Aalborg Universitet

Computational Problems in Modeling Milton Babbitt's Compositional Process

Bemman, Brian

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

Link to publication from Aalborg University

Citation for published version (APA):
Bemman, B. (2017). *Computational Problems in Modeling Milton Babbitt's Compositional Process*.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

Take down policy
If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from vbn.aau.dk on: April 29, 2024

COMPUTATIONAL PROBLEMS IN MODELING MILTON BABBITT'S COMPOSITIONAL PROCESS

BY
BRIAN BEMMAN

DISSERTATION SUBMITTED 2017

Computational Problems in Modeling Milton Babbitt's Compositional Process

Ph.D. Dissertation

Brian Bemman

Aalborg University
Department of Architecture, Design and Media Technology
Rendsburggade 14
DK-9000 Aalborg

Dissertation submitted: March, 2017

PhD supervisor: Assoc. Prof. David Meredith
Aalborg University

PhD committee: Associate Professor Amalia de Götzen
Aalborg University Copenhagen

Professor Andrew W. Mead
Indiana University

Professor Moreno Andreatta
IRCAM - CNRS UMR 9912 STMS - UPMC

PhD Series: Technical Faculty of IT and Design, Aalborg University

ISSN (online): 2446-1628
ISBN (online): 978-87-7112-916-8

Published by:
Aalborg University Press
Skjernvej 4A, 2nd floor
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

© Copyright: Brian Bemman

Printed in Denmark by Rosendahls, 2017

Abstract

This thesis provides a set of computational methods for analyzing Milton Babbitt's (1916–2011) 12-tone serial music and generating novel musical works in his style. A key component of the proposed methods is their basis in the actual techniques Babbitt himself used to compose his later works, supplemented by hypotheses in the form of algorithms regarding the specific processes that he used to apply these techniques. Two of these techniques, the *all-partition array* and *time-point system*, combine to form the musical surface of many of these works. Considerable attention in this thesis is paid to the structure found in the all-partition array. The problem of generating an all-partition array is challenging. However, solving this problem is necessary if one's aim is to model Babbitt's compositional process, as it greatly constrains the many other possible musical parameters in his works. A significant contribution of the work presented here is the demonstration that this problem can be formulated as a special case of the set-covering problem (SCP), a familiar problem in computer science and operations research. This thesis presents three methods for generating an all-partition array based on procedural backtracking with heuristics, integer programming (IP) and constraint programing (CP). A solution was found to the smallest instance of this problem using the proposed CP model. Generating the larger instances of the all-partition array found in Babbitt's music remains an unsolved problem. Finally, a method is proposed for generating from an all-partition array novel musical works based on the time-point system. This method has been used to automatically generate a symbolic score representation of a novel work for flute and string quartet, where the generated parameters of this piece include pitch, voice, onset, duration, dynamic level and meter. It is precisely because a computational method has been adopted here that the true difficulty in understanding Babbitt's music, his compositional process and the problems therein have been made clear.

Resumé

Denne afhandling giver et sæt computerbaserede beregningsmetoder til at analysere Milton Babbitts (1916–2011) 12-tone seriel musik og generere nye musikværker i hans stil. Et centralt element ved de givne metoder er deres basis i de faktiske teknikker Babbitt brugte til at komponere hans senere musikstykker, suppleret med hypoteser i form af algoritmer vedrørende de specifikke processer, han brugte, til at anvende disse teknikker. To af disse teknikker, *all-partition array* og *time-point system*, kombineres til at danne musikken i mange af disse musikstykker. I denne afhandling er der stort fokus på strukturen, der findes i all-partition arrayet. Problemet med generering af et all-partition array er udfordrende. Imidlertid er det nødvendigt at løse dette problem, hvis målet er at modellere Babbitts kompositoriske proces, da det begrænser de mange andre mulige musikalske parametre i hans stykker. Et væsentligt bidrag af arbejdet, der præsenteres her, er påvisningen af, at dette problem kan formuleres som et specialtilfælde af set-covering problemet (SCP), et velkendt problem i datalogi og operationsanalyse. Denne afhandling præsenterer tre metoder til generering af et all-partition array baseret på proceduremæssig backtracking med heuristikker, integer programmering (IP) og constraint programmering (CP). En løsning blev fundet til den mindste forekomst af dette problem ved brug af den foreslåede CP model. Generering af de større forekomster af all-partition arrays fundet i Babbitts musik er stadig et uløst problem. Endelig foreslås en metode til at generere nye musikalske stykker fra et all-partition array baseret på time-point systemet. Denne metode er blevet anvendt til automatisk at generere en symbolsk musikscorerepræsentation af et nyt stykke til fløjte og strygekvartet. De genererede parametre til dette stykke indeholder tonehøjde, stemmer, nodestart og -varighed, dynamikker og takt. Det er netop fordi computerbaserede beregningsmetoder er blevet brugt i denne afhandling, at den sande sværhedsgrad i at forstå Babbitts musik, hans kompositoriske proces og de problemer deri er blevet gjort klar.

Contents

Abstract iii

Resumé v

Thesis Details xi

Preface xiii

I Introduction 1

Introduction 3

1 Computational Music Analysis and Generation 4
1.1 Algorithms for Tonal Music Analysis and Generation . . 4
1.1.1 Algorithmic approaches 5
1.1.2 Statistical approaches 6
1.1.3 Mathematical approaches 7
1.2 Algorithms for Post-tonal and 12-tone Music Analysis and Generation 8
1.3 Evaluating Computational Models of Music 10
2 Music Perception and Cognition 11
2.1 Tonal and Common-practice Music 11
2.2 Post-tonal and 12-tone Music 12
2.3 Perceptual Models of Beat and Meter 12
3 Twelve-tone Music 12
3.1 Preliminary Definitions 13
3.2 Serialism and Maximal Diversity 14
4 Babbitt's Compositional Process 16
4.1 Philosophy 16
4.2 All-partition Array 17
4.3 Time-point System 19
4.4 Later Works 20
5 Previous Computational Work Related to Babbitt's Music . . . 21

5.1 Analysis of Works 22
5.2 Generation of Combination Matrices (CMs) 23
6 Contributions 25
7 Conclusion 29
References 30

II Papers **39**

A Exact Cover Problem in Milton Babbitt's All-partition Array **41**
1 Introduction 43
2 All-partition Array as Exact Cover 44
3 Solving the projection cover problem 45
4 Conclusion 48
References 48

B Predicting Babbitt's Time Points in None but the Lonely Flute and Around the Horn **51**
1 Introduction 53
2 Rothgeb's *Order Inversions* and Povel and Essens' *Clock Induction* model 54
2.1 Order Inversions 54
2.2 Clock Induction model 55
3 The number of possible rows without repetitions for an aggregate partition 57
3.1 Constructing rows 58
3.2 Computing rows 59
4 A heuristic based on the combined measures of order inversions and clock induction 61
4.1 Motivation 62
4.2 The heuristic 64
5 Predicting the time-point rows without repetitions in *None but the Lonely Flute* and *Around the Horn* 66
5.1 Relative impact of Order Inversions and Clock Induction 66
5.2 Results and Evaluation 68
6 Conclusion 72
References 73

C Generating Milton Babbitt's All-partition Arrays **75**
1 Introduction 77
2 The all-partition array 78
2.1 Preliminary definitions 79
2.2 Structural definitions 80

3 Previous work 84
4 Smalley Arrays 87
4.1 Organisational constraints on the row-form array 88
5 Generating a PcMatrix of the Smalley array class 89
5.1 Computing row operations in the PcMatrix 90
5.2 Computing transposition levels for the row operations in the row-form array 93
6 Generating the WPcMatrix 96
6.1 A backtracking algorithm for computing the sequence of integer compositions 99
6.2 Computing candidate compositions and OARPs 102
6.3 Ranking candidate compositions 105
6.3.1 Equal-lyne-length heuristic 105
6.3.2 Zero-gain segments heuristic 105
7 Parameter values for generating the WPcMatrices of three of Babbitt's works 106
8 Evaluation 107
8.1 Task 1: Predicting the sequences of integer compositions in Babbitt's works 108
8.2 Task 2: Generating a new Smalley array 110
9 Conclusion 112
References 113

D Integer Programming Formulation of the Problem of Generating Milton Babbitt's All-partition Arrays 117
1 Introduction 119
2 Previous work 122
3 Set-covering problem formulation of all-partition array generation 123
3.1 All-partition array generation as a set-covering problem (SCP) with additional constraints 124
4 IP implementation of conditions for set-covering in all-partition array generation 125
4.1 Conditions for C_k to contain twelve distinct integers in A (condition of containment) 126
4.2 Conditions for C_k to be integer compositions in A (condition of consecutiveness) 126
4.3 Condition for covering A 128
5 IP implementation of additional conditions in all-partition array generation 128
5.1 Left-to-right order of C_k and permissible overlaps 128
5.2 Conditions for C_k to be integer compositions defining distinct integer partitions (condition of distinctness) . . 129

6 Experiments 130
7 Conclusion and future work 131
8 Acknowledgements 131
References 132

E Constraint Programming Approach to the Problem of Generating Milton Babbitt's All-partition Arrays 135
1 Introduction 137
1.1 The structure of an all-partition array 138
2 CP constraints for the problem of generating an all-partition array from a given matrix 140
2.1 Consecutiveness 141
2.2 Containment 141
2.3 Covering all (i, j) in A 141
2.4 Restrictions on the left-to-right order of candidate sets and their overlaps 142
2.5 Candidate sets as all different integer partitions 142
3 Solution 143
4 Conclusion 144
References 145

F Generating New Musical Works in the Style of Milton Babbitt 149
1 Introduction 151
1.1 All-partition array 152
1.2 Time-point row 153
2 Babbitt's compositional process in his later works 153
2.1 Rows from ordered mosaics 154
2.2 Equal-note-value strings and rhythm 155
3 Generating rows from ordered mosaics 156
3.1 Pitch-class repetitions in rows containing equal-note-value strings 157
3.2 When and where pitch-class repetitions occur 158
3.3 Maximum length of an equal-note-value string 160
4 Generating rests and ties using equal-note-value strings 161
4.1 Rests and ties 162
5 Automating Babbitt's compositional process in his later works 164
6 Generated piece 166
7 Conclusion 175
References 175

Thesis Details

Thesis Title:	Computational Problems in Modeling Milton Babbitt's Compositional Process
Ph.D. Student:	Brian Bemman
Supervisor:	Assoc. Prof. David Meredith, Aalborg University

The primary contribution of this thesis consists of the following papers.

[A] Bemman, B. and Meredith, D. "Exact Cover Problem in Milton Babbitt's All-partition Array," *Mathematics and Computation in Music*, 2015.

[B] Bemman, B. and Meredith, D. "Predicting Babbitt's Time points in None but the Lonely Flute and Around the Horn," *Journal of Mathematics and Music*, in review.

[C] Bemman, B. and Meredith, D. "Generating Milton Babbitt's All-partition arrays," *Journal of New Music Research*, 2016.

[D] Tanaka, T., Bemman, B. and Meredith, D. "Integer Programming Formulation of the Problem of Generating Milton Babbitt's All-partition arrays," *International Society for Music Information Retrieval*, 2016.

[E] Tanaka, T., Bemman, B. and Meredith, D. "Constraint Programming Approach to the Problem of Generating Milton Babbitt's All-partition arrays," *International Conference on Principles and Practice of Constraint Programming*, 2016.

[F] Bemman, B. and Meredith, D. "Generating new musical works in the style of Milton Babbitt," *Computer Music Journal*, in review.

The following publications have also been produced as part of this Ph.D., but have not been included in this thesis.

[1] Bemman, B. and Meredith, D. "From analysis to surface: Generating the musical surface of Milton Babbitt's Sheer Pluck from a parsimonious encoding of an analysis of its pitch-class structure," *Music encoding Conference*, 2014.

[2] Bemman, B. and Meredith, D. "Comparison of heuristics for generating Milton Babbitt's All-partition arrays," *Computer Music Multidisciplinary Research Conference*, 2014.

This thesis has been submitted to the Technical Doctoral School of IT and Design at Aalborg University, Denmark in partial fulfillment of the requirements for a Ph.D. in Computer science and Engineering. The thesis consists of the scientific papers listed above which have either been published or submitted at the time of this writing. Parts of these papers are used directly or indirectly in the extended summary of the thesis which appears as part of the Introduction in Part I. The papers above appear in Part II. As part of the assessment, co-author statements have been made available to the assessment committee as well as to the Faculty at the university. This thesis is not available for open publication but only in limited and closed circulation due to copyright.

Preface

This thesis has been submitted to the Technical Doctoral School of IT and Design at Aalborg University, Denmark in partial fulfillment of the requirements for a Ph.D. in Computer science and Engineering. The research presented in this thesis was carried out in the period from December 2013 to February 2017 as part of the Music Informatics and Cognition (MusIC) group within the Media Technology section of the Department of Architecture, Design and Media Technology at Aalborg University, Denmark. This work was made possible by a research grant provided by a collaborative European project on machine modeling of creativity and music called *Learning to Create* (Lrn2Cre8), which was funded by the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission under FET grant number 610859.

I would like to thank my supervisor, David Meredith, for his kind and generous support and from whom I have learned a considerable amount. I would like to also thank the other members of my research group, Olivier Lartillot and Gissel Velarde, for our many thought-provoking discussions. Finally, I would like to thank Sidsel Nørholm and my parents, John and Judy, for their support during the past three years.

Brian Bemman
Aalborg University, March 16, 2017

Part I

Introduction

Introduction

This thesis addresses the problem of computational music analysis and generation of 12-tone music, using the works of the composer Milton Babbitt (1916–2011) as a case study. Computational music analysis and generation seeks to understand music, its structure and how it is created through the use of computers. The field of computer science has produced many powerful tools that can assist us in understanding the structure of music. Outside of this field, however, music-theoretical analysis is the standard method for attempting to understand musical structure. Experimental work in psychology and neuroscience provides a third approach that sheds light on the cognitive and perceptual processes involved in understanding music. Computer-generated descriptions of musical structure (i.e., analyses) have the potential to be both more efficient and more rigorous than those by humans. In tasks of musical generation, computer models must either learn or be given sufficient knowledge of these descriptions in order to produce output that is stylistically accurate. In this thesis, the latter strategy will be adopted.

The research presented here draws on methods in music theory, computer science and music perception to develop methods for analyzing Babbitt's music and generating novel works in his style. A key component of the proposed methods is their basis in the actual techniques Babbitt himself used to compose his later works, supplemented by hypotheses in the form of algorithms regarding the specific processes that he used to apply these techniques. It is precisely because a computational method has been adopted here (and not, for example, a traditional music-theoretical approach) that the true difficulty in understanding Babbitt's music, his compositional process and the problems therein have been made clear. Section 1 of this thesis introduces previous work in computational music analysis and generation, including different approaches and existing algorithms as well as a brief discussion of how previous approaches have been evaluated. Section 2 provides a short introduction to work on music perception and cognition. Section 3 provides an introduction to 12-tone and serial music. Section 4 provides an introduction to Babbitt's compositional process and two of his techniques—the all-partition array and time-point system, which are discussed extensively

throughout this thesis. Section 5 provides a brief overview of what little computational work has been done previously on Babbitt's music. Section 6 provides an overview of the papers in Part II which collectively present the proposed computational methods and main contribution of this thesis.

1 Computational Music Analysis and Generation

There exist many computer tools for assisting humans in analyzing and composing music (e.g., Humdrum (Huron, 1997), Max/MSP (Puckette, 2002), SuperCollider (McCartney, 2002) OpenMusic (Assayag et al., 1999), and AC Toolbox (Berg, 2011)). These tools are used by music theorists and composers in what has been called *computer-aided algorithmic composition* (CAAC) (Ariza, 2005). However, such tools require human intervention and are generally not capable of automating the complex processes involved in computing an analysis of a work or generating a novel piece of music (Fernandez and Vico, 2013). It is important to distinguish these tools from the goals of computational music analysis and generation and, in particular, the goals of this thesis, which seek to automate these processes of analyzing and composing music.

Within computational music analysis and generation, there are a number of both interesting and difficult computational problems, including composer recognition (Velarde et al., 2016), pattern finding (Meredith, 2015), automated expressive performance (Herwaarden et al., 2014), automatic harmonic analysis (Cambouropoulos et al., 2014) and polyphonic music generation (Cope, 1987; Ebcioğlu, 1987; Herremans and Sörensen, 2013). Many computational methods have been used to solve these problems, including Markov chains (Pearce, 2005), Bayesian statistical models (Temperley, 2009), machine learning methods (Dubnov et al., 2003; Herwaarden et al., 2014), as well as some constraint and optimization models (Laurson and Kuuskankare, 2001; Tanaka and Fujii, 2015). This section focuses on some of these methods in computational musicology as applied specifically to the problem of analyzing and generating the pitch structures found in music, as these will be of most importance to the work presented in this thesis.

1.1 Algorithms for Tonal Music Analysis and Generation

The constituent characteristics of music—pitch, timbre, key, harmony, dynamics, rhythm, meter and tempo—serve as focal points of research by computational musicologists. There are numerous algorithms capable of computing analyses by identifying many of these basic structural components given an encoding of a work's musical surface (e.g., Cambouropoulos, 2008; Chew, 2014; Meredith et al., 2002; Temperley, 2001, 2007). Similarly, there are

a number of proposed algorithms for generating these structures (Conklin, 2013; Pearce, 2005). However, musical structures involving strict or approximate repetition, that can be so generalized to comprise phrases, melodies, chord progressions, etc., require algorithms capable of broader scope and application.

Computational approaches to the analysis and generation of musical structure can be broadly divided into three categories: algorithmic, statistical and mathematical. Naturally, on some level, methods in each of these three categories rely on algorithms for their implementation. An algorithmic approach, however, refers to the sole use of algorithms which make no use of methods from either of the other two categories. Methods used in both algorithmic and statistical approaches must either learn or be given sufficient knowledge in order to produce a desired output. When learning from musical data, algorithmic and statistical approaches typically do so in one of two ways: unsupervised or supervised learning. In the former, an algorithm is given no initial assumptions regarding its data and is expected to bootstrap the desired output—that is, learn the structure of this data directly; while in the latter, an algorithm is provided with the desired output to its data and is expected to learn the rule which maps this input to its output. When learning is not a necessary component of some method, an algorithm is given explicit knowledge or rules regarding acceptable outputs. Algorithmic or statistical approaches may make use of explicit knowledge in this way, however, it is a necessary component of mathematical approaches. Thus, mathematical approaches are wholly distinct from learning and are non-statistical.[1] This thesis relies on a non-statistical algorithmic approach to music analysis and a combined algorithmic and mathematical approach to music generation which is neither learning nor statistical.

1.1.1 Algorithmic approaches

While algorithmic approaches to detecting, for example, tonality and key in tonal music have been some of the most enduring (Chew, 2014; Longuet-Higgins and Steedman, 1971), algorithmic approaches to the analysis of higher-level structures such as melody, pattern and segmentation are equally as important (Cambouropoulos, 2001; Meredith et al., 2003). In recent years, pattern-finding algorithms based on unsupervised learning using non-statistical methods such as COSIATEC (Meredith et al., 2003), SIATECCompress (Meredith, 2013) and Forth's (2012) algorithm, have demonstrated success in producing exhaustive analyses of music that agree, to some extent, with analyses by human experts. All of these algorithms are greedy approxi-

[1] In a limited sense, the process of mathematical optimization "learns" something of its input data, however, unlike a machine learning system which must optimize some parameters in order to learn, this knowledge is not generalizable to new instances of data.

mation algorithms, based on the earlier SIA and SIATEC algorithms (Meredith et al., 2002), that aim to compute compressed encodings of point-set representations of music. These compressed encodings take the form of sets of translational equivalence classes (TECs) of maximal translatable patterns (MTPs) that collectively “cover" all or most of the points in the input dataset. SIATECCompress, COSIATEC and Forth’s algorithm use different strategies for isolating more meaningful patterns from others through a heuristic that considers the amount of compression achievable by a TEC, the compactness of the MTPs in a TEC, and the amount of music accounted for or covered by the TEC. These criteria have also been perceptually confirmed (Collins et al., 2011). When the algorithm finds all meaningful MTPs, it generates as output a graph of these patterns, the analysis.

1.1.2 Statistical approaches

Additional algorithms based on statistical approaches such as Bayesian probability (Temperley, 2007), Markov models (Pearce, 2005) and multiple viewpoints (Conklin, 2013) have similarly shown success in several musical tasks of analysis and generation. Moreover, many of these approaches are perceptually motivated. A *Markov model* is a stochastic process represented as a graph where nodes specify possible values of random variables with edges corresponding to probability distributions where the *transition probability* of a given node, Y, is its conditional probability distribution given the *current state* or probability distribution of only the current node, X. A single random walk through this graph is called a *Markov chain*. A Markov chain for music generation works by generating, for example, one new musical pitch at a time, where given the current pitch, future pitches will be conditionally independent of past pitches. A multiple viewpoints approach allows for such a model to consider more attributes in a piece of music than only pitch, including onset, pitch interval, pitch contour, etc. In music analysis, both approaches have been used, for example, in style classification tasks (Conklin, 2013) and melodic pattern salience (Collins et al., 2011). In contrast to a Markov chain, a *Bayesian probability network* is a graph of random variables with edges specifying conditional dependencies between nodes where the *posterior probability* of a given node, Y, is its conditional probability distribution with the *prior probability* of all previously visited nodes. Unlike a Markov chain, a Bayesian probability network for music analysis works by predicting, for example, a next pitch in a melody which is conditionally dependent on past pitches. In music analysis, this approach has been used in tasks of modeling the perception of key and meter (Temperley, 2007).

1.1.3 Mathematical approaches

In some special cases, a task of analysis or generation in music can be modeled as a *mathematical optimization* or *constraint satisfaction problem* (Russell and Norvig, 2010). There are a number of programming paradigms for modeling such problems, for example, Integer Programming (IP) and Constraint Programming (CP), as well as many efficient algorithms and techniques for solving them (e.g., the Simplex algorithm and constraint propagation). One benefit of IP and CP, is that they allow for the separation of the formulation of a problem by users and the development by specialists of an algorithm for solving it. Compared to approximate computational strategies, such as genetic algorithms, IP and CP formulations and their solvers are suitable for searching for solutions that strictly satisfy necessary constraints.

Optimization Problems Using Integer Programming (IP) In an optimization problem, an algorithm selects according to some criteria the best element from possible valid alternatives, typically in the form of maximizing (or minimizing) a real function called the *objective function* which takes these elements as input (Cormen et al., 2009, p. 847). In other words, each valid solution to this type of problem will have an associated value, and the goal of the algorithm is to determine the solution with the best value. Integer programming (IP) is one example of a paradigm used in optimization problems wherein variables must be integers and most often, constraints are expressed as linear equations and inequalities. A classic optimization problem known in computer science and operations research as the *set-covering problem* (SCP) can be expressed in a straight-forward way using IP. The *set-covering problem* requires finding in a finite set, $S = \{1, 2, \ldots, m\}$, a family, F, of subsets of S, $F = \{F_1, F_2, \ldots, F_n\}$, such that every element of S belongs to at least one subset in F (Cormen et al., 2009, p. 1117). With IP, this problem can be expressed by defining an m-row, n-column boolean matrix $(a_{i,j})$, where a *covering* of the rows of this matrix is a vector, $\mathbf{x}$, such that $x_j = 1$ if column j occurs in the solution and $x_j = 0$, if not. Typically, as an optimization problem, the goal is to minimize the cardinality of the subset of columns in a solution. This is expressed in IP as a cost, c_j, where the task is to

$$\text{Minimize} \quad \sum_{j=1}^{n} c_j x_j \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{i,j} x_j \geq 1, \qquad \forall\, i = 1, \ldots, m \tag{2}$$

$$x_j \in \{0, 1\}, \qquad \forall\, j = 1, \ldots, n. \tag{3}$$

IP models have demonstrated success in some musical tasks from melodic pattern finding and segmentation (Tanaka and Fujii, 2014) to describing large-

scale form (Tanaka and Fujii, 2015). However, the use of IP applied to tasks in music is limited.

Constraint Satisfaction Problems Using Constraint Programming (CP) In contrast to optimization, in a constraint satisfaction problem, an algorithm attempts to satisfy the constraints governing the states of a set of objects. A constraint satisfaction problem is a set of variables, $\{X_1, X_2, \ldots, X_n\}$ and a set of constraints, $\{C_1, C_2, \ldots, C_m\}$ where each variable, X_i, has a non-empty domain, D_i, of possible values and each constraint, C_j, specifies for some subset of variables, a *relation* on the corresponding subset of domains, D_j (Russell and Norvig, 2010, p. 83). A solution to this problem is an *assignment* of values to every variable in which every constraint is satisfied. In a constraint satisfaction problem, there is typically no objective function and so the goal of the algorithm is to find any one feasible solution or determine if none exists (Russell and Norvig, 2010, p. 83). Constraint programming (CP) is one example of a paradigm for solving such problems where constraints can be stated as a relation between variables (as just described). Depending on the type of problem, constraints in CP can be more flexible than in IP, with not only linear constraints possible (as in IP) but boolean constraints and variable ranges over non-numeric domains (i.e., symbolic) as well. In recent years, CP has shown some success in select musical tasks, including orchestration (Carpentier et al., 2010) and part writing (Laurson and Kuuskankare, 2001).

How exactly a problem is modeled (i.e., which type of constraints are used and how many are required, etc.) is one of the most important factors in either IP or CP in being able to produce a solution using reasonable resources of time and space. A poor model, for example, may contain an order of magnitude more variables than is required of the problem. With too many variables, even the most powerful solvers might be unable to solve the problem. With proper models, both IP and CP offer similarly straightforward and powerful ways of modeling and solving difficult computational problems. For this reason, the application of these paradigms to the task of music generation will be of particular interest in this thesis (see, e.g., Papers D and E).

1.2 Algorithms for Post-tonal and 12-tone Music Analysis and Generation

As noted by Cope (2008), "Milton Babbitt's work with musical set theory (see Babbitt, 1955, 1960, 1961, 1965), although not in itself actual computer music analysis, set the stage for an extraordinary number of later programs based on his computational approach to analyzing post-tonal music" (Cope, 2008, p. 29). Indeed, many of the earliest algorithms for computational music

analysis and generation were written exclusively for post-tonal music, precisely because of its mathematical nature (Alphonce, 1974, 1980; Forte, 1966, 1973; Gill, 1963; Gross, 1975; Hiller and Isaacson, 1959). Most of these algorithms focused on identifying in a piece of music the set-class membership of collections of pitches (Forte, 1973) and the relationships between these collections (Alphonce, 1974, 1980; Forte, 1966; Harris and Brinkman, 1989) as well as more sophisticated approaches to serial analysis (Harris and Brinkman, 1989) and interval counting and rudimentary pattern finding with collections of fixed size (Gross, 1975).

The importance of post-tonal music to the development of algorithms for musical generation is clearly reflected in the first computer-generated musical composition, *Illiac Suite* (1957), which contained a post-tonal third movement. In this movement, a number of musical parameters, including rhythm and dynamics, were generated through the combination of a set of rules and a statistical Markov chain process. Shortly following the creation of the *Illiac Suite*, Gill (1963) devised a strictly rule-based system for generating post-tonal music in the style of Schoenberg. The methods devised by Gill's (1963) work in fact influenced one of the more important early contributions to tonal-music generation, namely Ebcioğlu's (1987) expert system for harmonizing chorales in the style of Bach.

Some more recent algorithms for tonal music applied to tasks in post-tonal analysis have emerged, from the analysis of contour (Morris, 1993) to segmentation (Chew, 2005), voice-leading (Tymoczko, 2011) and pitch-class set analysis (Cambouropoulos et al., 2014). Moreover, there are now a number of tools for assisting composers and music theorists in computing, for example, the normal form of a given set or generating the set-class of a given 12-tone row (Assayag et al., 1999; Berg, 2011). However, the scope of these tools in larger tasks of analysis (e.g., segmentation, form, etc.) is limited and they generally are not capable of automating these tasks directly from the input of the score.

Despite the early prevalence of algorithms for analyzing and generating post-tonal and 12-tone serial music, almost none have been applied to Babbitt's music (the single exception being Sward, 1981, discussed in Section 5) and their current use has since fallen relatively out of favor. To date, the algorithms and programming paradigms described in Section 1.1, for example, have only been used to analyze or generate music belonging to the domain of common-practice or tonal music. There is little reason to believe, however, that these more modern methods, for example IP or CP, could not be used for other types of music. Indeed, for some smaller, isolated problems in 12-tone music, such as generating an *all-interval series* row, CP has been used effectively (Chemillier and Truchet, 2001; Puget and Régin, 2007).[2] In this thesis,

[2]An *all-interval series* 12-tone row contains every possible interval (e.g., m2, P4, etc.) between

both IP and CP as well as a backtracking approach with heuristics (similar to that used in early methods by Ebcioğlu, 1987; Gill, 1963) are applied to the task of generating Babbitt's music.

1.3 Evaluating Computational Models of Music

Evaluating either a machine analysis of a work or a machine-generated piece of music is problematic and this is especially true of post-tonal and 12-tone music. Machine analyses of tonal music, for example, have typically been evaluated by using them for classification tasks and by comparing them quantitatively with human analyses (Meredith, 2013, 2015; Temperley, 2007). Classification tasks generally require extensive encodings of works and currently, there exist very few such encodings of post-tonal or 12-tone music. Comparing machine analyses with those of humans encounters a similar lack of analyses of 12-tone music when compared to, for example, the number of analyses that exist of Bach's works. Furthermore, there is generally no agreed upon "standard analysis" or consensus as to which of these post-tonal or 12-tone analyses theorists find best. Similar sorts of issues arise in tasks of machine-generated music as well.

Like their analogues in the tonal domain, evaluating models of 12-tone style music generation is particularly challenging. So-called "musical Turing tests" and subjective listener evaluations are frequently employed (Agres et al., 2016). At best, musically informed observers should be incapable of discriminating between pieces that are generated by such models and pieces in the styles that these models are intended to simulate. Assessing whether or not such a machine-generated piece of music could have been composed by a particular composer, however, is also difficult. Asking expert listeners, for example, whether or not they thought a given piece was composed by this composer would not suffice. This is because the most qualified individuals to make a reliable assessment (e.g., music theorists and musicologists) would likely know all of this composer's compositions, thereby disqualifying their response. Moreover, this says nothing of the possibility that different experts might have different ideas about what characteristics define this composer's music. It would therefore be necessary to frame the question differently. For example, by instead asking these same individuals who they think composed the piece. While this approach has never before been tried with Babbitt's music, it might, for instance, provide results indicating that the generated piece is representative enough of Babbitt's style, however, it would not indicate whether or not the piece was "good Babbitt". It is important to note of any computational model for generating music that merely satisfying the sufficient conditions for what it means of a generated piece to be "in the style of", for example, Babbitt, does not mean that the necessary conditions

its adjacent pitch classes.

for this piece to be “good Babbitt" have also been satisfied. A “good Babbitt" piece could mean any number of things and would likely be best evaluated using a combination of tasks, for example, listener experiments as suggested by Agres et al. (2016) and score analysis by experts. The research presented in this thesis, however, adopts a different methodology of evaluation, which is discussed in further detail in Section 6.

2 Music Perception and Cognition

Much work has been done on how humans perceive and cognize the structures found in music (see, e.g., Huron, 2006; Temperley, 2001). To a large extent, this research has focused on tonal music and its inherent hierarchy of pitch (Krumhansl, 1990), wherein some pitches in a scale are perceived to be more important than others. This pitch hierarchy helps establish a structure in tonal music that is perceptually coherent and predictable to listeners. Post-tonal and 12-tone music, on the other hand, generally lack this pitch hierarchy (Butler, 1990), despite the presence of a tonal center in the works of some composers of such music (e.g. Perle and Berg). It is precisely this lack of hierarchy in 12-tone music and how Babbitt sought to, in turn, establish other organizing principles that might make his music so interesting from a perceptual point of view.

