pattern Recognition*, second edn. Springer-Verlag.

Fujinaga, I., Pennycook, B., & Alphonce, B. 1989a. Computer recognition of musical notation. *Pages 87–90 of: Proceedings of the First International Conference on Music Perception and Cognition.*

Fujinaga, I., Alphonce, B., & Pennycook, B. 1989b (Nov.). Issues in the Design of an Optical Music Recognition System. *Pages 113–116 of: Proceedings of the International Computer Music Conference.* Ohio State University.

Fujinaga, I., Alphonce, B., Pennycook, B., & Hogan, K. 1991. Optical music recognition: Progress report. *Pages 66–73 of: Proceedings of the International Computer Music Conference.* Montreal.

Fujinaga, I., Alphonce, B., Pennycook, B., & Diener, G. 1992. Interactive optical music recognition. *Pages 117–120 of: Proceedings of the International Computer Music Conference.* San Jose.

Furtado, A.L. 1979. Transformation of database structures. *Lecture Notes in Computer Science*, **73**, 224–236.

Golin, E.J. 1991. *A Method for the Specification and Parsing of Visual Languages.* Ph.D. thesis, Brown University.

Göttler, H. 1979. Semantical description by two-level graph-grammars for quasihierarchical graphs. *Applied Computer Science*, **13**, 207–226.

Göttler, H., Günther, J., & Nieskens, G. 1991. Use Graph Grammars to Design CAD-Systems! *Pages 396–410 of:* Ehrig, H., Kreowski, H., & Rozenberg, G.

(eds), *Proceedings of the Fourth International Workshop on Graph Grammars and their Application to Computer Science (Lecture Notes in Computer Science, Vol. 532)*. Springer Verlag.

Katayose, H., Kato, H., Imai, M., & Inokuchi, S. 1989 (Nov.). An Approach to an Artificial Music Expert. *Pages 139–146 of: Proceedings of the International Computer Music Conference*. Ohio State University.

Kato, H., & Inokuchi, S. 1992. A recognition system for printed piano music using musical knowledge and constraints. *Pages 435–455 of:* Baird, H.S., Bunke, H., & Yamamoto, K. (eds), *Structured Document Image Analysis*. Springer-Verlag.

Kim, M.H., Lee, J.H., & Lee, Y.J. 1993. Analysis of fuzzy operators for high quality information retrieval. *Information Processing Letters*, **46**, 251–256.

Leroy, A., Muller, G., & Garnett, G.E. 1994. The Design of a Pen-Based Music Notation System. *Pages 286–292 of: Proceedings of the International Computer Music Conference*. Aarhus, Denmark.

Lindstrom, B. 1994. Musitek MIDISCAN. *Electronic Musician*, Feb., 151–154.

Malaviya, A. 1996. *On-line Handwriting Recognition with a Fuzzy Feature Description Language*. Ph.D. thesis, Technische Universität Berlin.

Malaviya, A., Leja, C., & Peters, L. 1996. Multi-Script Handwriting Recognition with FOHDEL. *Pages 147–151 of: Proceeding of the Biennial Conference of the North American Fuzzy Information Processing Society*. Berkeley.

Martin, P. 1989. Reconnaissance de Partitions Musicales et Réseaux de Neurones: une Etude. *Pages 217–226 of: Actes 7ème Congrès AFCET de Reconnaissance des Formes et Intelligence Artificielle*. Paris. (french).

Martin, P., & Bellissant, C. 1991. Low-Level Analysis of Music Drawing Images. *Pages 417–425 of: Proceedings of First International Conference on Document Analysis*, vol. 1. Saint-Malo, France.

Matsushima, T. 1985. Automated High Speed Recognition of Printed Music (WABOT-2 Vision System). *Page 477 ff. of: Proceedings of the 1985 International Conference on Advances Robotics.* Japan Industrial Robot Association (JIRA), Shiba Koen Minato-ku, Tokyo.

Matsushima, T., Ohteru, S., & Hashimoto, S. 1989 (Nov.). An integrated Music Information Processing System: PSB-er. *Pages 191–198 of: Proceedings of the International Computer Music Conference.* Ohio State University.

Nagl, M. 1979. A Tutorial and Bibliographical Survey on Graph Grammars. *Lecture Notes in Computer Science*, **73**, 70–126.

Nagl, M. 1987. Set Theoretic Approaches to Graph Grammars. *Lecture Notes in Computer Science*, **291**, 41–54.

Pal, S.K. 1992. Fuzziness, Image Informtion and Scene Analysis. *Pages 147–183 of:* Yager, Ronald R., & Zadeh, Lotfi A. (eds), *An Introduction to Fuzzy logic Applications and Intelligent Systems.* Kluwer Academic Publishers.

Pal, S.K., & Dutta Majumder, D.K. 1986. *Fuzzy Mathematical Approach to Pattern Recognition.* Wiley Eastern Limited.

Panel. 1991. Panel Discussion: The Use of Graph Grammars in Applications. *Pages 41–60 of:* Ehrig, H., Kreowski, H., & Rozenberg, G. (eds), *Proceedings of the Fourth International Workshop on Graph Grammars and their Application to Computer Science (Lecture Notes in Computer Science, Vol. 532).* Springer Verlag.

Pennycook, B. 1990. Towards Advanced Optical Music Recognition. *Advance Imaging*, Apr., 54–57.

Peyton Jones, S. 1987. *The Implementation of Functional Programming Languages.* Prentice Hall.

Pfaltz, J.L., & Rosenfeld, A. 1969. Web grammars. *Pages 609–619 of: Proceedings of the 1st Joint Conference in Artificial Intelligence.* Washington.

