

Rule-Based Analysis and Generation of Music

Thesis by

Randall Richard Spangler

In Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1999

(Submitted May 20, 1999)

© 1999

Randall R. Spangler

All Rights Reserved

Acknowledgements

There are many people who have contributed, directly and indirectly, to work presented here on rule-based analysis and generation of music. Indisputably first and foremost among them is my thesis advisor, Rodney Goodman. His investigations in rule-based systems provide an essential foundation for exploration of their nature and uses. Without that foundation, this work could not have even been conceived. In addition to his contributions to the technical side of the work, he also shared with me his inspiration and excitement about the potential of computer-generated music. His encouragement, support, and expert professional guidance cannot be understated – I can only say that I count myself extremely lucky among graduate students.

There are several people whose direct contributions to this work must also not go unmentioned. I thank Jim Hawkins for many engaging discussions on the musical implications of this work. I also thank William Lengefeld and Carol Moon, two of my professors at the Claremont Colleges, who provided me with important input during the initial formation of this project. Thanks must also be extended to the graduate students who helped develop the ITRULE algorithm and support algorithms: Padhric Smyth, Chris Ulmer, Charles Higgins, and John Lindal. Several undergraduates contributed to the work, providing not only exuberance, but also an enormous amount of transcription of sheet music into MIDI format. Among them are Heather Dean, Tony S. Chang, and Alex Lin.

Personal thanks must also be extended to those who have been helpful during the conceptual stages of this work and who served as subjects for the psychophysics experiments, enduring hours upon hours of mangled Bach chorales. Most notable in this group is Eric Grahn, whose insatiable curiosity and stamina for profound discussions helped me strive towards greater depths of understanding, both in the work itself and in the personal context in which I performed it.

Finally, I could not have managed to complete this work without my loving wife and best friend, Celeste. I cannot begin to express my appreciation for her years of hard work and sacrifice on my behalf, forgoing both her own advanced degree and career. With her steadfast devotion and faithfulness she has followed me through deep valleys and seen bits of the great heights to which I have soared. She has been my confidant, comforter, and sounding board for many ideas. Her firm confidence and faith that I could accomplish this task has kept me going through many a long night. She has been an unwavering source of the humor, food, and backrubs required to survive graduate research.

I dedicate this work to my parents, who filled my life with questions of the universe and taught me that the answers to those questions *are* out there, somewhere. Early on, they encouraged my interest in science, and fought to make sure I had the opportunities to develop that interest. Without their love and support, this work, and for that matter the author himself, would not have been possible.

Abstract

We develop a rule-based system for the purpose of analyzing musical examples to extract probabilistic rules of harmony; these rules are then used to generate new harmony in response to a melody input in real-time. A representation of music derived from the figured bass is developed which is suitable for embodying the harmonic content of a piece of music in a format suitable for machine learning. Algorithms are developed to convert music between this representation and standard MIDI files. An efficient algorithm for extracting raw rules from examples is presented, along with a comparison of its behavior to alternative methods such as hashing and hybrid algorithms. Processes to refine the rules produced by the previous algorithm into a more compact representation are shown, including considerations for weighting rules based on the types of errors they make in addition to their accuracy. Psychophysics experiments are performed to measure the perception of harmonic errors. The results of these experiments allow the development of new algorithms to generate rules which make less noticeable errors. The techniques developed above are used to build a rule-based system for real-time accompaniment.

Table of Contents

Acknowledgements	iii
Abstract	v
Table of Contents	vi
List of Figures	x
List of Tables.....	xii
CHAPTER 1 Introduction.....	1
1. Overview	1
2. Constraints Imposed by Real-Time Functionality	2
3. Advantages of a Rule-Based Algorithm.....	2
4. Choice of Music to Study.....	3
CHAPTER 2 Representation of Music	6
1. Existing Electronic Representations of Music	7
1.1. MIDI.....	7
1.2. Generalized Electronic Representations for Composition and Analysis	9
2. Figured Bass.....	9
3. Extended Figured Bass.....	10
3.1. Melody	12
3.2. Chord Function.....	13
3.3. Voice Positions.....	15
3.4. Duration.....	16

3.5. Accent.....	16
4. Windowed Example List.....	17
5. Conversion from MIDI to Extended Figured Bass	20
6. Conversion from Extended Figured Bass to Windowed Example List	24
6.1. Beat-based Conversion.....	27
6.2. Accent-based Conversion.....	28
7. Conversion from Extended Figured Bass to MIDI	28
CHAPTER 3 Representation of Knowledge.....	30
1. Definition of a Rule.....	31
2. Inferencing Using Rules.....	33
CHAPTER 4 Learning / Data Mining Algorithms.....	35
1. Data Mining.....	36
1.1. Inputs.....	36
1.2. Outputs	39
2. Algorithm Descriptions.....	39
2.1. The Hashing Algorithm.....	41
2.2. The SpanRULE Algorithm.....	48
3. Performance Analysis	53
3.1. Performance vs. Hash Table Fullness	53
1.2. 7-Segment LCD Data – Using Random Example Data	59
1.3. Performance vs. Number of Examples.....	63
1.4. Performance vs. Number of Potential Rule LHS's.....	65
1.5. Performance vs. Number of RHS Values	67

1.6. Performance vs. Number of LHS Attribute Combinations	68
1.7. Performance vs. Fraction of Unknown LHS Values	69
1.8. Memory Usage	70
2. Conclusions	71
2.1. Hashing Algorithm	71
2.2. SpanRULE Algorithm	72
2.3. Comparison	72
2.4. Hybrid Algorithms	73
2.5. Summary	74
CHAPTER 2 Refining Raw Rules	75
1. Subsumption Pruning	76
2. Measures for Rule Weight	77
2.1. Percent Correct	77
2.2. Classification Weights	77
2.3. Error-cost Weight	77
3. Measures for Rule Priority	79
3.1. J-measure	79
3.2. Error-cost Measure	82
4. Independence Pruning	82
4.1. Definition of Rule Dependence	82
4.2. Generation of Real-Time Dependency Information	83
5. Filtering and Segmentation of Rulesets	84
CHAPTER 3 Perception of Harmonic Errors in Bach Chorales	86

1. Background	87
2. Experiment Overview	88
2.1. Desired Output	88
2.2. Definition of Test Examples	88
2.3. Distribution of Test Examples.....	89
2.4. Generation of Test Examples	91
2.5. Test Procedure.....	93
3. Results	94
3.1. Perceived Magnitude vs. Beat Strength	96
3.2. Errors are Non-Symmetric	96
4. Summary	97
CHAPTER 4 A Rule-Based System for Real-Time Harmony	98
1.1. Input Data.....	99
1.2. Rule Generation.....	99
1.3. Testing.....	100
1.4. Analysis	103
1.5. Generated Harmony	105
1.6. Use of Generated Rules in a Real-Time Environment.....	105
CHAPTER 5 Summary	107
References	111

List of Figures

Figure 1.1: Bach Chorale no. 1	5
Figure 2.1: Example MIDI Data	8
Figure 2.2: Example of Extended Figured Bass.....	11
Figure 2.3: Input Window Used by Algorithm	19
Figure 2.4: Typical Example List.....	19
Figure 2.5: Input – Segmented into Chords	21
Figure 2.6: Input – With Accent Information	21
Figure 2.7: Input – With Identified Chord	23
Figure 2.8: Input – With Voice Positions.....	23
Figure 2.9: Input - Extended Figured Bass	23
Figure 2.10: Windowed Example List	26
Figure 2.11: Beat-Based Example List	26
Figure 2.12: Accent-Based Example List	26
Figure 2.13: Example Timestep	29
Figure 3.1: Example of a Rule with Order=2.....	32
Figure 4.1: Hashing Value Algorithm.....	43
Figure 4.2: Algorithm to Fill Hash Table.....	45
Figure 4.3: Algorithm to Empty Hash Table.....	47
Figure 4.4: Pseudocode for SpanRULE Hash Value Function	49
Figure 4.5: Process for Determining Hash Value Precision.....	49

Figure 4.6: Algorithm for Generating Rules From Attribute Combinations.....	52
Figure 4.7: Runtime vs. Hash Table Fullness for Hashing Algorithm.....	57
Figure 4.8: Comparison of Runtimes From Real and Random LCD Data	61
Figure 4.9: Runtime vs. Number of Examples.....	64
Figure 4.10: Runtime vs. Number of Rule LHS's	66
Figure 5.1: J-Measure vs. $p(x/y)$	81
Figure 6.1: Algorithm for Generating Psychophysics Examples.....	92
Figure 7.1: Generated Harmony for "Happy Birthday"	104
Figure 7.2: Data Flow in the Real-Time Harmonizer	104

List of Tables

Table 2.1: Significant MIDI Events	8
Table 2.2: Common Chord Functions in a Major Key.....	14
Table 2.3: Beat Accents vs. Timesteps for 3:4 Time	21
Table 3.1: Terminology of Rules	32
Table 4.1: Terminology for Rule Generation Algorithms.....	40
Table 4.2: Terminology for Hashing Algorithm	40
Table 4.3: Terminology for SpanRULE Algorithm.....	47
Table 4.4: Example Sets Used For Benchmarking	55
Table 6.1: Average Perceived Errors for Chord Pairs.....	95
Table 6.2: Average Perceived Error vs. Beat Strength	95
Table 7.1: Ruleset Segments	101
Table 7.2: Ruleset Performance	101
Table 7.3: Comparison of Information Theoretic Measures with Error-Cost Measures	102
Table 7.4: Some Traditional Music Theory Rules Found in Rulesets	102

CHAPTER 1

Introduction

Human understanding and intuition of music is often supported by substantial intuition developed as the result of extensive study of and experience with music (Alphonse, 1980). However, even the collective observations of hundreds of years of musical study are insufficient to provide a theory of musical composition complete enough to generate new music in the absence of human intuition (Loy, 1991)(Rothgeb, 1968). Recent advances in rule-based data mining algorithms allow a more thorough analysis of music to derive the underlying rules of music theory. These learned rules can then be applied in *real-time* to generate new music in a similar style.

1. Overview

The remainder of this chapter serves to introduce the problem of analysis and real-time generation of harmony, and discusses the choice of music studied. The main body of this work begins in Chapter 2 with the evolution of the Extended Figured Bass representation for musical harmony. Chapter 3 defines the type of rules used in our learning system, and describes how they are used to inference results. The hashing and SpanRULE algorithms for quickly searching through an example set to extract unrefined rules are analyzed and compared in Chapter 4. This is followed in Chapter 5 by several methods which can be applied to those rules to refine them into a more compact representation of knowledge. The perception of harmonic errors by human listeners is covered in Chapter 6, including derivation of a chord misclassification error table. The

techniques described in Chapters 2 through 6 are used to build a rule-based system for generation of Bach harmony, which is described and analyzed in Chapter 7. Conclusions are discussed as they naturally arise in each chapter. These are summarized in Chapter 8 and wider implications are discussed.

2. Constraints Imposed by Real-Time Functionality

A program which is to provide real-time harmony to accompany musicians at live performances faces several constraints. First, it must be fast enough to generate accompaniment without detectable delay between the musician playing the melody and the algorithm generating the corresponding harmony. This limits the complexity of the algorithm and the amount of information it can process for each timestep. Second, the algorithm must base its output only on information from previous timesteps. This differentiates our system from HARMONET (Hild et al., 1992) which required knowledge of the next note in the future before generating harmony for the current note. Third, the algorithm must be largely autonomous. It must not require constant attention by the performer to produce acceptable output, and must be robust enough that any errors it does produce are of a sort which are not as musically "bad" as other errors.

3. Advantages of a Rule-Based Algorithm

A rule-based neural network algorithm was chosen over a recurrent network or a non-linear feed-forward network. Neural networks have been previously used for harmonizing music with some success (Mozer and Soukup, 1991) (Shibata, 1991) (Todd, 1989). However, rule-based algorithms have several advantages when dealing with music. Almost all music has some sort of rhythm and is tonal, meaning both pitch and

duration of individual notes are quantized. This presents problems in the use of continuous networks, which must be overtrained to reasonably approximate discrete behavior. Rule-based systems are inherently discrete, and do not have this problem.

Another advantage of a rule-based network is that it is straightforward to determine why the system produced a given result by examining the rules which fired. This has advantages in the development of the algorithm, since it is easier to determine where mistakes are being made. It allows comparison of the results to existing knowledge of music theory as shown below, and may provide insight into the theory of musical composition beyond that currently available. Conversely, it is very difficult to determine why a non-linear multi-layer network makes a given decision or to extract the knowledge contained in such a network.

4. Choice of Music to Study

Chorale melodies harmonized by Johann Sebastian Bach were used as the input for the learning system. These are short (8 to 40 measure) pieces with a number of desirable characteristics. The rules of harmony which Bach used have been studied extensively by music theoreticians, so there is an existing base of knowledge to compare with the rules extracted in the course of research (Grout and Palisca, 1988). The pieces usually remain in one key and have few unresolved dissonances, making them easier to analyze. They have little rhythmic variation, and the harmonic content usually changes on a beat-by-beat basis; this reduces the time scale over which learning must be done to a few chords instead of several measures of music. Chorales have four voices, meaning that at any point in time, exactly four notes are being played. This simplifies chord

analysis, since the four voices usually form complete chords where all pitches for a given chord are present; for example, if a C Major chord were being played, there would be at least one C, one E, and one G being played). This also usually means that the combined pitches for the voices will match only one chord type at a time. The first few measures of Bach Chorale no. 1 are shown in Figure 1.1.

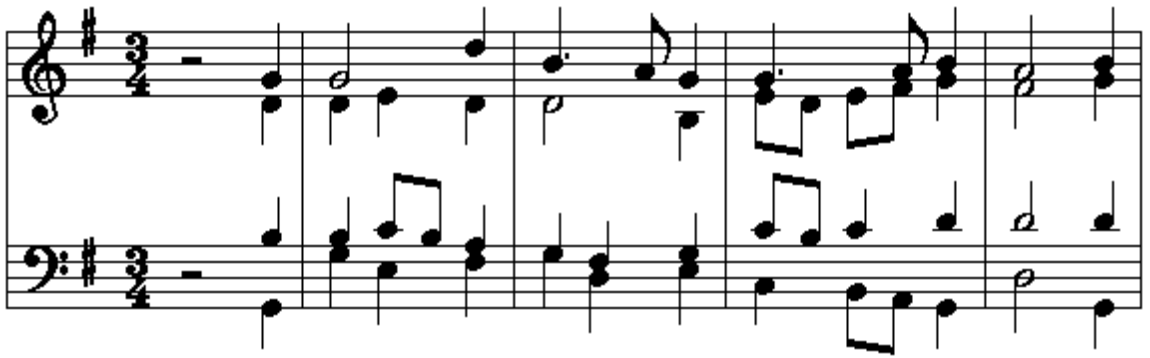


Figure 1.1: Bach Chorale no. 1

CHAPTER 2

Representation of Music

To extract useful information examples, it is necessary to choose a representation for those examples that accentuates the features being studied. In the case of musical harmony, the representation requires the following features. The representation needs to explicitly specify harmonic information; it needs to treat music as a progression of chords, as opposed to a collection of individual notes. Furthermore, the representation needs to be such that similar-sounding harmonies are represented similarly, and different-sounding harmonies are represented differently.

1. Existing Electronic Representations of Music

1.1. MIDI

The Musical Instrument Digital Interface (MIDI) file format is a specification for storage and transmission of musical data (MIDI Manufacturers Association, 1988). It was chosen as the medium for externally storing pieces of music because of its widespread use in the electronic music community, compatibility with existing sequencing software, and flexibility.

MIDI data is arranged as a stream of events, separated by delta times. This makes it ideal for sequencing applications where the goal is to send those events to synthesizers and effects generators. A typical stream of MIDI data is shown in Figure 2.1. The significant MIDI events are listed in Table 2.1.

```

Header format=0 ntrks=1 division=240
Track start
Delta time=0 Time signature=3/4 MIDI-clocks/click=24 32nd-
notes/24-MIDI-clocks=8
Delta time=0 Tempo, microseconds-per-MIDI-quarter-note=41248
Delta time=0 Meta Text, type=0x01 (Text Event) leng=23
    Text = <Chorale #001 in G Major>
Delta time=480 Note on, chan=1 pitch=67 vol=88
Delta time=0 Note on, chan=2 pitch=62 vol=72
Delta time=0 Note on, chan=3 pitch=59 vol=88
Delta time=0 Note on, chan=4 pitch=43 vol=65
Delta time=240 Note off, chan=4 pitch=43 vol=64
Delta time=0 Note off, chan=3 pitch=59 vol=64
Delta time=0 Note off, chan=2 pitch=62 vol=64
Delta time=0 Note off, chan=1 pitch=67 vol=64
Delta time=0 Note on, chan=1 pitch=67 vol=81
Delta time=0 Note on, chan=2 pitch=62 vol=75
Delta time=0 Note on, chan=3 pitch=59 vol=88
Delta time=0 Note on, chan=4 pitch=55 vol=60
Delta time=240 Note off, chan=4 pitch=55 vol=64
Delta time=0 Note off, chan=3 pitch=59 vol=64
Delta time=0 Note off, chan=2 pitch=62 vol=64
Delta time=0 Note on, chan=2 pitch=64 vol=58
Delta time=0 Note on, chan=3 pitch=60 vol=78
Delta time=1920 Meta Text, type=0x01 (Text Event) leng=7
    Text = <Fermata>

```

Figure 2.1: Example MIDI Data

Table 2.1: Significant MIDI Events

Event	Function	Relevant Parameters	Meaning
Time signature	Gives information about the timing of the piece	Time signature 32nd-notes/24-MIDI-clocks	needed to convert beats into measures, determine beat accents needed to convert current time into beat number
Note on / Note off	Turns a note on or off for a specific voice	Channel Pitch	which voice is changing (1=soprano, 2=alto, 3=tenor, 4=bass) which note is changing (pitch=60 is middle C)
Meta Text	Allows arbitrary messages to be sent	Text	“Chorale #001 in G Major” gives the name and key of the piece. “Fermata” states that there is a fermata on the chord starting at that time.