2.1 Tonal and Common-practice Music

Both musicologists (e.g., Huron and Parncutt, 1993; Meredith, 2012; Temperley, 2001) and psychologists (Deutsch and Feroe, 1981; Krumhansl, 1983; Shepard, 1982; Simon and Sumner, 1968) have developed theories for describing aspects of perceived musical structure precisely and parsimoniously. These models can be characterized as either statistical/probabilistic (e.g., Huron, 2006; Krumhansl, 1990; Temperley, 2007), structural or geometric (e.g., Chew, 2014; Shepard, 1982) or based on coding theories (Deutsch and Feroe, 1981; Meredith, 2012; Simon and Sumner, 1968). Some of the probabilistic models that have been proposed (e.g., Temperley, 2007) are in the tradition of the likelihood principle of perceptual organization (Helmholtz, 1910) while other systems, more in the tradition of generative linguistics and Gestalt psychology (e.g., Deutsch and Feroe, 1981; Lerdahl and Jackendoff, 1983; Lerdahl, 2001), are based on the “minimum" or “simplicity" principle which can be considered to derive from Koffka’s (1935) law of Prägnanz. The majority of experimental research has in fact reinforced much of the approach to analysis by traditional music theorists.

2.2 Post-tonal and 12-tone Music

Comparatively less psychological research has been done on post-tonal and 12-tone music (Butler, 1990; Dibben, 1990; Krumhansl et al., 1987; Lerdahl, 1989). Primary reasons for this are a relative lack of hierarchical structure in 12-tone music when compared to tonal music (Butler, 1990; Krumhansl et al., 1987) and a general difficulty among listeners in perceiving the structures found in 12-tone music (Dibben, 1990; Lerdahl, 1989). This lack of hierarchical structure in 12-tone music is a primary motivation for composers to seek alternative techniques for organizing their music, such as serialism (discussed in Section 3). With respect to Babbitt's music, Bernstein (2014) notes that Babbitt was interested in the perceptual implications of 12-tone music, particularly, in light of work by Miller (1956). Miller's work revealed that the channel capacity with respect to pitch in humans is approximately 2.5 bits, meaning that the maximum number of distinct categories (e.g., different pitches) that one can recognize in a single perceptual dimension (e.g., pitch) corresponds to 7 ± 2. Given the relative difficulty with which individuals are able to perceive the structures in 12-tone music, Miller's (1956) notion and the apparently innate process of grouping or *chunking* (Trehub and Hannon, 2006) likely had a significant impact on Babbitt's compositional process and the techniques he developed for organizing the structure of his works—in particular, his use of repetition.

2.3 Perceptual Models of Beat and Meter

An important structural component largely shared by both tonal and post-tonal music is beat and by extension, meter. Perhaps second to pitch, these are the most commonly studied features in music by psychologists (Povel and Essens, 1985), musicologists (Hasty, 1999; London, 2004) and those in the fields of signal processing, information retrieval and machine learning (Dubnov et al., 2003). Like perceptual models of pitch, models of beat induction and meter can be rule-based (Desain and Honing, 1999; Lee, 1991), probabilistic (Temperley, 2009) or based on coding theories (Povel and Essens, 1985). Furthermore, such models have demonstrated success in both the audio and symbolic (e.g., music notation) domains. Of particular importance to this thesis, are perceptual models of meter based on coding theories and Part II describes how one such model is used in the analysis of Babbitt's music and in particular, his technique of the time-point system (see Paper B).

3 Twelve-tone Music

Post-tonal and 12-tone music arose from progressive extensions to tonal music practice that occurred during the later years of the 19th century. At the

turn of the 20th century, the so-called Second Viennese School, comprised most notably of the composers, Berg, Webern and Schoenberg, developed practices (e.g., lack of hierarchical pitch structure) which led to post-tonal and later 12-tone music. The structural components of 12-tone music are not immediately salient in ways that those found in tonal music (e.g., key, functional harmony, etc.) are to the average listener. It is therefore critical to the understanding of 12-tone music to know the ways in which it is organized and how these differ from the structures found in tonal music.

3.1 Preliminary Definitions

As is normally done in pitch class set theory, we will denote the 12 musical pitch classes as integers (mod 12), where $C = 0$, $C\# = 1$, $D = 2$, ..., $B = 11$. We use the term *aggregate* in the sense in which it is usually used in pitch class set theory to mean the universe of pitch classes—that is, the set $\{0, 1, \ldots 11\}$. A *tone-row*, $\mathbf{r} = \langle p_0, p_1, \ldots p_{11} \rangle$, is then an ordered set of pitch classes that contains each element in the universe exactly once—that is, $\bigcup_{i=0}^{11} \{p_i\} = \{0, 1, \ldots, 11\}$. Each tone row belongs to an equivalence class of rows related by a group of canonical row transformations called *prime*, *inversion*, *retrograde* (i.e., reversal) and *retrograde inversion*. Such an equivalence class is known as a *row class* and the members of such a row class are called *row forms*. Traditionally, a row form is denoted by the transformation that produces it from the original or *prime* row form, P_0, using the abbreviations P_n, I_n, R_n and RI_n for transposition, inversion, retrograde and retrograde inversion, respectively, each with a transposition by n semitones.[3] We can define these canonical row transformations as follows:

$$\mathrm{P}_n(\mathbf{r}) = \bigoplus_{i=0}^{11} \langle (\mathbf{r}[i] + n) \bmod 12 \rangle \qquad \mathrm{I}_n(\mathbf{r}) = \bigoplus_{i=0}^{11} \langle (n - \mathbf{r}[i]) \bmod 12 \rangle$$

$$\mathrm{R}_n(\mathbf{r}) = \bigoplus_{i=0}^{11} \langle (\mathbf{r}[11 - i] + n) \bmod 12 \rangle \quad \mathrm{RI}_n(\mathbf{r}) = \bigoplus_{i=0}^{11} \langle (n - \mathbf{r}[11 - i]) \bmod 12 \rangle$$

where $0 \leq n \leq 11$ and $\bigoplus$ denotes concatenation, $\bigoplus_{i=1}^{n} \langle a_i \rangle = \langle a_1, a_2, \ldots, a_n \rangle$. The universe of tone rows can thus be strictly partitioned into row classes, each of which is a set of at most 48 distinct row forms that is closed under the operations $\{\mathrm{P}_0, \ldots \mathrm{P}_{11}, \mathrm{I}_0, \ldots \mathrm{I}_{11}, \mathrm{R}_0, \ldots \mathrm{R}_{11}, \mathrm{RI}_0, \ldots \mathrm{RI}_{11}\}$.

In Babbitt's music, additional 12-tone structural components, such as the *array* and *aggregate partition*, figure prominently. An *array* is a 2d arrangement of pitch classes—typically formed by concatenations of tone rows (Morris, 1987). When the row forms in such an array are represented using the standard abbreviations (i.e., P_n, I_n, R_n and RI_n) the array is called a *row-form array*

[3] Other transformations exist for defining equivalence classes and include for example, the M_5 and M_7 *multiplicative* transformations.

7	2	3	1	9	8	5	10	4	6	11	0
6	11	10	0	4	5	8	3	9	7	2	1
11	0	5	7	1	6	3	2	10	8	9	4
8	3	4	2	10	9	6	11	5	7	0	1
5	10	9	11	3	4	7	2	8	6	1	0
0	1	6	8	2	7	4	3	11	9	10	5

Fig. 1: An ordered mosaic (dark grey) in which each of the six ordered sets belongs to a different tone row in an array.

(Cuciurean, 1997, p. 8). Arrays (and row-form arrays) may be concatenated with one another from left-to-right and from top-to-bottom.[4] An *aggregate partition* is an unordered set of disjoint unordered sets whose union is the aggregate. For example, $\{\{0,1,2,3,4,5\},\{6,7,8,9,10,11\}\}$ is one possible aggregate partition. The pitch classes from each set in an aggregate partition may come from a single tone row or an array. In either a tone row or array, these sets may appear on the musical surface from left to right (melodically) or top to bottom (harmonically). In such cases, these sets become ordered. In order to further distinguish their left-to-right or top-to-bottom order, this thesis refers to an *ordered* set of ordered sets containing the aggregate as an *ordered mosaic*.[5] For example, $\langle\langle 5,11,0,2,6\rangle,\langle 4,1,9,3\rangle,\langle 7,8,10\rangle\rangle$ is one possible ordered mosaic. Figure 1 shows an ordered mosaic in which each of the six ordered sets belongs to a different tone row in an array. Note, in Figure 1, that the order in which the six ordered sets appear in the ordered mosaic corresponds to the ordering of the 12-tone rows in which these ordered sets originate. While various other ways to partition the aggregate appear in the works of Schoenberg, Berg, Webern, Dallapicolla and others, this representation (i.e., as an ordered mosaic) is important to the methods proposed in this thesis. The following sections discuss how the ordered mosaic and other methods of 12-tone composition were used by Babbitt.

3.2 Serialism and Maximal Diversity

Despite the seemingly strict definitions just given (Section 3.1), the 12-tone system of composition is largely free. Nonetheless, as Straus attests, its music “generally shares two structural features: (1) the aggregate of all twelve tones as a referential harmonic unit and (2) an ordered succession of tones as a source of motives, melodies, and harmonies" (Straus, 2009, p. xviii). The difference between these two features highlights the distinction between tech-

[4]See Babbitt’s *Arie da Capo* (1974) and *Sheer Pluck* (1986) for examples of works in which concatenated arrays may be found.

[5]Motivations for adopting the term “ordered mosaic" are provided in Papers B and C in Part II of this thesis.

niques of simply composing with twelve tones in (1) and what is known as *serialism* in (2).

In serial music, a composer organizes one or more musical parameters according to some recurring ordered series (Straus, 2009). For example, in 12-tone music this series could be the tone-row permutations found in a row class. Composers such as Stockhausen, Boulez and Messiaen are all known for their use of such serial techniques. In many ways, Babbitt's music is a continuation of the "composing with twelve tones" (number (1) above) practices developed by Schoenberg (Straus, 2009, p. 47). However, Babbitt sought more rigorous ways of organizing his music, developing techniques perhaps more in line with serialism (number (2) above) for governing the structures of pitch, rhythm, dynamic level and others.

One important way of characterizing Babbitt's music is through his use of what has been called *maximal diversity* in his application of serial techniques to all (or nearly all) musical parameters in a piece (Mead, 1994, p. 20). Maximal diversity is "an exhaustive completion of a list of possibilities" (Bernstein, 2014, p. 13) often manifesting in Babbitt's works as a presentation of as many musical parameters in as many different ways as possible. As Bernstein (2014) notes, Babbitt was often unable to complete these lists of musical parameters in his music. Over the years, Babbitt achieved varying degrees of maximal diversity in his music through a number of techniques of organization, including the *trichordal array*, *all-partition array* and *super array* (Mead, 1994, pp. 20–38). In many of his works based on, for example, the all-partition array, Babbitt used this structure in conjunction with additional techniques such as the *duration row* or *time-point system* to further serialize the rhythms and dynamic levels—often with these rhythmic or dynamic "lists" left incomplete.[6] This thesis argues that this feature of incomplete lists in Babbitt's music is a direct result of the consistency with which he adhered to certain systematic procedures in his compositional process. It is precisely these constraints then and Babbitt's refusal to violate them that make his music and his compositional process particularly suited to computational analysis. The primary contribution of this thesis—the collection of papers presented in Part II—focuses on the all-partition array and time-point system, as these techniques appear in a significant number of Babbitt's works. A brief introduction to Babbitt's compositional process and these techniques appears in the following section (Section 4), with more detailed explanations appearing in Part II.

[6]As Bernstein (2014) notes, complete arrays in the pitch domain are often found in Babbitt's works with incomplete arrays in the time domain.

4 Babbitt's Compositional Process

Babbitt's music and theoretical writings have had a profound impact on the composition of modern academic music. For over half a century, his thoughts regarding 12-tone music and composition, the techniques he developed, and his philosophy regarding the role of music theory in analysis have been of considerable interest to music theorists (see, e.g., Bernstein, 2014; Dubiel, 1997; Lake, 1986; Leong and McNutt, 2005; Martino, 1961; Mead, 1984, 1987, 1997). In recent years, the sketches for many of Babbitt's works have been made available.[7] Researchers are now able to examine many of Babbitt's contributions, including most importantly, his compositional process in much greater detail than before. As it turns out, this process that Babbitt devised for composing his works is highly constrained and as his sketches reveal, he was rigorous in his application of methodical procedures for satisfying these constraints. Part II of this thesis provides detail regarding the procedural and mathematical nature of Babbitt's compositional process while this section lays the ground work for understanding this process.

4.1 Philosophy

> "I like to believe that a not insignificant consequence of the proper understanding of a proper theory of music is to assure that a composer who asserts something such as: "I don't compose by system, but by ear" thereby convicts himself of, at least, an *argumentum ad populum* by equating ignorance with freedom, that is, by equating ignorance of the constraints under which he creates with freedom from constraints" (Babbitt and Peles, 2003, p. 191). — Milton Babbitt.

Babbitt's words shown in the quote above are interesting for a number of reasons. Foremost, it is clear that a keen awareness of the system by which a composer creates music and its constraints is important. Moreover, the link between analysis and composition it would seem is inextricable but it is not clear what a "proper" theory of music would be. He goes on, however, to state that such a theory would allow one to make "determinate and testable statements about musical compositions" (Babbitt and Peles, 2003, p. 191) and elsewhere, states that "a satisfactory theory is a satisfactory explanation of aspects of the empirical domain with which the theory is concerned" (Babbitt and Peles, 2003, p. 79). Babbitt's language here is clear and according to Schuijer (2008), it aligns closely with a philosophical movement in the

[7] The sketches for many of Babbitt's works have recently been made available through the Library of Congress (LLC), `https://lccn.loc.gov/2014565648`. Excerpts from some of Babbitt's sketches may be found in Bernstein (2014).

1920s called *Logical Positivism* which asserts that meaningful statements about the world must be either analytical or empirically verifiable (Schuijer, 2008, p. 251). Perhaps Babbitt's strongest words in support of a theory which accords with such a world view are made when he suggests that a theory is a "deductively interrelated system of laws, statable as a connected set of axioms, definitions, and theorems, the proofs of which are derived by an appropriate logic" and that "a musical theory reduces, or should reduce, to such a formal theory when uninterpreted predicates and operations are substituted for the terms and operations designating musical observables" (Babbitt and Peles, 2003, p. 79). In his seminal articles, Babbitt laid the ground work for a system of 12-tone analysis *and* composition that is decidedly as rigorous as his views above (Babbitt, 1955, 1960, 1961, 1962, 1973). For this reason, Babbitt's own music is a perfect candidate for examination by computational methods. In these papers, Babbitt presents both the all-partition array and the time-point system, which together form the basis upon which many of his works are constructed. These foundational concepts in his works will now be discussed in more depth.

4.2 All-partition Array

In an effort to express maximal diversity in his works, Babbitt devised a compositional structure known as the *all-partition array*. Construction of an all-partition array begins with the construction of an $I \times J$ array, A, of pitch-classes, where each ith horizontal sequence of pitch classes or *lyne* contains $J/12$ 12-tone rows. A complete all-partition array is a sequence of K ordered mosaics in A, each representing one of the K distinct ways in which 12 can be partitioned into I or fewer non-negative summands (e.g., $6 + 6$ or $5 + 4 + 3$).[8] Figure 2 shows an excerpt from an all-partition array with six lynes and the first three of its ordered mosaics corresponding to three distinct partitions of 12. Below each ordered mosaic in Figure 2 is a short-hand notation used by music theorists to denote how an aggregate is partitioned. For example, the notation, $3^2\,2^2\,1^2$, indicates that its ordered mosaic (shown above) contains two ordered pitch-class sets of length 3, two sets of length 2 and two sets of length 1. Note that this notation does not specify the order (i.e., lyne) in which each of these sets appears in the array. An important feature of every all-partition array is that the number of pitch classes required for its ordered mosaics exceeds the number of pitch classes in the array A. For example, a 6×98 all-partition array contains 576 pitch classes, but must also consist of 58 ordered mosaics, as there are 58 ways to partition 12 into 6 parts or fewer. However, $58 \times 12 = 696$ which is 120 more than the number of pitch classes in the array. Therefore, these missing pitch classes must be found

[8] This definition of the all-partition array is one proposed in this thesis and differs from that commonly given by music theorists.

1.	7 2		2 3 1 9 8 5
2.	6		6 11 10 0
3.	11	0 5 7	
4.	8 3 4	2 9 10	
5.	5 10 9	11 3 4	4 7
6.	0 1	1 6 8	
	$3^2\,2^2\,1^2$	3^4	6 4 2

Fig. 2: An excerpt from an all-partition array with six parts (i.e., lynes) and three of its ordered mosaics corresponding to three distinct partitions, $3^2\,2^2\,1^2$, 3^4 and 6 4 2. Note, this is the traditional representation used by music theorists, however, it is constructed from the same underlying array shown in the preferred representation used in this thesis in Figure 1.

by repeating elements in each lyne from one ordered mosaic to the next, without violating the order of pitch classes in the 12-tone rows from which each lyne is constructed (Mead, 1994, p. 34). In the third mosaic, shown in Figure 2, the first pitch class in each segment is a repetition from one of the previous ordered mosaics: pitch-classes 2 and 6 from the first ordered mosaic and another pitch-class 4, this time from the second ordered mosaic. Ensuring that all $I \times J$ pitch classes of A are contained by all K ordered mosaics represented by K distinct partitions, and that all $K \times 12 - (I \times J)$ pitch classes required in addition to those in A are horizontal repetitions, is a difficult task. An all-partition array which satisfies all of these constraints has been called *self-contained* (Mead, 1994, p. 218) and few known examples in Babbitt's music exist.[9]

A significant focus of this thesis will be on the difficult problem of generating self-contained, all-partition arrays (see Papers C, D and E). Indeed, Babbitt himself was unable to construct a six-part, self-contained all-partition array.[10] For this reason, all of his six-part arrays prior to the early 1980s are *non-self-contained*. In a non-self-contained array, the final ordered mosaic/partition can not be made to contain 12 distinct pitch classes without either (1) violating the constraint of horizontal repetitions (described above) or (2) failing to cover every pitch class in the array (i.e., there will be more than $K \times 12$ pitch classes). Examples of non-self-contained arrays in Babbitt's music may be found in, for example, *Arie da Capo* (1974) and *About Time* (1982). In the early-to-mid 1980s, a student of Babbitt's named David Smal-

[9]Mead (1994, p. 36) notes that there are in fact a number of all-partition arrays having distinct pitch-class organizations but that nonetheless, these are equivalent to one another by simple transformations (e.g., transposition, inversion or M5 operations.) The sequences of partitions in each of these arrays, however, are either the same or differ from one another by only a few partitions.

[10]David Smalley, personal email communication, February 25, 2015

ley created the first self-contained six-part all-partition array. The importance to Babbitt of a self-contained array is evidenced by the fact that, with one presently known exception, he continued to use Smalley's self-contained array in all subsequent works based on a six-part array.[11] The first time Babbitt used a Smalley self-contained array in a piece is not known as not all of his pieces have been analyzed, however, a strong candidate is in his work for solo guitar entitled *Sheer Pluck* (1984), which aligns nicely with when it was discovered.[12] The time period in which the self-contained six-part array appears in Babbitt's music is important as these arrays are one indicator of his third-period later works and mature compositional practice, which are the focus of this thesis. Part II of this thesis (in particular, in Papers C, D and E), provides a more formal definition of the self-contained all-partition array as well as a more detailed discussion regarding the importance of the array to Babbitt's works, such as how exactly it forms the basis from which many other parameters of his music (e.g., dynamics and rhythm) are derived.

4.3 Time-point System

Babbitt's time-point system sought to derive a correspondence between the 12-tone row and time. By replacing the interval of a half-step in a 12-tone row with a fixed period of time called a *unit* (Mead, 1987, p. 183), the time-point intervals (analogous to directed pitch-class intervals) between adjacent members become lengths of time measured in units rather than pitch intervals measured in semitones. In Babbitt's later works, the unit used by Babbitt in his time-point rows is typically a sixteenth note (Bernstein, 2014). Figure 3 shows an example of a time-point row using a sixteenth note as the unit and one possible rhythmic representation. Note, in Figure 3, how time points in a row denote onsets in time corresponding to new rhythmic events. In Figure 3(b), these rhythmic events have a duration equal to the inter-onset intervals of each time point. However, Babbitt often sought less straight-forward rhythmic interpretations for his time-point rows than that shown in Figure 3(b). In many of his later works based on the all-partition array, Babbitt constructed his time-point rows from the ordered mosaics in its underlying array. This process will now be considered in more detail.

[11]Following the discovery of a self-contained six-part array, Babbitt used a non-self-contained six-part array in his work, *Play it Again, Sam* (1989) (Andrew Mead, personal email communication, January 24, 2016). It is interesting to note of this title that, in addition to its reference to Woody Allen's Broadway play of the same name, it may be a clever allusion to Babbitt's own return to the use of the non-self-contained array.

[12]The title of *Sheer Pluck* is rich with puns and allusions. There are the clear references to the guitar, with "sheer" alluding to the fact that a performer can do nothing but "pluck" the instrument. There is also a more obscure reference to a quote by Horatio Alger (Robert Morris, personal email communication, October 15, 2015). If this piece is indeed Babbitt's first to use Smalley's self-contained array, then its title might be a clever way of suggesting that Smalley's discovery of this array was a matter of "sheer luck".

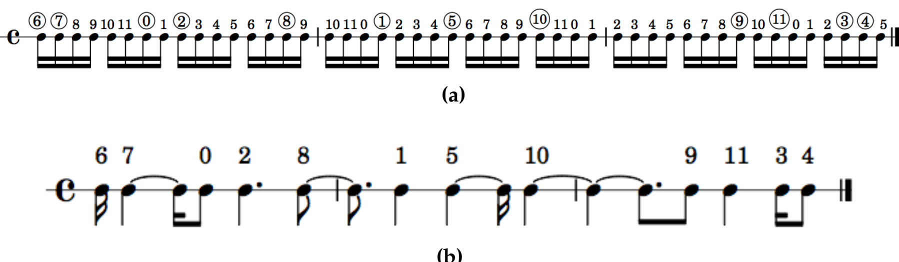


Fig. 3: Members of a time-point row in (a) set against a grid of sixteenth-note units and one possible rhythmic interpretation of this row in (b) where duration is equal to inter-onset interval.

4.4 Later Works

Both the all-partition array and the time-point system are characteristic of Babbitt's later works and he often used the former to derive material for the latter (Bernstein, 2014). The musical surface of these works is formed by uniting the pitch information specified by a pitch-class row constructed from an ordered mosaic in an all-partition array with the timing information specified by some rhythmic interpretation of a time-point row *also* constructed from an ordered mosaic from the same all-partition array.[13] Most often, pitch classes from the parts or lynes in an ordered mosaic are distinguished by register while time points are distinguished by dynamic level (Mead, 1994, p. 48). Figure 4 shows a possible pitch-class row, (a), and a possible time-point row, (b) taken from the same ordered mosaic in Babbitt's *None but the Lonely Flute* containing pitch classes distinguished by register and time points distinguished by dynamic level, respectively. Note, in Figure 4, that both the pitch-class and time-point rows are constructed from their respective ordered mosaics by selecting elements in a *left-to-right* order from each pitch-class segment (i.e., part or lyne). For example, the two pitch classes, 7, 2, in the pitch-class row of Figure 4(a), appear in the same order (despite intervening pitch classes) as the pitch-class segment, $\langle 7, 2 \rangle$, in the ordered mosaic from which they were constructed. Figure 5 shows the opening of Babbitt's *None but the Lonely Flute* and how the pitch-class row shown in Figure 4(a) and the rhythmic interpretation of the time-point row shown in Figure 4(b) have been united to form the musical surface.

A final component of the research presented in this thesis seeks to automate the process of first constructing pitch-class and time-point rows from an all-partition array, as shown in Figure 4, and then uniting these to form a

[13]It is important to note that, in his thesis, the term "row" is used to refer to what music theorists call a "realization" (Leong and McNutt, 2005). Unless otherwise stated, a "row" here refers not to some row in a row class but to a linear sequence of pitch classes or time points (without repetitions) from an ordered mosaic appearing on the musical surface.

(b)

Fig. 4: Opening pitch-class row in (a) and time-point row in (b) from Babbitt's *None but the Lonely Flute* (right) taken from their respective ordered mosaics (left). Note, in (a) that no time information is specified and in (b) that no pitch information is specified.

musical surface, as shown in Figure 5. In actuality, the details of this process prove more challenging both from a musical and computational standpoint than might at first be expected. The successful automation of this process, however, is the basis of the proposed model for generating novel musical works in the style of Babbitt provided in Part II of this thesis (see Paper F).

5 Previous Computational Work Related to Babbitt's Music

Despite the unusually explicit process by which Babbitt composed much of his music and its clear mathematical and procedural nature, almost no computational work on his music has been published (Bazelow and Brickle, 1976, 1979; Sward, 1981), despite comments by some suggesting its relevance (Morris, 2010; Starr and Morris, 1977). This limited computational work on Babbitt's music can be divided into efforts to either analyze his music or generate the structures related to those he used to compose his works—that is, not generate the music itself. It is important to note that at least one composer and music theorist has claimed to have developed a program to assist with constructing and editing all-partition arrays (Morris, 2010). Unfortunately, this program has not been made publicly available and the details of its design

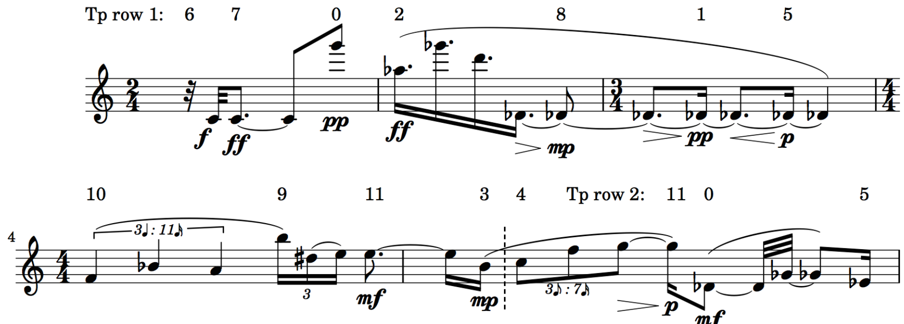


Fig. 5: Opening of Babbitt's *None but the Lonely Flute* corresponding to the pitch-class and time-point rows shown in Figure 4. Note, that the vertical dashed line in bar 5 marks the boundary between the first and second pitch-class rows (and ordered mosaics).

have not been published.[14] Of the remaining efforts to generate the structures in Babbitt's music that have been published, these have focused on the so-called *combination matrix (CM)*, a distinct but closely related structure to the all-partition array, which will be discussed below.

5.1 Analysis of Works

The only publicly available computational analysis of Babbitt's work, prior to that presented in this thesis, is that of Sward (1981). In her dissertation, Sward (1981) presents analyses of excerpts from three early pieces, *Three Compositions for Piano* (1947), *String Quartet No. 2* (1954) and *Duet* (1956). Sward applies to encodings of these excerpts a set of early algorithms, developed by Gross (1975), for discovering linear and vertical patterns of specified length. Gross' algorithms first catalogue the occurrences of pitch, duration, interval and dynamics in a piece. Then, by grouping any combination of these features, they search for respective matching sequences (Sward, 1981, p. 455). In her analysis, Sward evaluates these pieces according to two measures that she calls, *unity* and *variety*. Unity is the frequency of repetition of patterns and variety is the occurrences of different events (Sward, 1981, p. 521). The results of her findings largely confirm what one might suspect in a 12-tone system of composition, namely, that occurrences of pitch and interval are roughly uniform (Sward, 1981, p. 521). However, some interesting results were revealed in her comparative analysis of works by Xenakis. For example,

[14]In addition to Morris's (2010) algorithm, independent researchers, Ken Collins and Bob Arning, have purportedly developed a recursive algorithm using the Smalltalk programming language for generating 4, 6 and 12-part all-partition arrays (Ken Collins, personal email communication, November 30, 2016). Interested readers may write to Ken Collins at "ken-ncoll123@charter.net".

y	0 1 4 3 9 2	10 8 5 7 11	6
$T_5(y)$	5 6		9 8 2 7 3 1 10 0 4 11
$T_{11}I(y)$	11 10 7 8	2 9 1 3 6 4 0	5

Fig. 6: A three-row combination matrix (CM) formed from transformations of a given tone row, *y*. Note, that each row and column contains 12 distinct pitch classes. (From Starr and Morris, 1977, p. 18.)

non-immediate repetitions of pitch (e.g., C, D, C) occur frequently in Xenakis' works but not at all in Babbitt's early pieces. However, repetitions of patterns for both pitch and duration, in general, appear more frequently in Babbitt's music.[15] While the works Sward analyzed predate Babbitt's creation and use of the all-partition array, her recognition of the applicability of computational methods to Babbitt's music are much in line with the work presented in this thesis.

5.2 Generation of Combination Matrices (CMs)

While not explicitly computational, work by Starr and Morris (1977, 1978) established a systematic method for constructing structures known as *combination matrices (CMs)*. Combination matrices (CMs) are matrices of pitch classes with n 12-tone rows and n columns of aggregates containing n^2 row segments (Starr and Morris, 1977). Figure 6 shows one example of a CM. It is clear in Figure 6 how the columnar aggregates in a CM might prove useful in the construction of partitions in an all-partition array. In fact, Starr and Morris (1977, 1978) describe how different partitions in a CM can be found by *swapping* pitch classes across column boundaries (Starr and Morris, 1978, p. 58). The authors do not, however, propose a way for automating this process of swapping, despite suggesting this as a possibility (Starr and Morris, 1978, p. 59), nor do they provide a means for ensuring that with each swap, a different partition will be created—both crucial requirements for automatically generating an all-partition array. Nevertheless, it seems that at least parts of this process can be automated, as Kowalski (1985) has devised a set of algorithms for generating CMs. However, these algorithms fall short of generating an all-partition array.[16]

Bazelow and Brickle (1976) propose an algorithm for solving what they call Babbitt's "Partition Problem". They state this problem as follows: "Given an array of four forms of an arbitrary twelve-tone set, how many ways can

[15]Interestingly, Babbitt's later works contain many non-immediate repetitions. The prevalent use of both immediate (e.g., C, C) and non-immediate pitch repetitions in these later works may have been an explicit attempt to facilitate a listener's ability to cognitively "chunk" his music, as described in Section 2.2.

[16](David Kowalski, personal email communication, February 5, 2017).

3	4	11	0	7	8	2	1	6	5	10	9
2	1	6	5	10	9	3	4	11	0	7	8
8	7	0	11	4	3	9	10	5	6	1	2
9	10	5	6	1	2	8	7	0	11	4	3

Fig. 7: A solution to Babbitt's "Partition Problem" using the algorithm proposed by Bazelow and Brickle (1976).

the array be decomposed entirely into four-part, aggregate forming partitions?" (Bazelow and Brickle, 1976, p. 282). This problem is very similar to that of constructing a CM and can be viewed as a simplification of a central problem addressed in this thesis, namely, the generation of an all-partition array. However, the problem posed by Bazelow and Brickle differs from the problem in this thesis in four important respects. First, Bazelow and Brickle (1976) only consider partitions of exactly four parts; whereas, in a four-part all-partition array, partitions may have four or fewer parts (see the definition of an all-partition array in Section 4.2). Second, partitions that form a solution to their problem need not be all different. Third, the repetitions of pitch classes that must appear in an all-partition array, as shown, for example, in Figure 2, are not considered. Finally, any solution must *uniformly decompose* its array—that is, partition it into regions that must not overlap; whereas partitions may overlap under certain conditions in an all-partition array. That partitions must not form regions that overlap with one another is a crucial difference between this problem and the problem of generating an all-partition array as posed in this thesis. Bazelow and Brickle (1976) go on to lay the groundwork for an algorithm to solve this more simple problem and Figure 7 shows one solution using their proposed method.

Note that, in Figure 7, each of the four regions expresses the partition, 5421. This particular solution demonstrates a sufficient but not necessary property of a solution to their problem that they call *B-constructible*. The B-constructible property of an array means that it can be uniformly decomposed (i.e., partitioned) into four-part, aggregate-forming partitions in which parts in each partition can be paired into two pairs that each sum to six. For example, each of the four partitions, 5421, shown in Figure 7, can be paired into the two pairs, $5 + 1 = 6$ and $4 + 2 = 6$. As Bazelow and Brickle (1976) note, it is precisely because this array is B-constructible that its solution splits the array evenly into two, equally-sized halves (between columns 6 and 7).