Prerau, D. S. 1975. Do-Re-Mi: A Program that Recognizes Music Notation. *Computers and the Humanities*, **9**(1), 25–29.

Pruslin, D. 1966 (June). *Automatic Recognition of Sheet Music.* ScD dissertation, MIT.

Roads, C. 1986. The Tsukuba Musical Robot. *Computer Music Journal*, **10**(2), 39–43.

Sonka, M., Hlavac, V., & Boyle, R. 1993. *Image processing, Analysis and Machine Vision.* Chapman & Hall computing.

Stallings, W. 1977. Fuzzy set Theory Versus Bayesian Statistics. *Transactions on Systems, Man and Cybernetics*, **7**, 216–219.

Watkins, G.S. 1994. A Fuzzy Syntactic Approach to Optical Music Recognition. *Pages 297–302 of: Proceedings of the International Computer Music Conference.* Aarhus, Denmark.

Watkins, G.S. 1996. The use of fuzzy graph grammars for recognising noisy two-dimensional images. *Pages 415–419 of: Proceedings of the 1996 Biennial Conference of the North American Fuzzy Information Processing Society - NAFIPS.* Berkeley, California: IEEE.

Wolman, A., & Yaeger, T. 1994. Optical Music Recognition - progress report. *Pages 293–296 of: Proceedings of the International Computer Music Conference.* Aarhus, Denmark.

Yadid-Pecht, O., Brutman, E., Dvir, L., Gerner, M., & Shimony, U. 1992. RAMIT: Neural network for recognition of musical notes. *Pages 128–131 of: Proceedings of the International Computer Music Conference.* San Jose.

Zündorf, A., & Schürr, A. 1991. Nondeterministic Control Structures for Graph Rewriting Systems. *In: 17th International Workshop on Graph-theoretic Concepts in Computer Science.*

Appendices

Appendix A

Defining the grammar

A.1 The grammar definition language

A.1.1 Lex rules

```
nl "\n"
delim [ \t]
ws {delim}+
digit [0-9]
n {digit}+

%{
buffers attribs, applys;
%}

%%
{nl} { return(NEWLINE); }
{ws} { /* Do nothing */ }
{n} { yylval.i = atoi(yytext); return(N); }
"," { return(','); }
":" { return(':'); }
"(" { return('('); }
")" { return(')'); }
"{" { return('{'); }
"}" { return('}'); }
";" { return(';'); }
"=" { return('='); }
"#" { readtoeol(); return(COMMENT); }
```

```
Production { return(PRODUCTION); }
blob  { return(BLOB); }
line  { return(LINE); }
curve { return(CURVE); }

note { return(NOTE); }
notehead {return(NOTEHEAD); }
stem    { return(STEM); }
start { return(START); }
barline { return(BARLINE); }
clef    { return(CLEF); }
slur    { return(SLUR); }

bar { return(BAR); }
restofbar {return(RESTOFBAR); }
symbol { return(SYMBOL); }
notesstart {return(NOTESSTART); }
notesend { return(NOTESEND); }

GL { return(GL); }
GR { return(GR); }
Embed { return(EMBED); }
Attrib { readtocc (attribs[i1]); return(ATTRIB); }
Apply { readtocc (applys[i1]); return(APPLY); }
Nodes { return(NODES); }
Edges { return(EDGES); }
```

A.1.2 YACC rules

For the sake of simplicity, the C code assignments to set up the data structures have been omitted in this listing.

```
%token <i> N
%token NEWLINE PRODUCTION
%token GL GR EMBED ATTRIB APPLY COMMENT
%token NODES EDGES
%token BLOB LINE CURVE NOTE NOTEHEAD STEM START
%token BARLINE CLEF SLUR
%token BAR RESTOFBAR
%token SYMBOL NOTESSTART NOTESEND
```

```
%{

void readtocc (buffer_t buffer)
/* copies everything from { to } into buffer */
{ ...
}


void readtoeol ()
/* reads + ignores everything up to end of line */
{ ...
}

%}


%start file

%%
file : line NEWLINE
| file line NEWLINE
| error NEWLINE { yyerrok; };

line : heading
| gl
| gr
| embed
| ATTRIB
| APPLY
| COMMENT
| {/* blank line */};


heading : PRODUCTION N;

gl : GL ':' NEWLINE lsubgraph;
lsubgraph : lnodes NEWLINE edges;

lnodes : NODES '=' '{' nodelist '}';
nodelist : node
  | nodelist ',' node;
node : '(' N ',' nodelabel ')';

nodelabel : BLOB
   | LINE
           | CURVE
```

```
           | NOTE
           | NOTEHEAD
           | STEM
           | START
           | BARLINE
           | CLEF
           | SLUR
           | BAR
           | RESTOFBAR
           | SYMBOL
           | NOTESSTART
           | NOTESEND;

edges : EDGES '=' '{' edgelist '}';

edgelist : '(' N ',' N ')'
   | edgelist ',' '(' N ',' N ')'
   | {/* no edges */};


gr : GR ':' NEWLINE rsubgraph;
rsubgraph : rnodes NEWLINE edges;

rnodes : NODES '=' '{' rnodelist '}';
rnodelist : nodelist
     | {/* blank */};


embed : EMBED '=' '{' embedg '}'

embedg : '(' N ',' N ')'
| embedg ',' '(' N ',' N ')'
| { /* blank */ };


%%
```