There are several reasons MIDI data is not suitable for use as input for musical analysis. First, it is difficult to tell which notes are being played at a given point in time. The durations of notes are not explicitly coded. Second, it is equally difficult to determine rhythmic structure. Last, the format is extremely sensitive to the exact notes being played. If a piece is transposed up a semitone (from C to C-sharp, for example), every single pitch in the MIDI data changes. Even minor changes in the voicing of a chord have radically different representations in the MIDI data. For example, a C Major chord (C, E, G, C) could consist of pitches {60, 64, 79, 84}, or {67, 72, 76, 84}. The two voicings sound almost identical and have similar functions, but share only one common pitch.

1.2. Generalized Electronic Representations for Composition and Analysis

Several generalized representations for music have been proposed. These representations are often phrased in deeply-nested hierarchical structures (Smaill et al., 1993) or LISP-like lists (Brinkman, 1986), and treat musical information as collections of notes and modifiers. Others use semantic nets to hold the musical information (Camurri et al., 1993). While superior to MIDI for analyzing music, these representations are too general, and do not state specific harmonic representations clearly enough to be useful in learning probabilistic rules.

2. Figured Bass

Figured bass is a form for representing harmony that has been used since the early Baroque period (early 17th century). It focuses attention on the melody line, harmonic function of each harmony chord, and the bass line; the exact positions of the inner voices

were considered less important. It consists of a melody line, and a chord notation consisting of Roman numerals for the chords, sometimes with small Arabic numerals to provide more information about how the chord should be voiced (Grout and Palisca, 1988).

A modern-day equivalent of figured bass can be found in the "fake books" sold at music stores. These give the melody and chords in letter notation (Cm for C-minor, for example), but contain even less information about the voicing of the chord than the Baroque figured bass. These are often used by pianists hired to play at events, so that they can respond to requests for songs. The pianists are capable of filling in interesting harmony based on the melody and the chord.

Figured bass is a good representation for harmonic information; however, it is incomplete. This was a desirable feature in the Baroque period, since it allowed individual performers to distinguish themselves by playing the same piece in different ways. However, it requires a great deal of musical background and prior information to adequately translate, or *realize* the figured bass into full four-part harmony.

3. Extended Figured Bass

The extended figured bass format concisely states the harmonic content and rhythmic information for an accompaniment. Music is organized in terms of chords and beats instead of individual note on/off events. A typical extended figured bass is shown in Figure 2.2; the MIDI data in Figure 2.1 corresponds to the first few chords of this figured bass. This is similar to the Baroque figured bass discussed above, with additional attributes for each chord which specify all of the voice positions for the chord.

MEL	FUNC	BP	TP	AP	SP	DUR	ACC	OCTAVE
C	I	B0	T1	A2	S0	2	un	4
C	I	B0	T1	A2	S0	2	ACC	4
C	IV	B1	T0	A1	S2	1	un	4
C	vi	B0	T2	A0	S1	1	n	4
G	V	B1	T2	A0	S0	2	un	4
E	I	B0	T0	A2	S1	2	ACC	4
E	iii	B1	T2	A1	S0	1	un	4
D	V	B0	T1	A0	S2	1	n	4
C	vi	B0	T1	A2	S1	2	un	4
C	IV	B0	T0	A1	S2	1	ACC	4
C	-	-	-	-	-	1	n	4
C	-	B3	T0	A1	S2	1	un	4
D	vii07	B1	T2	A0	S1	1	n	4
E	I	B0	T2	A0	S1	2	un	4
D	V	B0	T0	A1	S2	4	FERM	4

Figure 2.2: Example of Extended Figured Bass

The extended figured bass format always represents music in terms of the key C-major. This is easier for an observer or user to visualize and understand, since it is the simplest key (since it has no sharps or flats). It also simplifies learning from extended figured bass examples, since C is always the first scale step in the key. Music in other keys is transposed to the key of C when it is placed in extended figured bass format, and transposed back when it is translated from extended figured bass to some other format.

Alternately, the melody can be represented as a number of semitones above the root of the key. For example, in the key of G-major, G=1, G-sharp=2, A=3, etc. While this provides true key-independence of the representation, it also proved to be much more difficult to read.

The extended figured bass consists of the following attributes.

3.1. Melody

Melody (MEL) is the pitch played by the soprano voice. This is the most important of the four voices. Harmony is generated downwards and built off of the melody.

The extended figured bass format separates the scale step of the melody (C, C#, D, etc.) from the octave of the melody note (4th, 5th, etc.). This reduces the number of different symbols which the melody attribute can take down to the twelve different scale steps present in western music. The octave is retained as a separate attribute (see below).

3.2. Chord Function

A chord function (FUNC) is uniquely determined by the tones which form it. For example, if the tones F, A, and C are present, the chord is determined to be a F-major chord, which is notated by the Roman numeral IV.

The most common functions in a major key in the works of Bach are listed in Table 2.2. The chord names and pitches shown assume the key is C major, as the extended figured bass requires. Other less common functions also exist; however, the chords shown in the table are sufficient to classify 98% of the chord tones in Bach chorales. If a chord cannot be classified into a known function, its function is left blank. This chord is probably the result of a passing tone or other non-chord tone.

Table 2.2: Common Chord Functions in a Major Key

Function	Chord Name	Pitches
I	C Major	C, E, G
I7	C7	C, E, G, B-flat
ii	D minor	D, F, A
V/V	D Major	D, F-sharp, A
V7/V	D7	D, F-sharp, A, C
iii	E minor	E, G, B
V/vi	E Major	E, G-sharp, B
V7/vi	E7	E, G-sharp, B, D
IV	F Major	F, A, C
vii07/V	F-sharp diminished 7th	F#, A, C, E-flat
V	G minor	G, B-flat, D
V	G Major	G, B, D
V7	G7	G, B, D, F-sharp
vi	A minor	A, C, E
V/ii	A Major	A, C-sharp, E
V7/ii	A7	A, C-sharp, E, G
IV/IV	B-flat Major	B-flat, D, F
vii07	B diminished 7 th	B, D, F, A-flat

3.3. Voice Positions

Bach harmony has four voices. From lowest to highest, these are bass, tenor, alto, and soprano. These can be represented as tones such as F, A, or C, or as positions within a chord function, such as 0, 1, or 2 in the chord function F-major.

Voices are represented as positions, since this reduces the number of possible values each voice can have. In western music, there are 12 tones: C, C#, D, D#, E, F, F#, G, G#, A, A#, B. However, there are only 4 voice positions: 0, 1, 2, 3, corresponding to the four tones in each chord function. To reduce confusion, voice positions are prefixed by the first letter of the voice name (B, T, A, or S).

Given a chord function (determined in the above section), we can determine the position of a voice from its tone by a simple lookup in Table 2.2. For example, if we have the chord function V7 (G7), we see from the table that the tones for that chord function are 0=G, 1=B, 2=D, and 3=F#. So if the voice's tone is G, it is given a position of 0; if the voice's tone is B, it is given a position of 1; and so on. If the voice in question were the bass, the voice position would be labeled "B0" or "B1" respectively. If the voice in question were the soprano, the voice position would be labeled "S0" or "S1" respectively.

Given a chord function, we can determine the tone of a voice given its voice position. For example, if we have the same V7 chord function, then the voice position A0 corresponds to a "G" in the alto, since A=alto and G is the first chord tone for V7. Similarly, voice position T3 would correspond to "F#" (tone 3 for the chord function V7) in the tenor.

3.4. Duration

The duration (DUR) is simply the length of the chord in eighth notes.

3.5. Accent

The accent (ACC) is the accent to be placed on the chord. There are four different strengths of accent. “FERM” is the strongest accent, and represents a fermata (or held chord) at the end of a phrase of music. “ACC” indicates the chord begins at the start of an accented beat. “un” indicates a chord beginning on an unaccented beat. Lastly, “n” is used to represent a chord which does not begin at the start of a beat. Chords with stronger accents tend to be less dissonant; most pieces end on a fermata to give a solid finish to the music.

The accent pattern for a measure is completely deterministic on the style of music and the time signature for the measure. For example, 4:4 time accents the first and third beats in a measure. This gives the following accent pattern in beats (ACC=accented beat, un=unaccented beat):

ACC-un-ACC-un

To allow for more frequent and interesting note changes in a piece, there are two eighth-note-long timesteps per beat. The second timestep in the beat is not at the start of a beat at all. This gives the following accent pattern in timesteps (n=not start of beat):

ACC-n-un-n-ACC-n-un-n

A similar measure in 3:4 time has the following accent pattern in eighth-note timesteps (this is the example given in the paper):

ACC-n-un-n-un-n

Other accent patterns are used in other styles of music. For example, a syncopated piece of music in 4:4 time might have an accent pattern such as:

ACC-n-n-un-ACC-n-n-un

The accent pattern serves as a way to know which timesteps are more musically relevant to the listener's sense of melody and key. The more accented a timestep is (FERM=most, n=least), the more a listener will focus on it; an awkward or misplaced note on an accented beat is certain to be noticed. In Bach harmony, accented beats almost always contain stable chords in the key of the melody.


4. Windowed Example List

Before rules can be generated, it is necessary to create examples from the figured bass data. Each example must contain all the data necessary to agree or disagree with a potential rule, including information about previous timesteps. Examples in the list may also be weighted, so that they count for more or less than a normal example; this is useful when learning rules in beat-based conversion (see Section 6.1).

For each chord, an input window was generated which included the current melody note and harmony information (shown in yellow) and the previous two timesteps of information. This windowed approach to scanning music has been shown to be sufficient to capture harmonic information, and can be thought of as analogous to short-term memory (Jones et al., 1993). The input window used is shown in Figure 2.3.

A portion of a typical example list used in generating harmony rules is shown in Figure 2.4. The first section of the example list provides in order the names of the fields in an example. Following this are examples at one example per line. Note that some

examples in this list have twice the weight of other examples. Each example contains information about the melody and chord function used at the current time and the previous two timesteps.



Melody G C C
 Function V vi
 Soprano S0 S1
 Bass B0 I0
 Alto A1 A0
 Tenor T0 T2

Figure 2.3: Input Window Used by Algorithm

```

%NAME 0 Weight
%NAME 1 Duration0
%NAME 2 Melody2
%NAME 3 Melody1
%NAME 4 Melody0
%NAME 5 Function2
%NAME 6 Function1
%NAME 7 Function0
1.0 1 C C C I I IV
1.0 1 C C C I I vi
1.0 2 C C G I IV V
1.0 2 C C G I vi V
1.0 2 C G E IV V I
1.0 2 C G E vi V I
1.0 1 G E E V I iii
1.0 1 G E D V I V
1.0 2 E E C I iii vi
1.0 2 E D C I V vi
0.5 1 E C C iii vi IV
0.5 1 D C C V vi IV
0.5 1 C C D vi IV vii07
0.5 2 C D E IV vii07 I
1.0 4 D E D vii07 I V

```

Figure 2.4: Typical Example List

5. Conversion from MIDI to Extended Figured Bass

To convert from MIDI format into a figured bass, an algorithm scans through the MIDI file and determines which notes the voices (bass, tenor, alto, soprano) are playing at which times. It extracts the key of the piece from an initial text event, and transposes the piece to the key of C Major, changing all pitches appropriately; this simplifies analysis of the data by making it easier to compare pieces. It then segments the piece into chords by beginning a new chord whenever a voice changes pitch.

This algorithm has one significant weakness in that it always treats passing tones and other ornaments as starting new chords, though they actually may serve no harmonic function. The best alternative from a musical standpoint would be to hand-code an algorithm to detect ornament-based chords and remove them; however, this would defeat one of the primary research goals of requiring no explicitly hand-coded musical knowledge. Currently, two fully automated schemes (beat-based and chord-based conversions) have been used to minimize the effects of this weakness.

TIME	DUR	B	T	A	S
000					
004	2	{ C3	E4	G4	C5 }
006					
006	2	{ C4	E4	G4	C5 }
008	1	{ A3	F4	A4	C5 }
009	1	{ A3	E4	A4	C5 }
010	2	{ B3	D4	G4	G5 }

Figure 2.5: Input – Segmented into Chords

Table 2.3: Beat Accents vs. Timesteps for 3:4 Time

Time	Accent
$6n + 0$	ACC
$6n + 1$	n
$6n + 2$	un
$6n + 3$	n
$6n + 4$	un
$6n + 5$	n

TIME	DUR	B	T	A	S	MEL	ACC
000							
004	2	{ C3	E4	G4	C5 }	C	un
006							
006	2	{ C4	E4	G4	C5 }	C	ACC
008	1	{ A3	F4	A4	C5 }	C	un
009	1	{ A3	E4	A4	C5 }	C	n
010	2	{ B3	D4	G4	G5 }	G	un

Figure 2.6: Input – With Accent Information

Once a piece of music has been segmented into chords, it looks like Figure 2.5. Each line represents one timestep and one chord. It contains information about the time the chord was started, its duration, and which note is being played in each voice. The next step is to determine the melody pitch. This is trivial, since it is just the soprano note without the octave. At this time, the algorithm also determines the accent of each chord. This is based on the time a chord starts and the time signature for the piece. In 3:4 time (the time signature for this example), a measure is 6 timesteps long. Accented beats occur every 6 timesteps, and unaccented beats occur every 2 timesteps. In Table 2.3, n is an integer representing the measure number. Fermatas are indicated by a text event in the MIDI data at the same time the chord starts (see the MIDI file format section for an example.) Fermatas represent strong chords at the ends of phrases of music, and are the strongest sort of accent. With melody and accent information, the data looks Figure 2.6.

The algorithm then attempts to determine the root and type of chord is being played by comparing each timestep with 120 common chords looking for a match. (This number of chords is sufficient to identify 99% of all chords occurring in Bach's music.) If all pitches being played could be part of a chord, the timestep is identified as that chord. For example, the chord at time=008 is identified as a F Major chord because all its pitches are either F, A, or C. If a chord cannot be identified, all its remaining fields are left unknown and processing starts on the next chord; this usually indicates that the chord was formed from a passing tone or other ornament and has no significant function in the piece anyway. The data now looks like Figure 2.7.

TIME	DUR	B	T	A	S	MEL	ACC	RT	TYPE
000									
004	2	{ C3	E4	G4	C5 }	C	un	C	Major
006									
006	2	{ C4	E4	G4	C5 }	C	ACC	C	Major
008	1	{ A3	F4	A4	C5 }	C	un	F	Major
009	1	{ A3	E4	A4	C5 }	C	n	A	Minor
010	2	{ B3	D4	G4	G5 }	G	un	G	Major

Figure 2.7: Input – With Identified Chord

TIME	DUR	B	T	A	S	MEL	ACC	RT	TYPE	IN	TP	AP	SP
000													
004	2	{ C3	E4	G4	C5 }	C	un	C	Major	I0	T1	A2	S0
006													
006	2	{ C4	E4	G4	C5 }	C	ACC	C	Major	I0	T1	A2	S0
008	1	{ A3	F4	A4	C5 }	C	un	F	Major	I1	T0	A1	S2
009	1	{ A3	E4	A4	C5 }	C	n	A	Minor	I0	T2	A0	S1
010	2	{ B3	D4	G4	G5 }	G	un	G	Major	I1	T2	A0	S0

Figure 2.8: Input – With Voice Positions

MEL	FUNC	IN	TP	AP	SP	DUR	ACC
C	I	I0	T1	A2	S0	2	un
C	I	I0	T1	A2	S0	2	ACC
C	IV	I1	T0	A1	S2	1	un
C	vi	I0	T2	A0	S1	1	n
G	V	I1	T2	A0	S0	2	un

Figure 2.9: Input - Extended Figured Bass

The next step is to determine the position of each voice. This is done by comparing the pitch of each voice with the allowed pitches for the chord and determining which chord note each voice is playing. The chord at time=008 has pitches {A, F, A, C}, which correspond to positions {I1, T0, A1, S2}. Now the data has the form shown in Figure 2.8.