The proposed methods for generating an all-partition array presented in this thesis are not based on any method by either Starr and Morris (1977, 1978) or Kowalski (1985) for constructing CMs, nor are they based on the algorithm devised by Bazelow and Brickle (1976). The following section of

this thesis, Section 6, provides an overview of these proposed methods and the primary contributions of each paper that appears in Part II.

6 Contributions

The contribution of this thesis is a set of computational methods for analyzing Babbitt's music and generating novel musical works in his style. A key component of the proposed methods is their basis in the actual techniques Babbitt himself used to compose his later works, supplemented by hypotheses in the form of algorithms regarding the specific processes that he used to apply these techniques. This thesis does not provide a means for directly evaluating either the proposed methods or the piece of music generated using these methods (see Paper F), either through comparisons to human output, listener ratings or score analysis, as suggested in Section 1.3. Instead, it is reasonable to suspect that machine-generated music that aims to be in the style of a composer can be at least partly evaluated based on how well the process used to create it aligns with the composer's own. The success of this particular methodology (and, indeed, whether or not the proposed hypotheses will be supported) depends on the accuracy of the knowledge given to the machine model. In this thesis, this knowledge is obtained through (1) extensive readings of Babbitt's own words regarding his compositional process, (2) critical study of manual analyses of his works by both the author and music theorists in the literature and (3) thorough inspection of the sketches for his works which are, arguably, the most important source of information regarding Babbitt's compositional process.

In the works of most composers, it is not immediately clear how modeling the compositional process would be done. As discussed in Section 4, however, Babbitt was unusually explicit about the techniques and processes by which he composed his works, which are inherently mathematical and procedural in nature. In this respect, his music is therefore, perhaps, less problematic for examination by computational methods of analysis and generation, than, for example, tonal music, where the compositional process is less explicit. Indeed, consider the nuance of many problems in tonal music such as part writing, where perhaps several musically "correct" answers are possible at any given point. In contrast, the problem of generating an all-partition array, for example, is much more well-defined mathematically. As it turns out, this problem as well as a number of others found in Babbitt's compositional process are computationally difficult to solve and it is these problems posed by the techniques found in his music, discussed in Part II, that will be the primary focus of this thesis. It is precisely because a computational method has been adopted here (and not, for example, a traditional music-theoretical approach) that the real difficulty in understanding Babbitt's

music, his compositional process and the problems therein have been made clear.

In Part II of this thesis, Papers A and B address the problem of analyzing Babbitt's music. Paper A provides a method for assessing the structure of an integral component of Babbitt's music, the all-partition array. Paper B presents a heuristic for predicting the time-point rows in Babbitt's music, which are responsible for determining the dynamic levels and in part, the rhythms, in many of his later works. Papers C, D and E address the problem of generating an all-partition array from a given array of 12-tone rows. Solving this problem is computationally difficult. Furthermore, few known solutions exist. Each of these three papers provides a different method for generating an all-partition array. Paper F provides a model based on the contributions of Paper B and E for generating from a complete all-partition array, a novel musical work in the style of Babbitt.

Paper A The first paper in this thesis demonstrates that the problem of assessing the structure of an all-partition array for its sequence of partitions, requires solving a special-case of the *exact covering problem* found in computer science. The proposed method for solving this problem is a procedural backtracking algorithm. This algorithm runs from left to right in a given input with unknown structure and returns a sequence of partitions if the input is determined to satisfy the criteria of an all-partition array. The paper concludes by using this algorithm to analyze an existing all-partition array in Babbitt's music (found in his work for solo guitar entitled *Sheer Pluck* (1984)) and its solution is a sequence of compositions (i.e., ordered partitions) which differs from those that have been found previously in the analyses by music theorists.

Paper B This paper provides a method for determining the exact number of possible time-point rows from the ordered mosaics in an all-partition array and proposes a heuristic based on Rothgeb's *order inversions* measure and Povel and Essens' *clock induction* model for predicting from these possible rows, the actual time-point rows chosen by Babbitt in two of his later works, *None but the Lonely Flute* and *Around the Horn*. The proposed heuristic hypothesizes that, when choosing time-point rows, Babbitt attempted to minimize their dissimilarity with the pitch-class rows in a piece and the amount of counter-evidence they provide against a preselected metrical beat. Results indicate that, individually, the order inversions measure performs better than the measure of clock induction at predicting the time-point rows chosen by Babbitt in these pieces. However, the best overall predictive performance is achieved by using a weighted combination of both measures, where in some ordered mosaics, the heuristic was able to place Babbitt's chosen time-point row in the top 2 percent of all pos-

sible rows. The performance of this heuristic across all ordered mosaics in a piece is slightly better in *None but the Lonely Flute* than in *Around the Horn*.

Paper C This paper proposes a procedural backtracking algorithm (similar to that proposed in Paper A) with two heuristics for generating from a given 12-tone row, a particular type of all-partition array found in many of Babbitt's works, called a *Smalley* array. Satisfying all the constraints necessary for constructing a Smalley array is difficult. The two proposed heuristics guide the algorithm through the search space towards a solution by selecting partitions that minimize the difference in lyne lengths as horizontal pitch repetitions are added when constructing the all-partition array. The proposed algorithm using these heuristics was unable to discover a solution after 100000 backtracks. This result demonstrates that a significant drawback to the proposed algorithm is any method of solving from left to right will make finding the final partition extremely difficult without good heuristics or some other means for drastically reducing the search space.

UPDATE: As of the submission of this thesis, solutions to the problem of generating a successful sequence of ordered mosaics each represented by a distinct partition from both a Smalley array and a non-self-contained array (found in Babbitt's *About Time*), have been found using the proposed backtracking procedure in Paper C with the addition of hard constraints for reducing the search space.

Paper D This paper provides an integer programming (IP) model of the problem of generating from a given array of 12-tone rows, a complete six-part all-partition array. A significant contribution of this paper demonstrates that the problem of generating an all-partition array can be formulated as a special-case of the set-covering problem (SCP) found in computer science and operations research, by representing the horizontal pitch repetitions required in an array as *overlaps* from one region to another. This fundamentally changes the nature of the problem in a way that can now be expressed as a mathematical problem of *covering* and not partitioning, as is commonly thought. The proposed IP model was unable to discover a solution to this problem, however, solutions to smaller, different-sized instances indicate that each represents a unique problem space and that, as their size increases, the solving time grows exponentially.

Paper E This paper provides a constraint programming (CP) model of the problem of generating from a given array of 12-tone rows, a complete four-part all-partition array. The focus of this paper is on Babbitt's smaller, four-part all-partition arrays and not the larger instances addressed in Papers C and D. The proposed CP model discovered a solution to this

problem after implementing a method for dividing its array into two, smaller instances, solving these and then re-joining them to form a complete solution. The discovered solution is the first known and published all-partition array to be generated entirely automatically by computer. Moreover, its structure differs from those previously discovered by Babbitt and others using presumably manual methods.

Paper F This paper proposes a method for generating from a completed all-partition array, music in the style of Babbitt's later works based on the time-point system and using what have been called *equal-note-value strings*. The proposed method uses the model described in Paper E to generate an all-partition array and the heuristic presented in Paper B to construct time-point rows that likely would have been chosen by Babbitt. From these, the proposed model outputs a sequence of equal-note-value strings containing pitch repetitions, rests and ties which collectively form the surface of a generated piece of music. As this model reveals, Babbitt's method for constructing equal-note-value strings requires an exhaustive partition of the pitch-class segments in an all-partition array by the collective sequence of equal-note-value strings which appear on the musical surface. Discovering such an exhaustive partition is a difficult computational problem and severely constrains the equal-note-value strings possible in Babbitt's music. Due to this difficulty, the proposed method for solving this problem adopts a greedy approach which relaxes the constraint for an exhaustive partition. The paper concludes with a generated piece for string quartet and flute that is the first-ever automatically generated music in the style of Babbitt. The generated parameters of this piece include pitch, rhythm (onsets and durations), voice, dynamic level, and meter.

A significant focus of this thesis is on generating an all-partition array (see Papers C, D and E). The primary reasons for this are that (1) it is not entirely clear how Babbitt and others (using presumably manual methods) have managed to construct an all-partition array, (2) generating this structure is a computationally difficult combinatorial problem, and (3) this structure forms the basis from which a number of additional parameters in many of his pieces are composed. Despite the significant amount of research on Babbitt's music, there has not been any attempt, beyond the construction of CMs (described in Section 5.2) to provide an explanation for how an all-partition array is constructed. It is likely, however, that the method proposed in Paper E of first dividing an array into smaller sections and then solving these smaller problems is how humans have approached the problem by hand.[17]

[17]David Smalley constructed his all-partition array by dividing the whole problem into smaller sections and solved these smaller problems. (David Smalley, personal email communication, February 25, 2015).

The research presented in this thesis represents the first published efforts to automatically generate an all-partition array. Despite these efforts, the problem largely remains unsolved, having found solutions to only the smallest instances found in Babbitt's music (i.e., a four part and six-part array) with two of the proposed methods. It should be noted, that the proposed methods have not been used in the larger, 12-part all-partition arrays containing 77 partitions found in Babbitt's music nor have any arrays not found in Babbitt's music been used as the basis for attempting to construct a new all-partition array.

Most models of music generation *undergenerate* with respect to the style of the music they are intended to output—that is, these systems are incapable of generating one or more structures or characteristics found in the style they are attempting to model. It is important to note that the model proposed in this thesis, however, does not undergenerate. The set of algorithms proposed by the model presented here collectively defines a class of all possible and "well-formed" pieces in the style of Babbitt according to the structures and techniques Babbitt himself used to compose his music. This model then generates as its output, members of this class—a class of which Babbitt's own pieces are also members. This is not to say, however, that all the pieces generated by this model are ones that Babbitt would have found to be "good". Nonetheless, initial results obtained using this model are promising, both from an analytical and aural perspective.

7 Conclusion

Over the years, a wide array of algorithms for the analysis and generation of both tonal and post-tonal music have been proposed. Almost none of these algorithms, however, have been applied to Babbitt's music, despite a mathematical and procedural nature to his compositional process strongly lending itself to their use. Moreover, despite the vast amount of literature concerning Babbitt's works, there remain several gaps in knowledge regarding his music and compositional process that this thesis hopes to fill. In manual analyses of Babbitt's works, it is generally sufficient to acknowledge the type of all-partition array in a given work without exploring, for example, how this structure was created to begin with or how severely its use constrains Babbitt's decisions at any one point in the compositional process regarding the possible pitches, dynamic levels or rhythmic figures. For this reason, there has not been a clear understanding of how an all-partition array is actually constructed or how difficult it was for Babbitt to satisfy the constraints needed to compose his works. In part, this thesis explores the answer to these questions by providing a more suitable and mathematical definition of the all-partition array which, in turn, allows for the proposal of

computational methods for solving the problem of generating an all-partition array. Having now a more appropriate definition and understanding of the all-partition array allows for completing certain tasks in Babbitt's music that otherwise were either not previously possible or extremely difficult to complete by hand, such as determining which all-partition array is used in a given work, predicting what time points Babbitt might have chosen or discovering an alternative sequence of partitions in a given array. Finally, this thesis provides a method for automatically generating from an all-partition array, the musical surface of entirely new works in the style of Babbitt. Generating such a musical work requires solving an altogether new and difficult computational problem. As this thesis shows, it is remarkable that Babbitt himself was able to manually satisfy all the constraints necessary to construct an all-partition array and subsequently create a piece of music based on this array.

References

Agres, B., Forth, J. and Wiggins, G. (2016). Evaluation of musical creativity and musical metacreation systems. *ACM Computers in Entertainment*, 14(3).

Alegant, B. (1993). *The seventy-seven partitions of the aggregate: Analytical and theoretical implications*. Ph.D. diss., University of Rochester.

Alphonce, B. (1974). *The Invariance Matrix*. Ph.D. diss., Yale University.

Alphonce, B. H. (1980). Music analysis by computer. *Computer Music Journal*, 4(2):26–35.

Anders, T., Anagnostopoulou, C., and Alcorn, M. (2005). Strasheela: Design and usage of a music composition environment based on the OZ programming model. In Roy, P., editors, *Second International Conference, MOZ 2004*, volume 3389 of *Multiparadigm Programming in Mozart/OZ*. Springer-Verlag, Berlin.

Ariza, C. (2005). Navigating the Landscape of Computer-Aided Algorithmic Composition Systems: A Definition, Seven Descriptors, and a Lexicon of Systems and Research. *Proceedings of the International Computer Music Conference*, pp. 765–772.

Assayag, G., Rueda, C., Laurson, M., Agon, C., and Delerue, O. (1999). Computer-Assisted Composition at IRCAM: From PatchWork to OpenMusic. *Computer Music Journal*, 23(3):59–72.

Babbitt, M. (1955). Some aspects of twelve-tone composition. *The Score and I.M.A. Magazine*, 12:53–61.

Babbitt, M. (1960). Twelve-tone invariants as compositional determinants. *Journal of Music Theory*, 46(2):246–259.

Babbitt, M. (1961). Set structure as a compositional determinant. *Journal of Music Theory*, 5(1):72–94.

Babbitt, M. (1962). Twelve-tone rhythmic structure and the electronic medium. *Perspectives of New Music*, 1(1):49–79.

Babbitt, M. (1965). The Structure and Function of Musical Theory. *College Music Symposium 5*.

Babbitt, M. (1973). Since Schoenberg. *Perspectives of New Music*, 12(1/2):3–28.

Babbitt, M. and Peles, S. (2003). *The collected essays of Milton Babbitt*. Princeton: Princeton University Press.

Bazelow, A. R. and Brickle, F. (1976). A partition problem posed by Milton Babbitt (Part I). *Perspectives of New Music*, 14(2):280–293.

Bazelow, A. R. and Brickle, F. (1979). A combinatorial problem in music theory: Babbitt's partition problem (I). *Annals of the New York Academy of Sciences*, 319(1):47–63.

Bemman, B. and Meredith, D. (2014). From analysis to surface: Generating the surface of Milton Babbitt's *Sheer Pluck* from a parsimonious encoding of an analysis of its pitch-class structure. In *The Music Encoding Conference, 20–23 May 2014*, Charlottesville, VA.

Bemman, B. and Meredith, D. (2015a). Comparison of heuristics for generating all-partition arrays in the style of Milton Babbitt. In *CMMR 11th International Symposium on Computer Music Multidisciplinary Research, 16–19 June 2015*, Plymouth, UK.

Bemman, B. and Meredith, D. (2015b). Exact cover problem in Milton Babbitt's all-partition array. In Collins, T., Meredith, D., and Volk, A., editors, *Mathematics and Computation in Music: Fifth International Conference, MCM 2015, London, UK, June 22–25, 2015, Proceedings*, volume 9110 of *Lecture Notes in Artificial Intelligence*, pages 237–242. Springer, Berlin.

Bemman, B. and Meredith, D. (2016). Generating Milton Babbitt's all-partition arrays. *Journal of New Music Research*, 45(2):1–21.

Berg, P. (2011). Using the AC Toolbox. Institute of Sonology, Royal Conservatory, The Hague.

Bernstein, Z. (2014). The problem of completeness in Milton Babbitt's music and thought. In *43rd Annual Meeting of The Music Theory Society of New York State (5–6 April 2014)*, New York, NY.

Butler, D. (1990). A study of event hierarchies in tonal and post-tonal music. *Psychology of Music*, 18(1):4–17.

Cambouropoulos, E. (2001). The Local Boundary Detection Model (LBDM) and its Application in the Study of Expressive Timing. In *Proceedings of the International Computer Music Conference, 17–22 September, 2001*, Havanna, Cuba.

Cambouropoulos, E. (2008). Voice and stream: Perceptual and computational modeling of voice separation. *Music Perception*, 26(1):75–94.

Cambouropoulos, E., Kaliakatsos-Papakostas, M., and Tsougras, C. (2014). An Idiom-independent Representation of Chords for Computational Music Analysis and Generation. In *Proceedings of the Joint ICMC/SMC Conference, 14–20 September, 2014*, Athens, Greece.

Carpentier, G., Assayag, G., and Saint-James, E. (2010). Solving the musical orchestration problem using multiobjective constrained optimization with a genetic local search approach. *Heuristics*, 16(5):681–714.

Chemillier, M. and Truchet, C. (2001). Two musical CSPs. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*.

Chew, E. (2005). Regards on two regards by Messiaen: Post-tonal music segmentation using pitch context distances in the spiral array. *Journal of New Music Research*, 34(4):341–354.

Chew, E. (2014). Mathematical and computational modeling of tonality: Theory and applications. volume 204 of *International Series on Operations Research and Management Science*. Springer, New York, NY.

Collins, T., Laney, R., Willis, A. and FGarthwaite, P. H. (2011). Modeling pattern importance in Chopin's mazurkas. *Music Perception*, 28(4):387–414.

Conklin, D. (2013). Multiple viewpoint systems for music classification. *Journal of New Music Research*, 42(1):19–26.

Cope, D. (1987). An expert system for computer-assisted composition. *Computer Music Journal*, 11:30–46.

Cope, D. (2008). *Hidden Structure: Music Analysis Using Computers*. A-R Editions, Madison, WI.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition.

Cuciurean, J. (1997). Self-similarity and compositional strategies in the music of Milton Babbitt. *Canadian University Music Review*, 17(2):1–16.

Desain, P. and Honing, H. (1999). Computational Models of Beat Induction: The Rule-Based Approach. *Journal of New Music Research*, 28(1):29–42.

Deutsch, D. and Feroe, J. (1981). The internal representation of pitch sequences in tonal music. *Psychological Review*, 88(6):503–522.

Dibben, N. (1990). The perception of structural stability in atonal music: The influence of salience, stability, horizontal motion, pitch commonality, and dissonance. *Psychology of Music*, 16(3):265–294.

Dubiel, J. (1997). What's the use of the twelve-tone system? *Perspectives of New Music*, 35(2):33–51.

Dubnov, S., Assayag, G., Lartillot, 0., and Bejerano, G. (2003). Using machine-learning methods for musical style modeling. *Computer Magazine*, 36(10):73–80.

Ebcioğlu, K. (1987). Report on the CHORAL project: An expert system for harmonizing four-part chorales. Technical Report RC12628. `http://global-supercomputing.com/people/kemal.ebcioglu/pdf/RC12628.pdf`.

Eck, D. (2001). A positive-evidence model for rhythmical beat induction. *Journal of New Music Research*, 30(2):187–200.

Eger, S. (2013). Restricted weighted integer compositions and extended binomial coefficients. *Journal of Integer Sequences*, 16(13.1.3):1–25.

Fernandez, J. and Vico, F. (2013). AI Methods in Algorithmic Composition: A Comprehensive Survey. *Journal of Artificial Intelligence Research*, 48:513–582.

Forte, A. (1966). A program for the analytic reading of scores. *Journal of Music Theory*, 10(2):330–364.

Forte, A. (1973). *The Structure of Atonal Music*. New Haven, London: Yale University Press.

Forth, J. (2012). *Cognitively-motivated geometric methods of pattern discovery and models of similarity in music*. Ph.D. diss., Goldsmiths, University of London.

Gill, S. (1963). A technique for the composition of music in a computer. *The Computer Journal*, 6(2):129–133.

Gross, D. (1975). *A set of computer programs to aid in musical analysis*. Ph.D. diss., Indiana University.

Harris, C. and Brinkman, A. (1989). An Integrated Software System for Set-Theoretic and Serial Analysis of Contemporary Music. *Journal of Computer-based Instruction*, 16(2):59–70.

Hasty, C. (1999). *Meter as Rhythm*. Oxford University Press, New York, NY.

Helmholtz, H. von. (1910/1962). Treatise on Physiological Optics. *3rd German edition of Handbuch der physiologischen Optik*. Voss, Hamburg and Dover, New York, 1910/1962.

Herremans, D. and Sörensen, K. (2013). Composing fifth species counterpoint music with a variable neighborhood search algorithm. *Expert systems with applications*, 40(16):6427–6437.

Herwaarden, S. Van., Grachten, M. and De Haas, W. B. (2014). Predicting expressive dynamics in piano performances using neural networks. In *15th International Society for Music Information Retrieval Conference, October 27–31, 2014, Proceedings*, Taipei, Taiwan.

Hiller, L., and Isaacson, L. (1959). *Experimental Music: Composition With an Electronic Computer*. New York: McGraw-Hill.

Huron, D. and Parncutt, R. (1993). An improved model of tonality perception incorporating pitch salience and echoic memory. *Psychomusicology*, 12(2):154–171.

Huron, D. (1997). Humdrum and Kern: selective feature encoding *Beyond MIDI: the handbook of musical codes*, MIT Press, Cambridge, MA.

Huron, D. (2006). *Sweet Anticipation: Music and the Psychology of Expectation*. MIT Press, Cambridge, Massachusetts.

Johnson, W. M. (1984). Time-point sets and meter. *Perspectives of New Music*, 23(1):278–293.

Kassler, M. (1963). A sketch of the use of formalized languages for the assertion of music. *Perspectives of New Music*, 1(2):83–94.

Kassler, M. (1967). Toward a theory that is the twelve-note-class system. *Perspectives of New Music*, 5(2):1–80.

Knuth, D. (2000). Dancing links. Technical report. Available online at `http://www-cs-faculty.stanford.edu/~uno/musings.html`.

Koffka, K. (1935). *Principles of Gestalt psychology*. Harcourt, Oxford, England.

Kowalski, D. (1985). *The array as a compositional unit: A study of derivational counterpoint as a means of creating hierarchical structures in twelve-tone music*. Ph.D. diss., Princeton University.

Kowalski, D. (1987). The construction and use of self-deriving arrays. *Perspectives of New Music*, 25(1/2):296–361.

Krumhansl, C. L. (1983). Perceptual structures for tonal music. *Music Perception*, 1:24–58.

Krumhansl, C. L., Sandell, G. J., and Sergeant, D. C. (1987). The perception of tone hierarchies and mirror forms in twelve-tone serial music. *Music Perception: An Interdisciplinary Journal*, 5(1):31–77.

Krumhansl, C. L. (1990). *Cognitive Foundations of Musical Pitch*. Oxford University Press, New York.

Lake, W. E. (1986). The architecture of a superarray composition: Milton Babbitt's String Quartet no. 5. *Perspectives of New Music*, 24(2):88–111.

Laurson, M. and Kuuskankare, M. (2001). A constraint based approach to musical textures and instrumental writing. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*.

Lee, C. (1991). The perception of metrical structure: Experimental evidence and a model. In P. Howell, R. West, and I. Cross, eds. *Representing Musical Structure*, volume 5 of *Cognitive Science Series*, pages 59–127. Academic Press, London.

Leong, D. and McNutt, E. (2005). Virtuosity in Babbitt's *Lonely Flute*. *Music Theory Online*, 11(1).

Lerdahl, F. and Jackendoff, R. (1983). *A Generative Theory of Tonal Music*. Cambridge, MA: MIT Press.

Lerdahl, F. (1989). Atonal prolongational structure. *Contemporary Music Review*, 3(2):65–87.

Lerdahl, F. (2001). *Tonal Pitch Space*. Oxford University Press, New York.

Lewin, D. (1976). On partial ordering. *Perspectives of New Music*, 14(2):252–257.

London, J. (2004). *Hearing in Time*. Oxford University Press, New York, NY.

Longuet-Higgins, H. C. and Steedman, M. J. (1971). On interpreting Bach. *Machine Intelligence*, 6:221–241.

Martino, D. (1961). The source set and its aggregate formations. *Journal of Music Theory*, 5(2):224–273.

McCartney, J. (2002). Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4):61–68.

Mead, A. (1984). Recent developments in the music of Milton Babbitt. *The Musical Quarterly*, 70(3):310–331.

Mead, A. (1987). About *About Time*'s time: A survey of Milton Babbitt's recent rhythmic practice. *Perspectives of New Music*, 25:182–235.

Mead, A. (1994). *An Introduction to the Music of Milton Babbitt*. Princeton University Press, Princeton, NJ.

Mead, A. (1997). Still being an American composer: Milton Babbitt at eighty. *Perspectives of New Music*, 35(2):101–126.

Meredith, D., Lemström, K. and Wiggins, G.A (2002). Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, 31(4):321–345.

Meredith, D., Lemström, K., and Wiggins, G. A. (2003). Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. In *Cambridge Music Processing Colloquium 2003, Department of Engineering, University of Cambridge*.

Meredith, D. (2012). A geometric language for representing structure in polyphonic music. In *13th International Society for Music Information Retrieval Conference (ISMIR 2012), 8–12 October 2012*, pages 133–138, Porto, Portugal.

Meredith, D. (2013). COSIATEC and SIATECCompress: Pattern discovery by geometric compression. In *MIREX 2013 (Competition on Discovery of Repeated Themes & Sections)*. Available online at `http://www.titanmusic.com/papers/public/MeredithMIREX2013.pdf`.

Meredith, D. (2015). Music analysis and point-set compression. *Journal of New Music Research*, 44(3), pp. 245–270.

Miller, G. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information.. *Psychological Review*, 63(2):81–97.

Morris, R. (1987). *Composition with Pitch-Classes: A Theory of Compositional Design*. Yale University Press, New Haven, CT.

Morris, R. (1993). New Directions in the Theory and Analysis of Musical Contour. *Music Theory Spectrum*, 15(2):205–228.

Morris, R. (2003). Pitch-class duplication in serial music: Partitions of the double aggregate. *Perspectives of New Music*, 41(2):96–121.

Morris, R. (2010). *The Whistling Blackbird: Essays and Talks on New Music*. The University of Rochester Press, Princeton, NJ.

Morris, R. and Alegant, B. (1988). The even partitions in twelve-tone music. *Music Theory Spectrum*, 10:74–101.

Orman, A. J. and Williams, H. P. (2006). A survey of different integer programming formulations of the travelling salesman problem. In Kontoghiorghes, E. J. and Gatu, C., eds. *Optimisation, Econometric and Financial Analysis*, volume 9 of *Advances in computational management science*, pages 93–108. Springer-Verlag Berlin Heidelberg, Berlin.

Page, D. (2013). Parallel algorithm for second-order restricted weak integer composition generation for shared memory machines. *Parallel Processing Letters*, 23(3):1350010.

Pearce, M. (2005). *The Construction and Evaluation of Statistical Models of Melodic Structure in Music Perception and Composition*. Ph.D. diss., Department of Computing, City University, London, UK.

Povel, D. J. and Essens, P. (1985). Perception of temporal patterns. *Music Perception: An Interdisciplinary Journal*, 23(3):411–440.

Puckette, M. (2002). Max at Seventeen. *Computer Music Journal*, 26(4):31–43.

Puget, J. F. and Régin, J C. (2007). Solving the all interval problem. Technical report. Available online at `https://ianm.host.cs.st-andrews.ac.uk/CSPLib/prob/prob007/puget.pdf`.

Rothgeb, J. (1967). Some ordering relationships in the twelve-tone system. *Journal of Music Theory*, 11(2):176–197.

Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey.

Schuijer, M. (2008). *Analyzing Atonal Music: Pitch-class Set Theory and Its Contexts*. Eastman Studies in Music, University of Rochester Press.

Scott, D. (1958). Programming a combinatorial puzzle. Technical Report 1, Dept. of Electrical Engineering, Princeton University.

Shepard, R. (1982). Geometrical approximations to the structure of musical pitch. *Psychological Review*, 89(4):305–333.

Simon, H. and Sumner, R. (1968). Pattern in music. In Kleinmuntz, B., ed., *Formal Representation of Human Judgement*. New York: John Wiley.

Starr, D. and Morris, R. (1977). A general theory of combinatoriality and the aggregate, part 1. *Perspectives of New Music*, 16(1):3–35.

Starr, D. and Morris, R. (1978). A general theory of combinatoriality and the aggregate, part 2. *Perspectives of New Music*, 16(2):50–84.

Starr, D. (1984). Derivation and polyphony. *Perspectives of New Music*, 23(1):180–257.

Straus, J. N. (2009). *Twelve-Tone Music in America*. Cambridge University Press, New York.

Sward, R. L. G. (1981). *An Examination of the Mathematical Systems Used in Selected Compositions of Milton Babbitt and Iannis Xenakis*. Ph.D. diss., Northwestern University.

Tamura, N. and Banbara, M. (2008). Sugar: A CSP to SAT translator based on order encoding. In *Proceedings of the 2nd International CSP Solver Competition*, pages 65–69.

Tanaka, T. and Fujii, K. (2014). Melodic pattern segmentation of polyphonic music as a set partitioning problem. In *International Congress on Music and Mathematics, ICMM 2014, Puerto Vallarta, Mexico, June 22–25, 2015, Proceedings*, Springer, Berlin.

Tanaka, T. and Fujii, K. (2015). Describing global musical structures by integer programming. In Collins, T., Meredith, D., and Volk, A., eds., *Mathematics and Computation in Music: Fifth International Conference, MCM 2015, London, UK, June 22–25, 2015, Proceedings*, volume 9110 of *Lecture Notes in Artificial Intelligence*, pages 52–63. Springer, Berlin.

Tanaka, T., Bemman, B., and Meredith, D. (2016a). Integer programming formulation of the problem of generating Milton Babbitt's all-partition arrays. In *17th International Society for Music Information Retrieval (ISMIR2016), 7–11 August 2016*, New York, USA.

Tanaka, T., Bemman, B., and Meredith, D. (2016b). Constraint programming approach to the problem of generating Milton Babbitt's all-partition arrays. In Rueher, M., ed., *22nd International Conference Principles and Practice of Constraint Programming, CP 2016, Toulouse, France, September 5–9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 802–810. Springer, Berlin.

Tani, N. and Bourboubi, S. (2011). Enumeration of the partitions of an integer into parts of a specified number of different sizes and especially two sizes. *Journal of Integer Sequences*, 14(11.3.6):1–12.

Temperley, D. (2001). *The Cognition of Basic Musical Structures*. MIT Press.

Temperley, D. (2007). *Music and Probability*. MIT Press.

Temperley, D. (2009). A Unified Probabilistic Model for Polyphonic Music Analysis. *Journal of New Music Research*, 38(1):3–18.

Trehub, S E. and Hannon, E E. (2006). Infant music perception: Domain-general or domain-specific mechanisms?. *Cognition*, 100(1):73-99.

Tymoczko, D. (2011). *A Geometry of Music: Harmony and Counterpoint in the Extended Common Practice*. Oxford University Press, New York.

Velarde, G., Weyde, T., Cancino-Chacon, C., Meredith, D. and Grachten, M. (2016). Composer Recognition based on 2D-Filtered Piano-Rolls. In *17th International Society for Music Information Retrieval Conference, August 7–11, 2016, Proceedings*, New York, NY.

Winham, G. (1970). Composition with arrays. *Perspectives of New Music*, 9(1):43–67.

Part II

Papers

Paper A

Exact Cover Problem in Milton Babbitt's All-partition Array

Bemman, B. and Meredith, D.

The paper has been published in
T. Collins, D. Meredith and A. Volk (eds.) Mathematics and Computation in Music: Fifth International Conference, MCM2015, London, UK, June 22–25, 2015 Proceedings, Lecture Notes in Artificial Intelligence, Vol. 9110, pp. 237–242, 2015.

© 2015 Springer, Berlin
The layout has been revised.

Abstract

One aspect of analyzing Milton Babbitt's (1916–2011) all-partition arrays requires finding a sequence of distinct, non-overlapping aggregate regions that completely and exactly covers an irregular matrix of pitch class integers. This is an example of the so-called exact cover problem. Given a set, A, and a collection of distinct subsets of this set, S, then a subset of S is an exact cover of A if it exhaustively and exclusively partitions A. We provide a backtracking algorithm for solving this problem in an all-partition array and compare the output of this algorithm with an analysis produced manually.

1 Introduction

The *exact cover problem* is a constraint satisfaction problem known to be NP-complete (Knuth, 2000, p. 2). It is defined as follows: given a collection of subsets, S, of a set, A, an *exact cover* of A is a sub-collection, s, of S that exhaustively and exclusively partitions A. The classic example of such a problem was provided by Scott and Trotter in 1958 Scott (1958). They found all ways to cover a chessboard with the 12 distinct, non-overlapping pentominoes while leaving the center four squares uncovered (see Fig. A.1).