A.2 Minimal grammar to do a full parse, up to the bar level

```
Production 1

# remove blobs that are too small
# just to filter out a lot of extra noise

GL:
  Nodes = {(1,blob)}
  Edges = {}

GR:
  Nodes = {}
  Edges = {}

Embed = {}

Attrib:
    {}

Apply = {_1.size < 0.07}


#------------------------

Production 2

# Map a line to a barline

GL:
  Nodes = {(1,line)}
  Edges = {}

GR:
  Nodes = {(1,barline)}
  Edges = {}

Embed = {(1,1)}

Attrib:
  {_1.x =(_1.x1 +_1.x2)/2;}
```

```
Apply = {Straight (1) && Vertical (1) &&
        Close (Very, Top(1), StavePoint (1, Top(1))) &&
Close (Very, Bot(1), StavePoint (9, Bot(1)))}

#--------------------------

Production 3

# map a blob to a notehead

GL:
  Nodes = {(1,blob)}
  Edges = {}

GR:
  Nodes = {(1,notehead)}
  Edges = {}

Embed = {(1,1)}

Attrib:
  {_1.x = _1.x;
   _1.y = _1.y;
   _1.solid =_1.solid;}

# _1.all = _1.all   would be a nice feature to have

Apply = {Trap(Size(1), 0.3, 0.6,1.0, 1.0, 1.4)}


#   ------------------------------

Production 4

# map line to stem

GL:
  Nodes = {(1,line)}
  Edges = {}

GR:
  Nodes = {(1,stem)}
  Edges = {}

Embed = {(1,1)}
```

```
Attrib:
  {_1.x1 =_1.x1;
   _1.y1 =_1.y1;
   _1.x2 =_1.x2;
   _1.y2 =_1.y2;
   _1.branches =0;}

Apply = {(Vertical (1) && FairlyStraight (1) && StemLength (1))}


#------------------------

Production 5

# map line and broken notehead onto
# stem and hollow notehead

GL:
  Nodes = {(1,line), (2,line)}
  Edges = {}

GR:
  Nodes = {(1,stem), (2,notehead)}
  Edges = {}

Embed = {(1,1), (2,2)}

Attrib:
  {/* stem */
   _1.x1 =_1.x1;
   _1.y1 =_1.y1;
   _1.x2 =_1.x2;
   _1.y2 =_1.y2;
   /* notehead */
   _2.x =MidPoint(2)->x;
   _2.y =MidPoint(2)->y;
   _2.solid =0.0;}

Apply = {Match (Length(1)/4.0, 1.0) &&
   Match (Length(2), 1.0) &&
   Close (Very, Top(1), Top(2))/* &&
   Straight (1)*/}


#------------------------
```

```
Production 6

# map a line onto a slur

GL:
  Nodes = {(1,line)}
  Edges = {}

GR:
  Nodes = {(1,slur)}
  Edges = {}

Embed = {(1,1)}

Attrib:
  {_1.x1 =_1.x1;
   _1.y1 =_1.y1;
   _1.x2 =_1.x2;
   _1.y2 =_1.y2;
  }

Apply = {Horizontal(1) && Match (_1.curve/1.2, 1)}


#-------------------------

Production 7

# form edges between a slur
# and the corresponding notes

GL:
  Nodes = {(1,slur), (2,notehead), (3,notehead)}
  Edges = {}

GR:
  Nodes = {(1,slur), (2,notehead), (3,notehead)}
  Edges = {(1,2), (1,3)}

Embed = {(1,1), (2,2), (3,3)}

Attrib:
  {
/* just copy accross all the attributes
  this is where .all would be really useful
 slur   (1) */
```

```
   _1.x1 =_1.x1;
   _1.y1 =_1.y1;
   _1.x2 =_1.x2;
   _1.y2 =_1.y2;
/* notehead (2) */
   _2.x =_2.x;
   _2.y =_2.y;
   _2.solid =_2.solid;
/* notehead (3) */
   _3.x =_3.x;
   _3.y =_3.y;
   _3.solid =_3.solid;}

Apply = {(Close (Fairly, Top(2), Left(1)) ||
  Close (Fairly, Bot(2), Left(1))) &&
         (Close (Fairly, Top(3), Right(1))||
  Close (Fairly, Bot(3), Right(1))) &&
          !Edge(1,2) && !Edge(1,3)}


#--------------------------------

Production 8

# form note from hollow notehead and stem

GL:
  Nodes = {(1,notehead), (2,stem)}
  Edges = {}

GR:
  Nodes = {(1,note)}
  Edges = {}

Embed = {(1,1), (2,1)}

 Attrib:
  {_1.pitch =GetPitch (Centre(1));
   _1.duration = 32;
   _1.x =_1.x;}


Apply = {(Close (Normal, Top(2), Left(1)) ||
          Close (Normal, Bot(2), Right(1))) &&
          1-_1.solid}
```

```
#------------------------

Production 9

# form note from solid notehead and stem

GL:
  Nodes = {(1,notehead), (2,stem)}
  Edges = {}

GR:
  Nodes = {(1,note)}
  Edges = {}

Embed = {(1,1), (2,1)}

Attrib:
  {_1.pitch =GetPitch (Centre(1));
   _1.duration = 16;
   _1.x =_1.x;}


Apply = {(Close (Normal, Top(2), Left(1)) ||
          Close (Normal, Bot(2), Right(1))) &&
          _1.solid}

#------------------------

Production 10

# map a complex curve onto a bass clef
# must still add stuff to deal with the 2 dots

GL:
  Nodes = {(1,curve), (2,blob), (3,blob)}
  Edges = {}

GR:
  Nodes = {(1,clef)}
  Edges = {}

Embed = {(1,1), (2,1), (3,1)}

Attrib:
  {_1.x =Centre(1)->x;
   _1.cleftype =bass;}
```

```
Apply = {Match (_1.size/6.8, 1) && Match (_1.aspect/1.9, 1) &&
         Match (_1.curve/2.8, 1) &&
         Close (Fairly, Centre(2), StavePoint (2,Right(1))) &&
         Close (Very, Centre(2), StavePoint (2,Centre(2))) &&
         Close (Fairly, Centre(3), StavePoint (4,Right(1))) &&
         Close (Very, Centre(3), StavePoint (4,Centre(3)))}


#------------------------

Production 11

# get rid of blobs
# ("noise" production)

GL:
  Nodes = {(1,blob)}
  Edges = {}

GR:
  Nodes = {}
  Edges = {}

Embed = {}

Attrib:
  {
  }

Apply = {NotOrCerts()}

#------------------------

Production 12

# get rid of lines
# ("noise" production)

GL:
  Nodes = {(1,line)}
  Edges = {}

GR:
  Nodes = {}
  Edges = {}
```

```
Embed = {}

Attrib:
  {
  }

Apply = {NotOrCerts()}

#-----------------------