The final step is to identify the function of each chord. This is done by comparing the root and type of each chord with a table of common functions (a portion of the table is shown in the section on figured bass notation above). If a chord does not match any of the common functions, its function is left unknown. As when the chord type was not identifiable, this chord probably is a result of an ornamental and serves no harmonic function.

At this point, the fields for absolute time and voice pitch can be removed, since they are not part of the figured bass notation. This has the side effect of losing information on voice leading and counterpoint, and also on the overall structure of the piece. However, the effect on the harmony rules currently being investigated has proved to be small. The final extended figured bass is shown in Figure 2.9.

6. Conversion from Extended Figured Bass to Windowed Example List

Examples are generated from a figured bass by moving a window down the list of chords and copying certain fields at each timestep. For example, an example list with the fields Function0 and Function1 (meaning the chord functions at the current and previous timestep) generated from the figured bass in the previous section would look like Figure

2.10. Note that for the first example, the function for the previous timestep is unknown because the chord is at the beginning of the piece.

Unfortunately, not all chords in the modified figured bass generated by the MIDI to figured bass algorithm are harmonically significant. Because that algorithm is unable to distinguish and remove passing tones and other ornaments, extra chords are present. Two schemes are used to minimize the effects of that shortcoming on the example set; these schemes are discussed below.

```
%NAME 0 Function1
%NAME 1 Function0
-      I
I      I
I      IV
IV     vi
vi     V
```

Figure 2.10: Windowed Example List

```
%NAME 0 Weight
%NAME 1 Function1
%NAME 2 Function0
1.0   -      I
1.0   I      I
0.5   I      IV
0.5   I      vi
0.5   IV     V
0.5   vi     V
```

Figure 2.11: Beat-Based Example List

```
%NAME 0 FunctionLastAccentedBeat
%NAME 1 FunctionLastBeat
%NAME 2 Function1
%NAME 3 Function0
- -   -      I
- I   I      I
I I   I      IV
I IV  IV     vi
I IV  vi     V
```

Figure 2.12: Accent-Based Example List

6.1. Beat-based Conversion

This scheme takes advantage of the fact that harmonic function usually changes between beats, but not within a single beat. Ornaments usually comprise only half of a beat, and the chords formed from them are less correlated with the surrounding music than the chords comprising the other half of the beat are. Thus, examples which include information from ornament chords will not correlate well with other examples and will produce only weak rules.

The beat-based algorithm is more complex than the chord-based algorithm because it must look at each chord for a beat and generate an example assuming that chord was the “real” chord for that beat. All examples for a timestep must then have their weights normalized so that the total weight for each timestep is one. The segment of figured bass used in Figure 2.10 would generate the beat-based example list shown in Figure 2.11.

This seems fairly straightforward when the examples are using only one previous beat of data. If, on the other hand, an example set was built from the current beat and four previous beats, and each beat had two chords (an ornament chord and the real chord), then each beat would result in $(2 \cdot 2 \cdot 2 \cdot 2 \cdot 2) = 32$ examples, each with weight 0.03125. This results in an example set which uses a great deal more memory. Beat-based conversion is therefore well-suited only to those example sets with a small time window.

6.2. Accent-based Conversion

The counterpart to beat-based conversion, accent-based conversion takes advantage of the fact that ornaments typically occur in the middle of beats or, less commonly, at the start of unaccented beats. Ornaments are only rarely placed at the start of accented beats, and never at the start of a fermata. Accent-based conversion allows additional example fields to be created for previous timesteps which started at the beginning of a beat, accented beat, or fermata. Since only one example is created per timestep, it is not necessary to weight the examples. The examples created from the segment of figured bass in Figure 2.10 are shown in Figure 2.11.

Note that it is possible for the first three fields to refer to the same timestep if the previous timestep was at the start of an accented beat. This redundancy can lead to the generation of highly interdependent rules, making independence pruning (see Chapter 5) essential.

7. Conversion from Extended Figured Bass to MIDI

The algorithm used to inference the harmony for a melody results in the generation of a figured bass. It is necessary to convert that figured bass back into MIDI data so that the harmony can be played on a synthesizer. The process for creating the MIDI data, or *realizing* the figured bass, is as follows. For each timestep:

- 1) Use the table of common functions to determine which pitches should be played for the chord.
- 2) Use the voice position fields to determine the pitch played by each voice.

- 3) Starting at just below the melody note, which is known (it was used as the input to the harmonizing algorithm), search down the scale until an unplayed note matching the alto's pitch is found. Send the MIDI code to turn that note on.
- 4) Repeat for tenor and bass notes.
- 5) Send the MIDI code for a delay equal to the note's duration.
- 6) Turn the notes off.

For example, take the following timestep:

MEL	FUNC	IN	TP	AP	SP	DUR	ACC
E	iii	I2	T1	A1	S0	1	un

Figure 2.13: Example Timestep

The table of common functions shows the “iii” chord has the pitches {E, G, B}. Based on the positions {I2, T1, A1, S0}, the voices will be playing (in order) {B, G, G, E}. Note the soprano pitch agrees with the melody field. If the melody note were at octave 5, this algorithm would turn on the notes {E5, G4, G3, B2}. In either case, ADMIRE would then send a delay corresponding to a duration of one eighth note, then turn the notes off. This algorithm assumes that pitches are never doubled. This is an acceptable assumption, since shifting notes down an octave does not significantly change the feel of the chord.

CHAPTER 3

Representation of Knowledge

Our system represents learned knowledge in the form of sets of parallel probabilistic rules. When input is received, all rules are fired in parallel to generate an output. This is more robust than representing rules in a many-layered decision tree (Kohonen et al., 1991), where a single bad rule can inhibit processing of many worthwhile rule deeper in the tree.

1. Definition of a Rule

Throughout the rest of this document, the terminology in Table 3.1 is used. Figure 3.1 shows an example of a rule with Order = 2.

Table 3.1: Terminology of Rules

Left-Hand Side (LHS)	Conditions that must be true for the rule to fire.
Right-Hand Side (RHS)	Result predicted by rule.
Order	The number of attribute=value pairs on the LHS.
P_{fire}	The probability the rule is able to fire.
P_{correct}	The probability the rule is correct if it fires.
Priority	The overall value of the rule over the entire input domain; several priority types are discussed below.
Weight	The value of the rule over the subset of the input domain where the LHS of the rule applies.

IF $Y_A=y_1$ AND $Y_B=y_2$ THEN $X=x_3$ WITH Priority P, Weight W

Figure 3.1: Example of a Rule with Order=2

2. Inferencing Using Rules

When a ruleset is used to infer a RHS value, each rule in the ruleset is checked in order of decreasing rule Priority. A rule can fire if it has not been marked dependent (see the next section on independence pruning) and all the attributes on its LHS are known. When a rule fires, its weight is added to the weight of the RHS value which it predicts. After all rules have had a chance to fire, the result is an array of weights for all possible values of the RHS attribute.

If all rules which fire on a given example inference the same RHS value, the result of the inference is clear. But if two or more rules fire and inference a number of different RHS values, one of two algorithms must be used to resolve the conflict. In either case, the weights of all rules inferencing a given RHS are accumulated to produce the weight of that RHS.

The simpler algorithm is termed “best-only.” The RHS with the highest weight is always chosen. This is the most correct method from the standpoint of probability theory. However, this tends to lead to monotonous music, since a given melody will always be harmonized in the exact same fashion. Furthermore, there is no single right way to harmonize any given melody (Shibata, 1991). Several identical chorale melodies are harmonized by Bach in two or more ways in his collection of harmonized chorales (Bach, 1941). These problems led to the development of a second algorithm.

The other option is to select between the possible RHS values based on a set of transition probabilities. The accumulated weights for the RHS values are exponentiated and normalized to produce probabilities for each value. The RHS value to be used is

chosen randomly based on these probabilities. It is important to note that the algorithm only chooses between values which had rules fire, not all possible values for the RHS attribute. Otherwise, there would always be a non-zero probability of picking any RHS value, even if no rules fired for that value.

If no rules for a given ruleset fire, there are two possibilities. If it is not the last part of a series of segmented rulesets, the next segmented ruleset will be given a chance to fire. If the ruleset is the last in the series, or is not part of a series of segmented rulesets, the RHS value is set to the most likely value of the RHS attribute based on the attribute's prior probability distribution. This is equivalent to classifying the RHS attribute with a zeroth-order Bayesian classifier.

This last problem can be avoided by training a first-order Bayesian classifier and using it as the last segment in a series of rulesets for a given RHS attribute. (For example, basing the current chord function only on the current melody pitch, and setting both the minimum probability for a rule and the minimum rule Priority to zero.) Since the first-order classifier will always have exactly one rule which fires, more information will be used to pick the RHS value than if no rules fired at all.

CHAPTER 4

Learning / Data Mining Algorithms

This section discusses two general algorithms for extracting rules from example data. The first algorithm discussed is the hashing algorithm, which builds up a large hash table of all possible rules. This was the algorithm used in previous research on the ITRULE algorithm (Smyth and Goodman, 1990). The second algorithm, called SpanRULE, is a more efficient algorithm I developed for quickly spanning an example set with all possible rules. The algorithms are described in detail. This section then discusses the performance of these algorithms for various types of datasets.

1. Data Mining

The goal of these algorithms is to quickly search through a database of examples and examine every potential rule which could be generated from those examples. These algorithms are not dependent on a specific measure of rule worth; they can be used with any measure of rule worth.

Each example is a set of attribute-value pairs. The value for a particular attribute for a given example may be unknown. The algorithms do not require that all examples have known values for all attributes. These algorithms currently work with discrete values only; they do not work with continuous-valued attributes unless those attributes are first binned into discrete bins (for example, a continuous attribute in the range 0-100 might be binned into 0-10, 10-20, 20-30, 30-40, etc.).

1.1. Inputs

These algorithms take a number of input parameters.

1.1.1 Example List

This is usually represented as a two-dimensional array of integers. Each row is a single example. Each column is an attribute. The value at location (R,C) in the array is therefore the value of attribute C for example R. For each attribute, we have an array of possible string tokens for that attribute. For example, an attribute "color" might have an array {"red", "yellow", "green", "blue"}. The value stored at (R,C) is an offset into the token array for the Cth attribute. So if that attribute is "color" and the value is 2, the Rth example would have the attribute-value pair "color=green" (since arrays are 0-based). If the value for that attribute is unknown for the example, (R,C) would contain the value -1.

Using 32-bit integers, each attribute can thus have up to 2 billion different discrete values. This is more than sufficient.

Using 16-bit integers would reduce the amount of memory consumed by the example list by half. This might be worthwhile in extremely low memory situations. However, since the current generation of 32-bit processors is slightly more efficient at reading in 32-bit integers than 16-bit integers, it might also involve a slowdown of a few percent when accessing the example list.

1.1.2 Function for Evaluating a Measure of Rule Worth

The algorithms discussed here are independent of the measure of rule worth used. This is true so long as the measure of rule worth can be determined independently for each rule and is not dependent on other rules. The measure of rule worth as used by these algorithms is a function which takes as input a rule and returns as output a floating point number. The measure is allowed to use other static variables such as the distributions of

the attributes for the LHS and RHS of the rule, etc. For testing these algorithms, the measure of rule worth used was the ITRULE J-measure.

1.1.3 Minimum Measure of Rule Worth

The rule generation algorithms can be set to keep only rules with worth above some threshold value. (Worth is determined by the above function.)

1.1.4 Maximum Rule Order

This is the maximum number of terms allowed on the LHS of a rule.

1.1.5 Minimum Examples On Which A Rule Must Fire Correctly

The algorithms can be set to keep only rules which fire correctly on more than one example. This reduces the risk that the generated rules simply memorize the training examples. As an example of this, consider a training set of 1000 examples and a generated set of 1000 rules, each firing 100% correctly on only a single example. This ruleset would generalize extremely poorly.

The SpanRULE algorithm can take advantage of this parameter to reduce the number of rules which must be examined closely.

1.1.6 Minimum Fraction Correct When Rule Fires

The algorithms can be set to keep only rules which are correct more than a threshold fraction of the time when they fire. This is a common measure of rule worth, and is often used in conjunction with a more complex measure of rule worth such as the J-measure.

The SpanRULE algorithm can take advantage of this parameter to reduce the number of rules which must be examined closely.

1.1.7 Number of Rules To Keep

Often, these algorithms will generate millions of potential rules. Keeping all of these rules around would consume large amounts of memory. However, most of these rules would be of extremely low rule worth. These algorithms sort the entire set of potential rules by the measure of rule worth, and keep at most this many rules (in descending order of rule worth).

1.2. Outputs

Both algorithms output the set of rules with the highest rule worth. Depending on the number of potential rules, this may be less than or equal to the maximum number of rules to keep. The algorithms also output a number of statistics used to analyze their performance. These outputs are discussed in more detail in the performance analysis section below.

2. Algorithm Descriptions

These algorithms make use of the terminology in Table 4.1.

Table 4.1: Terminology for Rule Generation Algorithms

MAXORDER	Maximum Rule Order (see above).
MAXRULES	Number of Rules To Keep (see above).
NVRHS	The number of different values the RHS attribute can take.
EXAMPLES	The set of training examples. We filter out any examples with unknown values for the RHS attribute at the time the examples are loaded, since those examples can't be used for generating rules.
EXAMPLE[n]	The nth training example.
EXAMPLE.RHSVAL	The RHS attribute's value for an example.
MINWORTH	Minimum Measure of Rule Worth (see above).
MINCORRECT	Minimum Fraction Correct When Rule Fires (see above).
MINEXAMPLES	Minimum Examples Rule Fires Correctly On (see above).
RULEWORTH(r)	Measure of rule r's worth.

Table 4.2: Terminology for Hashing Algorithm

HASHSIZE	Number of elements in the hash table.
HASHTABLE(hv)	The hash table element at offset hv. This contains the following elements: LHS (RULELHS for this element) and RHSDIST (distribution of RHS values for the LHS).
HASHVALUE(lhs)	A hash value derived from the attribute-value pairs in the specified rule LHS.
RULELHS	The left-hand side of a rule. This contains the following elements: ORDER (number of LHS terms), ATTR[] (attribute for each term), and VALUE[] (value for each term).
LHSCOMBOS	The set of permutations of 1...MAXORDER LHS attributes. For example, with the LHS attributes {A, B, C} and MAXORDER=3, this set would contain {A, B, C, AB, AC, BC, ABC}.

2.1. The Hashing Algorithm

This algorithm makes use of additional terminology from Table 4.2.

2.1.1 Stage 1: Allocate Memory and Determine Hashing Value Function

The hashing algorithm requires a hash table which is large enough to hold all possible rule LHS's and the RHS distributions for them. Determining the size of this hash table is the first problem with the hashing algorithm. If the table is too small, it will not be able to hold all of the rule LHS's which are actually generated, and the algorithm will halt and need to be restarted. If the table is too large, additional time will be wasted in stage 3 scanning the table for potential rules. If the table does not fit in available physical memory, excessive amounts of time will be spent managing virtual memory for the table; this can slow the algorithm by a factor of 100 or more. Section 3.1 below analyzes the effect of hash table size on the time and memory required for the hashing algorithm.

The hashing value function is an important part of the hashing algorithm. This takes a potential rule LHS and generates a number in the range $0 \dots \text{HASHSIZE}-1$. The choice of function here can make a non-trivial difference in how evenly the hash table is filled. A poor choice leads to uneven filling, which increases the number of hash entries that need to be examined when trying to add an example's contribution to the hash table. This increases the runtime, sometimes dramatically. This can be seen in the results as 'bumps' in the runtime graphs for the hashing method where the value function was not a good match for the example data. Unfortunately, again it is not possible before running the algorithm to determine whether a particular hashing value function is a good match

for the given example data or not. The hashing value used for these tests is generated by the pseudocode in Figure 4.1; this seemed to fill the hash table fairly evenly for the examples tested.

```
VALUE = LHS.ORDER  
  
Loop A from 0...LHS.ORDER-1  
    VALUE = (1234567 * VALUE) + LHS.ATTR[A]  
    VALUE = (7654321 * VALUE) + LHS.VALUE[A]  
    VALUE = (VALUE * 112344567) mod HASHSIZE
```

Figure 4.1: Hashing Value Algorithm

2.1.2 Stage 2: Fill the Hash Table

The advantage of the hashing algorithm is that it only needs to look at each example once. Once the hashing algorithm is done with that example, it never refers back to it again. This means that the examples do not need to be held in memory while the algorithm is running. This makes the hashing algorithm well-suited to problems which have tens or hundreds of millions of examples, so long as those problems don't generate so many potential rules that the hash table size is prohibitive.

For each example, examine all the rule LHS's which can be formed by the LHSCOMBO attributes and their corresponding values in the example. Skip any LHS's where the corresponding values in the example are unknown.