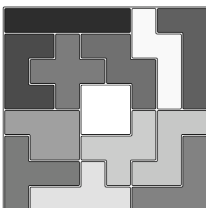


Fig. A.1: An exact covering of part of a chessboard using 12 pentominoes while leaving the center four squares uncovered. As taken from (Knuth, 2000, p. 2)

Following our definition above, the chessboard in Fig. A.1 would be A, the collection of 63 distinct pentominoes would be S, and the 12 distinct pentominoes selected to cover the chessboard would be s.

The exact cover problem is typically solved using a greedy backtracking algorithm that performs a depth-first search of the solution space (Knuth, 2000, p. 2). The backtracking process finds a complete solution to a problem by accumulating partial solutions to a set of constraints. It selects the first of these partial solutions until a complete solution is found, or, in the event that the constraints are no longer satisfied by the currently selected partial solution, it returns to the previous point and selects the next partial solution. It continues this process until either a solution is found or it fails.

11	4	3	5	9	10	10	1	1	8	2	0	7	6	5	5	4	4	11	9	3	10	1	2	6	8	7	0	9	10	3	5	11	4	4	1	0	0	8
6	7	7	0	2	2	8	1	10	9	5	5	3	4	4	11	11	0	7	8	8	6	2	1	10	3	9	11	4	5	8	1	0	2	6	7	7	10	5
5	6	11	1	7	0	9	9	8	4	2	2	3	10	1	1	0	7	5	11	6	9	10	2	4	3	8	5	0	0	1	11	7	6	3	8	8	2	4
2	9	10	8	4	3	3	3	0	5	11	1	6	7	7	4	9	8	10	2	3	6	1	7	5	0	11	2	3	8	10	4	9	6	5	1	11	11	0
0	0	5	4	6	6	6	10	11	2	9	3	1	8	7	4	5	10	0	6	11	8	7	3	1	2	9	4	11	0	10	6	5	2	7	1	3	3	8
1	8	8	9	7	3	2	11	11	4	10	10	0	5	6	3	2	9	9	7	1	8	11	0	4	6	5	10	9	2	1	3	7	8	11	6	0	10	10

Fig. A.2: The beginning of the WPcMatrix of Babbitt's *Sheer Pluck*.

11 4 3 5 9 10 10 1
6 7 7 0 2
5 6 11 1
2 9 10 8 4 3
0 0 5 4 6
1 8 8 9 7 3 2 11

(a) Area needing to be covered.

11 4 3 5 9 10 10 1
6 7 7 0 2
5 6 11 1
2 9 10 8 4 3
0 0 5 4 6
1 8 8 9 7 3 2 11

(b) 1st aggregate region.

11 4 3 5 9 10 10 1
6 7 7 0 2
5 6 11 1
2 9 10 8 4 3
0 0 5 4 6
1 8 8 9 7 3 2 11

(c) 2nd aggregate region (dashed lines).

11 4 3 5 9 10 10 1
6 7 7 0 2
5 6 11 1
2 9 10 8 4 3
0 0 5 4 6
1 8 8 9 7 3 2 11

(d) 3rd aggregate region (dashed lines). Area covered completely.

Fig. A.3: Process of forming a sequence of three aggregate regions in an excerpt from the WPc-Matrix shown in Fig. A.2.

2 All-partition Array as Exact Cover

A significant number of Milton Babbitt's (1916–2011) works are based on the *all-partition array* Mead (1994), which is a sequence of distinct, non-overlapping aggregate-forming regions that completely and exactly partition a matrix of pitch classes called a *projection*. Construction of a *six-part* all-partition array results in an irregular projection of six rows and 696 pitch classes that can be partitioned into 58 aggregate regions. Figure A.2 shows the beginning of the projection of Babbitt's *Sheer Pluck*. A WPcMatrix is not the musical surface but, rather, a framework upon which the surface is based. Figure A.3 illustrates the process of defining the first three aggregate regions in this projection.

Note in Figure A.3(b) that the first region contains an aggregate represented as a collection of row segments (from top to bottom) of length $3, 2, 1, 3, 1,$

and 2. We define an *integer partition*, denoted by $\text{IntPart}(s_1, s_2, ..., s_k)$, to be a representation of an integer $n = \sum_{i=1}^{k} s_i$, as an *unordered* sum of k positive integers. For example, if $n = 12$ and $k = 6$, then one possible integer partition is $\text{IntPart}(3, 3, 2, 2, 1, 1)$. We define an *integer composition*, denoted by $\text{IntComp}(s_1, s_2, ..., s_k)$, to be a representation of an integer $n = \sum_{i=1}^{k} s_i$, as an *ordered* sum of k positive integers. For example, if $n = 12$ and $k = 6$, then $\text{IntComp}(3, 3, 2, 2, 1, 1) \neq \text{IntComp}(3, 2, 1, 3, 2, 1)$. We define a *weak integer composition*, $\text{WIntComp}(s_1, s_2, ..., s_k)$, to be a representation of an integer, $n = \sum_{i=1}^{k} s_i$, as an ordered sum of k *non-negative integers*. For example, if $n = 12$ and $k = 6$, then $\text{WIntComp}(6, 6, 0, 0, 0, 0)$ is a weak integer composition. Thus, the first aggregate region in *Sheer Pluck*, shown in Fig. A.3(b), represents the integer partition, $\text{IntPart}(3, 3, 2, 2, 1, 1)$, and the integer composition, $\text{IntComp}(3, 2, 1, 3, 1, 2)$. We further define two relations, *partitionally equivalent* and *partitionally distinct*. Two integer compositions, c and d, are *partitionally equivalent* if and only if $[c] = [d]$, where $[c]$ and $[d]$ denote the partitions containing the compositions. Two integer compositions, c and d, are *partitionally distinct* if and only if $[c] \neq [d]$.

Using our definitions above, the problem we pose with respect to the all-partition array as an exact cover asks, "Given a universe of integer compositions (when $n = 12$ and $k = 6$), denoted by S, and a WPcMatrix of 696 pitch classes in six rows, denoted by A, does there exist a sequence of 58 partitionally distinct, and aggregate-forming integer compositions, s, that exactly covers A?" We call this the *projection cover problem*. Our efforts to answer this question continue work started by Bazelow and Brickle (Bazelow and Brickle, 1976, pp. 282–283), that asked a similar question of all-partition arrays in four parts. However, where their research sought to construct a WPcMatrix, this paper begins with a completed WPcMatrix and, as a method for musical analysis, seeks to efficiently reveal its all-partition array structure by discovering how (or if) it can be partitioned.

3 Solving the projection cover problem

Our proposed solution to the projection cover problem posed above, is the backtracking algorithm, BacktrackingBabbitt, shown in Fig. A.4. This algorithm takes a projection as input and returns a list of 58 partitionally distinct compositions chosen as partial solutions.

The algorithm begins in line 1 by computing a $6,188 \times 6$ list of compositions (in six parts), denoted by **Compositions**. Lines 2–4 initialize **CList**$[Cnt]$ to be an empty list of 58 lists, **Position** to be a 1×6 vector of indices (one for each row in **WPcMatrix**), and Cnt to be 1 (using 1-based indexing). Line 5 begins a **while** loop where Cnt is less than or equal to the number of required compositions, 58. First, it checks to see whether **CList**$[Cnt]$ has been

```
BacktrackingBabbitt(projection)
1   compositions ← ComputeCompositions(12, 6)
2   cList ← ⊕_{i=1}^{58} ⟨⟨⟩⟩
3   position ← ⊕_{i=1}^{6} ⟨1⟩
4   cnt ← 1      ▶ Index into cList
5   while cnt <= 58 and cnt > 0      ▶ Number of compositions
6      if |cList[cnt]| == ⟨⟩
7         ParseProjection(projection, compositions, partialSolutions, cnt, position)
          ▶ Returns cList and currentComp
8         if |cList[cnt]| == ⟨⟩      ▶ Backtrack
9            cnt ← cnt − 1
10           position ← position − currentComp
11           partialSolutions[cnt] ← ⟨⟩
12        else      ▶ Success
13           partialSolutions[cnt] ← currentComp
14           position ← position + currentComp
15           cnt ← cnt + 1
16     else
17        currentComp ← cList[cnt][currentComp.index + 1]      ▶ Select next composition
18        if cList[cnt][currentComp.index] > |cList[cnt]|      ▶ Backtrack
19           cnt ← cnt − 1
20           position ← position − currentComp
21           partialSolutions[cnt] ← ⟨⟩
22        else      ▶ Success
23           partialSolutions[cnt] ← currentComp
24           position ← position + currentComp
25           cnt ← cnt + 1
26  return partialSolutions
```

Fig. A.4: Pseudocode for implementation of BacktrackingBabbitt.

computed (line 6). **CList** contains candidate compositions at each *Cnt*. Candidate compositions are those compositions that satisfy the constraints of a partial solution (i.e., are partitionally distinct and form a region containing an aggregate).

If **CList**[*Cnt*] is empty (line 6), it has not been previously computed and so it calls ParseWPcMatrix, which returns **CList** and **CurrentComp** (line 7). **CurrentComp** is initialized by ParseWPcMatrix to be the first composition in **CList**[*Cnt*] if **CList**[*Cnt*] is not returned empty. If, after ParseWPcMatrix, **CList**[*Cnt*] remains empty, there are no candidate compositions at this *Cnt*. It must then backtrack, removing the previous composition from **PartialSolutions** (lines 8-11). **PartialSolutions** is a 58×6 list of candidate compositions at each *Cnt* selected by the algorithm to be a partial solution. If **CList**[*Cnt*] is not empty (line 12), then the algorithm has found at least one candidate composition at this *Cnt*. The **CurrentComp** is stored in **PartialSolutions**[*Cnt*] and both **Position** and *Cnt* are incremented (lines 13-15). **Position** is equivalent to counting from 1 a distance equal to the summation of like parts from each composition in **PartialSolutions** from 1 to $Cnt - 1$. **Position** is incremented at each *Cnt* by **CurrentComp**. In Figure A.3(d), **PartialSolutions** currently holds $\langle 3,2,1,3,1,2 \rangle$ and $\langle 3,3,3,3,0,0 \rangle$ and so **Position** would be $\langle 7,6,5,7,2,3 \rangle$.

If the first check for whether **CList** is empty (line 6) returns false,

```
ParseProjection(projection, compositions, partialSolutions, cList, cnt, position)
1    comps ← compositions
2    comps ← RemoveUsedCompositions(partialSolutions, comps)
3    k ← 1
4    for i ← 1 to |comps|        ► By row.
5       aggregate ← ∅
6       for j ← 1 to |comps|        ► By column.
7          if comps[i][j] ≠ 0
8             aggregate ← projection[j][position[j]..position[j] + comps[i][j] − 1]
              ► Add jth row segments to form region
9       aggregate ← RemoveDuplicates(aggregate)
10      if |aggregate| == 12
11         cList[cnt][k] ← comps[i][1..6]
12         k ← k + 1
13   currentComp ← cList[cnt][1]
14   return ⟨cList, currentComp⟩
```

Fig. A.5: The ParseWPcMatrix function.

CList[*Cnt*] has already been computed. This means the algorithm has backtracked at some point (line 16). It then attempts to select the next composition in **CList**[*Cnt*] by incrementing the index of **CurrentComp** (line 17). If there is not a next composition here because this index exceeds the size of **CList**[*Cnt*] (line 18), it must backtrack (lines 19-21). However, if there is another composition, it can proceed (lines 23-25). It continues this until it returns a complete **PartialSolutions** or fails (line 26).

Figure A.5 shows pseudocode for the ParseWPcMatrix function called in line 7 of BacktrackingBabbitt. ParseWPcMatrix begins by creating a copy of **Compositions** called **AllUnusedComps** (line 1). Next, it removes from **AllUnusedComps** compositions partitionally equivalent to those already selected as partial solutions (line 2). It then loops through the rows and columns of **AllUnusedComps** (lines 4-6) and initializes *MosaicPcs* to be an empty set (line 5). Next, it finds *j*th row segments in **WPcMatrix** parsed by **AllUnusedComps**[*i*][*j*] using the distance measured from **Position**[*j*] to the sum of **Position**[*j*] and **AllUnusedComps**[*i*][*j*] − 1. It stores these segments in *MosaicPcs* (lines 7–8). After removing any duplicate integers from *MosaicPcs* (line 10), if *MosaicPcs* is complete, it has found a candidate composition and saves this composition in **CList**[*Cnt*][*k*] (lines 10-12). The algorithm then assigns **CurrentComp** to be the first composition in **CList**[*Cnt*] and returns **CList** and **CurrentComp** (lines 13–14).

We conclude this section by providing the results of analyzing one of Babbitt's projections from both BacktrackingBabbitt and those found by a human analyst. Figure A.6(a) first shows the sequence of compositions found by a human analyst to partition the projection shown in Figure A.2. For comparison, Figure A.6(b) shows one of several sequences returned by BacktrackingBabbitt.

3	2	1	3	1	2	6	6	0	0	0	0
3	3	3	3	0	0	4	4	0	2	1	1
2	0	0	0	4	6	2	2	2	2	1	3
0	6	2	1	1	2	0	1	0	0	11	0
5	3	1	0	3	0	1	2	2	5	0	2
0	0	4	7	0	1	0	1	9	2	0	0
2	2	0	0	6	2	1	1	1	0	1	8*
0	2	4	4	0	2	0	4	0	3	4	1
0	2	1	0	6	3	1	1	0	7	3	0
2	0	1	4	0	5	0	0	0	0	0	12
0	7	3	0	0	2	8	0	2	1	1	0
0	0	7	5	0	0	5	5	1	1	0	0
8	0	2	0	2	0	1	1	7	2	1	0
3	0	0	0	6	3	1	0	0	0	10	1
3	9	0	0	0	0	0	1	1	9	0	1
1	0	5	6	0	0	0	0	0	4	4	4
2	0	2	0	7	1	5	4	0	3	0	0
3	5	1	1	1	1	0	1	3	1	1	6
0	0	3	1	0	8	5	5	2	0	0	0
5	2	2	3	0	0	1	3	0	1	5	2
0	2	2	2	4	2	0	0	10	0	2	0*
5	2	1	1	2	1	1	3	3	3	0	2
0	0	4	8	0	0	4	3	0	1	1	3
1	1	1	1	7	1	3	0	2	4	2	1
4	5	0	1	1	1	0	2	3	0	4	3
4	1	0	1	0	6	1	4	1	1	1	4
1	3	1	1	3	3	2	1	4	2	2	1
1	1	6	1	1	2	3	2	1	1	4	1
2	2	2	2	2	2	2	2	2	3	0	3

(a) Compositions found by human analyst.

3	2	1	3	1	2	6	6	0	0	0	0
3	3	3	3	0	0	4	4	0	2	1	1
2	0	0	0	4	6	2	1	2	2	2	3
0	6	2	1	1	2	0	1	0	0	11	0
5	3	1	0	3	0	1	2	2	5	0	2
0	0	4	7	0	1	0	1	9	2	0	0
2	2	0	0	6	2	1	1	0	1	1	8*
0	2	4	4	0	2	0	4	0	3	4	1
0	2	1	0	6	3	1	0	1	7	3	0
2	0	1	4	0	5	0	0	0	0	0	12
0	7	3	0	0	2	8	0	2	1	1	0
0	0	7	5	0	0	5	5	1	1	0	0
8	0	2	0	2	0	1	1	7	2	1	0
3	0	0	0	6	3	1	0	0	0	10	1
3	9	0	0	0	0	0	1	1	9	0	1
1	0	5	6	0	0	0	0	0	4	4	4
2	0	2	0	7	1	5	4	0	3	0	0
3	5	1	1	1	1	0	1	3	1	1	6
0	0	3	1	0	8	5	5	2	0	0	0
5	3	0	2	0	2	1	3	0	1	5	2
0	2	2	2	4	2	0	0	10	0	0	2*
5	2	1	1	2	1	1	3	3	3	2	0
0	0	4	8	0	0	4	3	0	1	1	3
1	1	1	1	7	1	3	0	1	4	2	2
4	5	0	1	1	1	0	2	3	0	4	3
4	1	0	1	0	6	1	4	1	1	1	4
1	3	1	1	3	3	2	1	4	2	2	1
2	1	6	1	1	1	3	2	1	1	4	1
2	2	2	2	2	2	2	2	2	3	0	3

(b) **partialSolutions** returned by BacktrackingBabbitt.

Fig. A.6: Distinct sequences of compositions that partition the WPcMatrix shown in Figure A.2 as found by a human analyst in (a) and returned by BacktrackingBabbitt in (b). Note asterisks (*) indicate where the sequences differ.

4 Conclusion

In this paper we suggest that analyzing Milton Babbitt's all-partition arrays represents a special case of a constraint satisfaction problem called an exact cover. We provide a backtracking algorithm called BacktrackingBabbitt as a solution to this problem. This algorithm finds a sequence of 58 partitionally distinct and aggregate-forming integer compositions that exactly covers a given projection of 696 pitch class integers. We believe this algorithm is not only a more efficient way (when compared to a human analyst) to perform this analytical task for a work based on the all-partition array, but that it can be used to discover alternative analyses to those offered previously by theorists.

References

Bazelow, A. R. and Brickle, F. (1976). A partition problem posed by Milton Babbitt (Part I). *Perspectives of New Music*, 14(2):280–293.

Knuth, D. (2000). Dancing Links. `http://www-cs-faculty.stanford.edu/~uno/musings.html`.

Mead, A. (1994). *An Introduction to the Music of Milton Babbitt*. Princeton University Press, Princeton, NJ.

Scott, D. (1958). Programming a Combinatorial Puzzle In: Technical Report No. 1. *Princeton University Department of electrical Engineering*, 10 June, Princeton, NJ.

Paper B

Predicting Babbitt's Time Points in None but the Lonely Flute and Around the Horn

Bemman, B. and Meredith, D.

The paper has been submitted to the
Journal of Mathematics and Music, 2017.

Currently in review
The layout has been revised.

Abstract

Milton Babbitt (1916–2011) is credited with developing several techniques of 12-tone composition that extend beyond pitch. One such technique, the time-point row, figures prominently in his mature rhythmic practice. In many of his works based on the all-partition array, the available time-point rows that may appear on the musical surface are the same as the available pitch-class rows. However, the number available is often large and his reasons for choosing one row over another are diverse and not clearly understood. In this paper, we propose that, when constructing time-point rows from an aggregate partition in an all-partition array, Babbitt attempted to minimize both (1) their dissimilarity to the pitch-class rows on the musical surface and (2) the amount of counter-evidence they provide against a preselected beat. We first review two existing measures, Rothgeb's dissimilarity measure using order inversions and Povel and Essens' clock induction model. We then present a way to determine the exact number of possible time-point rows (and pitch-class rows) without repetitions in a particular aggregate partition. Next, we introduce a novel heuristic, based on the aforementioned measures, for predicting from the available time-point rows, those particular rows chosen by Babbitt. We conclude by evaluating how well this heuristic predicts the time-point rows found in two of Babbitt's works, None but the Lonely Flute (1991) and Around the Horn (1993).

1 Introduction

In many of Milton Babbitt's (1916–2011) works using the time-point system and based on the all-partition array, the available pitch-class rows and time-point rows that may appear on the musical surface are the same, as they are both drawn from the same underlying array and its sequence of aggregate partitions.[1] Indeed, Bernstein (2014, p. 6) notes of these works, that "the sketches for numerous later pieces reveal that Babbitt tended to use the same array chart to work out both time-point and pitch-class structure." The number of ways to construct a linear row of elements from an aggregate partition is theoretically infinite, as certain elements may be repeated indefinitely. However, the number of ways to construct such a row without repetitions is finite and computable.[2] Nevertheless, this number is often very large and it is not at all obvious why Babbitt chose the particular rows that he did.

[1]See Bemman and Meredith (2016) and Mead (1994) for detailed discussions on the all-partition array.

[2]As a "row" in music theory necessarily contains 12 distinct elements, theorists use the term "realization" to refer to the linear sequence of elements from an aggregate partition that appear on the musical surface, which may or may not contain repetitions (Leong and McNutt, 2005). In this paper, we consider only sequences containing 12 distinct elements without repetitions and therefore use the term "row".

Babbitt appears to have taken various factors into consideration when choosing time-point rows, including, for example, the extent to which a time-point row implies a particular meter or the relationship that time points may have to the pitch classes in a work. Babbitt himself has suggested that "the metrical signature [is] probably determined by the internal structure of the time-point set" (Babbitt, 1962, p. 63). On the other hand, Mead (1994, pp. 48–49) has noted that "using the same abstract interval pattern for both time point and pitch class rows provides Babbitt with the necessary tools for pursuing his desired compositional ends." It would appear reasonable then, that a similar relationship may hold between the pitch-class and time-point rows that appear on the musical surface.

To this end, we propose that, when constructing the time-point rows from aggregate partitions in an all-partition array, Babbitt attempted to minimize both (1) their dissimilarity to the pitch-class rows that appear on the musical surface and (2) the amount of counter-evidence they provide against a preselected beat. We first review two existing measures that we later use to develop a heuristic for evaluating and selecting time-point rows. These two measures are Rothgeb's (1967) dissimilarity measure using *order inversions* and Povel and Essens' (1985) *clock induction* model. We then present a way to determine the exact number of time-point and pitch-class rows without repetitions in the aggregate partitions of an all-partition array. Next, we introduce a novel heuristic, based on the aforementioned measures, for predicting from the possible time-point rows without repetitions, those chosen by Babbitt. We conclude by evaluating how well this heuristic predicts the time-point rows found in two of Babbitt's works, *None but the Lonely Flute* (1991) and *Around the Horn* (1993).

2 Rothgeb's *Order Inversions* and Povel and Essens' *Clock Induction* model

2.1 Order Inversions

In some of his earliest writings, Babbitt suggested a way of understanding the organization of pitch classes in a 12-tone row as ordered pairs, in what have come to be called *order inversions* (Babbitt, 1960, 1961, 1962). If $\langle p_0, p_1, \ldots, p_k \rangle$ is a sequence of distinct numbers, then $\langle i, j \rangle$ is an *inversion* if and only if $i < j$ and $p_i > p_j$ (Cormen et al., 2009, p. 41). For example, the sequence $\langle 2, 0, 1, 3 \rangle$ has two inversions, $\langle 0, 1 \rangle$ and $\langle 0, 2 \rangle$.[3] In 1967, Rothgeb presented a method for measuring dissimilarity between 12-tone rows based on a generalization of these inversions (Rothgeb, 1967). He notes, however, that "the relevance

[3]Note that Cormen et al. (2009) define an inversion to be a pair of *indices*, not a pair of elements in the sequence.

of similarity relations evaluated on the basis of order is implicit in several writings of Babbitt" (Rothgeb, 1967, p. 183). An ordered pair in a 12-tone row, $S = \langle s_0, s_1, \ldots, s_{11} \rangle$, is a pair of pitch classes, $\langle s_i, s_j \rangle$, such that $i < j$. There are therefore 66 such ordered pairs in a 12-tone row (Rothgeb, 1967, p. 184) and we denote by $OP(S)$ this set of ordered pairs in some 12-tone row, S.[4] Given two 12-tone rows, A and B, the set of all *order inversions*, OI, in B with respect to A can be found by

$$OI = OP(B) \setminus OP(A), \tag{B.1}$$

where $\setminus$ denotes the complement or set difference. In other words, OI is the set of ordered pairs that appear in B but not in A (Rothgeb, 1967, pp. 183–184). We should note that these pairs need not be adjacent in a row, that is, intervening pitch classes are possible.

The maximum number of order inversions in one 12-tone row with respect to another is 66 (in the case that the rows are retrogrades of one another) while the minimum number is 0 (in the case that the rows are identical). Rothgeb's formula measures the dissimilarity between two 12-tone rows by determining the number of ordered pairs *not* shared by them. This number is

$$\begin{aligned} OI_{dis} &= 66 - |OP(A) \cap OP(B)| \\ &= |OI|, \end{aligned} \tag{B.2}$$

where $|.|$ denotes set cardinality and $\cap$ denotes set intersection (in this case, the set of shared ordered pairs) (Rothgeb, 1967, p. 184).[5] For example, given a 12-tone row, B, that shares 55 ordered pairs with a 12-tone row, A, the measure of their dissimilarity, OI_{dis}, is $66 - 55 = 11$.

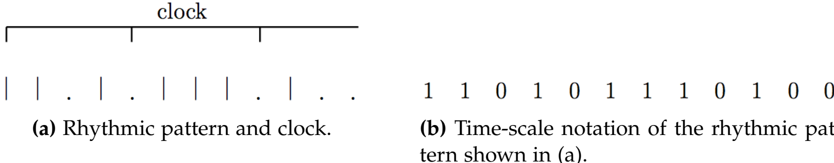
2.2 Clock Induction model

Povel and Essens (1985) introduced a model of beat induction in rhythmic patterns based on what they called *clocks*. A clock consists of equally spaced "ticks" set against a given rhythmic pattern. The time interval between consecutive ticks is called a *unit* and the duration of a unit can be no more than half the duration of the rhythmic pattern.

A unit represents an arbitrary but fixed duration of time most closely corresponding to the time interval (i.e., duration) between two consecutive beats. A unit's length is expressed as an integer multiple of inter-tick subdivisions, meaning that the time interval between two consecutive clock ticks may be subdivided into a whole number of intervals of equal duration, called *unit*

[4] The set of ordered pairs in a 12-tone row have been called *protocol pairs* (Lewin, 1976, p. 252).

[5] Notation for Equations B.1 and B.2 adapted from both Rothgeb (1967, p. 183) and Ilomäki (2008, p. 129).



(a) Rhythmic pattern and clock.

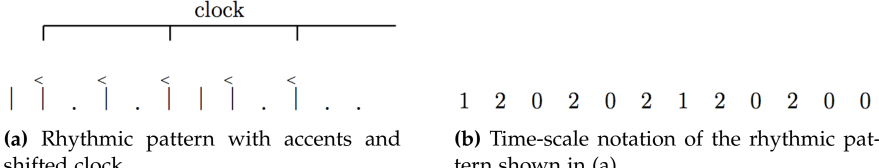
(b) Time-scale notation of the rhythmic pattern shown in (a).

Fig. B.1: Rhythmic pattern in (a) and its time-scale notation in (b). Note that the location of the clock is 1 and its unit length is four (see Povel and Essens, 1985, 415).

subdivisions (or just "subdivisions") (Povel and Essens, 1985, p. 414). For example, a unit of length 4 has four subdivisions, each with duration equal to 1/4 of the unit, in the same way as a quarter-note beat may be subdivided into four sixteenth notes. A clock may be shifted in time by any number of its subdivisions, however, a shift by a number of subdivisions equal to an integer multiple of its unit length, maps a clock onto itself. It is therefore only necessary to consider shifts by a number of subdivisions less than the unit length, since these shifts result in distinct clocks with ticks in different positions from the original, unshifted clock. The shifted or unshifted position of a clock is called its *location*. Any rhythmic pattern, set against a grid of a clock's ticks and its unit subdivisions, can be represented as a sequence of zeros and ones in *time-scale notation*, where a one indicates an event onset in the rhythmic pattern and a zero indicates that there is no event onset at that location. Figure B.1 shows an example of a rhythmic pattern with an example clock in (a) and its time-scale notation in (b). In Figure B.1(a), events in the rhythmic pattern have been indicated by vertical lines while locations that do not correspond to events have been indicated with dots. Note, in particular, that the three ticks of the clock shown above this pattern align with the first rhythmic event and two locations where there are no events.

Povel and Essens (1985, 415) note that, in earlier work, they had found that certain temporal events in a rhythmic pattern are perceived to be accented by listeners. These events are as follows: (1) relatively isolated events, (2) the second event of two adjacent events, and (3) the first and last events of a cluster of three or more events. Figure B.2(a) shows the rhythmic pattern from Figure B.1 with its clock's location shifted by one. Figure B.2(b) shows the same pattern using time-scale notation, where events predicted by these three rules to be accented are indicated by the value 2. Povel and Essens predict that the clock in Figure B.2(a) (with a unit equal to four and location 2) is more strongly induced by the given rhythm than the clock in Figure B.1(a) (with the same unit but location 1).

Povel and Essens argue that the strength with which a rhythmic pattern induces a particular clock is inversely related to the amount of *counter-evidence* the pattern provides against that clock. They further propose that this amount of counter-evidence, C, can be calculated using the following



(a) Rhythmic pattern with accents and shifted clock.

(b) Time-scale notation of the rhythmic pattern shown in (a).

Fig. B.2: Rhythmic pattern with accents in (a) and its time-scale notation in (b). Note that the location of the clock is 2, as its first tick is shifted one unit subdivision to the right (see Povel and Essens, 1985, p. 415).

formula:

$$C = Ws + u, \tag{B.3}$$

where W is a constant weight parameter (which they set to 4), s is the number of clock ticks that do *not* coincide with events in the rhythmic pattern (i.e., align with zeros in the time-scale notation), and u is the number of clock ticks that align with unaccented events (i.e., ones in the time-scale notation) (Povel and Essens, 1985, p. 417). For example, the amount of counter-evidence the clock shown in Figure B.2(a) encounters in its corresponding rhythmic pattern is $C = 0$, since, in this case, all the clock ticks align with accented events (i.e., twos in the time-scale notation). On the other hand, the amount of counter-evidence that the same rhythmic pattern provides against the clock in Figure B.1(a) is $4 \times 2 + 1 = 9$, since, over the whole rhythmic pattern, there are 2 clock ticks aligned with locations at which no events occur ($s = 2$) and one clock tick, the first, aligned with an unaccented event ($u = 1$).

The "best" clock for a given rhythmic pattern (i.e., the one that is most strongly induced by the pattern) is then, according to Povel and Essens' model, the one with the lowest value of C for that pattern. If two clocks result in the same amount of counter-evidence, then the better of the two is predicted to be the one whose unit evenly divides the total duration of the pattern (ensuring that the remaining clock ticks align only with accented events) (Povel and Essens, 1985, p. 419).

3 The number of possible rows without repetitions for an aggregate partition

Since Babbitt first laid the foundations for his time-point system (Babbitt, 1962), much work has been done on discovering the time points underlying many of his works (e.g., Bernstein, 2014; Leong and McNutt, 2005; Mead, 1987, 1994). However, less has been written about *how* Babbitt constructed his time-point rows and *why* he chose one and not another (Johnson, 1984; Mead, 1984). An understanding of the exact number of possible ways Babbitt could

construct time-point rows without repetitions from an aggregate partition will be informative in answering these questions.

3.1 Constructing rows

A partition of the aggregate or *mosaic*, in the sense in which it is used by Morris (2003, p. 103), is an unordered set of unordered pitch class (or time point) sets that collectively, exhaustively and exclusively partition the aggregate, e.g., $\{\{7,2,6,11,8\},\{3,4,5,10\},\{9,0,1\}\}$.[6] However, Babbitt's desire to maintain the order of elements in his rows when constructing time points and pitch classes in an array means that the unordered sets in these partitions become *ordered*. For this reason, an aggregate partition as it appears in an all-partition array is traditionally viewed to be an unordered set of *ordered* sets that we call *segments*, e.g., $\{\langle 7,2,6,11,8\rangle,\langle 3,4,5,10\rangle,\langle 9,0,1\rangle\}$, where each segment consists of consecutive elements in a single lyne in the array (Mead, 1994). Pitch classes or time points belonging to such a segment, however, when realized on the musical surface, may have intervening elements from another segment. Figure B.3 shows the opening time-point aggregate partition and time-point row from Babbitt's *None but the Lonely Flute* with one possible rhythmic interpretation. As indicated in Figure B.3(a), each lyne in this piece is associated with a distinct dynamic level. Figure B.3(a) shows the segment of this opening partition associated with each lyne. In this case, the smallest possible interval between adjacent time points (i.e., one) corresponds to a sixteenth note, as can be seen, for example, between time points 6 and 7 in Figure B.3(b).

This basic interval of time is called a *unit* (Mead, 1987, p. 183)—note that, perhaps confusingly, a "unit" between time points therefore corresponds more closely to what Povel and Essens call a "subdivision" than it does to what they call a "unit" in a clock.