Production 13

# get rid of noteheads that weren't
# converted into notes
# ("noise" production)

GL:
  Nodes = {(1,notehead)}
  Edges = {}

GR:
  Nodes = {}
  Edges = {}

Embed = {}

Attrib:
  {
  }

Apply = {0.001}

#-----------------------

Production 14

# insert start node into graph
# form a link from it to the clef
# at the beginning of the line
# if the 1st symbol isn't a clef, need
# extra productions

GL:
  Nodes = {(1,clef)}
  Edges = {}
```

```
GR:
  Nodes = {(1,start), (2,clef)}
  Edges = {(1,2)}

Embed = {(1,2)}

Attrib:
  {_2.cleftype =_1.cleftype;
   _2.x =_1.x;}

Apply = {Leftmost (1)}

#------------------------

Production 15

# insert start node into graph
# form a link from it to the first note
# at the beginning of the line
# if the 1st symbol isn't a clef,or a note need
# extra productions

GL:
  Nodes = {(1,note)}
  Edges = {}

GR:
  Nodes = {(1,start), (2,note)}
  Edges = {(1,2)}

Embed = {(1,2)}

Attrib:
  {_2.pitch =_1.pitch;
   _2.duration =_1.duration;
   _2.x =_1.x;}

Apply = {Leftmost (1)}

#------------------------

Production 16

# form an edge between a clef
# and the following note
```

```
GL:
  Nodes = {(1,clef), (2,note)}
  Edges = {}

GR:
  Nodes = {(1,clef), (2,note)}
  Edges = {(1,2)}

Embed = {(1,1), (2,2)}

Attrib:
  {/* clef */
   _1.cleftype =_1.cleftype;
   _1.x =_1.x;

   /* note */
   _2.pitch =_2.pitch;
   _2.duration =_2.duration;
   _2.x =_2.x;}

Apply = {Dleftof (1,2) && !Edge(1,2)}


#------------------------

Production 17

# form an edge between two adjacent notes


GL:
  Nodes = {(1,note), (2,note)}
  Edges = {}

GR:
  Nodes = {(1,note), (2,note)}
  Edges = {(1,2)}

Embed = {(1,1), (2,2)}

Attrib:
  {/* note 1 */
   _1.pitch =_1.pitch;
   _1.duration =_1.duration;
   _1.x =_1.x;
```

```
   /* note 2*/
   _2.pitch =_2.pitch;
   _2.duration =_2.duration;
   _2.x =_2.x;}

Apply = {Dleftof (1,2) && !Edge (1,2)}


#-------------------------

Production 18

# form an edge between a note and
# an immediately following barline


GL:
  Nodes = {(1,note), (2,barline)}
  Edges = {}

GR:
  Nodes = {(1,note), (2,barline)}
  Edges = {(1,2)}

Embed = {(1,1), (2,2)}

Attrib:
  {/* note */
   _1.pitch =_1.pitch;
   _1.duration =_1.duration;
   _1.x =_1.x;

   /* barline */
   _2.x =_2.x;}

Apply = {Dleftof (1,2) && !Edge (1,2)}


#------------------------

Production 19

# get rid of stems left over
# ("noise" production)
```

```
GL:
  Nodes = {(1,stem)}
  Edges = {}

GR:
  Nodes = {}
  Edges = {}

Embed = {}

Attrib:
  {
  }

Apply = {0.0001}

#------------------------

Production 20

# get rid of curves not used
# ("noise" production)

GL:
  Nodes = {(1,curve)}
  Edges = {}

GR:
  Nodes = {}
  Edges = {}

Embed = {}

Attrib:
  {
  }

Apply = {NotOrCerts()}


#--------------------
# Now for prods to do high-level part of the parsing


Production 21
```

```
# Everything reduces to a bar
# Check if the number of beats in the bar is correct

GL:
  Nodes = {(1,start), (2,restofbar)}
  Edges = {(1,2)}

GR:
  Nodes = {(1,bar)}
  Edges = {}

Embed = {}

Attrib:
  {_1.duration =_2.duration;
  }

Apply = {0.01+0.99*(Duration(2)==BEATSINBAR)}

#--------------

Production 22

GL:
  Nodes = {(1,symbol), (2,restofbar)}
  Edges = {(1,2)}

GR:
  Nodes = {(1,restofbar)}
  Edges = {}

Embed = {(1,1),(2,1)}

Attrib:
  {_1.duration =_2.duration +_1.duration;
  }

Apply = {TRUE}

#--------------

Production 23

GL:
  Nodes = {(1,symbol)}
  Edges = {}
```

```
GR:
  Nodes = {(1,restofbar)}
  Edges = {}

Embed = {(1,1)}

Attrib:
  {_1.duration=_1.duration;
  }

Apply = {Rightmost(1)}

#--------------

Production 24

GL:
  Nodes = {(1,notesstart)}
  Edges = {}

GR:
  Nodes = {(1,symbol)}
  Edges = {}

Embed = {(1,1)}

Attrib:
  {_1.x=_1.x;
   _1.duration=_1.duration;
  }

Apply = {TRUE}

#--------------

Production 25

GL:
  Nodes = {(1,notesend)}
  Edges = {}

GR:
  Nodes = {(1,symbol)}
  Edges = {}
```

```
Embed = {(1,1)}

Attrib:
  {
   _1.x=_1.x;
   _1.duration=_1.duration;
  }

Apply = {TRUE}

#--------------

Production 26

GL:
  Nodes = {(1,note)}
  Edges = {}

GR:
  Nodes = {(1,symbol)}
  Edges = {}

Embed = {(1,1)}

Attrib:
  {_1.x=_1.x;
   _1.duration =_1.duration;
  }

Apply = {TRUE}

#--------------

Production 27

GL:
  Nodes = {(1,clef)}
  Edges = {}

GR:
  Nodes = {(1,symbol)}
  Edges = {}

Embed = {(1,1)}

Attrib:
```

```
  {_1.x=_1.x;
   _1.duration =0;
  }

Apply = {TRUE}

#--------------

Production 28

GL:
  Nodes = {(1,barline)}
  Edges = {}

GR:
  Nodes = {(1,symbol)}
  Edges = {}

Embed = {(1,1)}

Attrib:
  {_1.x=_1.x;
   _1.duration =0;
  }

Apply = {TRUE}

#--------------

Production 29

GL:
  Nodes = {(1,slur),(2,note),(3,note)}
  Edges = {(1,2),(1,3)}

GR:
  Nodes = {(1,notesstart),(2,notesend)}
  Edges = {}

Embed = {(2,1),(3,2)}

Attrib:
  {_1.x=_2.x;
   _1.duration =_2.duration;
   _2.x=_3.x;
   _2.duration =_3.duration;
```

```
  }

Apply = {TRUE}
```