For each potential LHS, generate the corresponding HASHVALUE(lhs). Look in the that hash table element to see if it is (1) unused or (2) matches our current LHS. If the element is used but not by our LHS, skip to the next element and try again. Continue this process, wrapping from the end of the hash table to the beginning, until either (1) or (2) or we arrive back at our original element (in which case the hash table must be full – panic and stop calculating).

If the table element is unused, copy in the potential LHS. The element is now used, but has an empty distribution.

In either case, now increment the RHS distribution for that table element for the RHS attribute's value in the current example.

This stage can be represented by the pseudocode in Figure 4.2.


```
For EX in EXAMPLES
  For CO in LHSCOMBOS
    If EX contains unknown values for an attribute in CO,
      next EX

    ELHS = LHS with attributes from CO and values from EX
    HV = HASHVALUE(ELHS)

    Search HASHTABLE starting at HASHTABLE[HV]

      If HASHTABLE[HV] is unused, copy ELHS into
        HASHTABLE[HV].LHS and stop searching

      If HASHTABLE[HV].LHS = ELHS, stop searching

      Otherwise, update HV = (HV + 1) mod HASHSIZE

      If HV = HASHVALUE(ELHS), panic (hash table is
        full)

    Increment HASHTABLE[HV].RHSDIST[EX.RHSVAL]
```

Figure 4.2: Algorithm to Fill Hash Table

2.1.1 Stage 3: Empty the Hash Table

Once all the examples have been processed, the hash table contains the RHS distributions for all potential rule LHS's which can be generated from the example data. The next step is to empty the hash table and generate rules from those distributions.

For each hash table element in use, generate a potential rule per nonzero entry in the RHS distribution for that element. (For example, if the RHS distribution for an element has 3 nonzero bins, generate 3 potential rules.)

Evaluate each potential rule and make sure it meets the tests for MINEXAMPLES, MINCORRECT, MINWORTH. If it meets all the tests, add it to the output list of rules. If the output list of rules already contains MAXRULES rules, remove the lowest worth rule and replace it with the new rule if the new rule is of higher worth.

This stage can be represented by the pseudocode in Figure 4.3.

```

For TE in HASHTABLE
    If TE is unused entry, next TE
    For R = 0..NVRHS-1
        If TE.RHSDIST[R] = 0, next R
        Consider potential rule PR of the form "if TE.LHS
        then RHS=R"
        If TE.RHSDIST[R] < MINEXAMPLES, next R
        If TE.RHSDIST[R] is not at least MINCORRECT fraction
        of total counts in TE.RHSDIST, next R
        If RULEWORTH(PR) < MINWORTH, next R
        If output list of rules contains less than MAXRULES
        rules, add PR to list and next R
        Find rule RLW with lowest worth LW in output list.
        If RULEWORTH(PR) < LW, next R
        Remove rule RLW from the output list and replace it
        with rule PR

```

Figure 4.3: Algorithm to Empty Hash Table

Table 4.3: Terminology for SpanRULE Algorithm

HASHARRAY	Array of sortable elements. Each element contains the following parts: VALUE (hash value) and INDEX (index of the example this element refers back to).
CNVLHS[att]	Number of values attribute att in the current attribute combination can take.
CEXVAL[att]	Value for the current example for attribute att in the current attribute combination.
AMINWORTH	Adaptive minimum rule worth required to save a rule (starts out at MINWORTH).
RHSDIST[]	Distribution of RHS values for the current LHS.

2.2. The SpanRULE Algorithm

The discussion of the SpanRULE algorithm makes use of additional terminology from Table 4.3.

2.2.1 Stage 1: Determine Hash Value Precision

The SpanRULE algorithm works by creating a hash value for each example from the values in that example corresponding to the current attribute combination. This is generated by the pseudocode in Figure 4.4.

Obviously, VALUE must have sufficient precision to hold the product of the CNVLHS[]'s for the worst-case combination of attributes. The algorithm verifies this using the process in Figure 4.5. For all of the examples run so far, BITSREQUIRED \leq 32. This has allowed the hash values to be stored in normal 32-bit unsigned integers. For high-order rules where the attributes can have many values, it might be necessary to use 64-bit numbers. This would obviously impact sorting performance somewhat on the current generation of personal computers, because it would take longer to generate and compare the values, and because larger numbers would require more memory for the HASHARRAY.

```
VALUE = 0

For A = 0...CURRENTORDER - 1

    If CEXVAL[A] = -1 then fail (example contains unknown value
    for an attribute we're using)

    If A>0 then VALUE = (VALUE * CNVLHS[A-1])

    VALUE = VALUE + CEXVAL[A]
```

Figure 4.4: Pseudocode for SpanRULE Hash Value Function

```
MAXVALUE = 0

For CO in LHSCOMBOS

    MAXCO = 1

    For A = 0...CO.ORDER-1

        MAXCO = MAXCO * CNVLHS[A]

    If MAXVALUE < MAXCO then MAXVALUE = MAXCO

BITSREQUIRED = LOG2(MAXVALUE)
```

Figure 4.5: Process for Determining Hash Value Precision

2.2.2 Stage 2: Allocate Memory

The SpanRULE algorithm requires memory proportional to the number of examples. The entire example list needs to be held in memory, since it is referenced multiple times. SpanRULE also requires an array of sortable elements, one per example.

Because the memory requirements of SpanRULE are known at the start of the algorithm, if enough memory is available it is guaranteed that the algorithm will complete. This is an obvious advantage of SpanRULE over hashing, since hashing cannot determine beforehand how much memory will be required for the algorithm to complete.

2.2.3 Stage 3: Generate Rules From Attribute Combinations

In hashing, the outermost loop of the algorithm is over the examples. In SpanRULE, the outermost loop is over the attribute combinations LHSCOMBOS.

Within each combo, the algorithm scans down the list of examples and calculates the hash value for each example. If the example does not contain any unknown values for the current attribute combination, the hash value and the index for the current example are added to the HASHARRAY. This has performance advantages if the examples contain a high fraction of unknown values, since those examples are weeded out early in the processing of each combination.

The algorithm then sorts the HASHARRAY by HASHARRAY.VALUE. SpanRULE currently uses the quicksort algorithm, which has appealing performance, but this could easily be replaced by any other sorting algorithm. Because there is a 1:1

mapping of hash values to LHS values for each attribute combination, sorting has the effect of placing all elements with the same hash values adjacent to each other.

SpanRULE then traverses along the hash array, considering each group of elements which share a common hash value / LHS. It accumulates the distribution of the RHS values for the corresponding examples to the elements (remember that `HASHARRAY[i].INDEX` is the index of the example from which element `HASHARRAY[i]` was derived).

For each LHS, SpanRULE examines the RHS distribution in order of decreasing count. If the count drops below `MINEXAMPLES`, or the count divided by the total count in the distribution drops below `MINCOUNT`, processing proceeds to the next LHS / hash value, since the rest of the distribution will also be below `MINEXAMPLES` or `MINCOUNT`. This reduces the number of potential rules which need further consideration.

For each potential rule (LHS-RHS combo), SpanRULE then calculates the rule worth. If the worth is below `AMINWORTH`, the rule is too low worth to keep (its worth is either less than `MINWORTH`, or it is less than the worth of the `MAXRULES` rules already in the output list so it wouldn't be kept anyway), so processing moves to the next potential rule. Otherwise, the rule is added to the output list of rules. If the output list already contains `MAXRULES` rules, search for the rule with lowest worth. If that worth is less than the new rule's, remove that rule from the output list and replace it with the new rule. Update `AMINWORTH` to the worth of the removed rule, since that is \leq the worth of all other rules in the output list.

This stage can be represented by the code in Figure 4.6.

```
AMINWORTH = MINWORTH

For CO in LHSCOMBOS

    // (here we're filling the hash array)
    For EX in EXAMPLES

        If EX contains unknown values for an attribute in CO,
        next EX

        HASHARRAY[EX].INDEX = EX
        HASHARRAY[EX].VALUE = SHASHVALUE(CO,EX)

    // (here we're sorting the hash array)
    Quicksort HASHARRAY by HASHARRAY.INDEX

    // (here we're searching the sorted array for rules)
    For HDIST in subsets of HASHARRAY with same index

        ELHS = LHS with attributes from CO and values from EX

        For HE in HDIST

            Increment RHSDIST[HE.INDEX.RHSVAL]

        For R in RHSDIST, in order of decreasing RHSDIST[R]

            If RHSDIST[R] < MINEXAMPLES, skip to next HDIST
            (not just next R)

            If RHSDIST[R] is not at least MINCORRECT
            fraction of total counts in RHSDIST, skip to
            next HDIST

            Consider potential rule PR of the form "if ELHS
            then RHS=R"

            If RULEWORTH(PR) < AMINWORTH, next R

            If output list of rules contains less than
            MAXRULES rules, add PR to list and next R

            Find rule RLW with lowest worth LW in output
            list. If RULEWORTH(PR) < LW, next R

            Remove rule RLW from the output list and
            replace it with rule PR

        AMINWORTH = LW
```

Figure 4.6: Algorithm for Generating Rules From Attribute Combinations

3. Performance Analysis

This section examines the performance of the hashing and SpanRULE methods on a number of different data sets. Performance for an individual run is measured in two ways: amount of memory required by the algorithm, and time required for the algorithm to complete (including whether the algorithm completed at all)

Benchmarks were run on a dual Pentium II 266MHz system with 192MB RAM and 10000RPM ultra-wide SCSI hard drive, running Windows NT 4.0. The algorithms were limited to one of the processors; the operating system and benchmarking routines ran on the second processor, to minimize interference with the algorithm. The algorithms were allowed to consume up to 160MB RAM. More memory was available as virtual memory (paged off of a swapfile on the hard drive), but the performance of algorithms relying on virtual memory was several orders of magnitude slower (enough so that the test runs were stopped after 8 hours, and had still not completed by then).

Unless otherwise noted, all runs used the following settings:

- MINEXAMPLES = 2
- MINCORRECT = 0.5
- RULEWORTH() = ITRULE J-measure
- MINWORTH = 0.001
- MAXRULES = 2000

3.1. Performance vs. Hash Table Fullness

The hashing algorithm requires a large hash table to hold the RHS distributions for all the possible rule LHS's. If this table is too small, it will fill up completely and the

algorithm will halt. If the table is only barely big enough, the algorithm will be required to spend more time searching for LHS's and open slots in the table. The bigger the table gets, the more likely that the hash values for the LHS's will be unique, and the less time will be spent searching for LHS's and open slots in the table. However, when the hash table is initialized in step 1 or emptied in stage 3, a larger hash table takes longer to empty.

Ideally, the hashing algorithm would pick a table size which is large enough to hold all the LHS's without being too crowded, and without being so large that it spends extra time initializing and emptying the hash table. Unfortunately, there is no way to predict ahead of time the number of LHS's which will be generated from an example set.

For three different example sets, the SpanRULE algorithm was used first, to determine the number of LHS's generated from the example sets. With this information, it was possible to run the hashing algorithm with a series of table sizes which would result in the filled table being between 10% - 99% full.

The first two example sets were Bach harmony data from (Spangler, Goodman and Hawkins, 1998). The third example set was a chess endgame dataset (Bain, 1992). These sets are shown in Table 4.4.

Table 4.4: Example Sets Used For Benchmarking

Example set 1: Bach harmony, denoted as 1.20

- 5783 examples
- MAXORDER = 5
- LHS = "Melody0", "Melody1", "Melody2", "Function1", "Function2", "Accent0", "Accent1", "Bass1", "Tenor1", "Alto1", "Soprano1"
- RHS = "Function0"

Example set 2: Bach harmony, denoted as 1.21

- 6847 examples
- MAXORDER = 5
- LHS = "Melody0", "Melody1", "Melody2", "Function1", "Function2", "Accent0", "Accent1", "Bass1", "Tenor1", "Alto1", "Soprano1", "Function0"
- RHS = "Bass0"

Example set 3: Chess endgame, denoted as 1.22

- 28055 examples
- MAXORDER = 5
- LHS = "WKingCol", "WKingRow", "WRookCol", "WRookRow", "BKingCol", "BKingRow"
- RHS = "MovesToWin"

Subsets of each example set were also run, to test whether the shape of the performance curve for the hashing algorithm vs. hash table size varied based on the size of the run. These subsets were 30, 100, 300, 1000, 3000, 10000 examples (obviously, the 10000-example run was only applicable for the third example set).

Each example set and subset was run with hash table fill between 10% - 99% (based on the required hash table size as determined by the number of LHS's reported by SpanRULE). For each subset, the runtimes for the hash table sizes were scaled to the best runtime within the hash table sizes for that subset. This made it possible to compare the curve shapes between sets. Figure 4.7 shows this data.

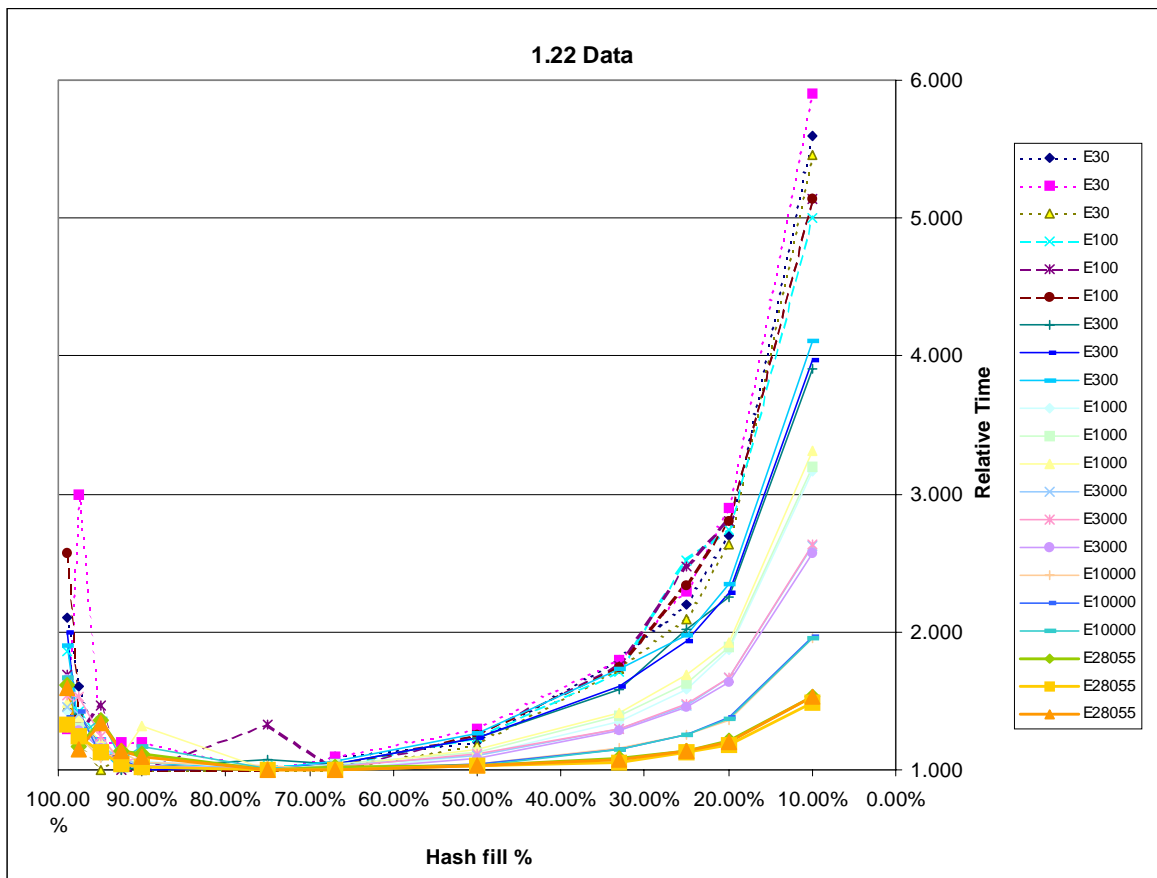


Figure 4.7: Runtime vs. Hash Table Fullness for Hashing Algorithm

When the table is too full, the runtime increases because the algorithm is spending more time searching for LHS's in the hash table. On average, a 99% full table results in search lengths an order of magnitude greater than a 75% full table. Using an even larger table such as a 10% full table can cut the average search length by another factor of 2, but requires 7.5x the memory.

When the table is too empty, it consumes large amounts of memory. On the first two example sets, a 10% full hash table for the entire example set wouldn't fit into the 160MB available physical RAM. With the full second example set, tables below 75% full wouldn't fit into memory. A larger example set with more LHS's might not be able to be processed by the hashing method in 160MB at all. Even when the table does fit into memory, the time required to empty it limits the speed of the algorithm for large tables. This effect is more pronounced when there are fewer examples in the input set.

The optimal hash table size falls where the table ends up 75% full. This has the best tradeoff of memory consumed vs. search time, and thus the best overall runtime. Since the runtime grows more slowly towards emptier tables than it does towards fuller tables, it is best to err on the side of too large a table than too small a table. That also avoids the problem of too small a table filling up completely.