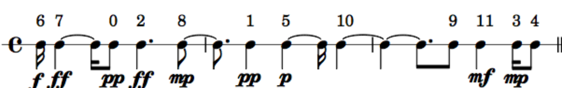
We know from the time-point aggregate partition shown in Figure B.3(a), that the left-to-right order of each segment imposes a partial order on any corresponding row of its elements (Lewin, 1976; Starr, 1978, 1984). Such a row is allowed to contain, for example, $\langle 7,2\rangle$, but not $\langle 2,7\rangle$. Furthermore, $\langle 7,8,2\rangle$ is an acceptable beginning to a row, but $\langle 7,3,2\rangle$ is not, because 3 must occur later than 8 in the row as it follows 8 in the segment for dynamic level ***mp*** in the aggregate partition in Figure B.3(a). On the other hand, in $\langle 7,8,2\rangle$, 8 is simply an intervening element, as described above. In other words, the general process of constructing a time-point row without repetitions from an aggregate partition amounts to choosing and then removing the left-most element of any one segment until all segments are empty.

[6] Our use of the term "mosaic" corresponds to that given by Morris (2003, p. 103), which differs from the way this term is used in his earlier writings (Morris and Alegant, 1988, p. 76).

3. The number of possible rows without repetitions for an aggregate partition

ff	7 2
f	6
mf	11
mp	8 3 4
p	5 10 9
pp	0 1

(a) Time-point aggregate partition containing segments of time points distinguished by dynamic level.



(b) One possible time-point row and corresponding dynamic markings with one possible rhythmic interpretation from the time-point aggregate partition in (a).

Fig. B.3: Opening time-point aggregate partition (represented by $3^2 2^2 1^2$) from Babbitt's *None but the Lonely Flute* in (a) and corresponding time-point row in (b) using a unit equal to a sixteenth note.

3.2 Computing rows

As noted by Alegant (1993, p. 61), there is a simple formula for calculating the total number of distinct ways to partition the aggregate by a single partition, p, of *unordered* segments.[7] As it turns out, this number is equal to the total number of distinct rows without repetitions from *one* p-partitioning of the aggregate containing *ordered* segments (as described above), and so the same formula can be used.

Let us suppose, then, that we have a p-partitioning of the aggregate, m, with k *ordered* segments, $s_1, s_2, \ldots, s_k$ (e.g., Figure B.3(a)). The total number of distinct time-point rows without repetitions, n, for this aggregate partition m can be found using the following formula for the *multinomial coefficient*:

$$n = \frac{(|s_1| + |s_2| + \ldots + |s_k|)!}{|s_1|!|s_2|! \ldots |s_k|!}, \tag{B.4}$$

where $|.|$ denotes segment length. From Equation B.4, we find that our time-point aggregate partition from Figure B.3(a) has $12!/3!3!2!2!1!1! = 3,324,000$ possible time-point rows without repetitions. By extension, this is the same number of possible pitch-class rows, if this were a pitch-class aggregate partition.

The multinomial coefficient is the number of distinct permutations of a *multiset*. This means we can uniquely express any linear row of elements from an aggregate partition as an ordered sequence of integers, which we call *segment numbers*, each indicating the segment to which a particular element belongs. As there are never more segments in an aggregate partition than

[7] It is possible that Sward (1981, p. 141) may have been the first to make this observation of partitions with respect to music.

there are elements in an aggregate, the resulting sequence of segment numbers is, in general, a multiset, in which a given segment number may occur more than once.[8] Thus, by assigning a number to each element in an aggregate partition corresponding to the lyne containing the segment to which it belongs (with lynes numbered from top to bottom), any row of elements constructed from such an aggregate partition has a corresponding sequence of lyne numbers. For example, the time-point row $\langle 6, 7, 0, 2, 8, 1, 5, 10, 9, 11, 3, 4 \rangle$ constructed from the time-point aggregate partition in Figure B.3(a), has the corresponding lyne number sequence $\langle 2, 1, 6, 1, 4, 6, 5, 5, 5, 3, 4, 4 \rangle$. This lyne number sequence is a permutation of the elements in the multiset $\{1, 1, 2, 3, 4, 4, 4, 5, 5, 5, 6, 6\}$.

At this point, it is important to note that an aggregate partition under this view must now be an *ordered* set of *ordered* segments so that the ordinal position of each segment in the set remains fixed to its lyne number and thus dynamic level (in the case of a time-point aggregate) or register (in the case of a pitch-class aggregate).[9] For example, it is clear that $\langle \langle 7, 2, 6, 11, 8 \rangle, \langle 3, 4, 5, 10 \rangle, \langle 9, 0, 1 \rangle, \langle \rangle, \langle \rangle, \langle \rangle \rangle$ contains three non-empty segments (from left to right) belonging to lynes 1, 2 and 3 (from top to bottom) of a six-part all-partition array. Such a set, corresponding to a bounded *weak composition* of the aggregate,[10] has been called an *ordered mosaic* (Bemman and Meredith, 2016, p. 3).[11] For the remainder of this paper, we refer to aggregate partitions in an all-partition array as ordered mosaics, as this representation retains the dynamic level and segment/lyne correspondence necessary for constructing the time-point rows used by our proposed heuristic.

Table B.1 shows the number of rows of elements without repetitions that can be constructed from a single ordered mosaic corresponding to each of the 77 partitions of 12 into 12 or fewer parts. This number varies considerably across the range of ordered mosaics, from 1 for partition 12^1 to 479,001,600 for partition 1^{12}. Incidentally, the maximum number of rows for any one ordered mosaic in an all-partition array corresponds to the so-called *even partitions*, which have been of interest to music theorists (Morris and Alegant, 1988). For

[8] The only exception is in an aggregate partition represented by 1^{12}, in which each distinct segment number occurs exactly once.

[9] In some of Babbitt's all-partition arrays, lynes formed by the segments in its pitch-class aggregates are distinguished by instrument and not register (Mead, 1994).

[10] A *weak composition*, which Bemman and Meredith (2014) denote by $\mathrm{WIntComp}(s_1, s_2, ..., s_k)$, is a representation of an integer $n = \sum_{i=1}^{k} s_i$, as an *ordered* sum of k *non-negative* integers, (i.e., including zero). For example, if $n = 12$ and $k = 6$ (i.e., the number of segments), then $\mathrm{WIntComp}(0, 6, 0, 6, 0, 0)$ is one possible weak integer composition.

[11] The term "composition" has been used previously by both Alegant (1993, p. 58) and Babbitt (1961, p. 83). Babbitt writes that "...the 2^{11} compositions of the number 12 all must be examined when the order of parts of a partition...is significant, as it is when the parts are – say – assigned linearly to classes of timbres, registers, dynamics, etc." and it is in this context that the terminology of "ordered mosaic" is used.

	Part.	Num.		Part.	Num.		Part.	Num.		Part.	Num.
1.	12	1	21.	6 3^2	18480	41.	6 2 1^4	332640	61.	3 2^4 1	4989600
2.	11 1	12	22.	7 2^2 1	23760	42.	3^4	369600	62.	4 2^2 1^4	4989600
3.	10 2	66	23.	5 4 3	27720	43.	4^2 2 1^2	415800	63.	3^2 2 1^4	6652800
4.	10 1^2	132	24.	6 4 1^2	27720	44.	5 2^3 1	498960	64.	2^6	7484400
5.	9 3	220	25.	5^2 1^2	33264	45.	4 3^2 1^2	554400	65.	3 2^3 1^3	9979200
6.	8 4	495	26.	4^3	34650	46.	5 3 1^4	665280	66.	4 2 1^6	9979200
7.	9 2 1	660	27.	7 2 1^3	47520	47.	6 1^6	665280	67.	3^2 1^6	13305600
8.	7 5	792	28.	6 3 2 1	55440	48.	4 3 2^2 1	831600	68.	2^5 1^2	14968800
9.	6 6	924	29.	5 4 2 1	83160	49.	4^2 1^4	831600	69.	3 2^2 1^5	19958400
10.	9 1^3	1320	30.	6 2^3	83160	50.	5 2^2 1^3	997920	70.	4 1^8	19958400
11.	8 3 1	1980	31.	7 1^5	95040	51.	3^3 2 1	1108800	71.	2^4 1^4	29937600
12.	8 2^2	2970	32.	5 3^2 1	110880	52.	4 2^4	1247400	72.	3 2 1^7	39916800
13.	7 4 1	3960	33.	6 3 1^3	110880	53.	3^2 2^3	1663200	73.	2^3 1^6	59875200
14.	6 5 1	5544	34.	4^2 3 1	138600	54.	4 3 2 1^3	1663200	74.	3 1^9	79833600
15.	8 2 1^1	5940	35.	5 3 2^2	166320	55.	5 2 1^5	1995840	75.	2^2 1^8	119750400
16.	7 3 2	7920	36.	5 4 1^3	166320	56.	3^3 1^3	2217600	76.	2 1^{10}	239500800
17.	8 1^4	11880	37.	6 2^2 1^1	166320	57.	4 2^3 1^2	2494800	77.	1^{12}	479001600
18.	6 4 2	13860	38.	4^2 2^2	207900	58.	3^2 2^2 1^2	3326400			
19.	7 3 1^2	15840	39.	4 3^2 2	277200	59.	4 3 1^5	3326400			
20.	5^2 2	16632	40.	5 3 2 1^2	332640	60.	5 1^7	3991680			

Table B.1: The number of distinct linear rows of elements without repetitions that can be constructed from a single ordered mosaic corresponding to each of the 77 partitions of 12 into 12 or fewer parts.

example, in a six-part all-partition array, the maximum number is 7,484,400 rows corresponding to partition 2^6 and the maximum in a four-part array is 369,600 rows corresponding to partition 3^4. For any one time-point ordered mosaic (hereafter abbreviated "TpOM") in an array, predicting which time-point row without repetitions Babbitt chose out of the sometimes millions possible, would appear to be a difficult task. In the following sections, we will introduce a heuristic based on order inversions and beat induction using clocks that goes some way towards predicting Babbitt's choice of time-point rows.

4 A heuristic based on the combined measures of order inversions and clock induction

As mentioned above, we propose that, when constructing time-point rows, Babbitt attempted to minimize both (1) their dissimilarity to the pitch-class rows in a piece and (2) the amount of counter-evidence they provide against a preselected beat—that is, a particular clock. In this section, we motivate this view and describe how, in particular, we have used order inversions as a measure of (1) and clock induction as a basis for measuring (2).

4.1 Motivation

It may, perhaps, be necessary to emphasize that, by using Povel and Essens' clock-induction model as a basis for our proposed heuristic, we are *not* intending to suggest that Babbitt explicitly used this particular computational model of beat perception as a basis for predicting the extent to which any given time-point row would induce any particular beat in the mind of a listener. What we are hypothesizing is that one of the factors that determined Babbitt's choice of time-point rows was the extent to which a row induced a desired beat and we have chosen to use Povel and Essens' model in our heuristic to predict this. We could have used some other computational model of beat induction—for example, that of Lee (1991)—but Povel and Essens' model is simple to implement, already provides a way of quantifying the compatibility between a rhythm and a given beat, and is supported by empirical evidence.

In implementing our heuristic, we have tried to make no analytical assumptions about the pieces we will look at, other than what can be gleaned from the score or what has been written about Babbitt's late practice. However, we make several concessions regarding the implementation of our heuristic for use with time points. For example, we have removed from all rows any repetitions of time points, retaining only the first occurrence of each of the 12 elements in each ordered mosaic. In practice, Babbitt often repeated certain time points. Assuming otherwise, however, is necessary in order to have both fixed clock and row lengths—i.e., *minimum total temporal durations* (Babbitt, 1962, p. 64). It is also possible that Babbitt first constructed his time-point rows without repetitions and then added repetitions using criteria along the lines of those he used to add repetitions in realizations of pitch class rows.[12]

By using clocks we treat a time-point row as a "rhythmic pattern," where time points become the "events" of this pattern, subject to accenting or not, based upon Povel and Essens' rules (described in Section 2.2). The assumption here is that, if Babbitt intended some meter to be evident from the organization of time points, then we would expect to find that time points (i.e., events) in a row would induce some beat. The use of clocks is well suited for measuring beat induction in a time-point row, as unit subdivisions in a clock, like time-point units, can represent any arbitrary (short) duration (e.g., a sixteenth note, 32nd note, etc.). As many of Babbitt's later works based on the time-point system have a unit equal to a sixteenth note (Bernstein, 2014, p. 7), so too would the unit subdivisions of their corresponding clocks. This observation motivates the particular way in which we use clocks, as the pieces we will look at are representative of Babbitt's later practice.

Povel and Essens originally developed their clock-induction model to *pre-*

[12]See Bemman and Meredith (2016) for a detailed discussion of this process.

dict which of a universe of possible beats (clocks) a particular rhythm would be most likely to induce in the mind of a listener. In our usage of their model, we consider only those clocks with a unit equal to four, with each tick corresponding to the beat in the meter found predominantly throughout a piece (i.e., what we have proposed to be Babbitt's preselected beat). This means that, for any given time-point row, the possible clocks are those with the same unit, differing only from each other in their location (i.e., their shifted position by unit subdivision). For example, the presence of a predominantly simple meter would mean we would consider only a single type of clock with a unit equal to four and its four locations, giving a total of four clocks. Of these, we choose the clock with the best location. While clocks are general enough to accommodate clock units and their subdivisions equal to any note duration, we assume that both the time-point unit and duration between consecutive beats is unchanging in a piece, which may or may not be true. Under these assumptions, however, a repeated time point at the level of the beat (not meter) contributes no additional information beyond its first occurrence, as repetitions occur at integer multiples of 12—that is, a time point which falls on a beat can only ever fall on subsequent beats when repeated (and vice-versa). Naturally, this is true only in cases where the number of time-point units to a beat evenly divides 12, but under our assumptions of a piece, this remains true. We should note, also, that in general the performance of a clock decreases as the total temporal duration of the time-point row increases. As repetitions generally increase the total temporal duration of a row and as we do not currently have any means for fairly comparing rows which differ in their number of time points, it would prove difficult to evaluate such rows using clocks.

In general, we have found that, from one pitch-class ordered mosaic (hereafter abbreviated "PcOM") to another in an all-partition array, it is virtually impossible to ensure that pitch-class rows (without repetitions) constructed from each, according to the process described in Section 3, will belong to the same row class. Even within one PcOM, it is difficult to construct two such distinct rows that belong to the same row class. Confronted with this difficulty, it would seem reasonable to assume that Babbitt sought additional ways to draw meaningful connections between the pitch-class and time-point rows he used, beyond the mere fact that they are drawn from the same ordered mosaics.[13] Nevertheless, we suspect that the way in which he might have drawn these connections in many of his works is by minimizing the differences these time-point rows on the musical surface have with the pitch-class rows from the *same* ordered mosaic. We are motivated here to use order inversions and not intervallic content (as suggested by Mead (1994, pp. 48–

[13]With respect to the pitch-class rows taken from different PcOMs, Babbitt has used equal-note-value-strings and array references to draw these connections (Bernstein, 2014, p. 11).

49)) to measure the dissimilarity between the pitch-class and time-point rows taken from the same ordered mosaic in a work, primarily because Babbitt's own writings reveal that he conceived of the organization of rows in dyads (Babbitt, 1960, 1961, 1962). While a comparison with other (dis)similarity measures would be interesting, we have not done this here. Likewise, there have been numerous more recent models of meter (Hasty, 1999; London, 2004), beat induction and onset detection (Eck, 2001) that have been developed, however, we do not consider these.

4.2 The heuristic

Recall that, for both OI_{dis} from Rothgeb's order inversions measure (shown in Equation B.2) and C from Povel and Essens' clock induction model (shown in Equation B.6), a lower value is better. This is therefore true of their summed value as well. However, the possible values taken by these two measures range over different intervals. As mentioned in Section 2.1, the values of OI_{dis} lie withing the range $[0, 66]$. The values of C, on the other hand, lie within the range $[10, 121]$ when using a clock unit equal to four and Povel and Essens' weighting parameter, $W = 1$. The lower and upper bounds of this range for values of C correspond to the rows $\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \rangle$ and $\langle 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 \rangle$, respectively. Thus, in order to combine OI_{dis} and C, we must first scale their respective values. There are a number of methods to scale data, normalization and standardization being most common, and we have chosen to normalize each of the values for OI_{dis} and C, which we denote by $\widehat{\cdot}$.[14] Our heuristic therefore consists of minimizing the sum of these normalized values:

$$\widehat{OI_{\text{dis}}} + \widehat{C}. \tag{B.5}$$

We conclude this section by describing what steps we took at each point in implementing our heuristic and Figure B.4 shows this procedure. For a given piece, we first compute all distinct permutations of a multiset of segment/lyne numbers (as described in Section 3) for a given mosaic in its underlying array. We then do steps 1–7 in Figure B.4 for each of the permutations of this multiset and for all ordered mosaics. In step 1, we generate the time-point row from the current ordered mosaic using the current permutation of segment numbers. This row may or may not be the time-point row Babbitt chose. Next, we convert this time-point row to Povel and Essens' time-scale notation (step 2). As this sequence of zeros and ones can be quite long, we have shown in Figure B.4 only the first three and last three time points of this row. In step 3, we add accents to this time-scale notation, according to the rules described in Section 2.2. In the next step, we compute

[14] In this paper, we use feature scaling to normalize, given by the formula: $\frac{x_i - \min(x)}{\max(x) - \min(x)}$.

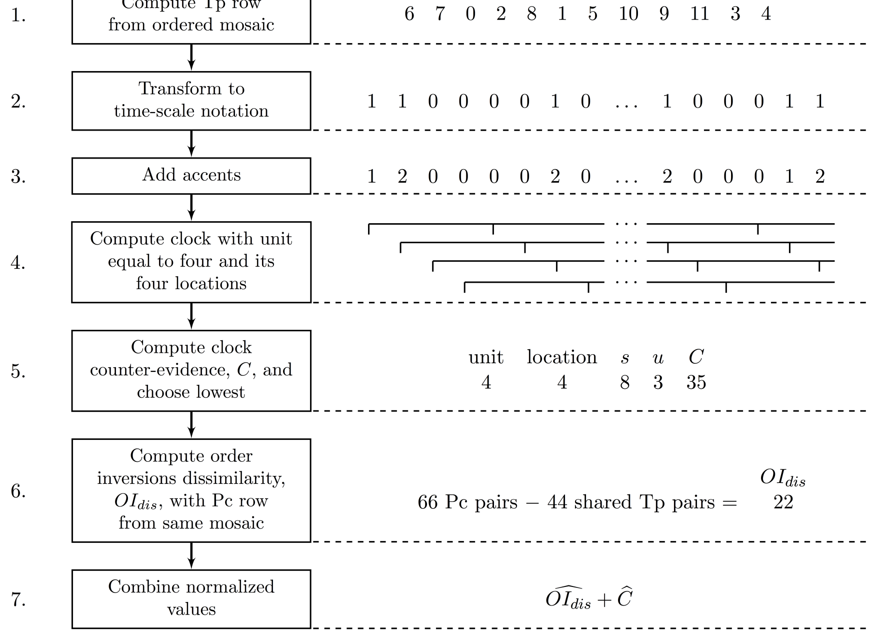


Fig. B.4: Proposed procedure for predicting the time-point rows without repetitions in Babbitt's later works based on the all-partition array. We follow this procedure for every possible time-point row in each ordered mosaic of an all-partition array.

a clock with a unit equal to four and all its locations (step 4). In step 5, we compute the counter-evidence encountered by each of these four clocks (using Povel and Essens' weighting parameter, $W = 1$, defined in Equation B.6) in the time-scale notation (with accents) of the current time-point row. We then select the best of these clocks—that is, the one having the least amount of counter-evidence, as measured by the value C. We then use the pitch-class row (on the musical surface) taken from the current ordered mosaic and its corresponding current time-point row and compute their number of shared ordered pairs, along with the resulting order inversions dissimilarity measure, OI_{dis} (step 6). Finally, we combine the normalized values of both measures, $\widehat{OI_{\text{dis}}}$ and $\widehat{C}$ (step 7).

5 Predicting the time-point rows without repetitions in *None but the Lonely Flute* and *Around the Horn*

In this section, we evaluate how well our heuristic, as measured by the combined values of $\widehat{OI_{\text{dis}}}$ and $\widehat{C}$, predicts the time-point rows in Babbitt's *None but the Lonely Flute* and Part I of *Around the Horn*.[15] We have chosen these pieces as they are relatively short and their use of the time-point system is relatively straightforward, allowing for a simple implementation and test of our heuristic. Both pieces assign a consistent set of dynamic levels to each part (lyne) of their respective arrays with *None but the Lonely Flute*, a six-part array, ranging from ***ff*** to ***pp*** (with one dynamic level to a part) and *Around the Horn*, a four-part array, ranging from ***fff*** to ***ppp*** (with two possible dynamic levels to a part).

5.1 Relative impact of Order Inversions and Clock Induction

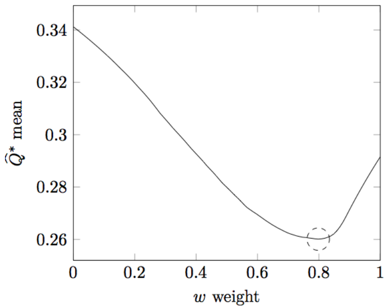
In each piece, the relative impact of the two measures we have combined in our proposed heuristic differs both from ordered mosaic to ordered mosaic and over the whole of the work. The relative impact of the two measures can be expressed by weighting one or the other by a constant parameter (much like W in Equation B.6). This weighting parameter we will call w and it takes a value between 0 and 1. We have chosen here to use a value for w which yields the best results across the whole of each piece, as opposed to "fine tuning" this value for each ordered mosaic. In order to incorporate this weighting parameter, we have used a weighted mean where, given a set of values, $x_1, x_2 \ldots, x_n$, one attempts to minimize a quantity, $w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$ over the sum $w_1 + w_2, \ldots, w_n$. From this, we can modify our heuristic to be weighted in the following way:

$$Q = w\widehat{OI_{\text{dis}}} + (1 - w)\widehat{C}, \tag{B.6}$$

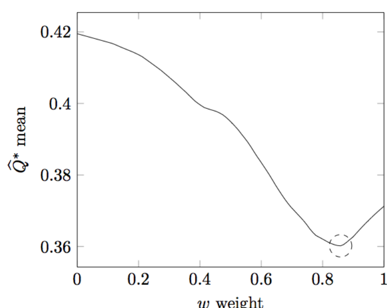
where our task is to discover the value of w that minimizes the sum, Q, for all time-point rows chosen by Babbitt in a given piece. Values of w greater than 0.5 will assign more weight to $\widehat{OI_{\text{dis}}}$ than $\widehat{C}$; while values of w less than 0.5 mean that $\widehat{C}$ contributes more to predicting the time point rows than $\widehat{OI_{\text{dis}}}$.

In order to discover the "best" weighting for each piece, we first selected from a range of 0 to 1, 100 equally-spaced values for w. Next, for each time-point row chosen by Babbitt, we obtained its value of Q, which we denote by Q^*, and then normalized it over all the rows for its mosaic, producing a

[15] Bernstein (2014) and Leong and McNutt (2005) have analyzed *None but the Lonely Flute* while Mead (1997) and Dubiel (1997) has analyzed *Around the Horn*.



(a) The lowest mean, 0.26, corresponds to $w = 0.798$ in *None but the Lonely Flute.*



(b) The lowest mean, 0.36, corresponds to $w = 0.859$ in *Around the Horn*.

Fig. B.5: Mean of values for $\widehat{Q}^*$ for *None but the Lonely Flute* in (a) and *Around the Horn* in (b) after applying 100 different, equally-spaced values between 0 and 1 for the weighting parameter, w, to $w\widehat{OI_{\text{dis}}} + (1-w)\widehat{C}$. Note that the dashed circles indicate the lowest mean.

value that we denote by $\widehat{Q}^*$. Normalizing over the rows within each mosaic ensures that we are able to fairly compare the predictive performance of our heuristic over different mosaics, as the number of possible time-point rows varies widely from one mosaic to another. Finally, we chose the value of w that minimized the arithmetic mean of the values of $\widehat{Q}^*$ over all mosaics in a piece. In other words, we chose the value of w that worked best on average over all mosaics. Figure B.5 shows the mean values of $\widehat{Q}^*$ for *None but the Lonely Flute* in (a) and *Around the Horn* in (b) after applying 100 different, equally-spaced weights between 0 and 1 to $w\widehat{OI_{\text{dis}}} + (1-w)\widehat{C}$.

The results shown in the plots in Figure B.5 indicate that the lowest mean for values of $\widehat{Q}^*$ in *None but the Lonely Flute*, 0.26, is considerably lower than the lowest mean for values of $\widehat{Q}^*$ in *Around the Horn*, 0.36. Interestingly, the lowest mean in each piece aligns with approximately the same weights, $w = 0.798$ in *None but the Lonely Flute* and $w = 0.859$ in *Around the Horn*. Moreover, both curves are fairly smooth, with a single, well-defined minimum. This suggests that the clocks measure is substantially less important than the order inversions measure, but its relative contribution remains roughly the same in both pieces, despite their differing means. It is clear, however, from the upwards curves on either side of the lowest mean, that using both measures permits more accurate prediction of the time point rows chosen by Babbitt than is achievable by using either order inversions alone (i.e., where $w = 1$) or clocks alone (i.e., where $w = 0$).

5.2 Results and Evaluation

In the following discussion we will use $\widehat{Q}^*$ values as a primary means for evaluating the performance of our heuristic. However, while the value of $\widehat{Q}^*$ indicates how well Babbitt's chosen row for a mosaic satisfies our heuristic, relative to the row that satisfies our heuristic *best* for that mosaic, it does not tell us anything about the *proportion* of possible time-point rows for a given mosaic that satisfy our heuristic at least as well as the one that Babbitt chose. We can get a better indication of the latter by *ranking* each of the possible time-point rows for a mosaic according to its value of Q, and then dividing this rank, r, by the total number of distinct ranks, N. We choose to use *dense ranking*, where equal values of Q receive the same rank and where the next time-point row which differs in Q is assigned the immediately following rank (e.g., if the sequence of values of Q is $\langle 0.01, 0.02, 0.02, 0.03 \rangle$, then the corresponding sequence of ranks would be $\langle 1, 2, 2, 3 \rangle$).

Using the best weights found for each piece in Figure B.5, we now look to how our heuristic performs in each ordered mosaic using $\widehat{Q}^*$ and r/N. Tables B.2 and B.3 show the results of applying our heuristic based on $w\widehat{OI_{\text{dis}}} + (1 - w)\widehat{C}$ to predict the time-point rows found in Babbitt's *None but the Lonely Flute* and *Around the Horn*, respectively.[16] In Tables B.2 and B.3, $\widehat{OI_{\text{dis}}}$, $\widehat{C}$, $\widehat{Q}^*$, r and r/N are as defined above. We evaluate the performance of our heuristic on each piece by giving the mean values of $\widehat{Q}^*$ and r/N over all mosaics for that piece.

The fact that $\widehat{Q}^*$ and r/N give different information for a given mosaic is evident from, for example, the results for the first and twelfth mosaics in Table B.2. Here, the value of r/N for the first mosaic, 0.022, is *lower*—i.e., *better*—than r/N for the twelfth mosaic (0.03), indicating that there are proportionally fewer possible rows in the first mosaic that satisfy our heuristic at least as well as Babbitt's chosen row.[17] Conversely, the value of $\widehat{Q}^*$ for the first mosaic (0.063) is *higher*—i.e., *worse*—than that for the twelfth mosaic (0.026), indicating that the row chosen by Babbitt in the twelfth mosaic satisfies our heuristic better (relative to the best row for the twelfth mosaic), than the row he chose for the first mosaic (relative to the best row for that mosaic).

The mean value of r/N shown in Table B.2 indicates that, on average, the time-point rows chosen by Babbitt in *None but the Lonely Flute* have a rank, when evaluated using our heuristic, that lies within the top 23% of possible ranks.

[16] As Babbitt has done in his sketches of *Around the Horn*, we use "sounding pitch" and not "notated pitch" to denote the pitch classes and time points of this piece.

[17] Actually, because r is *dense* rank, r/N is the proportion of distinct *ranks* that are at least as good as that of Babbitt's chosen row. To find the proportion of *rows* for a mosaic that are at least as good as Babbitt's row, one should use standard competition ranking ("1224" ranking), and then look at the rank of the next best row after the one chosen by Babbitt.

	None but the Lonely Flute				Clocks	Measures		Evaluation		
		Babbitt's rows								
	Tp/PcOMs	Num. possible	Tp	Pc	Location	$\widehat{OI}_{\text{dis}}$	$\widehat{C}$	$\widehat{Q}^*$	r	r/N
1.	$3^2 2^2 1^1$	3324000	< 6, 7, 0, 2, 8, 1, 5, 10, 9, 11, 3, 4 >	< 0, 7, 8, 6, 2, 1, 5, 10, 9, 11, 3, 4 >	4	0.106	0.292	0.063	20	0.022
2.	3^4	369600	< 11, 0, 5, 7, 1, 6, 2, 10, 3, 4, 8, 9 >	< 11, 0, 5, 7, 1, 6, 3, 2, 10, 8, 9, 4 >	4	0.095	0.261	0.05	12	0.018
3.	642	13860	< 6, 2, 4, 7, 11, 3, 10, 1, 9, 0, 8, 5 >	< 6, 4, 11, 2, 3, 7, 1, 9, 8, 10, 0, 5 >	1	0.276	0.294	0.274	66	0.192
4.	$62^2 1^2$	166320	< 8, 9, 1, 5, 0, 2, 6, 7, 10, 4, 3, 11 >	< 8, 1, 0, 2, 6, 7, 9, 4, 5, 3, 11, 10 >	1	0.333	0.167	0.293	140	0.235
5.	5321^2	332640	< 0, 4, 11, 3, 5, 7, 2, 8, 6, 10, 1, 9 >	< 7, 2, 8, 6, 1, 0, 4, 3, 11, 9, 10, 5 >	3	0.625	0.35	0.58	469	0.612
6.	741	3960	< 4, 9, 3, 6, 2, 10, 8, 11, 5, 7, 0, 1 >	< 3, 4, 2, 10, 9, 6, 11, 5, 7, 0, 1, 8 >	3	0.355	0.429	0.328	93	0.304
7.	62^3	83160	< 4, 8, 0, 6, 3, 9, 7, 2, 10, 1, 5, 11 >	< 0, 4, 10, 5, 6, 8, 3, 9, 7, 2, 11, 1 >	1	0.459	0.389	0.438	222	0.422
8.	$4^2 2^2$	207900	< 1, 5, 11, 0, 8, 9, 4, 6, 7, 10, 3, 2 >	< 1, 11, 5, 0, 6, 8, 9, 4, 10, 7, 3, 2 >	3	0.119	0.286	0.071	24	0.036
9.	6321	55440	< 9, 1, 10, 2, 7, 11, 4, 6, 0, 5, 8, 3 >	< 1, 2, 7, 9, 10, 8, 11, 4, 6, 3, 0, 5 >	4	0.286	0.444	0.293	125	0.236
10.	5421	83160	< 6, 3, 11, 1, 4, 7, 2, 8, 5, 9, 0, 10 >	< 11, 6, 4, 10, 8, 3, 1, 7, 2, 5, 9, 0 >	2	0.432	0.5	0.415	232	0.397
11.	732	7920	< 2, 6, 10, 1, 0, 8, 7, 11, 5, 4, 9, 3 >	< 1, 6, 10, 11, 2, 0, 8, 5, 7, 4, 9, 3 >	3	0.278	0.4	0.297	92	0.227
12.	75	792	< 7, 0, 1, 11, 3, 4, 6, 8, 10, 9, 2, 5 >	< 0, 7, 1, 11, 3, 4, 8, 6, 10, 9, 5, 2 >	4	0.136	0.231	0.026	5	0.03
13.	82^2	2970	< 11, 2, 6, 10, 5, 3, 9, 1, 4, 7, 8, 0 >	< 10, 2, 11, 6, 5, 3, 1, 9, 4, 7, 8, 0 >	3	0.179	0.25	0.122	18	0.071
14.	63^2	18480	< 2, 9, 0, 1, 7, 11, 8, 3, 6, 10, 4, 5 >	< 2, 9, 0, 1, 6, 7, 11, 8, 4, 3, 10, 5 >	1	0.171	0.278	0.144	38	0.091
15.	93	220	< 5, 3, 10, 4, 11, 2, 7, 9, 6, 8, 0, 1 >	< 5, 3, 4, 10, 9, 11, 2, 7, 6, 8, 0, 1 >	2	0.174	0.667	0.222	14	0.146
16.	651	5544	< 6, 7, 11, 8, 5, 9, 1, 2, 4, 0, 10, 3 >	< 8, 11, 6, 9, 7, 2, 5, 4, 1, 10, 0, 3 >	3	0.44	0.438	0.405	125	0.484
17.	$72^2 1$	23760	< 5, 9, 10, 2, 6, 3, 4, 0, 11, 8, 1, 7 >	< 5, 9, 10, 6, 3, 2, 4, 0, 11, 8, 1, 7 >	4	0.051	0.2	0.024	4	0.009
18.	531^4	665280	< 9, 0, 8, 10, 2, 1, 4, 7, 11, 6, 5, 3 >	< 8, 1, 4, 9, 10, 2, 0, 11, 7, 5, 6, 3 >	4	0.359	0.45	0.345	226	0.343
19.	831	1980	< 11, 7, 9, 1, 2, 8, 10, 5, 0, 3, 6, 4 >	< 7, 9, 8, 1, 2, 5, 0, 10, 6, 11, 4, 3 >	3	0.6	0.417	0.58	107	0.626
20.	532^2	166320	< 4, 6, 3, 7, 10, 2, 9, 0, 1, 5, 11, 8 >	< 3, 4, 7, 10, 6, 5, 2, 11, 0, 1, 9, 8 >	2	0.361	0.3	0.31	160	0.289
21.	42^4	1247400	< 9, 11, 2, 4, 5, 6, 8, 10, 1, 3, 7, 0 >	< 5, 9, 11, 4, 2, 8, 6, 1, 7, 3, 10, 0 >	1	0.286	0.125	0.182	86	0.133
							Mean	0.26	—	0.23

Table B.2: Results of using the heuristic, $w\widehat{OI}_{\text{dis}} + (1-w)\widehat{C}$, to predict the time-point rows in Babbitt's *None but the Lonely Flute*. Note that $\widehat{Q}^*$ shows the normalized value of $w\widehat{OI}_{\text{dis}} + (1-w)\widehat{C}$ assigned to each time-point row chosen by Babbitt (shown in the column headed "Tp"), where $w = 0.798$.