Appendix B

Defining the control diagram

B.1 The control diagram definition language

B.1.1 Lex rules

```
nl "\n"
delim [ \t]
ws {delim}+
digit [0-9]
n {digit}+

%%
{nl} { return(NEWLINE); }
{ws} { /* Do nothing */ }
{n} { yylval.i = atoi(yytext); return(N); }
"," { return(','); }
":" { return(':'); }
"(" { return('('); }
")" { return(')'); }
"#" { return(COMMENT); }
"I" { return (INIT); }
"F" { return (FINAL); }
"Y" { return ('Y'); }
"N" { return ('N'); }
"O" { return (OUTPUT); }
Nodes { return(NODES); }
Edges { return(EDGES); }
```

B.1.2 YACC rules

```
%union {
   int i;
}

%token <i> N
%token NEWLINE
%token NODES EDGES COMMENT
%token OUTPUT INIT FINAL

%{
  #include "ctrl.in.h"
  #include <stdio.h>
  #include <stdlib.h>
  #include <malloc.h>

  ctrlnode_t ctrl [MaxNodes];

  int curnode, thisnode, n1,n2;
  char thischoice;

void readtoeol2 ()
/* reads + ignores everything up to end of line */

{ char ch;
  while ((ch=getchar ())!='\n');
}

%}


%start file

%%
file : part
| file part /*NEWLINE */
| error NEWLINE { yyerrok; };

part : nodes
| edges
| COMMENT { readtoeol2(); }
| NEWLINE;
```

```
nodes: NODES ':' NEWLINE nodelist;

nodelist: node
| nodelist node
| NEWLINE {/* blank line */};

node:  NodeNo ':' plist NEWLINE;

NodeNo: N {curnode = $1;

plist:    N {ctrl[curnode].prods.data[ctrl[curnode].prods.n++] = $1;
}
| plist ',' N
{ctrl[curnode].prods.data[ctrl[curnode].prods.n++] = $3;
};


edges : EDGES ':' NEWLINE edgelist;

edgelist : '(' NodeNo1 ',' NodeNo2 ',' choice ')'
     {if (thischoice == 'Y')
ctrl[n1].yedges.data[ctrl[n1].yedges.n++] =n2;
      else
ctrl[n1].nedges.data[ctrl[n1].nedges.n++] =n2;
     }
   | edgelist ',' '(' NodeNo1 ',' NodeNo2 ',' choice ')'
     {if (thischoice == 'Y')
                ctrl[n1].yedges.data[ctrl[n1].yedges.n++] =n2;
              else
                ctrl[n1].nedges.data[ctrl[n1].nedges.n++] =n2;
     }
   | edgelist ',' NEWLINE edgelist;

NodeNo1 :  N {n1 = $1;}
| INIT {n1 = INITIALNODE;}
| FINAL {n1 = FINALNODE;}
| OUTPUT {n1 = OUTPUTNODE;};

NodeNo2 :  N    {n2 = $1;}
        | INIT   {n2 = INITIALNODE;}
        | FINAL   {n2 = FINALNODE;}
| OUTPUT {n2 =OUTPUTNODE;};


choice :  'Y' {thischoice = 'Y';}
```

```
| 'N' {thischoice = 'N';};


%%

#include "lctrl.in.C"


void yyerror (char *str)

{
  printf("\n*** Error while parsing control diagram: %s\n", str);
}
```