In Figure 4.7, some of the curves have spikes in them. These spikes are caused by uneven hash table filling. This is another weakness of the hashing method. If the hashing function does not evenly spread values across the table, part of the table may fill up and cause longer average search lengths for that run. This does not prevent the algorithm from finishing, but may degrade performance on some runs.

3.2. 7-Segment LCD Data – Using Random Example Data

In order to generate performance statistics for the rule generation algorithms, it was necessary to obtain a number of extremely large datasets with varying properties. However, large datasets of sufficient complexity are difficult and expensive to obtain. One way around this is to use datasets generated algorithmically from some model. A program encoding this model can be used to generate datasets of any size relatively easily.

One model which has been used previously is the 7-segment LCD problem. This has seven inputs on the LHS, representing the seven LCD segments. On the right-hand side is the digit represented by the segments. To make the problem non-trivial, the examples are corrupted by 10% random noise (a 10% chance that any given segment is flipped from off to on or vice versa). This dataset was used by Goodman, et al. (1992) in their work on developing the ITRULE algorithm for classification.

Example sets from between 50 – 1,000,000 examples were generated using that algorithm, and run through both rule generation algorithms.

We then decided to see whether the performance of the algorithms would differ if run on example sets completely corrupted by noise. Our hypothesis was that the performance should not change significantly. Both the real and random datasets were big enough to generate all possible rule LHS's (only 1610 LHS's are possible if rules are limited to five or fewer LHS attributes), so the size of the hash tables would be equal. This means memory usage would be equivalent between the real and random data. While the time spent in SpanRULE in emptying the RHS distribution for each LHS would be

slightly more efficient for the random data, due to the MINEXAMPLES and MINCORRECT cutoffs, that represents only a small fraction of the total runtime.

The data from these the runs is shown in Figure 4.8.

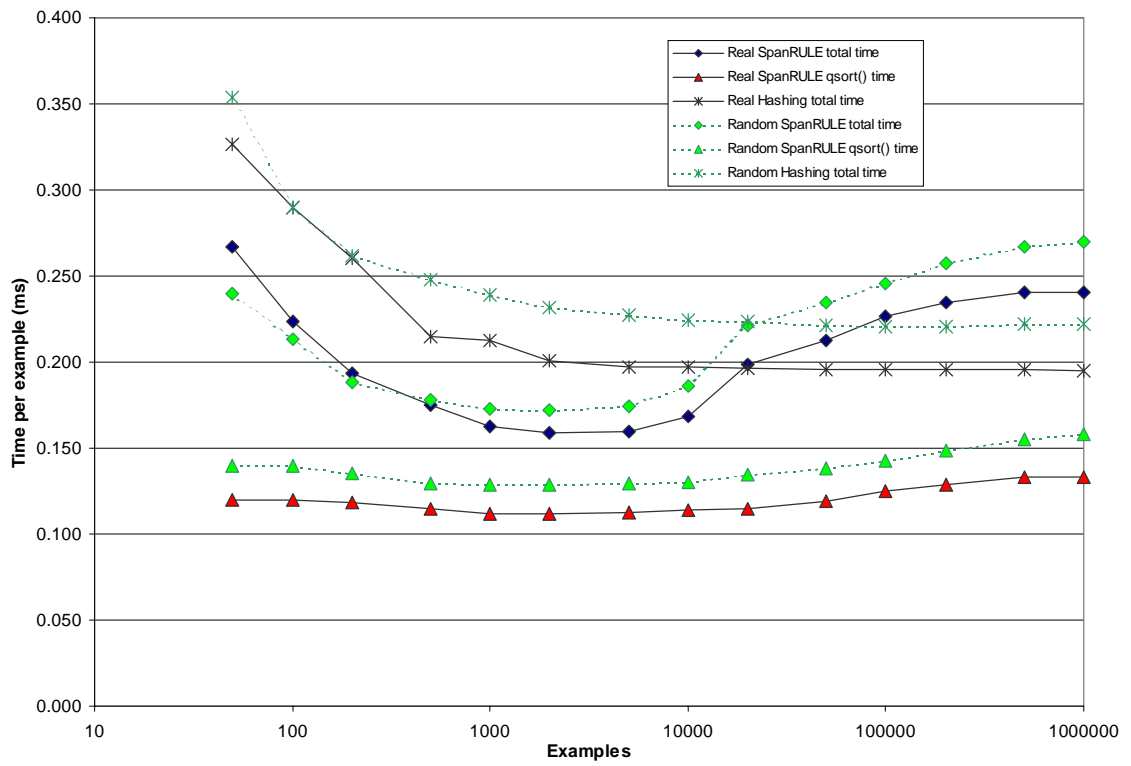


Figure 4.8: Comparison of Runtimes From Real and Random LCD Data

For SpanRULE, the runs on random data varied from 90% - 112% of the runtimes for real data. Runs with few examples were the only ones where the algorithm was faster on random data. Virtually all of this time increase seems to be spent in the quicksort. Since the LHS's for random data have an even distribution, they represent one of the worst cases for the quicksort algorithm. This is reflected in quicksort times 113% - 119% of the quicksort times for real data.

For hashing, the runs on random data varies from 100% - 114% of the runtimes for real data. Virtually all of this time increase seems to be spent in filling the hash table. The cause appears to be that the hashing algorithm had a longer average search length to find a matching entry or empty hash entry. For real data, the average hash search length was 1.49; for the random data, the average hash search length was 1.83. Again, the difference in the LHS distributions between real and random data was likely the cause. For real data, the more likely LHS's would also be more likely to be found in the hash table after a shorter search; this would reduce the average search length. For random data, the LHS's would be evenly distributed and so there would be no LHS's which would benefit more than the others from having a shorter-than-average search length.

Aside from those differences, neither algorithm showed a significant behavior change between real and random data. Thus, it seems plausible to further examine the behaviors of the algorithms by generating a series of different random example sets with different parameters. This allowed isolating a number of different factors which affected the performance of the algorithms. These factors are examined in the following sections.

3.3. Performance vs. Number of Examples

To examine the performance of the algorithms vs. number of examples, several test example sets were generated. These had 7 LHS attributes, with between 2 – 11 values per attribute, and a RHS attribute with 10 values. Example sets were generated with between 50 – 1,000,000 examples. The varying number of values per LHS attribute simulates example sets of varying complexity.

For runtime, both algorithms had a strong linear dependence on the number of examples. To view the underlying trends, runtime is graphed as runtime *per example*.

Since the quicksort upon which the SpanRULE algorithm depends scales as $N \cdot \log(N)$, we would expect to see SpanRULE's runtime have some $\log(N)$ dependence. The hashing algorithm should scale more linearly, as long as memory for the hashing table is available.

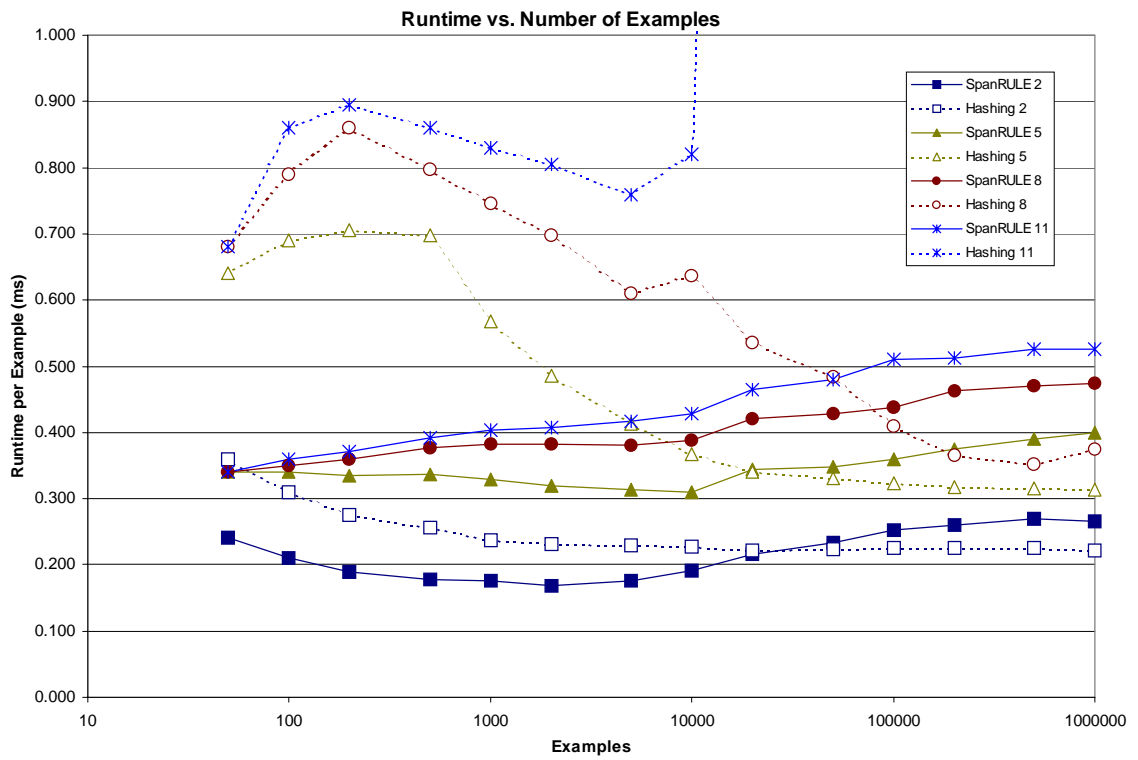


Figure 4.9: Runtime vs. Number of Examples

Figure 4.9 shows the results of these runs. For all the example sets, SpanRULE is more efficient for small numbers of examples. The more complex the set, the larger the number of examples before hashing is faster than SpanRULE. At the most complex set (11 values per LHS attribute), the hashing algorithm fails to complete when run on example sets containing over 50,000 examples – the possible rules LHS's for those runs did not fit in a 160MB hash table. For this set, SpanRULE is always more efficient.

As the number of examples gets larger, the hashing algorithm gets more efficient (when it is able to complete). However, it is never even twice the speed of the SpanRULE algorithm. The SpanRULE algorithm gets gradually more inefficient as the number of examples increases, but this is an extremely gradual inefficiency (less than a factor of two as number of examples grows from 50 to 1,000,000).

As expected, the memory required for the SpanRULE algorithm grows linearly in the number of examples.

3.4. Performance vs. Number of Potential Rule LHS's

More complex data sets should impact hashing more than SpanRULE, since the former needs to maintain a larger hash table to keep track of all of the potential rule LHS's. This was tested using the example sets from the previous section. While holding the number of examples constant, the number of values per LHS attribute was varied from 2 – 11. The runtime for each run was graphed against the number of rule LHS's considered for that run. Figure 4.10 shows this data.

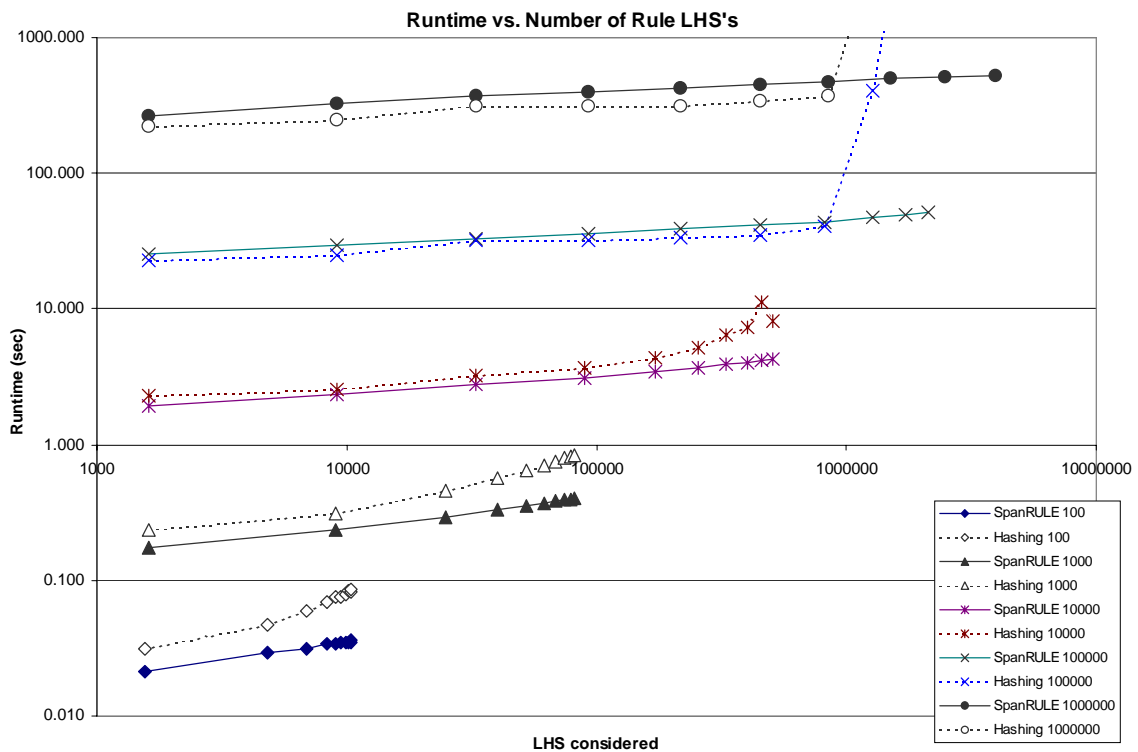


Figure 4.10: Runtime vs. Number of Rule LHS's

For the three smallest series, SpanRULE is always slightly faster. As the number of potential rules increases, hashing runtime also increases faster than SpanRULE runtime. For the larger two series (100,000 and 1,000,000 examples), SpanRULE is slightly slower. However, hashing was unable to complete the more complex runs in those series, while SpanRULE was able to complete them without seriously decreased performance.

As expected, the memory required for the hashing algorithm grows linearly in the number of potential rule LHS's.

3.5. Performance vs. Number of RHS Values

To examine the performance of the algorithms vs. the number of values the RHS attribute could take, several additional example sets were generated. These had between 50 – 1,000,000 examples, and either 2, 5, 8, or 11 values per LHS attribute. Sets were generated with 2, 10, 100, and 1000 values for the RHS.

Since the hashing algorithm needs to hold all RHS distributions for *all* the potential rule LHS's, its memory usage scales linearly with the number of RHS values. SpanRULE should not be as severely impacted by the number of RHS values, since it only needs to examine the distribution for one LHS at a time.

The maximum available space for the hashing method's hash table was 160MB. With 2 values per RHS, this allowed 2,000,000 potential rule LHS's to fit into the table – enough for all but the largest and most complex 2-RHS runs. However, with 1000 values per RHS, only 40,000 LHS's fit into the table. That only allowed a small fraction of the

1000-RHS runs to complete using the hashing method. The SpanRULE method was able to complete all of the runs.

As the number of RHS values increased, the runtimes slowed somewhat for both algorithms. This was due to additional time required to search the RHS distribution for each LHS for nonzero elements.

3.6. Performance vs. Number of LHS Attribute Combinations

Both algorithms need to examine every LHS attribute combination for every example. However, the SpanRULE algorithm also has the overhead of (# of combos) quicksorts, traded off against the hashing algorithm's overhead of needing to loop through the combinations once per example.

Since runtime is strongly dependent on the number of possible rules, several example sets were generated with different numbers of LHS attributes and values per attribute according to the following formulas:

$$(\# \text{ LHS attributes}) + \log_2(\text{values per LHS attribute}) = 12$$

$$(\# \text{ LHS attributes}) + \log_3(\text{values per LHS attribute}) = 12$$

Using these formulas, example sets were generated with 2, 3, 4, or 6 LHS attributes (and thus 64, 16, 8 or 4 values per LHS attribute for the top formula and 729, 81, 27, or 9 values per LHS attribute for the bottom formula). By using these formulas, and setting the max rule order to 6, the number of potential rule LHS's for a set of examples spanning the input space should be constant for each formula. Example sets had between 50 – 1,000,000 examples.

For both algorithms, runtime scales with the number of LHS attribute combinations. However, the ratios of the runtimes for the hashing and SpanRULE algorithms does not show that one is significantly more efficient than the other as the number of LHS attribute combinations increases.

3.7. Performance vs. Fraction of Unknown LHS Values

Some real-world data sets, especially medical databases, are somewhat sparse, containing a large fraction of unknown values. For these data sets, large numbers of examples may be required to get accurate statistics from which to generate rules, since an example cannot contribute to a rule unless all of the LHS attributes for that rule have known values in the example.

The example sets used in this section had 7 LHS attributes, with between 2 – 11 values per attribute, and a RHS attribute with 10 values. Example sets were generated with between 50 – 1,000,000 examples. The examples were then corrupted with 10%, 20%, 40%, and 80% unknown values.