Table B.2 also shows that the mean value of $\widehat{Q}^*$ over the mosaics in *None but the Lonely Flute* is 0.26. These results suggest that the factors taken into consideration by the heuristic may be among those used by Babbitt when choosing his time-point rows in this piece. On the other hand, as we can see by the mean of r/N in Table B.3, we achieve a significantly reduced performance for *Around the Horn* when compared to *None but the Lonely Flute*, despite selecting the best-performing value of w. While there remain several mosaics in *Around the Horn* in which our heuristic performs well (e.g., ordered mosaics two and fourteen), it is worth noting that these are distributed relatively widely across the whole of the work. For example, r/N values lower than, 0.01, are found in mosaic numbers 8, 16, 20 and 22. These observations suggest either (1) the presence of entirely new metrical structures (for which a clock with a unit equal to four is no longer appropriate) or (2) longer periods of metrical displacement by the respective time points than in *None but the Lonely Flute*.

The findings in both pieces provide some evidence to support Babbitt's claim that the organization of time points suggests a particular meter (Babbitt, 1962, 63), insofar as beat induction can be used as a proxy for meter. To a greater extent, our results support our hypothesis that the pitch-class and time-point rows are generally related in some way. Noting the poor predictive performance of the heuristic in some of the ordered mosaics, it is clear that there are additional criteria we have not considered. However, even in these mosaics in which the predictive performance is low, the results may prove interesting from an analytical point of view. For example, the "stepping out of time" portion of *None but the Lonely Flute*, as described by Leong and McNutt (2005), coincides with two rather poorly performing ordered mosaics (fifteen and sixteen) which lie between two rather high performing ordered mosaics (fourteen and seventeen) in Figure B.2. These findings appear to support the metrical displacement by the time-point rows observed in this passage by Leong and McNutt.

The results summarized in Tables B.2 and B.3 suggest a number of possibilities worthy of further exploration. First, the relatively poor predictive performance in some of the ordered mosaics (just described) may suggest that a global heuristic across all mosaics of a piece is not the way forward—as each mosaic corresponds to a musically unique point in a piece, perhaps it would be preferable to consider it in isolation. It could be, for example, that some factors that depend more on local context need to be taken into account, such as which time-point rows have been used previously in the piece and which partitions remain to be used in the array. Indeed, if we "fine tune" the values for each measure of our heuristic in each mosaic, we already achieve a significant improvement. In other words, the value of w that performs best globally over a whole piece is not always the value of w that performs best for an individual ordered mosaic.

Around the Horn

		Babbitt's rows			Clocks	Measures		Evaluation		
	Tp/PcOMs	Num. possible	Tp	Pc	Location	$\widehat{OI}_{\text{dis}}$	$\widehat{C}$	$\widehat{Q}^*$	r	r/N
1.	$4^{2}31$	138600	$< 2,6,11,3,0,10,8,7,9,4,5,1 >$	$< 6,3,2,11,9,4,10,5,1,0,8,7 >$	1	0.439	0.524	0.411	275	0.446
2.	532^{2}	166320	$< 2,5,9,4,8,10,1,6,11,0,7,3 >$	$< 5,9,4,10,8,1,0,11,6,2,7,3 >$	3	0.31	0.4	0.26	144	0.235
3.	6321	55440	$< 5,1,10,4,0,9,3,6,7,11,8,2 >$	$< 10,3,5,0,9,6,1,4,8,7,11,2 >$	3	0.469	0.706	0.493	193	0.466
4.	63^{2}	18480	$< 8,9,3,2,11,6,0,5,4,10,1,7 >$	$< 3,8,4,11,9,2,6,5,0,1,10,7 >$	1	0.364	0.556	0.364	161	0.438
5.	921	660	$< 9,7,0,2,6,4,5,8,1,11,3,10 >$	$< 2,7,6,9,0,4,5,8,1,11,3,10 >$	3	0.207	0.667	0.217	28	0.144
6.	651	5544	$< 0,6,1,11,3,2,10,5,8,9,4,7 >$	$< 6,1,2,0,10,11,3,8,5,4,9,7 >$	1	0.346	0.529	0.333	102	0.381
7.	641^{2}	27720	$< 11,7,0,3,6,1,10,9,5,4,8,2 >$	$< 0,7,11,6,1,3,10,9,5,4,8,2 >$	1	0.161	0.533	0.169	52	0.136
8.	$10\ 1^{2}$	132	$< 2,7,11,6,3,10,0,8,1,5,4,9 >$	$< 7,11,2,6,3,10,0,8,1,5,4,9 >$	1	0.105	0.333	0.144	6	0.085
9.	3^{4}	369600	$< 0,8,11,3,9,1,5,7,10,2,6,4 >$	$< 9,7,0,5,4,1,8,10,6,11,3,2 >$	2	0.674	0.348	0.631	492	0.696
10.	$43^{2}2$	277200	$< 4,8,5,1,7,0,3,11,6,10,2,9 >$	$< 3,11,8,6,10,1,5,4,7,2,9,0 >$	3	0.596	0.478	0.593	492	0.652
11.	$5^{2}1^{2}$	33264	$< 10,1,8,11,7,0,2,4,9,5,3,6 >$	$< 11,10,2,7,9,0,4,5,8,1,3,6 >$	2	0.667	0.368	0.669	293	0.711
12.	84	495	$< 8,1,9,11,6,3,4,2,5,10,7,0 >$	$< 8,1,9,11,6,3,2,10,4,7,0,5 >$	2	0.278	0.25	0.264	25	0.248
13.	543	27720	$< 6,9,0,7,10,2,3,8,5,1,4,11 >$	$< 3,8,6,9,0,1,5,7,4,11,10,2 >$	3	0.657	0.368	0.645	338	0.741
14.	6^{2}	924	$< 10,11,0,8,3,7,9,1,5,6,2,4 >$	$< 10,11,0,9,8,3,1,6,2,7,5,4 >$	4	0.4	0.118	0.319	49	0.405
15.	$5^{2}2$	16632	$< 0,4,5,11,9,10,1,8,2,6,7,3 >$	$< 5,4,11,0,8,9,10,1,2,7,6,3 >$	1	0.308	0.267	0.281	71	0.227
16.	$72^{2}1$	23760	$< 5,9,2,4,0,1,3,7,8,10,6,11 >$	$< 9,0,5,4,2,1,3,8,7,10,6,11 >$	4	0.194	0	0.105	23	0.06
17.	93	220	$< 3,6,2,11,9,1,8,4,5,7,0,10 >$	$< 6,11,9,1,3,8,4,5,0,2,7,10 >$	1	0.6	0.444	0.597	73	0.709
18.	10 2	66	$< 7,10,5,0,3,1,9,4,8,6,11,2 >$	$< 7,5,10,0,1,9,4,8,3,6,11,2 >$	1	0.455	0.5	0.496	19	0.543
19.	11 1	12	$< 2,7,4,0,11,8,3,5,1,6,10,9 >$	$< 7,2,4,0,11,8,3,5,1,6,10,9 >$	1	0.1	0.667	0.189	3	0.25
20.	91^{3}	1320	$< 2,6,10,3,11,7,9,4,1,0,8,5 >$	$< 6,3,10,2,11,7,9,4,1,0,8,5 >$	1	0.148	0.444	0.145	18	0.087
21.	82^{2}	2970	$< 7,1,5,10,3,6,9,8,2,0,4,11 >$	$< 1,5,3,8,6,7,10,9,2,0,4,11 >$	2	0.385	0.667	0.389	63	0.318
22.	732	7920	$< 5,8,10,6,1,3,7,11,4,0,2,9 >$	$< 5,10,6,3,8,7,11,4,1,0,2,9 >$	2	0.2	0.267	0.148	38	0.099
23.	741	3960	$< 11,1,4,6,8,9,0,5,10,7,2,3 >$	$< 1,11,4,8,10,7,6,9,0,2,5,3 >$	2	0.478	0.214	0.419	89	0.42
							Mean	0.36	—	0.37

Table B.3: Results of using $w\widehat{OI}_{\text{dis}} + (1-w)\widehat{C}$ to predict the time-point rows in Babbitt's *Around the Horn*. Note that $\widehat{Q}^*$ shows the normalized value of $w\widehat{OI}_{\text{dis}} + (1-w)\widehat{C}$ assigned to each time-point row chosen by Babbitt (shown in the column headed "Tp"), where $w = 0.859$.

What this suggests is that the relative weighting of order inversions to compliance with a meter may not be the same throughout a piece—that is, Babbitt may not have applied a constant relative prioritization of these two factors throughout a piece. However, the fact that the value of w that worked best globally is almost identical for the two pieces considered here would seem to suggest that he applied an approximately constant relative prioritization of the two factors across different pieces.

It is also possible that the considerations modelled by our heuristic were indeed important to Babbitt, but that, due to the large number of time-point rows in some ordered mosaics, he was unable to consistently choose the best of all possible rows at each point. Instead, we can hypothesize that he just chose the best that he was able to find using, presumably, manual methods of searching the space of possible rows for each mosaic. Indeed, in some ordered mosaics with relatively few possible rows—for example, mosaic 12 in *None but the Lonely Flute* and mosaic 8 in *Around the Horn*—we achieve low values for both $\widehat{Q}^*$ and r/N. However, further analysis would be required in order to determine how significant these particular results are.

6 Conclusion

In this paper, we have introduced a heuristic based on Rothgeb's dissimilarity measure of order inversions and Povel and Essens' clock induction model, for predicting the time-point rows in two of Babbitt's later works, *None but the Lonely Flute* and *Around the Horn*. The results show that, while the performance of our heuristic differs between the two pieces, the relative importance of order inversions over clock induction remains the same—that is, similar values of w in both pieces resulted in globally optimal performance. The use of clock induction, however, did improve the overall performance when compared to using order inversions alone. We believe that our results serve as a validation of Babbitt's own words that he considered meter when constructing his time-point rows, despite the lesser importance of the clocks measure compared to order inversions in our findings. A full exploration of how clocks might reveal interesting aspects in Babbitt's time-point pieces is beyond the scope of this paper. However, it might prove interesting to explore how these measures perform in some of Babbitt's works based on a more complex use of the time-point system that does not use equal-note-value-strings and where units often vary in size (e.g., 32nd notes or sixteenth note triplets). As presented, our use of clocks assumes an unchanging beat, however, with little modification and some additional constraints, we believe clocks could accommodate time-point rows containing repetitions and account for instances in Babbitt's music where his use of the time-point system conflicts with the notated meter.

Acknowledgements

The work reported in this paper was carried out as part of the EC-funded collaborative project, “Learning to Create" (Lrn2Cre8). The Lrn2Cre8 project acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET grant number 610859.

References

Alegant, B. (1993). *The seventy-seven partitions of the aggregate: Analytical and theoretical implications.* Ph.D. diss., University of Rochester.

Babbitt, M. (1960). Twelve-tone invariants as compositional determinants. *Journal of Music Theory*, 46(2):246–259.

Babbitt, M. (1961). Set structure as a compositional determinant. *Journal of Music Theory*, 5(1):72–94.

Babbitt, M. (1962). Twelve-tone rhythmic structure and the electronic medium. *Perspectives of New Music*, 1(1):49–79.

Bemman, B. and Meredith, D. (2014). From analysis to surface: Generating the surface of Milton Babbitt's *Sheer Pluck* from a parsimonious encoding of an analysis of its pitch-class structure. In *The Music Encoding Conference, 20–23 May 2014*, Charlottesville, VA.

Bemman, B. and Meredith, D. (2016). Generating Milton Babbitt's all-partition arrays. *Journal of New Music Research*, 45(2):1–21.

Bernstein, Z. (2014). The problem of completeness in Milton Babbitt's music and thought. In *43rd Annual Meeting of The Music Theory Society of New York State (5–6 April 2014)*, New York, NY.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition.

Dubiel, J. (1997). What's the use of the twelve-tone system? *Perspectives of New Music*, 35(2):33–51.

Eck, D. (2001). A positive-evidence model for rhythmical beat induction. *Journal of New Music Research*, 30(2):187–200.

Hasty, C. (1999). *Meter as Rhythm.* Oxford University Press, New York, NY.

Ilomäki, Tuukka. (2008). *On the similarity of twelve-tone rows.* Ph.D. diss., Sibelius Academy.

Johnson, W. M. (1984). Time-point sets and meter. *Perspectives of New Music*, 23(1):278–293.

Lee, C. (1991). The perception of metrical structure: Experimental evidence and a model. In P. Howell, R. West, and I. Cross, eds. *Representing Musical Structure*, volume 5 of *Cognitive Science Series*, pages 59–127. Academic Press, London.

Leong, D. and McNutt, E. (2005). Virtuosity in Babbitt's *Lonely Flute*. *Music Theory Online*, 11(1).

Lewin, D. (1976). On partial ordering. *Perspectives of New Music*, 14(2):252–257.

London, J. (2004). *Hearing in Time*. Oxford University Press, New York, NY.

Mead, A. (1984). Recent developments in the music of Milton Babbitt. *The Musical Quarterly*, 70(3):310–331.

Mead, A. (1987). About *About Time*'s time: A survey of Milton Babbitt's recent rhythmic practice. *Perspectives of New Music*, 25:182–235.

Mead, A. (1994). *An Introduction to the Music of Milton Babbitt*. Princeton University Press, Princeton, NJ.

Mead, A. (1997). Still being an American composer: Milton Babbitt at eighty. *Perspectives of New Music*, 35(2):101–126.

Morris, R. and Alegant, B. (1988). The even partitions in twelve-tone music. *Music Theory Spectrum*, 10:74–101.

Morris, R. (2003). Pitch-class duplication in serial music: Partitions of the double aggregate. *Perspectives of New Music*, 41(2):96–121.

Povel, D. J. and Essens, P. (1985). Perception of temporal patterns. *Music Perception: An Interdisciplinary Journal*, 23(3):411–440.

Rothgeb, J. (1967). Some ordering relationships in the twelve-tone system. *Journal of Music Theory*, 11(2):176–197.

Starr, D. (1978). Sets, invariance and partitions. *Journal of Music Theory*, 22(1):1–42.

Starr, D. (1984). Derivation and polyphony. *Perspectives of New Music*, 23(1):180–257.

Sward, R. L. G. (1981). *An Examination of the Mathematical Systems Used in Selected Compositions of Milton Babbitt and Iannis Xenakis*. Ph.D. diss., Northwestern University.

Paper C

Generating Milton Babbitt's All-partition Arrays

Bemman, B. and Meredith, D.

The paper has been published in the
Journal of New Music Research Vol. 45(2), pp. 184–204, 2016.

© 2016 Taylor and Francis Group, Routledge
The layout has been revised.

Abstract

In most of Milton Babbitt's (1916–2011) works written since the early 1960s, both the pitch and rhythmic content is organised according to a highly constrained structure known as the all-partition array. The all-partition array provides a framework that ensures that as many different forms of a tone row as possible (generated by any combination of transposition, inversion or reversal) are expressed "horizontally" and that all 58 integer partitions of 12 are expressed as 58 adjacent "vertical" aggregates. We present a greedy backtracking algorithm for generating a particular type of all-partition array found in Babbitt's works, known as a Smalley array. Constructing such an array is a difficult task, and we present two heuristics for helping to generate this type of structure. We provide the parameter values required by this algorithm to generate the specific all-partition arrays used in three of Babbitt's works. Finally, we evaluate the algorithm and the heuristics in terms of how well they predict the sequences of integer partitions used in two of Babbitt's works. We also explore the effect of the heuristics on the performance of the algorithm when it is used in an attempt to generate a novel array.

1 Introduction

The *all-partition array* is a mathematical structure that has been used in the composition of certain twelve-tone works by select composers, of whom Milton Babbitt (1916–2011) is the most notable. An all-partition array provides a framework that ensures that as many different forms of a tone row as possible (generated by any combination of transposition, inversion or reversal) are expressed "horizontally" and that all 58 integer partitions of 12 are expressed as 58 adjacent "vertical" aggregates (see Figure C.3 for an example of an all-partition array and three of its 58 partitions). Nearly all of Babbitt's compositions since the early 1960s (marking the start of his second period) use this structure. Understanding the structure of the all-partition array is important for helping to explain aspects of both Babbitt's compositional process and the detailed structure of his works, as it was used to help achieve his goal of *maximal diversity*.[1]

The purpose of this paper is to present an algorithm for generating a particular type of all-partition array found in Babbitt's works, known as a *Smalley array* (discussed in detail in sections 2 and 4). Smalley arrays were developed by the composer and mathematician, David Smalley, while he was working with Babbitt during the early 1980s (Mead, 1994, p. 220). These arrays are interesting and worthwhile to study for several reasons. For the music theorist,

[1]Babbitt's *principle of maximal diversity* is the presentation of as many musical parameters in as many different possibilities as possible. See Mead (1994, pp. 33–34) and Bernstein (2014) for discussions on the various ways Babbitt achieved maximal diversity in his works.

analysing a piece based on a Smalley all-partition array is a challenging task. In fact, Bemman and Meredith (2015b) have shown that it involves solving a special case of the well-known *set-covering problem* which can be shown to be NP-hard (Cormen et al., 2009). *Constructing* a Smalley array is also a hard problem. However, the fact that Smalley succeeded in discovering such an array suggests the possibility of devising a practical algorithm that can generate such an array using reasonable resources of time and space. Indeed, other individuals have since successfully constructed arrays of various types (discussed further in section 3). However, the process that Smalley used to discover his array remains unclear. Furthermore, that this apparently computationally complex problem was solved without the use of a computer is of considerable interest and could potentially shed light on the mechanisms that underlie creative processes more generally.

An algorithm for generating Smalley arrays, could serve as a hypothesis for how such an array was constructed by Smalley himself. However, it is possible, of course, that none, some or all of the steps taken by the algorithm were used by Smalley in constructing his arrays by hand. Nonetheless, having such an algorithm would provide certain benefits. For example, it would allow us to determine whether there exist other Smalley arrays that were not used by Babbitt. This is significant as only a handful of Smalley arrays have been found in Babbitt's music. Attempting to devise an algorithm for generating arrays of this type could provide valuable insight into why so few distinct Smalley arrays are known to exist. Assuming we are able to generate a new array, we could then study it and potentially use it as the basis for constructing novel musical works.

In the remainder of this paper, we begin by defining the structure of the all-partition array (section 2). Next, we review previous work relating to the construction of all-partition arrays (section 3). Then, in section 4, we define the sub-class of all-partition arrays known as *Smalley arrays*. In sections 5 and 6, we propose an algorithm for generating Smalley arrays. In section 7, we provide the parameter values that need to be given to this algorithm in order for it to generate the all-partition arrays for three of Babbitt's works. Finally, in section 8, we evaluate the algorithm and the heuristics in terms of how well they predict the sequences of integer partitions used in two of Babbitt's works, *Sheer Pluck* (1984) and *About Time* (1982). We also explore the effect of the heuristics on the performance of the algorithm when it is used in an attempt to generate a novel array.

2 The all-partition array

The all-partition array and the many different ways in which it has been used in practice have been written about in great detail (see, in particular,

Mead, 1994).[2] We will therefore only provide a summary here, focusing in particular on the structures found in a specific type of all-partition array known as a *Smalley array* (discussed here and in section 4). Insofar as possible, we use existing and well-established terminology to describe the structure of the all-partition array. However, we also introduce some new terminology where necessary or if we feel that the existing terminology is ambiguous or excessively overloaded.

2.1 Preliminary definitions

Before considering the all-partition array in detail, we first review some concepts from pitch class set theory that will be used extensively throughout this paper. We use the term *aggregate* in the sense in which it is usually used in pitch class set theory to mean the universe of pitch classes—that is, the set $\{0, 1, \ldots 11\}$. A *tone-row*, $A = \langle p_1, p_2, \ldots p_{12} \rangle$, is then an ordered set of pitch classes that contains each element in the aggregate exactly once—that is, $\bigcup_{i=1}^{12} \{p_i\} = \{0, 1, \ldots 11\}$.[3] Each tone row belongs to an equivalence class of rows related by any combination of transposition, inversion or retrograde (i.e., reversal). Such an equivalence class is known as a *row class* and the members of such a row class are called *row forms*. Traditionally, each row form is denoted by the transformation that produces it from the original or *prime* form, using the abbreviations P_n, I_n, R_n and RI_n for transposition, inversion, retrograde and retrograde inversion, respectively, each combined with a transposition by n semitones.[4] The universe of tone rows can thus be strictly partitioned into row classes, each of which is a set of 48 row forms that is closed under the operations $\{\mathrm{P}_0, \ldots \mathrm{P}_{11}, \mathrm{I}_0, \ldots \mathrm{I}_{11}, \mathrm{R}_0, \ldots \mathrm{R}_{11}, \mathrm{RI}_0, \ldots \mathrm{RI}_{11}\}$.

Combinatoriality is a property of certain pitch class sets whereby the union of an unordered set and one or more transformations of this set form an aggregate. For example, a *hexachord*, $H_1 = \{p_1, p_2, \ldots p_6\}$, is said to be combinatorial if and only if, for some transformation, T, $H_1 \cup T(H_1) = \{0, 1, \ldots 11\}$ (Babbitt, 1961, p. 78). A tone row constructed from combinatorial sets may be used to construct additional tone rows having predictable properties of *invariance* and *complementation* with respect to these sets and their locations in each row. If $A_1 = \langle p_1, p_2, \ldots p_{12} \rangle$, and $A_2 = \langle q_1, q_2, \ldots q_{12} \rangle$ are two forms of the same tone row, then A_1 and A_2 are said to be *hexachordally combinatorial* (henceforth *hc-related*), if and only if $\{p_1, p_2, \ldots p_6\} = \{q_7, q_8, \ldots q_{12}\}$ (Babbitt, 1961, p. 78). For example, if $\mathrm{P}_0 = \langle 0, 11, 6, 4, 10, 5, 8, 9, 1, 3, 2, 7 \rangle$, then this

[2] The origins of the all-partition array can be traced to the *trichordal array*, a structure that Babbitt used in pieces such as his *Composition for Four Instruments* (1948). See Mead (1994) for a thorough discussion of Babbitt's techniques.

[3] We consistently use $\langle \cdot \rangle$ for ordered sets and $\{\cdot\}$ for unordered sets (in the normal mathematical sense).

[4] Other transformations exist for defining equivalence classes and include for example, the M_5 and M_7 *multiplicative* transformations.

	1	2	3	4	5	6	7	8
1	RI_6	P_5	I_9	R_8	I_3	P_{11}	RI_0	R_2
2	I_6	R_5	RI_3	P_2	RI_9	R_{11}	I_0	P_8
3	I_5	P_1	R_{10}	RI_2	R_4	P_7	I_{11}	RI_8
4	R_7	RI_{11}	I_2	P_{10}	I_8	RI_5	R_1	P_4
5	RI_7	I_4	R_9	P_6	R_3	I_{10}	RI_1	P_0
6	R_6	P_3	RI_4	I_1	RI_{10}	P_9	R_0	I_7

Fig. C.1: An example of a 6×8 row-form array containing 24 *hc*-related row forms, where $P_0 = \langle 0, 11, 6, 4, 10, 5, 8, 9, 1, 3, 2, 7 \rangle$.

row form is *hc*-related to the row form $I_7 = \langle 7, 8, 1, 3, 9, 2, 11, 10, 6, 4, 5, 0 \rangle$. A pair of row forms are said to *share the same disjunct hexachords* if and only if $\{p_1, p_2, \ldots p_6\} = \{q_1, q_2, \ldots q_6\}$ (Babbitt, 1961, p. 74). For example, if $P_0 = \langle 0, 1, 3, 6, 7, 9, 2, 4, 5, 8, 10, 11 \rangle$, then it shares the same disjunct hexachords with the row form $P_6 = \langle 6, 7, 9, 0, 1, 3, 8, 10, 11, 2, 4, 5 \rangle$.

2.2 Structural definitions

When beginning the construction of a Smalley all-partition array, pairs of *hc*-related row forms, called *arrays*, are made (Winham, 1970).[5] When the row forms in such an array are represented using the standard abbreviations (i.e., P_n, I_n, R_n and RI_n) the array is called a *row-form array* (Cuciurean, 1997, p. 8). Row-form arrays may be concatenated with one another from left-to-right and from top-to-bottom. Perhaps confusingly, music theorists have used the term "row-form array" both for single row-form arrays and these larger structures formed by concatenating single row-form arrays, distinguishing between them on the basis of size. Adopting this traditional terminology, we focus here on row-form arrays that contain all (and only) the 48 row forms of a row class. Mead (1994, p. 34) refers to all-partition arrays based on row-form arrays of this type as *hyperaggregates* or *row-class aggregates*. Figure C.1 shows an example of such a row-form array.

In Figure C.1, tone rows appear in *hc*-related pairs that are grouped into 8 columns called *blocks* (Mead, 1994, p. 18). Following traditional usage, we refer to each row in a row-form array as a *lyne* (Kassler, 1963, p. 93; Kassler, 1967, p. 14; Mead, 1994, p. 18). Lynes are often expressed explicitly as a sequence of pitch class integers and we call this in extenso representation of a row-form array a *PcMatrix*. A 6×8 row-form array such as the one in Figure C.1 then corresponds to a 6×96 matrix of pitch classes that we

[5]Arrays exhibiting any form of combinatoriality can be constructed and are not limited to equal divisions of a tone row into hexachords. See Starr and Morris (1977, p. 4) for a discussion on so-called *uneven combinatoriality*.

denote by $\text{PcMatrix}_{6,96}$. The row-form array and its corresponding PcMatrix are alternative representations of the same underlying matrix of pitch classes. The difference between the two representations is that the row-form array indicates how the pitch classes are grouped into row forms and how these row forms are related to the prime form of the row used in the work; whereas the PcMatrix simply specifies, in extenso, the pitch class at each position in the matrix.

The various possible types of all-partition array can be classified according to the number of lynes that they contain (Mead, 1994, p. 18). In principle, the number of lynes in an all-partition array can be any value less than or equal to 12 (assuming 12 pitch classes per octave). However, as explained by Mead (1994, pp. 31–32), by constraining himself to constructing arrays from all-combinatorial hexachords, Babbitt limited himself to all-partition arrays containing four, six or twelve lynes.

In this paper, we focus on all-partition arrays whose tone rows are constructed from the all-combinatorial hexachord known as the *D-hexachord*.[6] Single all-partition arrays using this hexachord generally have six lynes (Mead, 1994, p. 32). The D-hexachord is a collection of six pitch classes comprised of "two disjunct 3-pc chromatic clusters a tritone apart", the prime form for which is $\langle 0,1,2,6,7,8 \rangle$ (Cuciurean, 1997, p. 11). Such a hexachord can be mapped onto both itself and its complement by transposition or inversion and transposition. If S is a pitch class set, then we denote transposition by n semitones by $\mathrm{T}_n(S)$ and inversion around 0 followed by transposition by n semitones by $\mathrm{I}_n(S)$. We denote the complement of S in the aggregate by $\overline{S}$. Thus, if $S = \{0,1,2,6,7,8\}$, then $\mathrm{T}_3(S) = \mathrm{T}_9(S) = \mathrm{I}_{11}(S) = \mathrm{I}_5(S) = \overline{S}$ and $\mathrm{T}_0(S) = \mathrm{T}_6(S) = \mathrm{I}_2(S) = \mathrm{I}_8(S) = S$. For this reason, there exist only six distinct D-hexachords. In turn, the row class of a tone row constructed from two D-hexachords can be partitioned into six equivalence classes, each containing eight tone rows that share the same disjunct hexachords (Mead, 1994, pp. 31–32). Typically, each of the six lynes in an all-partition array built on the D-hexachord employs the eight tone rows from one of these equivalence classes.[7]

The most important constraint on the structure of an all-partition array is that it must be possible to partition its PcMatrix into a sequence of *ordered mosaics*. We define an *ordered mosaic* to be an ordered aggregate partitioned into n ordered pitch class sets, each belonging to a lyne of the PcMatrix, where n is less than or equal to the number of lynes. Our terminology here is a refinement of Robert Morris' concept of *mosaic* which he defined to be an

[6] See Babbitt (1955) and Martino (1961) for discussions on all six of the all-combinatorial hexachords and their letter nomenclature.

[7] Notable exceptions, where the number of lynes is not equal to the number of distinct hexachords used in an array, can be found in Babbitt's *String Quartet no. 3* (1970), *Post-Partitions* (1966) and *Sextets* (1966) (Mead, 1994, p. 32).

$$\begin{vmatrix} 11 & 4 & 3 & 5 & 9 & 10 & 1 & 8 & 2 & 0 & 7 & 6 \\ 6 & 7 & 0 & 2 & 8 & 1 & 10 & 9 & 5 & 3 & 4 & 11 \\ 5 & 6 & 11 & 1 & 7 & 0 & 9 & 8 & 4 & 2 & 3 & 10 \\ 2 & 9 & 10 & 8 & 4 & 3 & 0 & 5 & 11 & 1 & 6 & 7 \\ 0 & 5 & 4 & 6 & 10 & 11 & 2 & 9 & 3 & 1 & 8 & 7 \\ 1 & 8 & 9 & 7 & 3 & 2 & 11 & 4 & 10 & 0 & 5 & 6 \end{vmatrix}$$

(a) One possible ordered mosaic.

$$\begin{vmatrix} 11 & 4 & 3 & 5 & 9 & 10 & 1 & 8 & 2 & 0 & 7 & 6 \\ 6 & 7 & 0 & 2 & 8 & 1 & 10 & 9 & 5 & 3 & 4 & 11 \\ 5 & 6 & 11 & 1 & 7 & 0 & 9 & 8 & 4 & 2 & 3 & 10 \\ 2 & 9 & 10 & 8 & 4 & 3 & 0 & 5 & 11 & 1 & 6 & 7 \\ 0 & 5 & 4 & 6 & 10 & 11 & 2 & 9 & 3 & 1 & 8 & 7 \\ 1 & 8 & 9 & 7 & 3 & 2 & 11 & 4 & 10 & 0 & 5 & 6 \end{vmatrix}$$

(b) A second possible ordered mosaic.