B.2 Control diagram for applying productions in appendix B

```
Nodes:
1 : 1
2 : 2,4,5,12,6
3 : 3,10,20,11
4 : 7
5 : 8,9,13,19
6 : 14
7 : 16
8 : 17
9 : 18
10 : 29
11 : 27
12 : 28
13 : 26
14 : 25
15 : 24
16 : 23
17 : 22
18 : 21
19 : 15

# Y-edge from each node to itself,
# N-edge from each node to next node
# except for nodes 6 and 16, since
# productions 14 and 23 must only be applied once

Edges:
(I,1,Y), (1,1,Y), (1,2,N), (2,2,Y), (2,3,N),
(3,3,Y), (3,4,N), (4,4,Y), (4,5,N), (5,5,Y), (5,6,N),
(6,7,Y), (6,19,N), (19,7,Y),
(7,7,Y), (7,8,N), (8,8,Y), (8,9,N), (9,9,Y), (9,0,N), (0,10,Y),
(10,10,Y), (10,11,N), (11,11,Y),(11,12,N), (12,12,Y),(12,13,N),
(13,13,Y),(13,14,N), (14,14,Y),(14,15,N),
(15,15,Y),(15,16,N), (16,17,Y), (17,17,Y),(17,18,N), (18,F,Y)
```

Appendix C

Internal representation of productions

```
typedef enum    /* node labels  (terminals and nonterminals) */
         { blob, line, curve,
           notehead, stem,
           note, start, barline,
           clef, slur,
           bar, restofbar,
           symbol, notesstart,notesend,
         } ptypes;

typedef struct     /* list of edges */
         { int n; /* number of edges */
           int data [smaxedges]; /* edges */
         } sedges;

typedef struct /* node in subgraph associated with a production */
         { ptypes ptype; /* node label */
           sedges edges; /* edges */
         } snode;

typedef struct /* subgraph associated with a production */
         { int n; /* number of nodes */
           snode data [smaxnodes]; /* nodes */
         } sgraph;

typedef char embedgraph[smaxnodes]; /* stores embedding transformation */

typedef el *sglist_t [smaxnodes]; /* temporary array used in some functions */
```

```
typedef struct  /* storage structure for a production */
         {sgraph *gl, *gr;         /* left- and right-hand subgraphs */
          void (*attrib)(list_t list,sglist_t sglist);
                       /* pointer to Attr transfer function */
          Fval (*apply)(list_t list);  /* pointer to Applic predicate */
          embedgraph embed; /* embedding transformation */
         } prod_t;

typedef struct  /* storage structure for a number of productions */
         {int n;
          prod_t data [maxprods];
         } prods_t;
```

Appendix D

Fuzzy parsing algorithm

Some function used by the algorithm

```
void MarkGraphNodes (list_t &NodesList, list_item spos, Fval cert)
/* Mark graph nodes that are in isomorphic subgraph NodesList
   <spos> is pos of state in TmpStatesList
   Modifies global var <UsedNodesList>
   Used for pruning branched in search tree resulting from distinct
   subgraphs */



void AppendStatesToQ (Q_DEF (state) &Q)
/* uses global vars <UsedNodes> and <TmpStatesList>
   destroys some of the data in global var <UsedNodes>
   deletes states that are distinct and
   appends the rest to Q
   postcondition : TmpStatesList is empty

Function operates as follows.
   1) deal with cases where we must calculate OR of all certainties of
      states using a particular node  (denoted by state with mu >=2)
   2) find the state with the highest certainty value
   3) find all states associated with subgraphs that are directly or
      indirectly conjunct with that state's associated subgraph
   4) delete states not marked in above step from TmpStatesList
   5) Now append all states in list onto Q
    Calculate certainty and priority of each state as follows:
    heuristic =0.0;     /* alternative = tmpstate.node/38.0; */
/* (19 nodes in ctrl diag) */
    priority =1.0 - (tmpstate.cert +heuristic);
```

```
    if (tmpstate.cert >0.0)
      Q_INS(tmpstate, priority);  /* Insert into queue */
  }
}
```

```
graph_t *CopyGraph (graph_t &graph);
/*  make a complete copy of the graph
    and returns a pointer to the new copy */
```

```
<state>.set (int node, graph_t graph, Fval mu, long st_node, graph_t *output)
/* Set the components of a state variable
node  : position in control diagram
graph : current graph
mu : certainty value of parse path to state
st_node : number of node in search tree (used for search tree output
output  : output graph associated with parse path */
```

```
<list_of_states>.append (state S)
/* Add state S to list of states */
```

```
WRITE_ST (long st_node, long st_parent, int prod, Fval mu,
                double totcert)
/* Used purely to output state in state space search tree
   Search tree output in the form of a daVinci graph   */
```