The hashing method examines each attribute combination for an example at the last stage before adding that example's contribution to the hash table. Gains in speed for the hashing method should be due to needing to update the hash table fewer times.

The SpanRULE method examines the attribute combination for each example when calculating the hash value for that example and combination. This happens before the hash values are quicksorted. If the example set has a large number of unknown values, this reduces the number of examples which must be quicksorted and processed further.

Relative runtimes for the two algorithms for 10% - 40% unknowns were roughly comparable to runtimes without unknown values. For the 80% unknowns runs, SpanRULE gained a 40% speed increase over hashing. This is likely because for unknown examples it could skip over more processing than the hashing method could.

3.8. Memory Usage

As expected, the memory required for the SpanRULE algorithm grows linearly in the number of examples. The most memory required was 38.2MB, for the 1,000,000 example run with 1000 RHS values.

SpanRULE should be able to use virtual memory (disk space) to hold the example list, since it only needs to read down the example list (# LHS combos) times to calculate the hash values. Sorting of the examples is done using the calculated hash values, which require much less memory and so can be held in real memory. Since the example list is read in order, there will not be excessive page swapping as the example list is cycled into memory.

As expected, the memory required for the hashing algorithm grows linearly in the number of potential rule LHS's and linearly with the number of RHS values. The most memory required would have been 20378MB for the 1,000,000 example run with 1000 RHS values. However, that was far more than the 160MB available for running the rule generation benchmark.

Hashing is not able to use virtual memory (disk space). If the hashing algorithm is good, it will evenly fill the hash table. However, this means that each hash table access is essentially at a random location in the table. Random access is the worst possible case

for virtual memory usage, since there is no way to gain advantage by holding frequently accessed pages of memory in real memory (since all pages are accessed randomly and evenly).

4. Conclusions

4.1. Hashing Algorithm

The hashing method requires a good hashing function. Nonetheless, in a non-trivial fraction of the runs, even a relatively good hashing function fills the hash table, greatly impacting runtimes. If the hashing function fills the table unevenly, performance can be far worse even for small datasets and large hash tables.

The hashing algorithm performs optimally when the hash table is sized so that after adding all the potential LHS's the hash table is 75% full. If the hash table is barely big enough (almost 100% full), runtime is greatly affected by the time spent searching for hash table entries when filling the table. If the hash table is too small (over 100% full), the algorithm will completely fill the hash table and halt. If the hash table is too large (less than 25% full), the overhead required to empty the hash table slows the algorithm considerably. Unfortunately, there does not seem to be a good way to determine ahead of time what size hash table will be required.

The memory required by hashing scales linearly with the number of potential rules and with the number of RHS attributes. If there are a large number of potential rules, or if the RHS has many possible values, the hashing algorithm's performance suffers because of increased memory consumption for the hash table. Hashing does not

require any memory per example. Thus, the hashing algorithm is most efficient for large example sets which generate a relatively small number of rules.

4.2. SpanRULE Algorithm

The SpanRULE algorithm performed well in all scenarios. Its runtime was not severely impacted by varying any of the parameters. Although its performance did not scale as well as hashing for large number of examples, the decrease in performance for large example sets was not major (less than a factor of two over five orders of magnitude of example set size).

The memory required by SpanRULE scales with the number of examples, since it requires holding the entire training set in memory at once. This might present a problem for running SpanRULE on extremely large datasets (tens of millions of examples). Partially, compensating for this limitation, the memory required by SpanRULE *is* known before the algorithm is run – allowing the user to know ahead of time whether there is sufficient memory for the algorithm to complete. Furthermore, SpanRULE requires virtually no additional memory as the number of RHS values increases.

4.3. Comparison

For small datasets, the performance of the two algorithms is roughly comparable.

For large datasets of low complexity (many examples, few potential rules), hashing is up to twice as fast as SpanRULE, since its runtime per example is unaffected by the number of examples.

For datasets of high complexity (many potential rules or many RHS values), hashing is often unable to finish at all, because its memory consumption scales with both

those quantities. For example sets with hundreds or thousands of RHS values, hashing is unable to run except with the simplest datasets. The SpanRULE algorithm was always able to finish, regardless of the complexity of the dataset.

Some data sets have a high fraction of values which are unknown. SpanRULE gains a performance benefit for those data sets, since for a given combination of LHS attributes it only needs to hash/quicksort the examples which have no unknown values for those attributes. Hashing also gains some benefit (since there are fewer combinations of attributes for each example for which all the values are known), but not as significant a benefit as SpanRULE.

4.4. Hybrid Algorithms

It seems natural to examine whether there might exist a hybrid algorithm somewhere between pure hashing and pure SpanRULE. Such an algorithm would combine the advantages of both existing algorithms. When both existing algorithms are able to finish, their runtimes are roughly comparable; the only times where an algorithm is unable to finish is when it runs out of memory. For a hybrid algorithm to offer any advantage, it will thus need consume less memory.

The advantage of hashing is that it only needs to look at each example once, so it can be used on very large example sets. The downside is that it needs enough memory to hold all possible rule LHS's and their respective RHS's in its hash table.

The advantage of SpanRULE is that it examines one potential rule LHS at a time, so it does not require large amounts of memory to hold many LHS and RHS distributions.

The downside is that it needs enough memory to hold all the examples, since it must examine and sort the example list once per LHS attribute combination.

Assume that the example list fits in memory. Why not empty the hash table more frequently, so that it doesn't need to be as big?

One possible hybrid algorithm might through the examples once per rule order and empty the hash table after each rule order. The problem here is that most of the potential rules (on the order of 80%-90%) are of the highest rule order allowed. Emptying the hash table saves only 10-20% on memory, but requires (rule order) passes through the example list to fill the hash table and (rule order) passes through the hash table to empty it. This does not significantly reduce memory usage, and it removes the one real advantage the hashing method has (only looking at each example once).

Another hybrid could sift through the examples (# LHS combos) times in a manner similar to the outer loop of SpanRULE. Instead of qsort()'ing the potential rules, it could put them in a hash table. The problem here is that the number of rules for a given combination isn't known, and potentially could be large. To ensure the hash table doesn't overflow, we need to use a large hash table. But this introduces the overhead of emptying a large hash table (# LHS combos) times – which is very slow.

4.5. Summary

The SpanRULE method is a robust method for generating potential rules. It performs comparably fast to the hashing method, but does not have the hashing method's weakness of using an unpredictably large amount of memory. Hybrid algorithms utilizing features of both SpanRULE and hashing do not appear promising.

CHAPTER 5

Refining Raw Rules

The algorithms described above generate all possible rules that can be derived from the example data. Further processing is necessary to narrow that set of rules down to a sufficiently small set of the most valuable rules.

1. Subsumption Pruning

Subsumption pruning is an algorithm to remove those rules which contribute no new information to the ruleset. Subsumption pruning works in the following manner.

- Consider two rules A and B which predict the same right-hand side attribute and value.
- If rule B is of higher order than rule A (B has more attributes on its LHS),
- and if each attribute on the LHS of rule A is present on the LHS of rule B and has the same value as it does for rule A,
- and if rule B is correct a an equal or smaller fraction of the time than rule A,
- then rule B is removed from the ruleset.

The justification for subsumption pruning is as follows. Any time rule B fires, rule A will also fire. Both rules predict the same RHS value. Since rule B is not correct more of the time than rule A, rule B adds no new information. Therefore, we might as well remove it, and save the memory and CPU time.

Subsumption pruning must be done after any filtering and segmentation (see Section 5 below). If rule A in the previous example were filtered out, then we shouldn't have removed rule B and we have lost information.

2. Measures for Rule Weight

When two or more rules fire with conflicting results, some means needs to be available for determining which of the conflicting outcomes will prevail. For each rule, a weight is calculated using one of the following measures. This allows higher-weight rules to dominate lower-weight rules for a given situation.

2.1. Percent Correct

One of the simplest measures which can be used is to give greater weight to rules which are correct a greater fraction of the time. This weight is simply

$$W_{correct} = p(x/y)$$

This weight works well if there are only a few rules firing for each LHS. However, it works very poorly if many rules fire. It is readily apparent that using this weight, two rules which are each correct 41% of the time outweigh a rule which is correct 80% of the time.

2.2. Classification Weights

(Goodman et al., 1992) derived the following rule weight for classification:

$$W = \log\left(\frac{p(x|y)}{p(x)}\right)$$

This has been shown to be a good weight for classification.

2.3. Error-cost Weight

The previous rule weights are based on the assumption that all incorrect classifications are equally bad. However, as the psychophysics data below shows, this is

not the case. It is thus desirable to derive a weight which is based on the cost of making each mistake.

We define the cost of guessing x_1 when the answer should be x_2 as $C(x_2 \rightarrow x_1)$ – the cost of misclassifying an x_2 as a x_1 . We further require that the cost for a misclassified result is always positive. We define $C(x_1 \rightarrow x_1) = 0$; there is no cost for guessing the right answer. The misclassification cost table of $C(x_2 \rightarrow x_1)$ for all x_1, x_2 is specified as input to this algorithm.

We define the average cost for guessing x_1 as $C(x_1)$. This is

$$\sum_x p(x)C(x \rightarrow x_1)$$

Similarly, the average cost for guessing x_1 given we know y is $C(x_1|y)$. This is

$$\sum_x p(x|y)C(x \rightarrow x_1)$$

Given a rule of the form "IF y THEN x ", the simplest cost-based rule weight would be one of the form

$$C_{\max} - C(x|y)$$

Where C_{\max} is the highest cost in the misclassification table specified as input. This ensures that the weights are always positive. This first simple weight is roughly analogous to the percent correct weight above.

We can also define a rule weight as

$$C(x) - C(x|y)$$

This is the average reduction in cost given that the rule has fired.

Since cost is always positive, we could also define the weight as

$$\log\left(\frac{C(x)}{C(x|y)}\right)$$

or,

$$\log\left(\frac{1+C(x)}{1+C(x|y)}\right)$$

The (1+) parts would help keep the weights from exploding if $C(x|y)$ is very close to zero. The optimal form for this rule weight is dependent on the range of values present in the misclassification cost table, though the latter weight will be well-formed for any positive-valued misclassification cost function.

3. Measures for Rule Priority

The algorithms for rule generation discussed above generate many more rules than are practical to hold in memory or quickly analyze. It is thus important to order the rules by some measure of rule worth. This allows keeping some fraction of the rules which have the highest worth.

3.1. J-measure

The first measure of rule worth used was the J-measure from the ITRULE algorithm. (Goodman et al., 1992) All possible rules are considered and ranked by a measure of the information contained in each rule defined as

$$J(\mathbf{X}; \mathbf{Y} = y) = p(y) \cdot \left[p(x|y) \cdot \log\left(\frac{p(x|y)}{p(x)}\right) + (1 - p(x|y)) \cdot \log\left(\frac{(1 - p(x|y))}{(1 - p(x))}\right) \right]$$

This measure trades off the amount of information a rule contains against the probability of being able to use the rule. Rules are less valuable if they contain little

information. Thus, the J-measure is low when $p(x/y)$ is not much higher than $p(x)$. Rules are also less valuable if they fire only rarely ($p(y)$ is small), since those rules are unlikely to be useful in generalizing to new data. A plot of the J-measure is shown in Figure 5.1.

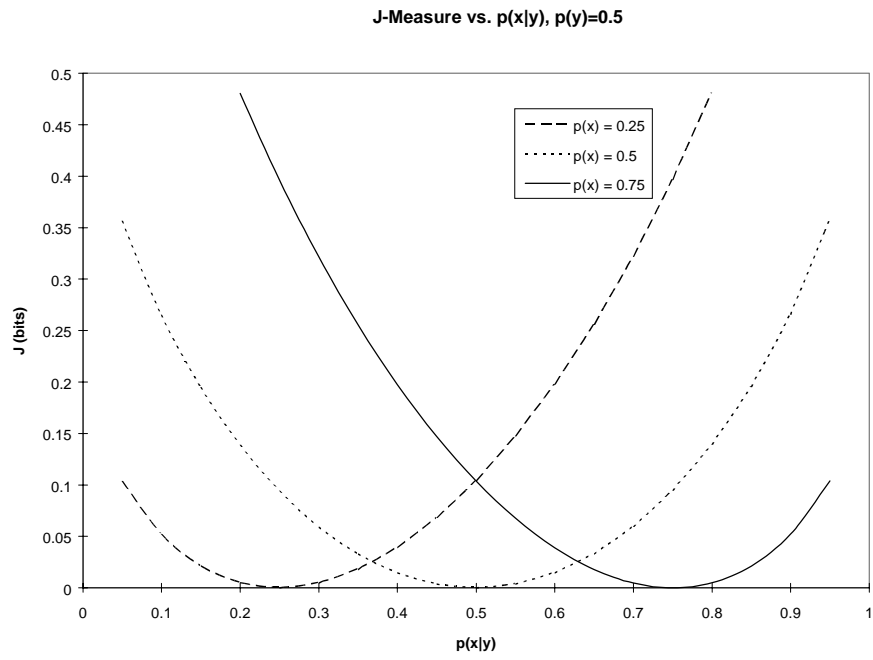


Figure 5.1: J-Measure vs. $p(x|y)$

3.2. Error-cost Measure

Given a misclassification cost table of the sort described in Section 2.3 above, it is possible to derive a measure of the average misclassification cost C_R of a rule:

$$C_R(X/Y = y) = \sum_{x_2} p(x_2 | y) \cdot C(x_2 \rightarrow x)$$

Note that if the misclassification cost is the trivial cost where $C(x_2 \rightarrow x) = 1$ for all $x_2 \neq x$ that the rule cost simplifies to:

$$C_R = 1 - p(x/y)$$

which is just the probability that the rule is incorrect. As with the J-measure above, it is desirable for the priority to take into account the frequency with which the rules fires.

We propose the following error-cost rule priority:

$$P_{EC} = p(y) \cdot [C_{MAX} - C_R(x | y)]$$

4. Independence Pruning

For the classification weight to work properly, all rules which are allowed to fire must be independent of one another. Otherwise, one good rule could be overwhelmed by the combined weight of twenty mediocre but virtually identical rules. To prevent this problem, each ruleset is analyzed to determine which rules are dependent with other rules in the same ruleset.

4.1. Definition of Rule Dependence

- 1) Consider two rules R_A and R_B which predict the same RHS and value.
- 2) Let \mathbf{A} be the set of examples for which rule R_A fires.
- 3) Let \mathbf{B} be the set of examples for which rule R_B fires.

4) Define the overlap O_{AB} as

$$O_{AB} = \frac{|A \cap B|}{|A \cup B|}$$

(the number of examples for which both R_A and R_B fire, divided by the number of examples for which either R_A or R_B fires.)

5) If $O_{AB} > 0.5$, the rules are dependent.

For each rule, the algorithm maintains a list of lower priority rules which are dependent with the rule. This list is used in real time independence pruning (see below).

It would seem at first that it would be easiest to remove all dependent rules at the time a ruleset is created. However, this actually degrades the quality of the ruleset. As an example, assume a ruleset containing only the following two rules, and assume the rules are dependent:

IF $A_1 = a_{1,2}$ THEN $A_{RHS} = a_{RHS,3}$ with priority 0.013
 IF $A_2 = a_{2,5}$ THEN $A_{RHS} = a_{RHS,3}$ with priority 0.009

Now assume we are trying to inference A_{RHS} and that the value of A_1 is currently unknown. Only the second rule would be able to fire. However, if we removed the second rule at the time of ruleset creation, no rules would be able to fire and we would not be able to inference a value for A . We can avoid this problem by only independence pruning those rules which can fire for a given LHS. This method is discussed below.

4.2. Generation of Real-Time Dependency Information

As explained in the section above on generation of dependence data, all rules which fire for a given LHS should be independent. However, we cannot prune rulesets ahead of time to remove rules without losing information. The solution to this dilemma

is to keep track of which rules are dependent on other rules, and only allow rules which are still independent to fire. This algorithm is described below.

Start by allocating and zeroing an array F , where f_i is zero if rule R_i is allowed to fire. Then for each rule R_i in order of decreasing Priority,

- 1) If f_i is non-zero, the rule is not allowed to fire. Skip to the next rule.
- 2) If the rule can't fire, skip to the next rule.
- 3) The rule can fire. Add its weight to the weight for the RHS value it predicts.
- 4) For each rule R_j in the list of rules dependent with R_i , set the corresponding f_j non-zero.

This algorithm is very fast, since it requires only array lookups and does no complex calculations. In fact, it is faster than using the same ruleset without dependency information, since if a rule is forbidden from firing the program does not spend time determining if the rule is allowed to fire. (With no dependency information, all rules are checked to see if they can fire.)

5. Filtering and Segmentation of Rulesets

The measures described above are unable to take into account any additional *a priori* knowledge about the nature of the problem - for example, that harmony rules which use the current melody note as input are more desirable because they avoid dissonance between the melody and harmony.