Fig. C.2: Two possible ordered mosaics in an excerpt from a $\text{PcMatrix}_{6,96}$ corresponding to the first block of its row-form array (which is shown in Figure C.1). Each ordered mosaic consists of the pitch classes on the left of the boundary marked by the ragged line. In (a) the ordered mosaic is $\langle\langle 11,4,3\rangle, \langle 6,7\rangle, \langle 5\rangle, \langle 2,9,10\rangle, \langle 0\rangle, \langle 1,8\rangle\rangle$ and in (b) the ordered mosaic is $\langle\langle 11\rangle, \langle\rangle, \langle\rangle, \langle\rangle, \langle 0,5,4,6,10\rangle, \langle 1,8,9,7,3,2\rangle\rangle$,

unordered aggregate partitioned into *unordered* pitch class sets (Morris, 2003, p. 103). Figure C.2 shows two possible ordered mosaics in an excerpt from a $\text{PcMatrix}_{6,96}$ corresponding to the first block of its array in Figure C.1.

Note in Figure C.2(a) that the region to the left of the ragged boundary line contains an aggregate that can be represented as a collection of lyne *segments* with lengths (from top to bottom) of 3, 2, 1, 3, 1 and 2. As noted by Mead (1994, p. 32), these aggregate regions are traditionally represented by listing the segment lengths in descending order of size. For example, the region in Figure C.2(a) would be $3,3,2,2,1,1$ (often written $3^2 2^2 1^2$, where each exponent denotes the number of occurrences of segments with length equal to its base) and its mosaic (in the sense used by Morris) might be $\{\{11,4,3\},\{2,9,10\},\{6,7\},\{1,8\},\{5\},\{0\}\}$.[8] Clearly, however, these representations of a mosaic and the lengths of its segments fail to specify the lyne within which each segment occurs and fail to indicate the lynes that do not contain any segments in a given mosaic. We therefore adopt a more informative representation in which the lengths of the segments in an aggregate region are listed in a vector (e.g., $\langle 1,0,0,0,5,6\rangle$ for the example in Figure C.2(b) rather than 651) and the ordered mosaic itself is represented as an ordered set of ordered sets (e.g., $\langle\langle 11\rangle, \langle\rangle, \langle\rangle, \langle\rangle, \langle 0,5,4,6,10\rangle, \langle 1,8,9,7,3,2\rangle\rangle$ for the region in Figure C.2(b)). We now define some terms that allow us to use this more informative representation.

An *integer partition*, which we denote by $\text{IntPart}(s_1, s_2, ..., s_k)$, is a representation of an integer, $n = \sum_{i=1}^{k} s_i$, as an unordered sum of k positive integers, $s_1 \ldots s_k$ (Eger, 2013; Tani and Bouroubi, 2011). For example, if $n = 12$ and $k = 6$, then one possible integer partition is $\text{IntPart}(3,3,2,2,1,1)$. An

[8]It should be noted that Mead does not refer to these regions as mosaics, but as *partitions*.

integer composition, which we denote by $\text{IntComp}(s_1, s_2, ..., s_k)$, is a representation of an integer, $n = \sum_{i=1}^{k} s_i$, as an *ordered* sum of k positive integers (Eger, 2013; Page, 2013). For example, if $n = 12$ and $k = 6$, then $\text{IntComp}(3, 3, 2, 2, 1, 1) \neq \text{IntComp}(3, 2, 1, 3, 2, 1)$.[9] A *weak* integer composition, which we denote by $\text{WIntComp}(s_1, s_2, ..., s_k)$, is a representation of an integer $n = \sum_{i=1}^{k} s_i$, as an *ordered* sum of k *non-negative* integers, (i.e., including zero) (Page, 2013). For example, if $n = 12$ and $k = 6$, then $\text{WIntComp}(6, 6, 0, 0, 0, 0)$ is a weak integer composition.

We denote by ℓ the total number of distinct integer partitions possible when $n = 12$ and k is the number of lynes. In those of his works based on a single all-partition array, Babbitt ensures that each of these ℓ integer partitions is represented exactly once (Mead, 1994, pp. 31–32). Indeed, it is for this reason that the resulting structure is called an *all-partition* array.

If c is the integer composition, $\text{IntComp}(s_1, s_2, ..., s_k)$, then we define the *integer partition associated with* c, denoted by $[c]$, to be $\text{IntPart}(s_1, s_2, ..., s_k)$. That is, the integer partition $[c]$ is the unordered set containing all and only those elements in the integer composition, c. We further define that two integer compositions, c and d, are *partitionally equivalent* if and only if $[c] = [d]$. This terminology is compatible with that of Morris (2003, p. 103), who refers to all mosaics formed by the same integer partition as belonging to the same *partition class*. Two integer compositions, c and d, are *partitionally distinct* if and only if $[c] \neq [d]$. Partitional equivalence is a true equivalence relation on the set of integer compositions that partitions them into classes, each of which corresponds to a distinct integer partition. From these definitions, we may say more precisely that each of the ℓ possible integer partitions is represented in an all-partition array by an ordered mosaic in which the lyne segment lengths are given by an integer composition that is partitionally distinct from every other integer composition used in the array.

The number of pitch classes required in any single all-partition array is equal to 12ℓ. However, a PcMatrix of the type discussed here must contain $48 \times 12 = 576$ entries, since its array contains each of the 48 row forms belonging to a specific row class. When the number of lynes, k, is 6, as is typically the case when the array is based on the D-hexachord, then $\ell = 58$. This implies that the number of pitch classes required to populate the aggregate regions corresponding to the $\ell = 58$ integer partitions, i.e., $12 \times 58 = 696$, exceeds the number of entries in the PcMatrix by $696 - 576 = 120$. In order to satisfy the constraint that all 58 integer partitions are represented by ordered mosaics, Babbitt had to insert 120 additional pitch classes into these PcMatrices. These additional entries are found by repeating certain pitch classes in each lyne. In this way, Babbitt was able to preserve the order of pitch classes in the underlying row forms (Mead, 1994, p. 32). All-partition arrays having

[9] Babbitt (1961, p. 83) also uses the term “composition" in this sense.

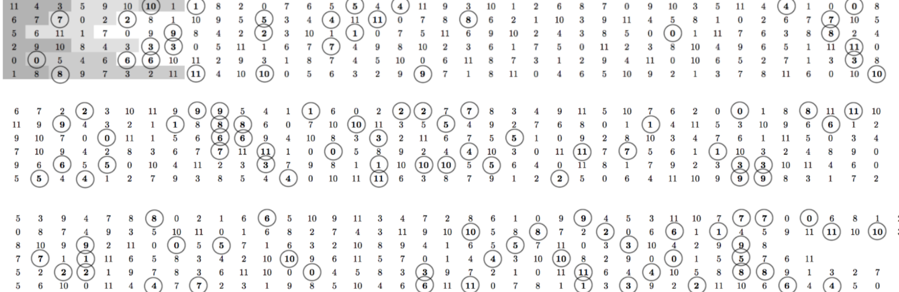


Fig. C.3: The complete WPcMatrix for the six-lyne, self-contained and horizontally weighted all-partition array in *Sheer Pluck* (1984), represented as an irregular matrix of pitch classes (note the unequal lyne lengths). The first three of its 58 partitions have been indicated in three different shades of grey. Outer-aggregate repeated pcs (OARPs) are indicated with circles.

these additional pitch classes are thus said to be *horizontally weighted* (Morris, 2010, p. 44). We name these additional pitch classes found in a horizontally weighted all-partition array, *outer-aggregate repeated pcs* (*OARPs*).

For any horizontally weighted all-partition array, its PcMatrix with these additional OARPs will contain 12ℓ entries, and we call this "unpartitioned" matrix a WPcMatrix. A WPcMatrix is always equal in size to its PcMatrix along its first dimension (i.e., the number of lynes, k), but larger along its second dimension due to the way OARPs must be added. Moreover, a WPc-Matrix can be a regular matrix or an irregular matrix (i.e., with unequal lyne lengths), depending on the distribution of its OARPs. Figure C.3 shows the entire six-lyne WPcMatrix found in Babbitt's *Sheer Pluck*, represented as an irregular matrix of pitch class integers with its 120 OARPs indicated by circles.[10] Following on from our definitions above, we define an *all-partition array* to be a partitioning of a WPcMatrix into a sequence of ℓ ordered mosaics, such that the shape of each mosaic is defined by a partitionally distinct integer composition.

3 Previous work

The earliest work on constructing all-partition arrays was carried out by Babbitt himself, however, several other composers and theorists have since made further contributions. We focus here on these more recent efforts, as they present clear, yet contrasting, methodologies on how arrays might be constructed. In particular, we consider those methods presented by Starr and Morris (1977, 1978), Bazelow and Brickle (1976, 1979) and Morris (2010). Writ-

[10]The OARPs in many of Babbitt's all-partition arrays can be found similarly encircled in his sketches (see Bernstein, 2014, p. 8).

y	0 1 4 3 9 2	10 8 5 7 11	6
$\mathrm{T}_5(y)$	5 6		9 8 2 7 3 1 10 0 4 11
$\mathrm{T}_{11}\mathrm{I}(y)$	11 10 7 8	2 9 1 3 6 4 0	5

Fig. C.4: A three-row combination matrix (CM) formed from transformations of a given tone row, y. Note, that each row and column contains 12 distinct pitch-classes. (From Starr and Morris, 1977, p. 18.)

ings by Babbitt related to aspects of array construction include discussions on set combinatoriality (Babbitt, 1955, 1961), general properties of invariance in sets (Babbitt, 1960) and partitions of the aggregate (Babbitt, 1973).

Many of these previous efforts have focused on a lower level of construction than we present here. Early work by both Starr and Morris, for example, examines certain combinatorial and modulo invariant properties of pitch classes in a given set (Starr and Morris, 1977, 1978). In particular, they introduce the notions of *begin-set*, *end-set* and *n-clique*[11] (Starr and Morris, 1977, pp. 12–14) and found that a tone row constructed from these, while satisfying certain constraints, will produce subsequent rows under some transformations with both predictable properties and qualities desirable for the construction of an all-partition array. Vertically concatenating such rows can form a so-called *combination matrix*. Combination matrices (CMs) are matrices of pitch classes with n tone rows and n columns of aggregates containing n^2 row segments (Starr and Morris, 1977, p. 8). Figure C.4 shows an example of a three-row combination matrix constructed from a given tone-row, $y = \langle 0, 1, 4, 3, 9, 2, 10, 8, 5, 7, 11, 6 \rangle$.

It is clear from Figure C.4 how the presence of such columnar aggregates in a CM are of importance to the construction of an all-partition array. Indeed, it is because of this property that CMs can be concatenated from left to right and from top to bottom to form blocks in a row-form array that have a higher chance of being partitionable than ones in which the rows have been chosen at random.[12] Starr and Morris go on to describe how suitable partitions in such a sequence of CMs can be found by swapping pitch classes across column boundaries. At some point in the process of performing these swaps, however, human intervention seems inevitable as the authors do not propose a way for automating their method, despite suggesting this as a possibility (Starr and Morris, 1978, p. 59). Nevertheless, it seems that at least parts of the processes proposed by Starr and Morris can be automated, since Morris (2010) and Kowalski (1985) claim to have developed computer programs to assist with constructing and editing arrays. Unfortunately, these programs

[11]See Babbitt (1961) for a discussion of n-cliques.

[12]Successful all-partition arrays constructed using this method can be found in Morris' musical works.

have not been made publicly available and details of their design have not been published.

Bazelow and Brickle (1976, 1979) and Morris (2010) lay the ground work for automatically constructing arrays. Like Starr and Morris, Bazelow and Brickle are interested in the relationship between the pitch-class content of a tone row and how various transformations can be informative when attempting to find partitions. However, these authors tackle a much simpler problem than that which we address here, namely, what they call *Babbitt's Partition Problem*: "Given an array of four forms of an arbitrary twelve-tone set, how many ways can the array be decomposed entirely into four-lyne, aggregate forming partitions?" (Bazelow and Brickle, 1976, p. 283). One possible solution to this problem, they argue, "would be, roughly speaking a step-by-step method which would examine a given area of set forms over all the possibly four-part partitions of twelve together with their respective permutations" (Bazelow and Brickle, 1976, p. 288). They reject such a brute force approach on the grounds that some partitions will fail to form an aggregate precisely because not all configurations of pitch-classes in a row-form array, determined by the transformations of its tone rows, can have a solution. Indeed, testing for such cases, if such an a priori way exists for rejecting partitions, may be inefficient. Nonetheless, exhaustively testing whether given partitions are aggregate-forming at any given point in a row-form array is computationally trivial and it is the approach we adopt here. Further differences between our methods exist and these will be discussed in more detail below.

In his later work, Morris (2010, p. 74) notes that if one wanted to construct a horizontally weighted array "one simply took the collection of unpartitioned rows and started partitioning aggregates from the left end, duplicating notes on the edges of row segments if desired, until the process finished on the right side". He goes on to state that in 1989 he "wrote a computer program to help manage the process". Unfortunately, this algorithm has not been published, but it would be of considerable relevance to the work we present here, as our model closely follows the described procedure. While the general process is straight-forward, there remain many practical problems that need to be solved in order for this strategy to be practical. Morris acknowledges as much, stating that "one must keep track of the types of partitions, which notes have been duplicated, and backtrack when there are no aggregates to the right available" (Morris, 2010, p. 74). However, satisfying all constraints necessary for constructing a successful all-partition array in this manner is intractable, as the search space is enormous and the number of successful sequences is very small. For example, given a 6×8 row-form array such as the one used in Babbitt's *None but the Lonely Flute* (1991) along with the corresponding configuration of 120 OARPs required to form a horizontally weighted array, Bemman and Meredith (2015a) estimated that the search

space contains $\approx 78^{58}$ possible sequences and they were unable to use this brute-force method to find a single successful sequence even after running the algorithm for several months (Bemman and Meredith, 2015a, p. 772).

It is not surprising, then, that most other efforts to explain the construction of arrays rely on modifying existing arrays in some way—for example, by switching the order of blocks or by using circle-of-fifths transformations (i.e., M_5 and M_7) (Mead, 1994, p. 36). It is intriguing that, despite the difficulties of generating all-partition arrays automatically, human composers have been constructing such arrays for decades. This suggests that, although an exhaustive search approach to finding an all-partition array would be intractable, it may nevertheless be possible to find a set of heuristics that can be used in conjunction with an approximation algorithm to generate all-partition arrays using realistic time and memory resources. The work reported in the remainder of this paper represents an initial attempt to explore this possibility.

4 Smalley Arrays

The six-lyne all-partition arrays that Babbitt used in several of his works were devised by the composer and mathematician, David Smalley, while he was working with Babbitt during the early 1980s (Mead, 1994, p. 220). The arrays created by Smalley differ from those created by Babbitt in terms of both the organisation of row forms in their row-form arrays and the sequence of integer partitions used. The arrays devised by Smalley (henceforth, *Smalley arrays*) satisfy more rigorous constraints than those satisfied by Babbitt's arrays and are therefore harder to construct. For this reason, Smalley arrays were often re-used by Babbitt. Figure C.5 shows how Smalley arrays can be classified according to Babbitt's most common practices.

The first constraint satisfied by a Smalley array (labelled (1) in Figure C.5) is that the WPcMatrix and sequence of ℓ integer compositions must be *self-contained*. This means that the all-partition array must only contain the 576 pitch-classes from the PcMatrix and the 120 outer-aggregate repeated pcs (OARPs). This is the hardest constraint to satisfy. The requirement that any additional pitch class must be an OARP is itself greatly constrained by the particular sequence in which the $\ell = 58$ integer partitions are used. This sequence, however, is itself constrained by the arrangement of tone rows in the row-form array.

The second constraint on a Smalley array (labelled (2) in Figure C.5) is that a PcMatrix must contain 48 tone rows belonging to the same row class. The third constraint (labelled (3) in Figure C.5) is that all the tone rows in a given lyne of the PcMatrix must contain the same disjunct hexachords (which must be distinct from the hexachords used in the other lynes of the PcMatrix). At this point, there is more than one way to arrange the tone rows of a row-

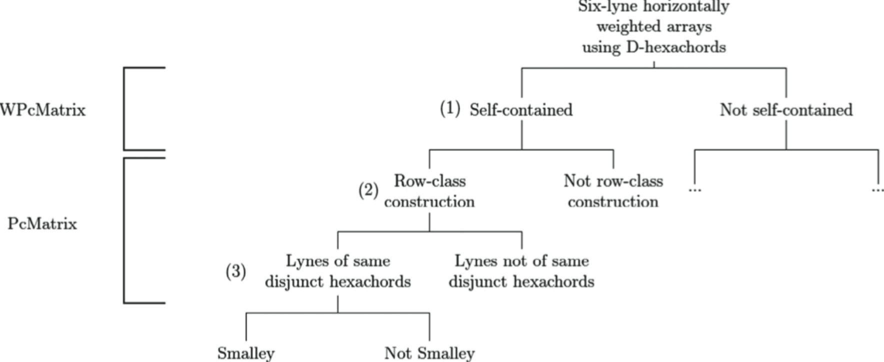


Fig. C.5: Classification of six-lyne, horizontally weighted arrays using D-hexachords according to Babbitt's most common practices. Note, in particular, the constraints satisfied by the Smalley arrays.

form array while satisfying constraints (2) and (3) above. A Smalley array represents one such way and we discuss this arrangement next.

4.1 Organisational constraints on the row-form array

A number of authors have discussed the various ways in which Babbitt organised the tone rows in his works (Bemman and Meredith, 2014; Lake, 1986; Mead, 1994; Morris, 1987). We have found that in his six-lyne, horizontally weighted all-partition arrays using the D-hexachord, two distinct organisational methods are used to construct the row-form arrays. The first of these methods corresponds to the arrangement found in a Smalley array and examples occur in Babbitt's *Sheer Pluck* (1984), *Joy of More Sextets* (1986), and *None but the Lonely Flute* (1991). The second organisational method is found, for example, in Babbitt's *Arie da Capo* (1974) and *About Time* (1982). Despite their differing arrangements, both methods satisfy constraints (2) and (3) in Figure C.5. However, the latter method results in a WPcMatrix that is not self-contained (constraint (1)). In this paper, we focus on Smalley row-form arrays.

In a Smalley row-form array, pairs of tone rows are *hc*-related so as to exhaust the possible 2-combinations of row operations, P, I, R and RI. Moreover, operations in horizontal groups of four *hc*-related pairs form 6 distinct 2×4 Latin rectangles. Figure C.6 shows a row-form array in which the organisational constraints satisfied by a Smalley array are shown. This template illustrates one way in which the 48 tone rows of a row class can be arranged in order to fulfil these constraints.

In Figure C.6, arrows labelled T_3 and T_9 indicate relationships by transposition between pairs of consecutive tone rows in one lyne to consecutive tone

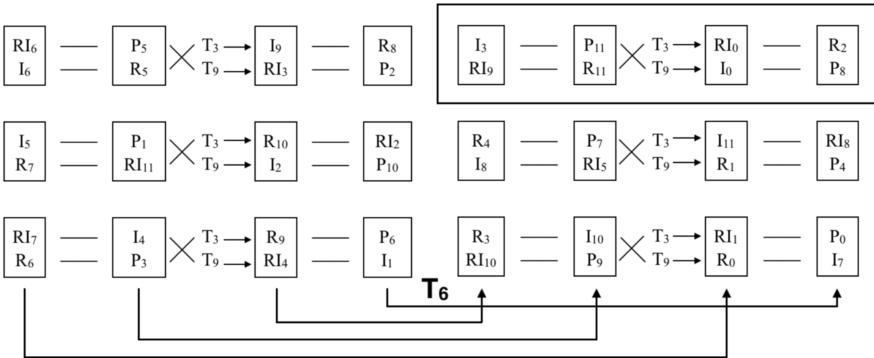


Fig. C.6: A Smalley row-form array and organisational constraints. Each box contains an *hc*-related pair of rows. Each horizontal sequence of four *hc*-related pairs forms a distinct 2×4 Latin rectangle of operations. One of these is indicated by the rectangle drawn in the upper right. Note that this particular arrangement of tone rows represents one of many possible ways in which the constraints of a Smalley row-form array may be satisfied. See text for further explanation.

rows in an adjacent lyne. The order of these transpositions, however, may be reversed such that T_9 occurs on top. Arrows below the template, labelled T_6, indicate blocks similarly related by transposition.

5 Generating a PcMatrix of the Smalley array class

In this section and the next, we present a method for generating the WPc-Matrix of a Smalley array. Figure C.7 gives an overview of this method. In this section, we focus on generating the PcMatrix; in the next, we describe a method for adding OARPs to the PcMatrix in order to produce the WPcMatrix.

Figure C.7 shows that our method takes four pieces of information as input, of which the first is the tone row on which the array is based. At various points during the execution of the method, the algorithm is required to select one from a choice of candidates. For example, at an early stage in the execution, the algorithm must select one of the four different row operations to omit in the first column of the row-form array (see Figure C.8). At this stage in the process, the algorithm may choose any of the four row-form operations to omit. However, we can specify which operation it should select by providing this information in the input. The second to fourth items of input information shown in Figure C.7 are simply encodings of the choices that the algorithm should make at each point in its execution when it needs to select from a number of options. The algorithm can also be run in a generative mode in which it makes random selections when faced with a set

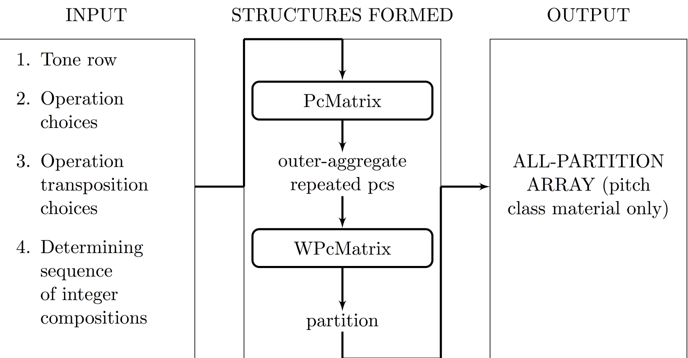


Fig. C.7: Basic flow of constructing an all-partition array using our proposed method. Arrows indicate one or more functions necessary for generating one structure from a previous one.

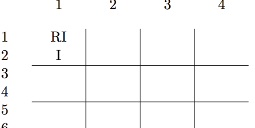
of possibilities.

5.1 Computing row operations in the PcMatrix

We generate the PcMatrix of a Smalley array using two functions that pair *hc*-related tone rows according to the constraints described in Figure C.6. The ComputeOperations function, shown in Figure C.9, organises the row form operations in a row-form array. The second function, ComputeTranspositions, shown in Figure C.11, then assigns transpositions to these operations. Figure C.8 shows how the process of filling a row-form array with operations unfolds in ComputeOperations. Note in Figure C.8 that, after operation pairs in (a) and (b) have been placed, operation pairs in columns 3 and 4 in (c) are simply the reverse of the operation pairs found in columns 1 and 2, respectively. Figure C.8(d) repeats the process illustrated in (a), (b) and (c) for the remaining row pairs until the row-form array is completely filled.

The ComputeOperations function, shown in Figure C.9, takes a single input variable, **Choices**, which is a vector of integer values indicating the specific choices made at each decision point in the algorithm. The range of possible parameter values at each position in **Choices** is given by $\langle$1–4, 1–6, 1–2, 1–2, 1–2, 1–2, 1–2, 1–2$\rangle$. This implies that there are 1536 distinct arrangements of row operations within a row-form array that would be acceptable in a piece based on a Smalley all-partition array.

Figure C.9 shows pseudocode for an implementation of ComputeOperations. The pseudocode conventions that we employ in this paper are based on those used by Meredith (2006). In this pseudocode, "$\leftarrow$" is the assignment operator and block structure is indicated by indentation alone. Ordered sets are notated with angle brackets ("$\langle\cdot\rangle$") and their names are rendered in bold



(a) Step 1. Place an ordered pair of operations in rows 1–2 of column 1 (in this case, ⟨RI, I⟩).

	1	2	3	4
1	RI	P		
2	I	R		
3				
4				
5				
6				

(b) Step 2. Choose an ordering for the other two operations and place in rows 1–2 of column 2.

	1	2	3	4
1	RI	P	I	R
2	I	R	RI	P
3				
4				
5				
6				

(c) Step 3. In column 3 of rows 1–2, place the same operations as in column 1 but in reverse order. In column 4, place the operations in column 2 in reverse order.

	1	2	3	4
1	RI	P	I	R
2	I	R	RI	P
3	I	P	R	RI
4	R	RI	I	P
5	RI	I	R	P
6	R	P	RI	I

(d) Repeat Steps 1–3 for rows 3–4 and 5–6.

Fig. C.8: Process of filling columns 1–4 of a row-form array with operations according to the constraints shown in Figure C.6.

font. The length of an ordered set, $\mathbf{A}$, is denoted by $|\mathbf{A}|$. The ith element of an ordered set, $\mathbf{A}$, is denoted by $\mathbf{A}[i]$. Thus, $\mathbf{A}[1]$ is the first element in $\mathbf{A}$ and $\mathbf{A}[|\mathbf{A}|]$ is the last element in $\mathbf{A}$. If each element in an ordered set, $\mathbf{A}$, is itself an ordered set, then $\mathbf{A}[i][j]$ denotes the jth element of the ith element of $\mathbf{A}$. $\mathbf{A}[i..j]$ denotes the segment of the ordered set, $\mathbf{A}$, containing the ith to the jth elements in $\mathbf{A}$. Similarly, $\mathbf{A}[i..j][k..\ell]$ denotes the rectangular region of the two-dimensional array, $\mathbf{A}$, containing the elements that occur in both the ith to jth rows and the kth to ℓth columns. If $\mathbf{A}$ and $\mathbf{B}$ are two ordered sets, then $\mathbf{A} \oplus \mathbf{B}$ is the concatenation of $\mathbf{A}$ and $\mathbf{B}$—that is, $\mathbf{A} \oplus \mathbf{B} = \langle \mathbf{A}[1], \ldots \mathbf{A}[|\mathbf{A}|], \mathbf{B}[1] \ldots \mathbf{B}[|\mathbf{B}|] \rangle$. If $\langle \mathbf{A}_1, \mathbf{A}_2, \ldots \mathbf{A}_n \rangle$ is a sequence of ordered sets, then

$$\bigoplus_{i=1}^{n} \mathbf{A}_i = \mathbf{A}_1 \oplus \mathbf{A}_2 \oplus \ldots \oplus \mathbf{A}_n \,.$$

In Figure C.9, ComputeOperations begins by initializing an empty 6×8 row-form array called **RFA** (line 1). **RFA** is a 2-dimensional array of nils that will hold an operation at each position. Next, an ordered set, **Operations**, is initialized, containing the four standard row operations, ⟨P,I,R,RI⟩ (line 2). In line 3, three of the four row-form operations are chosen for use in the first column of the row-form array and stored in the variable, **OpsMinusOne**. The decision as to which operation to omit from column 1 of **RFA** is specified by the first element in **Choices**. For example, in Figure C.8, the operation P is omitted from the first column and this would be indicated by setting **Choices**[1] to 1, indicating that the first element in ⟨P,I,R,RI⟩ is to be omitted. In line 4 of ComputeOperations, the Permute2Combinations function is used to compute the unordered 2-combinations of **OpsMinusOne** (sorted in lexicographical order) and find all permutations of these three 2-combinations. These permutations are stored in a 6×3 array, called **AllTwoCombSeqPerms**, in which each row contains one of these permutations and the rows are sorted in reverse lexicographical order. In line 5, a single row (i.e., permutation) in **AllTwoCombSeqPerms** is chosen by **Choices**[2] and saved in a variable called **TwoCombSeq**. For example, if **Choices**[1] $= 1$, as in our previous example, and **Choices**[2] $= 2$, then **TwoCombSeq** would be ⟨{I,RI},{I,R},{R,RI}⟩. ComputeOperations then begins a **while** loop that iterates once for each pair of rows in **RFA** (lines 9 to 19) (that is, the loop iterates 3 times for a six-lyne all-partition array). Inside this loop, i indexes a pair of rows in **RFA** while k indexes a 2-combination in **TwoCombSeq** and m indexes a parameter value in **Choices**. In line 10, the two permutations of the kth unordered pair of operations in **TwoCombSeq** are stored in the variable **Pairs** and, in line 11, the value of **Choices**[m] is used to select one of these ordered pairs of operations for placement at $\mathbf{RFA}[i..i+1][1]$. m is then incremented in line 12. Figure C.8(a) illustrates the process in lines 10–12 for $i = 1$. The algorithm then computes the com-

```
ComputeOperations(Choices)
1   RFA ← An empty 6 × 8 array.
2   Operations ← ⟨P, I, R, RI⟩
3   OpsMinusOne ← Remove(Choices[1], Operations)   ► Choose one operation to remove.
4   AllTwoCombSeqPerms ← Permute2Combinations(OpsMinusOne)
5   TwoCombSeq ← AllTwoCombSeqPerms[Choices[2]]   ► Choose one sequence of three, unordered 2-combinations.
6   k ← 1   ► Index into TwoCombSeq (1-based indexing).
7   m ← 3   ► Index into Choices.
8   i ← 1   ► Indexes first row in current row pair in RFA.
9   while i < |RFA|
10      Pairs ← Permute(TwoCombSeq[k])   ► Permute unordered 2-combination.
11      RFA[i..i + 1][1] ← Pairs[Choices[m]]   ► Step 1 in Figure 8(a).
12      m ← m + 1
13      Pairs ← Permute(SetDiff(Operations, RFA[i..i + 1][1]))
14      RFA[i..i + 1][2] ← Pairs[Choices[m]]   ► Step 2 in Figure 8(b).
15      m ← m + 1
        ► Reverse pairs in columns 1 and 2 to fill columns 3 and 4.
16      for j ← 3 to 4
17          RFA[i..i + 1][j] ← RFA[i..i + 1][j − 2]′   ► Step 3 in Figure 8(c).
18      k ← k + 1
19      i ← i + 2   ► Increment row pair.
    ► Copy columns to fill entire row-form array.
20  for j ← 5 to 8
21      RFA[1..6][j] ← RFA[1..6][(7 − j) mod 4 + 1]
22  return RFA
```

Fig. C.9: Pseudocode for the ComputeOperations function.

plement of **RFA**$[i..i+1][1]$ in **Operations** (line 13), giving the two remaining operations to be placed in column 2 of rows i and $i+1$. Again, the two possible orders for this remaining pair of operations are stored in the variable **Pairs** (line 13). One of these orderings is then chosen in line 14 and placed in the row-form array. The decision as to which of these two orderings to choose is given by the value of **Choices**$[m]$. m is then again incremented in line 15. The process in lines 13–15 is illustrated in Figure C.8(b).

ComputeOperations needs only to choose the operations in columns one and two, as the remaining operations are then determined. For a given pair of rows, columns three and four are the reverse (denoted by $'$) of columns 1 and 2, respectively (see lines 16–17 and Figure C.8(c),(d)). Moreover, columns $5, 6, 7$ and 8 are exact repetitions of columns $3, 2, 1$ and 4, respectively (see lines 20–21 and the complete row-form array in Figure C.1).

5.2 Computing transposition levels for the row operations in the row-form array

The ComputeTranspositions function, shown in Figure C.11, proceeds in a similar manner to ComputeOperations, except that it finds transpositions individually and not in pairs. Figure C.10 shows how the process of assigning transpositions to the operations in the intermediary row-form array found by ComputeOperations unfolds in ComputeTranspositions.

ComputeTranspositions takes three input variables: (1) the array of row-form operations, **RFA**, generated by ComputeOperations; (2) a tone row, **Row**, assumed to be constructed of D-hexachords; and (3) a vector of transposition choices, **Trs**. Each element of **Trs** encodes a selection to be made at some point in the process of assigning transpositions to the operations in

	1	2	3	4
1	RI_6	P	I	R
2	I	R	RI	P
3	I	P	R	RI
4	R	RI	I	P
5	RI	I	R	P
6	R	P	RI	I

(a) Step 1. Place a transposition in **RFA**[1][1].

	1	2	3	4
1	RI_6	P	I_9	R
2	I_6	R	RI_3	P
3	I	P	R	RI
4	R	RI	I	P
5	RI	I	R	P
6	R	P	RI	I

(b) Steps 2 and 3. Place a transposition in **RFA**[2][1] *hc*-related to **RFA**[1][1] and place T_3 and T_9 transpositions of these in **RFA**[1][3] and **RFA**[2][3], respectively.