The parsing algorithm

```
int main (int argc, char *argv [])

{
  /* get file names gramfname and graphname  (deleted for simplicity) */

  ReadIn (graphfname);  /* read in graph */

  ReadStave (stavefname, &stave);  /* read in stave data */
  ReadProds (gramfname);                /* parse productions */
```

```
assignfuncs (producs);                /* assign pointers to functions */
/* Productions should now be set up + ready to be applied */
CtrlIn (ctrl_fname);                  /* read in control diagram */

SetupUsedNodes (graph);

node =ctrl[INITIALNODE].yedges.data[0];  /* follow edge from node I */
p=0;
prod =ctrl[node].prods.data[p];       /* use 1st production in node */
                 /* (we will use every production in the node, in order) */
NoSuccess =1;
cont =FALSE;
while (node <MaxNodes || !Q.empty())
  {
   prodno =prod -1;
   findsubg (producs->data[prodno].gl, &graph, NodesList, cont, &success);
              /* find isomorphic subgraph */
              /* if cont is FALSE, start searching from the beginning of */
              /* the graph, else continue from where we stopped last time */
   if (success)
     {mu = (*(producs->data[prodno].apply)) (NodesList);
      if (mu>0.0)      /* if certainty function has value >0 */
        {                      /* apply production */
         NoSuccess =0;
         sgtosglist (producs->data[prodno].gr, &sg1, sglist);
         (*(producs->data[prodno].attrib)) (NodesList, sglist);
         newgraph =CopyGraph (graph);
         NewList (NodesList, graph, *newgraph, newlist);
         replace (newgraph, newlist, producs->data[prodno].gl->n,
                 &sg1, producs->data[prodno].embed, sglist);

         if (ctrl[node].yedges.n==0)
           cout <<"No Y-edges to follow from node "<<node<<"\n";
         else
          {newnode =ctrl[node].yedges.data
             [rand()%ctrl[node].yedges.n];

           if (newnode ==OUTPUTNODE)
            {outputgraph =CopyGraph (*newgraph);
             if (ctrl[OUTPUTNODE].yedges.n ==0)
               printf ("ERROR - no Y-edge from OUTPUT node\n");
             newnode =ctrl[OUTPUTNODE].yedges.data
               [rand()%ctrl[OUTPUTNODE].yedges.n];
                                        /* follow a Y-edge */
                                        /* choose one randomly */
            }
```

```
        if (newnode ==FINALNODE)
         {st_node++;
          WRITE_ST (st_node, st_parent, prod,mu,S.cert&&mu);

          cout << "\n"<< st_node;
          FinalNode (S.output, (S.cert&&mu));
          if (S.cert >highestcert)
            {highestcert =S.cert;
             highestnode =st_node;
            }
         }
        else
         {if (mu==2.0)
               /* Special case caused by NOISE productions */
               /* actual certainty is calculated in AppendStatesToQ */
            newS.set (newnode, *newgraph, (S.cert+2.0), ++st_node, outputgraph);
          else
            newS.set (newnode, *newgraph, (S.cert&&mu), ++st_node, outputgraph);
          spos =TmpStatesList.append (newS); /* insert new state in tmp
                                                list */
          MarkGraphNodes (NodesList, spos, mu);  /* mark graph nodes in
                                                isomorphic subgraph */
          WRITE_ST (st_node, st_parent, prod,mu,newS.cert);
                    /* write new node in search tree output file */
         }
       }
      }
    cont =TRUE;
   }
 else                           /* isomorphic subgraph was not found */
   {mu=0.0;
    if (p ==ctrl[node].prods.n-1)
                      /* all prods in node have failed */
      {if (NoSuccess)
        if (ctrl[node].nedges.n ==0)
            cout <<"No N-edges to follow from node "<<node<<"\n";
        else
        {
         node =ctrl[node].nedges.data[rand()%ctrl[node].nedges.n];
           /* follow an N-edge (choose one randomly) */
        if (node ==OUTPUTNODE)
         {outputgraph =CopyGraph (graph);
            /* follow Y-edge */
          if (ctrl[node].yedges.n ==0)
            printf ("ERROR - no Y-edge from OUTPUT node\n");
          node=ctrl[node].yedges.data[rand()%ctrl[node].yedges.n];
```

```
        }
        if (node ==FINALNODE)
          {st_node++;
           WRITE_ST (st_node, st_parent, prod,0.0,S.cert);

           cout << "\n"<< st_node;
           FinalNode (S.output, S.cert);
           if (S.cert >highestcert)
            {
             highestcert =S.cert;
             highestnode =st_node;
            }
          }
        else
          {S.set (node,graph,S.cert,++st_node, outputgraph);
         spos =TmpStatesList.append (S); /* insert new state in tmp
                                                   list */
         WRITE_ST (st_node, st_parent, prod,mu,S.cert);
                         /* write new node in search tree output file */
          }
       }

      /* Append appropriate new branches to Q */
      AppendStatesToQ (Q);
      if (Q.empty())        /* if finished */
        {/* close state space search tree output file */
         fclose (st_file);
         cout << "Final node with highest certainty" <<highestnode<<"\n";
         return (0);
        }
      S =Q_POP();           /* go to a new state in search tree */
      S.get (node,graph, st_parent, outputgraph);
      p=0;
      prod =ctrl[node].prods.data[p];
      NoSuccess =1;
      SetupUsedNodes (graph);
                /* first production in node */
    }
   else
    {if (++p <ctrl[node].prods.n)
      prod =ctrl[node].prods.data[p];
                /* next production in node */
    }
   cont =FALSE;    /* start looking at beginning of graph */
  }
}
```

```
  return (OK);
}
```

Appendix E

State space search trees

E.1 Search tree for the simple example (14.1)

E.2 Search tree showing ambiguity (14.2)

E.3 Search tree showing more ambiguity (14.3)

Appendix F

Components of the prototype system

F.1 Low-level components

The low-level parts of the system are performed by three separate programs. Each of these programs reads in a TIFF image and writes either a TIFF image or a text data file (or both). All of the filenames used by these programs are #defined near the top of the .c file. When a filename is given (as an optional argument), it overrides the default input filename. Figure F.1 shown the relationship between these programs.

rm_staves finds the staff lines and removes them from the image (section 9.2). It writes the staff data to a text file, and writes a TIFF images with the staff lines removed. The input TIFF file should be saved as a black and white image (with or without compression). The associated makefile is makestaves.
Usage: rm_staves [input_image_filename]
Limitations: This program is not very robust and it only works for a single staff (although the extension to multiple staves would not be difficult). If it cannot find the staff, it fails ungracefully (it tends to crash).