Filtering is performed to force all rules in a ruleset to use a given attribute. This is done when it is known that a certain attribute is key to determining the RHS value for the ruleset. For example, it is very desirable for rules which determine the current chord

function to take into account the current melody note. Rules which do not contain the desired attribute are permanently removed from the ruleset.

A priori knowledge of this nature can also be incorporated by segmenting rulesets into more- and less-desirable rules based on the presence or absence of a desired LHS attribute such as the current melody note (Melody0). Rules lacking the attribute are removed from the primary set of rules and placed in a second "fallback" set. Only in the event that no primary rules are able to fire is the secondary set allowed to fire. This gives greater impact to the primary rules (since they are used first) without the loss of domain size (since the less desirable rules are not actually deleted).

Segmentation also has performance benefits. A large number of rules can be kept in the fallback rulesets without significant reduction in the speed of the inferencing engine using the rules, because those rules are only considered a small fraction of the time when the primary segment fails to fire. This is important in the application of musical rules to generate accompaniment in real-time.

CHAPTER 6

Perception of Harmonic Errors in Bach Chorales

This section discusses a psychophysics experiment performed to measure how listeners perceive errors in harmony. We discuss the background and need for this information. The design of the experiment is described. We conclude with an analysis of the experimental data.

1. Background

In developing the rule-based harmony generator, one shortcoming of the basic rule engine became apparent. Some musical errors in harmony are more readily perceived than others. For example, if a melody note that should be accompanied by a V7 chord (GBDF) is accompanied by a V chord (GBDG), the change is not likely to be noticeable, even to a musically proficient listener familiar with the piece. However, if a melody note which should be accompanied by a I chord (CEGC) is accompanied by a vii07 chord (BDF \flat), the change is easily noticeable even to someone with no musical training or knowledge of the piece of music in question. Therefore, it is worse to make the mistake of replacing I with vii07 than it is to replace V7 with V.

To the rule engine, "V" and "V7" and "I" and "vii07" are all just character string tokens. At each point in harmonizing a melody, the rule engine attempts to choose the correct token based on the input data provided to it. The input data is also in the form of a list of string tokens, for melody note, previous chord, etc. If the rule engine chooses the correct harmony token for a set of inputs, this is labeled correct, and if another token is chosen, this is labeled incorrect. All incorrect tokens are treated as equally bad, because the rule engine has no way of distinguishing which mistakes are more noticeable than others to human listeners.

A system for automated harmony which *could* take into account which errors are more readily perceived should thus sound better than a naïve system which treats all errors as equally bad. This system would be designed so that the mistakes it tends to make are of the less noticeable variety. While it might have a higher percentage of total errors in the statistical sense, it would have a much lower percentage of *perceived* errors. Studies of professional musicians (Repp, 1991) show that the majority of their technical errors are less noticeable because they are harmonically plausible in the context of the surrounding music, so there is justification for seeking a computer algorithm which would produce comparable results.

To design such a system, it is necessary to first collect data on how listeners perceive musical errors. Then the system can be designed to minimize the incidences of errors which are more often perceived. The following experiment attempts to build a table of knowledge about how listeners perceive errors in harmonic function.

2. Experiment Overview

2.1. Desired Output

The desired output of this experiment is a misclassification cost table. This is a table

Cost(correct chord function A, corrupted chord function B)

which represents the cost of playing chord function B when chord function A should have been played. This table can be used in the development of a new rule algorithm.

2.2. Definition of Test Examples

100 harmonized Bach chorales were used as input. These were broken up into phrases. A phrase consists of a series of chords ending on a fermata. This yielded 585

phrases. Of these, phrases shorter than 4 chords or longer than 20 chords were discarded. The shortest phrases do not provide sufficient harmonic context in which to measure perception of errors. Using the longest phrases would slow down testing time, since they were 11-30 seconds in length, as opposed to 4-10 seconds for phrases between 4-20 chords long. Removal of the shortest and longest phrases left 553 phrases.

A test example consists of a phrase with a single erroneous chord in a random position. The error is not placed in the first two chords; errors there are less noticeable because the harmonic context of the phrase has not been established (Thompson, 1993). The error is created by replacing the function for that chord with another function. The bass, alto, and tenor rules from (Spangler, Goodman and Hawkins, 1998) are then fired to determine the voice positions for that replacement chord.

2.3. Distribution of Test Examples

Of the 23 different chord functions present in the phrases, 5 were not present in a sufficient number of phrases to be significant. The 18 remaining functions were divided into two groups:

Group A: I, V, V7, IV, vi, ii, V7/V, vii07, iii, V/V, I7

and

Group B: V7/ii, V7/vi, V/vi, vii07/V, V/ii, IV/IV, v

Group A consists of chords in common usage, and the secondary dominants of the V and IV chords. Group B are less common chords.

The strength of the beat an error occurs on should contribute a great deal to how strongly that error is perceived. There are 4 different beat strengths used in the extended

figured bass representation: not on beat (n), unaccented beat (un), accented beat (ACC), and fermata (FERM). To examine the effect of beat strength, we generated examples of each misclassification at each beat strength.

There were thus $18 \times 4 = 72$ possible combinations of source chords and beat strengths. However, not all of these combinations were present in the input phrases; for example, there are no vii07 chords present on fermatas. This reduced to 69 the number of available input combinations. For each input combination, there were 17 chords it could be corrupted to, plus the trivial case where the chord was corrupted to itself. This yields $18 \times 69 = 1242$ combinations of inputs and outputs.

To obtain more accurate data on the more commonly occurring chords, the system was weighted to generate more examples of misclassifying the more common chords in group A. Examples were generated using the following proportions per beat type:

- For a chord not even reharmonized (original music), 2 examples.
- For a chord misclassified to itself, 2 examples.
- For a chord in group A misclassified to another chord in group A other than itself, 2 examples.
- For all other chord pairs, 1 example.

Chords which were misclassified to themselves were still re-voiced by firing the bass, alto, and tenor rules. This provided a control group which would allow separation of the increased noticeability of corrupting a chord from the increased noticeability of re-phrasing a chord.

2.4. Generation of Test Examples

To generate the test examples, the input phrases were arranged in random order in a list. The quota of examples to generate was placed in a 3-dimensional array QUOTA with indices for (beat type), (correct function), (corrupt function). Each element in that array contained the number of examples left to generate with that beat type and chord functions, initially set from 1 to 5 based on the proportions specified above. The list of phrases was then scanned repeatedly with the algorithm in Figure 6.1.

This was done repeatedly until fewer than 5 new examples were generated on a complete pass through the phrase list. This cutoff was employed to ensure that there were not a large number of examples generated from a few phrases that happened to contain some of the less common chords. Generation of a batch of examples thus usually halted after 10-15 passes through the phrase list, after generating 94% to 97% of the total example quota.

Each example generated was logged by example number, then written to a filename consisting of 3 random letters and the 5-digit example number. Playing the files in alphabetical order would thus play the examples in random order. Furthermore, no information about the nature of the corrupt chord or the source phrase could be determined from the example filenames.

This process was entirely automated, allowing a separate set of examples to be generated for each subject.

```
For PH in PHRASES

    Start at random offset in phrase and scan entire phrase
    (wrapping around at the end, skipping the first 2 chords)

    For each chord's corresponding beat type BT and function
    CF, see if there are nonzero entries for corrupt chord in
    the column QUOTA[BT][CF] of the quota array.

    If there are

        Choose one corrupt chord type CC at random.

        Decrement QUOTA[BT][CF][CC].

        Alter the chord function at the current position in
        the phrase from CF to CC.

        Fire the Major8 Bass, Tenor, and Alto rules to
        determine voice positions for the chord.

        Write the altered phrase to a new example file.
```

Figure 6.1: Algorithm for Generating Psychophysics Examples

2.5. Test Procedure

The testing program was written to work under Windows 95, using the media player built into Windows to play MIDI files. Each subject was given a copy of the testing program, with their own set of 1000 randomly generated examples. When the subject pressed the spacebar, the program played a MIDI file of the next example. The subject could press the spacebar to hear the example again, or could press a number between 1 and 5 to indicate how noticeable any error was, where 1 indicated no noticeable error and 5 indicated a very noticeable error. The subject could also enter comments into the program at any time. The test procedure was self-paced. Subjects were encouraged to do 100-200 examples per sitting, using headphones if available. If the subject exited and re-entered the test program, testing continued with the next example. If the subject completed their batch of examples, testing continued with the first example of the batch. The test program recorded the following for each example:

- time/date
- subject name
- example filename
- number of times subject played example before entering result
- result (from 1-5)
- any comments

Subjects were asked whether they played a musical instrument or had education in music theory. However, this information was not used, except to verify that a group of

subjects with varying musical training was present. Studies by Brand (1981) have shown that amount of musical knowledge does not affect perception of musical errors.

3. Results

Five users were tested, each with a unique set of examples. This yielded 4394 data points (one user did not complete the test). The average perceived errors for chord function errors are shown in Table 6.1. In the 3560 examples where errors were present, subjects reported an average magnitude of 2.65. In the 690 examples where no harmonic errors were present but the original chords were revoiced with the bass/alto/tenor rules, subjects reported an average magnitude of 1.82. In the 144 examples of uncorrupted Bach chorales, subjects reported an average magnitude of 1.24.

Table 6.1: Average Perceived Errors for Chord Pairs

	I	I7	V/V	ii	V7/V	iii	V/vi	V7/vi	IV	vii07/V	V	v	V7	V/ii	vi	V7/ii	IV/IV	vii07
I	1.4	2.6	3.3	2.2	2.3	1.5	2.0	1.5	2.4	5.0	2.9	2.0	2.6	2.6	1.9	2.5	3.3	3.8
I7	1.6	2.2	3.5	2.1	2.8	2.0	3.8	2.5	2.5	4.0	2.4	2.6	2.8	2.5	2.6	3.4	1.7	3.6
V/V	1.7	4.1	1.5	2.0	2.2	2.8	1.8	4.5	3.0	3.5	2.4	2.2	3.2	2.8	2.6	1.0	2.7	3.4
ii	1.7	2.5	2.7	1.7	3.2	2.4	2.7	2.3	2.2	4.0	1.9	2.6	2.2	2.6	2.9	3.0	1.0	2.5
V7/V	2.5	3.1	1.6	1.6	1.6	2.8	5.0	3.0	2.0	3.2	2.5	2.0	2.4	4.0	1.6	2.3	3.0	3.4
iii	1.9	3.5	2.9	2.3	2.8	2.1	2.3	1.6	2.9	2.4	3.0	2.5	2.6	3.4	2.8	2.7	3.7	4.0
V/vi	2.3	4.0	1.5	1.6	3.3	1.3	1.6	1.2	2.8	4.8	1.8	2.0	3.2	3.4	1.7	3.3	4.0	1.8
V7/vi	2.4	3.6	2.5	2.1	3.0	1.5	2.2	1.6	2.9	1.7	2.5	2.5	2.0	4.0	1.5	3.2	3.3	2.3
IV	2.1	3.1	3.2	1.7	3.2	2.0	3.5	2.5	1.5	3.0	1.8	4.0	2.8	2.8	2.6	2.0	3.5	3.1
vii07/V	2.0	3.0	1.8	2.0	1.0	2.0	3.0	3.0	2.6	2.8	3.0	1.0	2.5	3.5	2.0	1.6	3.3	4.0
V	2.2	3.7	2.3	2.2	2.1	1.9	3.4	1.8	2.0	3.9	1.4	1.7	1.9	2.6	2.3	2.0	2.5	3.2
v	2.5	2.8	2.0	2.4	2.7	2.0	3.5	2.8	3.6	3.8	2.3	1.6	2.8	3.5	2.0	3.3	3.0	4.0
V7	1.5	2.8	1.2	2.2	3.0	2.3	3.2	5.0	2.1	3.9	1.8	2.8	2.2	1.0	2.7	4.3	2.6	3.7
V/ii	2.7	3.7	2.3	2.6	4.0	2.8	3.5	2.4	2.3	4.2	3.4	2.7	3.3	2.1	2.5	2.3	2.0	4.2
vi	3.0	3.3	2.8	1.7	2.8	2.9	3.2	1.7	2.5	3.7	2.4	2.0	2.2	2.2	1.6	2.0	3.0	2.5
V7/ii	1.0	4.0	1.5	2.2	3.0	2.8	1.0	1.7	2.8	5.0	4.0	3.0	3.2	1.7	2.0	1.8	1.6	4.0
IV/IV	4.0	3.5	5.0	2.0	3.8	3.0	4.0	4.0	2.5	3.8	3.0	1.4	3.3	2.3	2.6	3.4	1.6	3.3
vii07	2.6	2.9	2.7	2.0	2.7	1.3	2.4	3.0	2.4	3.0	1.5	1.5	2.0	4.3	2.7	3.3	4.0	2.8

Table 6.2: Average Perceived Error vs. Beat Strength

Beat Strength	Examples	Average Perceived Error
N	852	2.32
un	1412	2.35
ACC	1390	2.45
FERM	612	3.23
(overall)	4250	2.51

3.1. Perceived Magnitude vs. Beat Strength

We expected to see that errors on stronger beats would be more noticeable because notes on those beats are harmonically more significant in a piece of music. Data from the experiment confirms this prediction. This is shown in Table 6.2.

3.2. Errors are Non-Symmetric

For two chords A and B, it is *not* necessarily the case that the error of misclassifying A→B is the same as misclassifying B→A. This can be readily seen in the table above; the table is not symmetric. This complicates efforts to map out the distances between two chords (since the distance from A→B is not the distance from B→A). Similar asymmetries in harmony have been reported by Krumhansl et al. (1982).

This is also not surprising given the generally accepted rules of chord transitions from music theory (Ottman, 1989). If the replacement chord forms a valid chord transition, the error would not likely be as noticeable. For example, replacing the middle chord in the progression V/V→vii07/V→V with a I chord to form V/V→I→V still forms a valid sequence. Replacing the middle chord in the progression vi→I→IV with vii07/V to form vi→vii07/V→IV does *not* form a valid sequence. There are far more places where vii07/V can be replaced with I than vice versa, which matches the experimental result that replacing vii07/V with I has a perceived error of 3.0 vs. 5.0 for the reverse replacement.

4. Summary

This experiment produced a table of the noticeability of errors in chord function. This table can be used with the error-cast measures for rule worth and weight from Chapter 5 to minimize the noticeability of errors generated from those rules.

CHAPTER 7

A Rule-Based System for Real-Time Harmony

This chapter describes a rule-based system for real-time harmony generation built using the techniques developed in the preceding chapters, including ruleset segmentation, real-time dependency pruning, and error-based rule measure.

1.1. Input Data

This algorithm was trained on a set of 100 harmonized Bach chorales. These were translated from MIDI format into extended figured bass using accent-based conversion (see Chapter 2, Section 6.2) and a window which included the current timestep and the previous two timesteps. This produced a set of 7630 training examples.

1.2. Rule Generation

Rulesets were generated for each attribute using the SpanRULE algorithm with the following settings:

- Maximum Rules per Set = 2048.
- Maximum Rule Order = 5.
- Minimum Fraction Correct = 0.3.
- Minimum Rule Priority = 0.001, using J-measure priority.

The generated rules were weighted using the classification weight. The Function0 rules were also tested using the error-cost rule weight

$$\log\left(\frac{1 + C(x)}{1 + C(x | y)}\right)$$

from Chapter 5, Section 2.3 and the misclassification cost table generated by the psychophysics experiments in Chapter 6 (Table 6.1). The diagonal of the misclassification cost table was set to zero for calculation of error-cost, due to the

requirements of the error-cost rule weight. Another set of rules was also generated using both the error-cost rule weight and the error-cost rule priority.

Rulesets were then segmented into the sets shown in Table 7.1 and subsumption pruned, and real-time dependency information was generated. Note that all of the segments for the Function0 rules contain the current melody note (Melody0). In this case, requiring all rules to use an attribute (Melody0) did not reduce the size of the input domain; all examples from the testing set were still fired on by at least one of the remaining rules.

1.3. Testing

These rulesets were tested on 1721 examples derived from 27 chorales not used in the training set. Table 7.2 shows the fraction of examples correctly inferred for each ruleset before and after segmentation. Also shown is the average number of rules evaluated per test example; the speed of inferencing is proportional to this number.

To determine whether segmentation was in effect only removing lower priority rules, we generated a second unsegmented ruleset for each attribute, consisting of the highest priority rules. This second set was limited in size so that it had the same average number of rules evaluated per test example as the segmented ruleset. This set is denoted as "Unsegmented #2" in the table.

To determine whether the error-cost rule measure was reducing the noticeability of errors in the output of the rulesets, the total perceived error for the test examples as inferred by the rulesets was summed for both the classification weights and the error-cost weights.

Table 7.1: Ruleset Segments

RHS Attribute	LHS Attributes	Required LHS Attributes For Segment	Rules
Function0	Melody0, Melody1, Function1, Bass1, Melody2, Function2	Melody0, Function1, Function2	180
		Melody0, Function1	386
		Melody0	326
Soprano0	Melody0, Function0	Melody0, Function0	74
Bass0	Function0, Soprano0, Function1, Bass1	Function0, Soprano0	125
		(none)	182
Alto0	Function0, Soprano0, Bass0, Function1, Alto1	Soprano0, Bass0	267
		(none)	533
Tenor0	Function0, Soprano0, Bass0, Alto0, Function1, Tenor1	Soprano0, Bass0, Alto0, Function0	52
		Soprano0, Bass0, Alto0	164
		(none)	115

Table 7.2: Ruleset Performance

Attribute	Ruleset	Total Rules	Avg Rules/Example	Correct
Function0	Unsegmented	1721	1721	50.4%
	Segmented	892	617	50.8%
	Unsegmented #2	617	617	45.7%
Soprano0	Unsegmented	74	74	94.6%
Bass0	Unsegmented	307	307	70.1%
	Segmented	307	162	70.5%
	Unsegmented #2	162	162	65.5%
Alto0	Unsegmented	800	800	63.5%
	Segmented	800	275	63.3%
	Unsegmented #2	275	275	59.3%
Tenor0	Unsegmented	331	331	73.6%
	Segmented	331	180	74.4%
	Unsegmented #2	180	180	67.0%

Table 7.3: Comparison of Information Theoretic Measures with Error-Cost Measures

Priority Measure	Weight	Examples Correct	Avg Cost/Error	Avg Cost/Example
J-Measure	Classification	50.8%	2.34	1.15
J-Measure	Error-Cost	51.6%	2.25	1.09
Error-Cost	Error-Cost	53.1%	2.25	1.06

Table 7.4: Some Traditional Music Theory Rules Found in Rulesets

Harmony rules, filtered to always use Melody0 and Function1:						
1.	IF	Melody0 E	THEN	Function0 I	0.83	0.89 0.0601
		AND		Function1 V		
	The strongest cadence (ending) in classical harmony (G Major → C Major)					
3.	IF	Melody0 F	THEN	Function0 IV	0.98	3.12 0.0499
		AND		Function1 V		
	Another common transition (G Major → F Major)					
Bass rules, filtered to always use Function0:						
1.	IF	Function1 V	THEN	Bass0 B1	0.98	1.59 0.0255
		AND		Function0 IV		
	Combined with rule 3 above, always places the V→IV transition in first Bass					
3.	IF	Function1 V	THEN	Bass0 B0	0.86	0.20 0.0179
		AND		Function0 I		
	Combined with rule 1 above, always places the V→I cadence in root position (the strongest position for an ending chord)					
26.	IF	Function0 vii07	THEN	Bass0 B1	0.53	0.17 0.0098
	Always places diminished 7th chords (GBDF) in first inversion (in classical harmony, diminished 7th chords are always placed in inversion). This rule is of lower priority because diminished 7th chords do not appear very often.					

1.4. Analysis

The generated rules for harmony have a great deal of similarity to accepted harmonic transitions (Ottman, 1989). Examples of these rules are shown in Table 7.4. Most of the common harmonic transitions listed in Ottman are present in the rulesets.

In all cases, segmenting the rulesets reduced the average rules fired per example without lowering the accuracy of the rulesets (in some cases, segmentation even increased accuracy by up to 4%). Speed gains from segmentation ranged from 80% for Tenor0 up to 279% for Function0. In comparison, simply reducing the size of the unsegmented ruleset to match the speed of the segmented ruleset reduced the number of correctly inferred examples by up to 6%. Ruleset segmentation is thus an effective method for incorporating *a priori* knowledge into learned rulesets. It provides significant speed increases over unsegmented rulesets with no loss of accuracy.

The error-cost measures for rule priority and weight perform better than the information theoretic measures, as shown in Table 7.3. Using the error-cost rule weight both increases the accuracy of the ruleset and reduces the average cost of misclassifications made by the ruleset. Using the error-cost priority further increases accuracy.



Figure 7.1: Generated Harmony for "Happy Birthday"

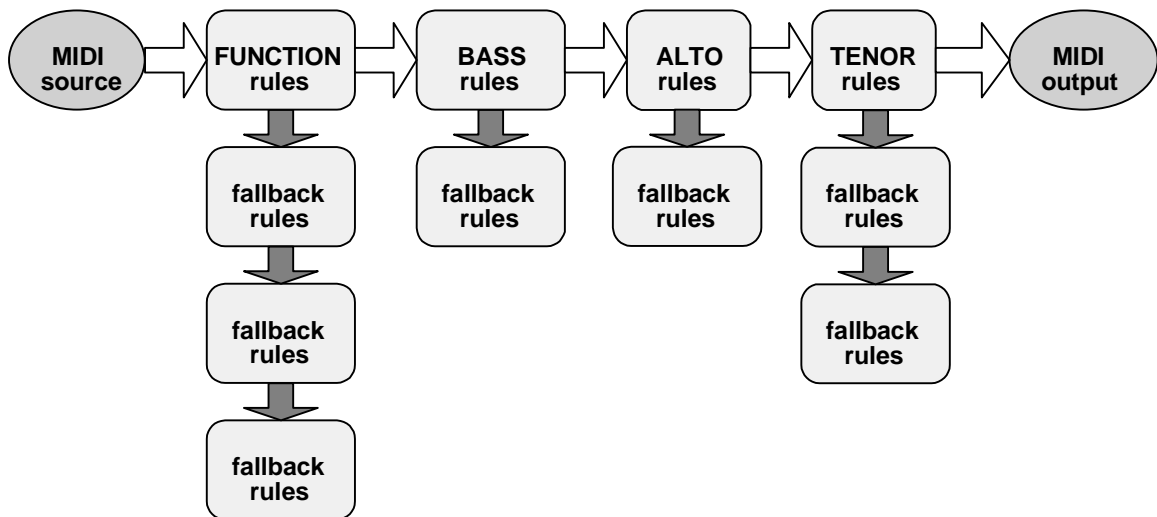


Figure 7.2: Data Flow in the Real-Time Harmonizer

1.5. Generated Harmony

Given the melody for "Happy Birthday," the segmented rulesets produce the harmony shown in Figure 7.1. The harmony is certainly not the traditional harmony sung for that melody; it is closer to the Bach chorales in sound than it is to its original harmony, which is what we would hope for.

1.6. Use of Generated Rules in a Real-Time Environment

A program was written in Microsoft Visual Basic to accept melodic input from a MIDI keyboard. As each note was received, the previous timesteps of harmony were shifted in the input window to provide two timesteps of history, and the generated rules were used to produce values for the chord function and voice positions for the current timestep, as shown in Figure 7.2. This information was used to construct harmony notes as described in Chapter 2, Section 7. These notes were then sent to a MIDI synthesizer.

On a typical Intel Pentium-based personal computer, the program is able to generate harmony fast enough to keep up with all but the fastest performers. The harmony is appropriate in most cases, especially with classical melodies. The algorithm seems to have more difficulty generating harmony for melodies which are drastically different than the chorale melodies harmonized by Bach. When the input melody is substantially different, the top (most-specialized) segments of each ruleset are often unable to fire since the input is outside their input domain, and the rule engine must fall back on the lower (more general) segments, which are also of lower information content and accuracy.

When a given melody is repeated several times, a somewhat different harmony is generated each time. This is due to the way in which results are resolved from conflicting rules. The qualitative effect of this is superior to simply picking the most likely result at each timestep, which results in music which is somewhat monotonous.

CHAPTER 8

Summary

Work presented here describes the evolution of tools required to perform analysis of musical examples to extract rules and real-time generation of harmony based on those rules.

Existing representations for music proved unsuitable for generation of harmony rules. A representation for musical information was developed to facilitate rule extraction. This representation, the Extended Figured Bass, was based on a form of the figured bass used by many musicians and composers, including J. S. Bach. The representation was extended to more explicitly specify the positions of the chord voices. Algorithms were developed to automatically transcribe music from the MIDI format to and from the Extended Figured Bass.

The choice of a system based on probabilistic provided several advantages over other knowledge systems. Foremost among these was the explicit representation of knowledge, which allows for comparison with existing music theory. Many of the generally accepted rules of classical harmony were found in the rulesets generated by the system. The randomized process by which conflicting rules are resolved generates harmony which is non-deterministic, and thus more interesting..

A new data mining algorithm for extracting rules from examples, SpanRULE, is introduced. This algorithm performs well under a wide range of input parameters, and has easily determinable limits based on the amount of available memory. In comparison with rule generation algorithms based on hash tables and hybrid algorithms, SpanRULE proved to be equally fast and more predictable.

Several methods were developed for improving raw rules generated from examples. Error-cost rule weight and probability measures which are based on the cost

of mistakes made by rules were introduced. These measures avoid a frequent shortcoming in learning systems of only considering the frequency of errors as opposed to taking into account the nature of those errors. A method of including *a priori* knowledge about input attributes was developed. This method, ruleset segmentation, was shown to provide increased accuracy and speed of inferencing.

In order to effectively utilize the error-cost measures developed above, it was necessary to perform psychophysics experiments to measure the perceived cost of errors in musical harmony. These data from these experiments was used to develop a misclassification cost table for harmonic function.

The preceding techniques were used to build a system employing learned rules which was capable of real-time generation of harmony in response to an input melody. The performance of this system was analyzed, verifying the validity of the error-cost measures and ruleset segmentation approaches to refining extracted rules. Finally, an application of this system was constructed and used to generate real-time harmony.

The methods developed here suggest a range of directions in which future research could proceed. These methods could be applied to other styles or composers of music. Preliminary research in these directions highlights several problems which will need to be solved. Many styles of music, string quartets for example, are less harmonically compact than harmonized chorales. To analyze these styles, new algorithms will need to be developed to convert them into a format conducive to rule extraction, and then to convert the rule output back into the original style. This introduces issues with orchestration and resolving conflicts between the generated harmony and physical limitations of the instruments. If similar collections of music from

two different composers can be analyzed, the rules generated from each composer could be compared to quantitatively establish differences between their styles, or combined to create a new composite style embodying features of each composer. The generated rules could also be used for verification of the origin of pieces of music, by measuring the degree to which a piece of music agrees with the rules generated from music of known origin.

Applications based on these musical rules also have potential. For example, the rules could be used to develop a program to aid in education of future musicians. A computerized expert could be trained on examples of a musical style. Students attempting to write music in the style could then ask the expert to check their compositions for errors and suggest alternatives. Because of its rule-based nature, the expert could also provide explanations for why its suggestions fit the style.

Furthermore, the algorithms for extraction and improvement of rules have many potential non-musical applications, from analysis of medical databases to economic forecasting and real-time control. All of these can benefit from a system of explicit knowledge which is learned from examples and designed to minimize the cost of its inferences.

References

- Alphonse, B. H. (1980). Music Analysis by Computer – A Field for Theory Formation. *Computer Music Journal* **4**(2):26-35.
- Bach, J. S. (Ed.: A. Riemenschneider). (1941). 371 Harmonized Chorales and 96 Chorale Melodies. Milwaukee, WI: G. Schirmer.
- Bain, M. (1992). Learning Optimal Chess Strategies. In "ILP 92: Proc. Intl. Workshop on Inductive Logic Programming" (S. Muggleton, ed.). Tokyo, Japan: Institute for New Generation Computer Technology.
- Brand, M. (1981). Music Abilities and Experiences as Predictors of Error-Detection Skill. *Journal of Research in Music Education* **29**(2):91-96.
- Camurri, A., M. Frixione and R. Zaccaria. (1993). A Music Knowledge Representation System Combining Symbolic and Analogic Approaches. *Computers and the Humanities* **27**:25-30.
- Brinkman, A. R. (1986). Representing Musical Scores for Computer Analysis. *Journal of Music Theory* **30**:225-275.
- Dannenberg, R. B. (1993). Music Representation Issues, Techniques, and Systems. *Computer Music Journal* **17**(3):20-30.
- Gibson, D. B. (1988). The Aural Perception of Similarity in Nontraditional Chords Related by Octave Equivalence. *Journal of Research in Music Education* **36**(1):5-17.

- Grout, D. J. and C. V. Palisca. (1988). *A History of Western Music*. New York: W. W. Norton.
- Goodman, R. M., P. Smyth, C. M. Higgins, and J. Miller. (1992). Rule-Based Neural Networks for Classification and Probability Estimation. *Neural Computation* **4**(6):781-804.
- Ha, T. M. (1997). The Optimum Class-Selective Rejection Rule. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **19**(6).
- Hamidzadeh B. and S. Shekhar. (1992). Can Real-Time Search Algorithms Meet Deadlines? *In "Proceedings of the Tenth National Conference on Artificial Intelligence,"* 486-491. Menlo Park: AAAI Press/MIT Press.
- Higgins C. M. and R. M. Goodman. (1993). Learning Fuzzy Rule-Based Neural Networks for Control. *In "Advances in Neural Information Processing Systems 5"* (Hanson S.J., et al., eds), 350-360. San Mateo, CA:Morgan Kaufmann.
- Hild, H., J. Feulner and W. Menzel. (1992). HARMONET: A Neural Net for Harmonizing Chorales in the Style of J. S. Bach. *In "Advances in Neural Information Processing Systems 4"* (J. Moody, ed.), 267-274. San Mateo, CA: Morgan Kaufmann.
- Jaffe, D. A. and W. A. Schloss. (1994). The Computer-Extended Ensemble. *Computer Music Journal* **18**(2):78-86.
- Jones, J., D. Scarborough and B. Miller. (1993). GTSIM: A Computer Simulation of Music Perception. *Computers and the Humanities* **27**:19-23.
- Katayose, H. and S. Inokuchi. (1993). Learning Performance Rules in a Music Interpretation System. *Computers and the Humanities* **27**:31-40.

- Kimura, M. (1997). Computers for Performers - Crossing the Boundaries for the Future. *Computer Music Journal* **20**(4):25-26.
- Kohonen, T., P. Laine, K. Tiits and K. Torkkola. (1991). A Nonheuristic Automatic Composing Method. In "Music and Connectionism" (P. M. Todd and D. G. Loy, eds.), 229-242. Cambridge, Massachusetts: MIT Press.
- Krumhansl, C. L., J. Bharucha and M. A. Castellano. (1982). Key Distance Effects on Perceived Harmonic Structure in Music. *Perception and Psychophysics* **32**(2):96-108.
- Loy, D. G. (1991). Connectionism and Musiconomy. In "Music and Connectionism" (P. M. Todd and D. G. Loy, eds.), 20-38. Cambridge, MA: MIT Press.
- MIDI Manufacturers Association. (1988). MIDI 1.0 Detailed Specification. Los Angeles, CA: International MIDI Association.
- Mookerjee, V. S. and M. V. Mannino. (1997). Sequential Decision Models for Expert System Optimization. *IEEE Transactions on Knowledge and Data Engineering* **9**(5).
- Mozer, M. and T. Soukup. (1991) Connectionist Music Composition Based on Melodic and Stylistic Constraints. In "Advances in Neural Information Processing Systems 3" (R. Lippmann, ed.). San Mateo, CA: Morgan Kaufmann.
- Nishijimi, M. and K. Watanabe. (1993). Interactive Music Composer Based on Neural Networks. *Fujitsu Scientific Technical Journal* **29**(2):189-192.
- Orem, P. W. (1924). Theory and Composition of Music. Philadelphia, PA: Theo. Presser Co.
- Ottman, R. (1989). Elementary Harmony. Englewood Cliffs, NJ: Prentice Hall.

- Repp, B. H. (1996). The Art of Inaccuracy: Why Pianists' Errors Are Difficult to Hear. *Music Perception* **14**(2):161-184.
- Rothgeb, J. E. (1968). Harmonizing the Unfigured Bass: A Computational Study. Ph.D. Dissertation, New Haven, Connecticut: Yale University.
- Shibata, N. (1991). A Neural Network-Based Method for Chord/Note Scale Association with Melodies. *NEC Research & Development* **32**(3):453-459.
- Smaill, A., G. Wiggins and M. Harris. (1993). Hierarchical Music Representation for Composition and Analysis. *Computers and the Humanities* **27**:7-17.
- Smyth, P. and R. M. Goodman. (1990). An Information Theoretic Approach to Rule Induction from Databases. *IEEE Transactions on Knowledge and Data Engineering*.
- Spangler, R. R., R. M. Goodman and J. Hawkins. (1998). Bach in a Box: Real-Time Harmony. In "Advances in Neural Information Processing Systems 10" (M. I. Jordan, M. J. Kearns and S. A. Solla, eds.), 957-963. Cambridge, MA: MIT Press.
- Thompson, W. F. (1993). Modeling Perceived Relationships Between Melody, Harmony, and Key. *Perception and Psychophysics* **53**(1):13-24.
- Todd, P. (1989) A Connectionist Approach to Algorithmic Composition. *Computer Music Journal* **13**(4):27-43.
- Witten, I. H., L. C. Manzara and D. Conklin. (1994). Comparing Human and Computational Models of Music Prediction. *Computer Music Journal* **18**(1):70-80.