thinlines reads in a TIFF file (the output of rm_staves), applies masks to thin the lines in the image and then writes the resulting image to another TIFF file (section 11.1).The associated makefile is makethinlines.
Usage: thinlines [no_staves_image_filename]

getprimits reads in the TIFF files resulting from rm_staves and thinlines and the stave data file, and outputs a text file containing the primitive elements in the input image (Chapters 11 and 12). The associated makefile is makeprimits.
Usage: getprimits [thinned_image_filename]

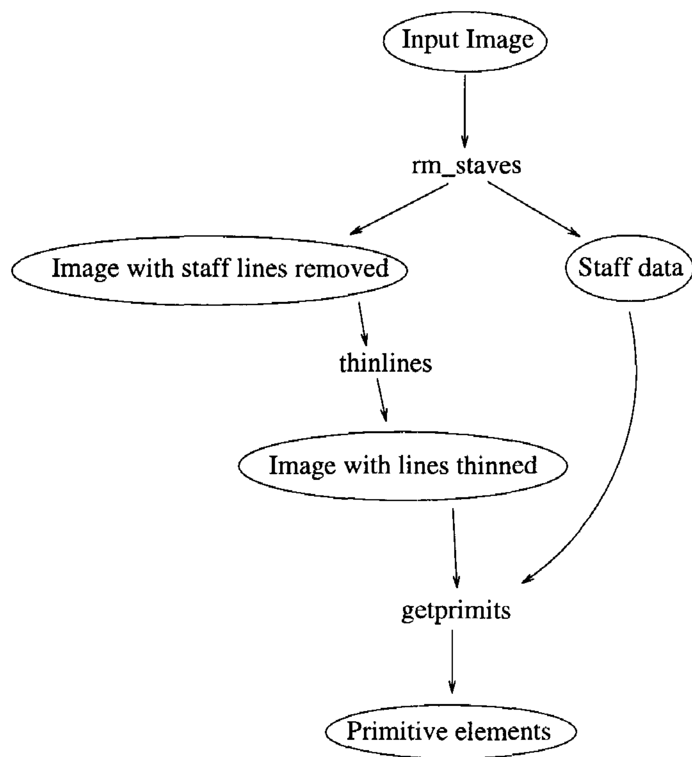


Figure F.1: Relationship between the low-level programs.

F.2 Parsing components

Parsing consists of two stages. **parsegrammar** parses the grammar and writes C-code functions for the attribute transfer functions and certainty functions (prodfuncs.C). A library of functions that these use is found in supfuncs.C. After this stage has been completed, prodfuncs.C can be compiled and parsegraph can be linked. The makefile associated with parsegrammar is makestep1.
Usage: parsegrammar [grammar_filename]

parsegraph can then be used to parse the input graph (read from the textfile produced by the low-level stages) according to the grammar. The associated makefile is Makefile.
Usage: parsegraph [grammar_filename graph_filename]
#defines near the top of the .C file specify the name of the control diagram file and also allow a choice of search type and the possible output of the search tree, in the form of a **daVinci** graph. For information on the daVinci graph visualisation system, view the URL http://www.informatik.uni-bremen.de/~davinci/ The number of beats (in hemidemisemiquavers) in a bar is currently hard-coded as a #define in the file supfuncs.h.

Appendix G

Systems and languages used

All of the work described in this thesis was done under various flavours of the UNIX operating system, predominantly Solaris 2.2 and 2.5 running on a Sun SPARCclassic and on a Sun SPARCserver 10, and Linux (various releases) running on an Intel 80486 PC.

We used two scanners at various stages for image acquisition: a Logitek hand-held scanner and a HP scanjet flatbed scanner. We used xv 3.00 to display the images, and to save them in postscript format for printing.

Images are read and written as black and white TIFF files, using the libraries written by Sam Leffler at Silicon Graphics (version 3.30beta). Information on these libraries can be found at the URL http://tecstar.cv.com/ dan/tiff/. The TIFF file format is very powerful, and is the one most commonly supported by scanner software.

The low-level primitive extraction routines were written in C, using the GNU C complier (gcc) version 2.4. The code for these routines totalled about 4200 lines (including spacing and comments).

The parsing algorithms were also initially written in C. However, when we implemented the fuzzy version of the parser, we ported the code to C++. This was done simply so that we could define a fuzzy certainty value (0..1) class and overload the standard boolean logic operators (and,or,not) to work on the fuzzy certainty values. This simplified the process of fuzzifying the parsing algorithms. Once again, we used the GNU compiler (g++) version 2.4. The code for these algorithms totalled about 6400 lines (including spacing and comments).

The parsing program reads in information such as the grammar, control diagram and input graph from various files. We used LEX and YACC to generate parsers for these files. This means that it is easy to modify the format used or to add additional symbols.

LEDA (Library of Efficient Data types and Algorithms) libraries (version 3.2) provided the priority queue and stack definitions and functions that are used by the parsing algorithms. LEDA is available by anonymous ftp from ftp.cs.uni-sb.de (134.96.252.31) /pub/LEDA.

We initially wrote routines to output graphs in xfig's file format, so that we could use xfig to visualise them. We also used xfig (release 2.1.6 and later 3.1.4) for drawing most of the diagrams in this thesis. We later discovered the daVinci graph visualisation system, and used this for all our graph visualisation. We wrote a routine to output the search tree as a daVinci graph so that it could be visualised. Information on daVinci can be found at the URL http://www.informatik.uni-bremen.de/~davinci/.

Finally, this thesis was typeset using LaTeX.