	1	2	3	4
1	RI_6	P_5	I_9	R
2	I_6	R	RI_3	P
3	I	P	R	RI
4	R	RI	I	P
5	RI	I	R	P
6	R	P	RI	I

(c) Step 4. Place a transposition in **RFA**[1][2], such that the row form shares disjunct hexachords with **RFA**[1][1].

	1	2	3	4
1	RI_6	P_5	I_9	R_8
2	I_6	R_5	RI_3	P_2
3	I	P	R	RI
4	R	RI	I	P
5	RI	I	R	P
6	R	P	RI	I

(d) Steps 5 and 6. Place a transposition in **RFA**[2][2] *hc*-related to **RFA**[1][2] and place T_3 and T_9 transpositions of these in **RFA**[1][4] and **RFA**[2][4], respectively.

	1	2	3	4
1	RI_6	P_5	I_9	R_8
2	I_6	R_5	RI_3	P_2
3	I_5	P_1	R_{10}	RI_2
4	R_7	RI_{11}	I_2	P_{10}
5	RI_7	I_4	R_9	P_6
6	R_6	P_3	RI_4	I_1

(e) Repeat steps 1–6 for remaining rows until complete.

Fig. C.10: Process of assigning transpositions to operations in columns 1–4 in order to complete a row-form array according to the constraints shown in Figure C.6.

```
ComputeTranspositions(RFA, Row, Trs)
1   RowClass ← ComputeRowClass(Row)
2   k ← 1     ► Index into Trs (1-based indexing).
3   i ← 1
4   while i < |RFA|
5      OpTrans ← TranspositionsFromRowClass(RFA[i][1], RowClass)     ► Available transpositions.
6      RFA[i][1].trans ← OpTrans[Trs[k]]     ► Step 1 in Figure 11(a).
7      k ← k + 1
8      OpTrans ← TranspositionsFromRowClass(RFA[i + 1][1], RowClass)
9      HexTrans ← FindHRelation(OpTrans, RFA[i][1], RowClass)     ► Available hc-related transpositions.
10     RFA[i + 1][1].trans ← HexTrans[Trs[k]]     ► Step 2 in Figure 11(b).
11     k ← k + 1
12     if Trs[k] = 1     ► Step 3 in Figure 11(b).
13        RFA[i][3].trans ← (RFA[i + 1][1].trans + 3) mod 12
14        RFA[i + 1][3].trans ← (RFA[i][1].trans + 9) mod 12
15     else     ► Reverse order of transpositions.
16        RFA[i][3].trans ← (RFA[i + 1][1].trans + 9) mod 12
17        RFA[i + 1][3].trans ← (RFA[i][1].trans + 3) mod 12
18     k ← k + 1
19     OpTrans ← TranspositionsFromRowClass(RFA[i][2], RowClass)
20     HexTrans ← FindHexContent(OpTrans, RFA[i][1], RowClass)
21     RFA[i][2].trans ← HexTrans[Trs[k]]     ► Step 4 in Figure 11(c).
22     k ← k + 1
23     OpTrans ← TranspositionsFromRowClass(RFA[i + 1][2], RowClass)
24     HexTrans ← FindHRelation(OpTrans, RFA[i][2], RowClass)
25     RFA[i + 1][2].trans ← HexTrans[Trs[k]]     ► Step 5 in Figure 11(d).
26     k ← k + 1
27     if Trs[k − 3] = 1     ► Step 6 in Figure 11(d).
28        RFA[i][4].trans ← (RFA[i + 1][2].trans + 3) mod 12
29        RFA[i + 1][4].trans ← (RFA[i][2].trans + 9) mod 12
30     else
31        RFA[i][4].trans ← (RFA[i + 1][2].trans + 9) mod 12
32        RFA[i + 1][4].trans ← (RFA[i][2].trans + 3) mod 12
33     i ← i + 2     ► Increment row pair.
    ► Transpose columns according to Figure 6 to complete row-form array.
34  for j ← 5 to 8
35     RFA[1..6][j].trans ← (RFA[1..6][(7 − j) mod 4 + 1].trans + 6) mod 12
36  return GetPcMatrix(RFA)
```

Fig. C.11: The ComputeTranspositions function.

RFA.

In Figure C.11, ComputeTranspositions begins by computing the row class of the tone row, **Row**, provided as input. This row class is stored in the variable, **RowClass** (line 1). The algorithm then begins a **while** loop that iterates once for each pair of adjacent lynes in **RFA** (lines 4–33). Inside this loop, i indexes a pair of lynes in **RFA** while k is an index into **Trs**.

The function, TranspositionsFromRowClass, called in line 5, returns a list of the so-far-unused transposition levels from **RowClass** for the operation given as its first argument. The transposition levels used so far for each row form operation are tracked in **RowClass**. **RowClass** also stores a copy of **Row**. In line 5, TranspositionsFromRowClass finds the available transpositions for the operation at **RFA**$[i][1]$ and saves them in the variable **OpTrans**. This variable also stores the row-form operation to which its list of transpositions apply. **Trs**$[k]$ then specifies the transposition level to be chosen from **OpTrans** and stores it as a property of **RFA**$[i][1]$ in **RFA**$[i][1]$.trans. k is then incremented (lines 6–7, see also Figure C.10(a)).

Next, in line 8, TranspositionsFromRowClass returns a new list of unused transpositions, this time for the operation at **RFA**$[i+1][1]$. In line 9,

FINDHRELATION finds all those row forms whose transpositions are currently in **OpTrans** that are *hc*-related to the row form at $\mathbf{RFA}[i][1]$ (for which the transposition level has just been computed), and stores these in **HexTrans**. In line 10, the transposition indicated by $\mathbf{Trs}[k]$ is selected and stored in $\mathbf{RFA}[i+1][1]$.trans, and then k is incremented in line 11.

In lines 12–17, the value in $\mathbf{Trs}[k]$ is used to select how to transform adjacent rows by T_3 or T_9 according to the row-form array template in Figure C.6. k is then incremented again in line 18. An example of the process in lines 12–18 is shown in Figure C.10(b). Lines 19–22 carry out the process exemplified in Figure C.10(c), that is, selection of a transposition level for the second row form in lyne i. In line 19, the so-far-unused transpositions for the row-form operation at $\mathbf{RFA}[i][2]$ are computed using TRANSPOSITIONS-FROMROWCLASS. In line 20, the FINDHEXCONTENT function returns a list of possible transpositions for $\mathbf{RFA}[i][2]$ (whose operation is stored with the so-far-unused transpositions in **OpTrans**) that allow the resulting row form to share disjunct hexachords with the row form at $\mathbf{RFA}[i][1]$—recall that all the rows in a single lyne must share disjunct hexachords. In line 21, one of these allowable transpositions is selected and stored in $\mathbf{RFA}[i][2]$.trans. In lines 23–25, the algorithm uses the same procedure as in lines 8–10 to determine a transposition level for the second row form in lyne $i+1$ that is *hc*-related to the second row form in lyne i (see Figure C.10(d)). In lines 27–32, the algorithm then uses the same order of adjacent row transpositions (decided in lines 12–17) to determine the transposition levels of the fourth row forms in lynes i and $i+1$ (see Figure C.10(d)).

After performing all the steps in the main **while** loop for each of the three adjacent lyne pairs, the transposition levels for the first four row forms in each lyne have been computed. The algorithm then places transpositions of these levels in columns 5–8 according to Figure C.6 (lines 34–35) and return the **PcMatrix** representation of the row-form array (line 36).

6 Generating the WPcMatrix

Generating the WPcMatrix of a Smalley array class from a $\text{PcMatrix}_{6,96}$ requires finding a sequence of 58 integer compositions that satisfy certain constraints. If we imagine determining these integer compositions in a left-to-right manner, then we can define a *candidate composition* to be one that

1. forms an ordered mosaic in the PcMatrix containing a complete aggregate or a *sufficiently complete aggregate* that can be made complete with some combination of outer-aggregate repeated pcs (OARPs); and

2. is partitionally distinct from all other compositions found in the sequence before it.

11 4 3 | 5 9 10 ¦ 1 8 2 0 7 6
6 7 | 0 2 8 ¦ 1 10 9 5 3 4 11
5 | 6 11 1 ¦ 7 0 9 8 4 2 3 10
2 9 10 | 8 4 3 ¦ 0 5 11 1 6 7
0 | 5 4 6 10 11 2 9 3 1 8 7
1 8 | 9 7 3 2 11 4 10 0 5 6

Fig. C.12: IntComp$(3,2,1,3,1,2)$ (indicated by the ragged solid line boundary), forming an ordered mosaic containing a complete aggregate, followed by WIntComp$(3,3,3,3,0,0)$ (indicated by the dashed line), forming an ordered mosaic containing a sufficiently complete aggregate, in an excerpt from a PcMatrix. Note that the sufficiently complete aggregate is missing pitch class 7 and has two occurrences of pitch class 8.

We define a *sufficiently complete aggregate* as an incomplete aggregate in which the number of missing pitch classes is less than or equal to the number of lynes in the array (in this case, six). As each ordered mosaic formed by a candidate composition must contain an aggregate (i.e., 12 distinct pitch classes), a sufficiently complete aggregate missing n pcs will need to contain n OARPs. Figure C.12 shows an example of a pair of ordered mosaics in a PcMatrix, one containing a complete aggregate and one containing a sufficiently complete aggregate.

In Figure C.12, the second composition forms an ordered mosaic containing an incomplete aggregate, as it is missing pitch class 7 and contains a duplicate pitch class 8. We speak generally of the boundary between two ordered mosaics as a *mosaic boundary* and, more specifically, of that part of a mosaic boundary in a particular row of the PcMatrix as a *segment boundary*. Thus, the two ordered mosaics in Figure C.12 form a mosaic boundary with four segment boundaries (in rows 1–4).

When describing the process by which OARPs are generated, we use the term *segment boundary pc* to refer to a pitch class lying directly to the *left* of a segment boundary. Segment boundary pcs residing in the ordered mosaic formed by the *current composition* (to the left of the dashed boundary line in Figure C.12) we call *potential pushed pcs*, while those found in the previous composition (to the left of the solid boundary line) we call *potential repeated pcs*. In Figure C.12, the potential pushed pcs are 10, 8, 1 and 3, while the potential repeated pcs are 3, 7, 5 and 10. Only by repeating a potential repeated pc across a segment boundary to the first position of the current segment and thereby “pushing" the potential pushed pc from this segment (and into the next ordered mosaic), can one complete a sufficiently complete aggregate. As noted by Mead (1994), this practice allowed Babbitt to preserve the order of pitch classes in each tone row. Figure C.13(a) and (b) show two possible solutions to completing the sufficiently complete aggregate formed by the

11 4 3 5 9 10 1 8 2 0 7 6

6 7 **7** 0 2 8 1 10 9 5 3 4 11

5 6 11 1 7 0 9 8 4 2 3 10

2 9 10 8 4 3 0 5 11 1 6 7

0 5 4 6 10 11 2 9 3 1 8 7

1 8 9 7 3 2 11 4 10 0 5 6

(a)

11 4 3 **3** 5 9 10 1 8 2 0 7 6

6 7 **7** 0 2 8 1 10 9 5 3 4 11

5 6 11 1 7 0 9 8 4 2 3 10

2 9 10 **10** 8 4 3 0 5 11 1 6 7

0 5 4 6 10 11 2 9 3 1 8 7

1 8 9 7 3 2 11 4 10 0 5 6

(b)

11 4 3 5 9 10 1 8 2 0 7 6

6 7 0 2 8 1 10 9 5 3 4 11

5 6 11 1 7 0 9 8 4 2 3 10

2 9 10 8 4 3 0 5 11 1 6 7

0 5 4 6 10 11 2 9 3 1 8 7

1 8 9 7 3 2 11 4 10 0 5 6

(c)

Fig. C.13: (a) Simple and (b) complex solutions (in bold) to completing the sufficiently complete aggregate in the region formed by the candidate composition, WIntComp$(3,3,3,3,0,0)$, shown in Figure C.12. (c) No solution exists for the sufficiently complete aggregate in the region formed by the composition, WIntComp$(3,3,0,3,3,0)$. This composition is therefore not a candidate composition.

current composition in Figure C.12. Figure C.13(c) provides an example of a sufficiently complete aggregate that does not have a solution.

In Figure C.13(a), pitch class 7 (row 2) from the previous ordered mosaic is repeated across its segment boundary, subsequently “pushing" the last pitch class of the candidate segment out of consideration (pitch class 8). In this way we have found the most simple solution to completing this sufficiently complete aggregate. There are, however, two solutions for this candidate composition and (b) shows the second and slightly more complex of these. By first repeating pitch class 3 (row 1) across its segment boundary, 7 and 10 are now missing while 3 and 8 become duplicates. By then repeating pitch class 7 (row 2), only pitch class 10 remains missing and pitch class 3 remains a duplicate as 8 has been pushed. We can complete the aggregate by repeating the last remaining missing pitch class 10 and pushing the last remaining duplicate, pitch class 3 (row 4). By contrast, Figure C.13(c) shows a non-candidate composition (representing the same integer partition) whose sufficiently complete aggregate cannot be made complete.

We should note that an integer composition, whether it forms a sufficiently complete aggregate or not, defines a region similar to what Bazelow

and Brickle (1979) have called a *segmented block*. Further, a *proper block* is a segmented block that contains a complete aggregate. A proper block thus qualifies as a candidate composition. However, not all candidate compositions are proper blocks, since neither sufficiently complete aggregates nor ordered mosaics containing non-contiguous segments qualify as proper blocks (Bazelow and Brickle, 1979, p. 55). The weak integer composition, $\mathrm{WIntComp}(3,3,0,3,3,0)$, shown in Figure C.13(c) is an example of an ordered mosaic containing non-contiguous segments that does not constitute a proper block.

6.1 A backtracking algorithm for computing the sequence of integer compositions

If one attempts to determine a sequence of integer compositions for an all-partition array in an incremental fashion (i.e., “from left to right" as suggested by Morris (2010, p. 74)), then it is possible that not every remaining unused integer composition at a given stage in the process will be a candidate composition (as illustrated by the example in Figure C.13(c)). Moreover, it is possible that, at some point in the sequence, there will be no candidate compositions at all. In fact, if a left-to-right, incremental approach is adopted, both of these situations turn out to be highly likely, making it difficult to generate an entire sequence. To overcome this difficulty, we propose a solution involving a backtracking algorithm.

Backtracking algorithms are a depth-first way of searching for all possible solutions to a set of constraints. The backtracking process finds a complete solution to a problem by accumulating partial solutions to a set of constraints. It selects the first of these partial solutions until a complete solution is found; or, in the event that the constraints cannot be satisfied by the currently selected partial solution, it returns to a previous point and selects the next possible partial solution. It continues this process until either a solution is found or it fails. Figure C.14 shows pseudocode for an algorithm, called BacktrackingBabbitt, for generating a WPcMatrix from a PcMatrix by adding OARPs.

The algorithm begins by computing a list of integer compositions (when $n = 12$ and $k = 6$), stored in the variable named **Compositions** (line 1). **Compositions** is a 6188×6 array of integers, in which each row represents a distinct integer composition. Line 2 initializes **CList** to be a list of 58 empty lists that will hold all candidate compositions and their OARP solutions at each point in the sequence. Line 3 initializes **PartialSolutions** to be a 58×6 array that will be used to hold both the final sequence of integer compositions computed and the partial solution at each stage during the algorithm's execution. Each row in **PartialSolutions** stores a single integer composition as a sequence of 6 summands. *Cnt*, used to store the current position being

```
BacktrackingBabbitt(PcMatrix)
1   Compositions ← ComputeCompositions(12, 6)   ▶ Integer compositions of 12 into 6 or fewer parts.
2   CList ← ⊕_{i=1}^{58} ⟨⟨⟩⟩
3   PartialSolutions ← ⊕_{i=1}^{58} ⟨⊕_{j=1}^{6} ⟨nil⟩⟩
4   Cnt ← 1   ▶ Index into CList (1-based indexing).
5   Position ← ⊕_{i=1}^{6} ⟨1⟩   ▶ Vector giving current starting position of current segment in each row.
6   C ← ⊕_{i=1}^{58} ⟨0⟩   ▶ Vector of indices into elements in CList.
7   R ← ⊕_{i=1}^{58} ⟨0⟩   ▶ Vector of indices into list of OARP solutions in each element in CList.
8   while Cnt ≤ 58 and Cnt > 0
9     if CList[Cnt] = ⟨⟩   ▶ ComputeCandidateCompositions has not been called here before.
10      ⟨CList, C, R⟩ ← ComputeCandidateCompositions(PcMatrix, C, R, Compositions, Cnt, Position, CList, PartialSolutions)
11      if CList[Cnt] = ⟨⟩   ▶ There are no more candidate compositions.
          ▶ Failure.
12        ⟨PcMatrix, Cnt, C, R, Position, PartialSolutions⟩ ← Backtrack(PcMatrix, CList, Cnt, C, R, Position, PartialSolutions)
13      else   ▶ There is at least one candidate composition.
          ▶ Success.
14        ⟨PcMatrix, Cnt, Position, PartialSolutions⟩ ← Advance(PcMatrix, CList, Cnt, C, R, Position, PartialSolutions)
15    else   ▶ The algorithm has backtracked to this position.
16      R[Cnt] ← R[Cnt] + 1   ▶ Select next OARP solution for current composition.
17      if R[Cnt] > |CList[Cnt][C[Cnt]].oarps|   ▶ There are no more OARP solutions for this candidate composition.
18        C[Cnt] ← C[Cnt] + 1   ▶ Select next candidate composition.
19        if C[Cnt] > |CList[Cnt]|   ▶ There are no more candidate compositions for this position in the sequence.
            ▶ Failure.
20          ⟨PcMatrix, Cnt, C, R, Position, PartialSolutions⟩ ← Backtrack(PcMatrix, CList, Cnt, C, R, Position, PartialSolutions)
21        else   ▶ There is another candidate composition.
            ▶ Success.
22          if CList[Cnt][C[Cnt]].oarps = ⟨⟩   ▶ Current composition forms a complete aggregate.
23            R[Cnt] ← 0
24          else   ▶ Current composition forms a sufficiently complete aggregate.
25            R[Cnt] ← 1   ▶ Select first OARP solution for current composition.
26          ⟨PcMatrix, Cnt, Position, PartialSolutions⟩ ← Advance(PcMatrix, CList, Cnt, C, R, Position, PartialSolutions)
27      else   ▶ There is an OARP solution.
          ▶ Success.
28        ⟨PcMatrix, Cnt, Position, PartialSolutions⟩ ← Advance(PcMatrix, CList, Cnt, C, R, Position, PartialSolutions)
29  WPcMatrix ← PcMatrix
30  return ⟨WPcMatrix, PartialSolutions⟩
```

Fig. C.14: The BacktrackingBabbitt algorithm.

processed in **CList**, is initialized to 1 (using 1-based indexing) in line 4. Line 5 initializes **Position** to be a 6-vector that will hold the starting position in each row in **WPcMatrix** of the corresponding segment in the ordered mosaic for the composition at **PartialSolutions**[*Cnt*]. Lines 6–7 initialize **C** and **R** to be 58-vectors in which the entries index candidate compositions in **CList**[*Cnt*] and OARP solutions in **CList**[*Cnt*][**C**[*Cnt*]].oarps, respectively.

On each iteration of the **while** loop that starts in line 8, the algorithm first checks if a list of candidate compositions has yet been computed for the current position, *Cnt*. If this has not been done, then **CList**[*Cnt*] will be empty in line 9 and ComputeCandidateCompositions will be called in line 10, which returns **CList**, **C** and **R**. **C**[*Cnt*] is initialized to 1 by ComputeCandidateCompositions while **R**[*Cnt*] is initialized to 1 if the composition at **CList**[*Cnt*][1] forms a sufficiently complete aggregate (otherwise, **R**[*Cnt*] is initialized to 0). If, after ComputeCandidateCompositions has executed, **CList**[*Cnt*] is empty in line 11, then the algorithm must backtrack by calling Backtrack (line 12). If, on the other hand, there is at least one candidate composition in **CList**[*Cnt*], then the algorithm can advance to the next position in the sequence by calling Advance (line 14).

If the list of compositions at the current position in **CList** is not empty in line 9, then this implies that the algorithm has backtracked at least once to this position from some later position. In this case, the algorithm needs

```
ADVANCE(PcMatrix, CList, Cnt, C, R, Position, PartialSolutions)
1   ADDOARPS(PcMatrix, CList, Cnt, C, R, Position)
2   PartialSolutions[Cnt] ← CList[Cnt][C[Cnt]]
3   Position ← Position + CList[Cnt][C[Cnt]]
4   Cnt ← Cnt + 1
5   return ⟨PcMatrix, Cnt, Position, PartialSolutions⟩
```

(a) ADVANCE function called in the event that a candidate composition is found.

```
BACKTRACK(PcMatrix, CList, Cnt, C, R, Position, PartialSolutions)
1   Cnt ← Cnt − 1
2   Position ← Position − CList[Cnt][C[Cnt]]
3   PartialSolutions[Cnt] ← ⊕_{i=1}^{6} ⟨nil⟩
4   R[Cnt] ← 0
5   C[Cnt] ← 0
6   REMOVEOARPS(PcMatrix, Position)
7   return ⟨PcMatrix, Cnt, C, R, Position, PartialSolutions⟩
```

(b) BACKTRACK function called in the event that no candidate compositions are found.

Fig. C.15: The (a) ADVANCE and (b) BACKTRACK functions called by BACKTRACKINGBABBITT.

to first try any remaining untried OARP solution for the current candidate composition. $\mathbf{R}[Cnt]$ is therefore incremented in line 16 and line 17 checks if we have run out of OARP solutions for this candidate composition. If there is still a possible OARP solution, then the algorithm advances in line 28. If no remaining OARP solution exists for the current candidate composition at this position, then the next candidate composition for this position (if there is one) must be tried. This is done in lines 18–26. If there is no remaining untried candidate composition for this position (i.e., $\mathbf{C}[Cnt] > |\mathbf{CList}[Cnt]|$ in line 19), then the algorithm backtracks in line 20. Otherwise, the algorithm must check in line 22 whether the new composition is complete or sufficiently complete. The list of OARP solutions stored in the oarps property of the current candidate composition, $\mathbf{CList}[Cnt][\mathbf{C}[Cnt]]$, will only be empty in line 22 if this candidate composition is complete. If the current composition is complete, then $\mathbf{R}[Cnt]$ is set to 0 in line 23, otherwise it is set to index the first possible OARP solution in line 25. Either way, the algorithm then advances in line 26.

The **while** loop continues to iterate until a candidate composition has been chosen for each possible position from 1 to 58. At this point, a complete sequence of integer compositions will be stored in **PartialSolutions** and all the necessary OARPs will have been inserted into the PcMatrix, meaning that it has been transformed into a WPcMatrix. The algorithm terminates by returning both the sequence of integer compositions in **PartialSolutions** and the WPcMatrix in the variable **WPcMatrix**.

Figure C.15 shows pseudocode for the ADVANCE and BACKTRACK functions called by the BACKTRACKINGBABBITT algorithm. As shown in Figure C.15(a), ADVANCE begins by adding OARPs to **PcMatrix**, if necessary

(line 1). It then adds the candidate composition at **CList**[*Cnt*][**C**[*Cnt*]] to **PartialSolutions**[*Cnt*] and increments **Position** by the values in this composition (lines 2–3). At any given *Cnt*, **Position** is equivalent to counting from 1 a distance equal to the summation of like parts from each composition in **PartialSolutions** from 1 to $Cnt - 1$. Finally, it increments *Cnt* (line 4) and returns a sequence containing the new values for **PcMatrix**, *Cnt*, **Position** and **PartialSolutions** (line 5). The BACKTRACK function in Figure C.15(b) begins by decrementing *Cnt* by 1 and then **Position** by the current composition, **CList**[*Cnt*][**C**[*Cnt*]] (lines 1–2). Next, it removes the composition at **PartialSolutions**[*Cnt*] and resets **R**[*Cnt*] and **C**[*Cnt*] to zero (lines 3–5). Finally, it removes OARPs from **PcMatrix**, if necessary (line 6), and then returns a sequence containing the new values of **PcMatrix**, *Cnt*, **C**, **R**, **Position** and **PartialSolutions**.

6.2 Computing candidate compositions and OARPs

BACKTRACKINGBABBITT relies on two functions, COMPUTECANDIDATECOMPOSITIONS and COMPUTEOARPs, for computing candidate compositions and their associated OARPs (if the aggregates corresponding to the integer compositions are sufficiently complete). Pseudocode for these two functions is given in Figures C.16 and C.17. COMPUTECANDIDATECOMPOSITIONS finds all integer compositions that are partitionally distinct from those that have been previously used and that form either complete or sufficiently complete aggregates in **PcMatrix** at the row positions encoded in **Position**. COMPUTEOARPs then takes each integer composition computed by COMPUTECANDIDATECOMPOSITIONS that forms a sufficiently complete aggregate and determines whether or not it can be made complete with OARPs.

The COMPUTECANDIDATECOMPOSITIONS algorithm, shown in Figure C.16, begins by making a list, **AllUnusedComps**, of all the integer compositions that are partitionally distinct from ones that have been used so far (line 1). **AllUnusedComps** is an $n \times 6$ matrix in which each row represents an integer composition. In line 2, the variable, k, which will be used to track the number of candidate compositions discovered, is initialized to 0. The **for** loop that starts in line 3 then iterates over all the integer compositions in **AllUnusedComps** and adds each composition that could potentially work at the current position to the list of candidate compositions stored in **CList**[*Cnt*]. For each composition in **AllUnusedComps**, the algorithm first makes a list, **MosaicPcs**, of the pcs that would occur in the ordered mosaic defined by that composition (lines 4–12). The algorithm also notes the potential repeated and pushed PCs, and stores these in **PotRepPCs** and **PotPushPCs**, respectively. The aggregate region defined by an integer composition may contain duplicate pcs. In line 13, the algorithm therefore computes the set of pcs (i.e., without duplicates) in the ordered mosaic and stores this set in *MosaicPcsNoDups*.

```
ComputeCandidateCompositions(PcMatrix, C, R, Compositions, Cnt, Position, CList, PartialSolutions)
1   AllUnusedComps ← RemoveUsedCompositions(PartialSolutions, Compositions)
2   k ← 0      ▶ Stores the number of candidate compositions discovered for this position.
3   for i ← 1 to |AllUnusedComps|
4       MosaicPcs ← ⟨⟩      ▶ Stores list of pcs in this ordered mosaic (possibly with duplicates).
5       PotRepPCs ← ⊕_{i=1}^{6} ⟨nil⟩      ▶ Used to store potential repeated pcs.
6       PotPushPCs ← ⊕_{i=1}^{6} ⟨nil⟩      ▶ Used to store potential pushed pcs.
7       for j ← 1 to 6      ▶ j indexes a column in AllUnusedComps.
8           if AllUnusedComps[i][j] ≠ 0      ▶ There is a segment in the jth lyne.
                ▶ Add jth segment to aggregate region.
9               MosaicPcs ← MosaicPcs ⊕ PcMatrix[j][Position[j]..Position[j] + AllUnusedComps[i][j] − 1]
10              if Position[j] > 1      ▶ There will be potential repeated pcs.
11                  PotRepPCs[j] ← PcMatrix[j][Position[j]]
12                  PotPushPCs[j] ← PcMatrix[j][Position + AllUnusedComps[i][j] − 1]
13      MosaicPcsNoDups ← RemoveDuplicates(MosaicPcs)      ▶ The set of distinct pcs in this ordered mosaic.
14      if |MosaicPcsNoDups| = 12      ▶ Ordered mosaic contains a complete aggregate.
            ▶ Found a candidate composition.
15          CList[Cnt] ← CList[Cnt] ⊕ ⟨AllUnusedComps[i][1..6]⟩
16          k ← k + 1
17      else if |MosaicPcsNoDups| ≥ 6 ∧ Cnt > 1      ▶ Sufficiently complete aggregate.
18          OARPs ← ComputeOARPs(MosaicPcs, MosaicPcsNoDups, PotRepPCs, PotPushPCs)
19          if OARPs ≠ ⟨⟩      ▶ Ordered mosaic has at least one OARP solution.
                ▶ Found a candidate composition.
20              CList[Cnt] ← CList[Cnt] ⊕ ⟨AllUnusedComps[i][1..6]⟩
21              k ← k + 1
22              CList[Cnt][k].oarps ← OARPs
23  if |CList[Cnt]| > 0      ▶ Candidate compositions were found for this position.
24      ⟨CList⟩ ← RankCandidates(CList, C, Cnt)      ▶ Uses heuristics to rank the found compositions.
25      C[Cnt] ← 1      ▶ C[Cnt] now indexes first composition in CList[Cnt].
26      if CList[Cnt][1].oarps ≠ ⟨⟩      ▶ First composition in CList has an OARP solution.
27          R[Cnt] ← 1      ▶ R[Cnt] set to index first OARP solution for first composition.
28      else
29          R[Cnt] ← 0      ▶ R[Cnt] set to 0, indicating this is a complete aggregate.
30  else
31      C[Cnt] ← 0      ▶ No candidate compositions.
32  return ⟨CList, C, R⟩
```

Fig. C.16: The ComputeCandidateCompositions algorithm.

If the current integer composition generates a complete aggregate, then it is added to **CList**[*Cnt*] and *k* is incremented (lines 14–16). Alternatively, if the set of distinct pcs in the ordered mosaic is potentially large enough to be completed by the addition of OARPs, the function ComputeOARPs is called in line 18 to compute all the possible OARP solutions for this sufficiently complete aggregate. If at least one such OARP solution exists, the integer composition is added to **CList**[*Cnt*], *k* is incremented and the oarps property of the added composition is set to contain the possible OARP solutions (lines 19–22).

Once the **for** loop has completed and all the integer compositions in **AllUnusedComps** have been checked, the algorithm ranks the discovered candidate compositions according to heuristics that will be described in section 6.3 (line 24). The values of **C**[*Cnt*] and **R**[*Cnt*] are then initialized to appropriate values (lines 25–29 and 31) and the algorithm returns the updated values of **CList**, **C** and **R** (line 32).

Figure C.17 shows pseudocode for the ComputeOARPs function, used in line 18 of ComputeCandidateCompositions to compute all possible OARP solutions that allow for a sufficiently complete aggregate to be completed.

```
COMPUTEOARPS(MosaicPcs, MosaicPcsNoDups, PotRepPCs, PotPushPCs)
1   OARPs ← ⟨⟩      ▶ Stores list of OARP solutions.
2   MissingPcs ← COMPUTEMISSINGPCS(MosaicPcsNoDups)      ▶ Missing pcs from this ordered mosaic.
3   DuplicatePcs ← COMPUTEDUPLICATEPCS(MosaicPcs)      ▶ Duplicate pcs in this ordered mosaic.
    ▶ Compute all k-combinations of segment boundary pcs.
4   ⟨PotRepCombs, PotPushCombs⟩ ← COMPUTEKCOMBINATIONS(PotRepPCs, PotPushPCs)
5   for i ← 1 to |PotRepCombs|      ▶ i indexes rows in PostRepCombs and PostPushCombs.
6      MissingPcsCopy ← SORT(MissingPcs ⊕ PotPushCombs[i])
7      DuplicatePcsCopy ← SORT(DuplicatePcs ⊕ PotRepCombs[i])
8      if MissingPcsCopy = DuplicatePcsCopy      ▶ OARP solution found.
9         OARPs ← OARPs ⊕ ⟨PotRepCombs[i]⟩
10  return OARPs
```

Fig. C.17: The ComputeOARPs function.

The first step in this function is to initialize an empty list, **OARPs**, that will contain the OARP solutions discovered (line 1). Line 2 computes the set of missing pitch classes, **MissingPcs**, from *MosaicPcsNoDups*. Line 3 computes the set of duplicate pitch classes, **DuplicatePcs**, from **MosaicPcs**. The function ComputeKCombinations is called in line 4 and computes all unordered *k*-combinations from **PotRepPCs** and **PotPushPCs** taken from 1 to the number of pitch classes found in each at a time. It saves those *k*-combinations from **PotRepPCs** and **PotPushPCs** not containing any nil in two lexicographically ordered lists, **PotRepCombs** and **PotPushCombs**, respectively. The **for** loop that begins in line 5 then iterates over all the combinations in **PotRepCombs**. For an OARP solution to be found, two conditions must be met:

1. all pitch classes that are missing (either from the ordered mosaic to begin with or those from **PotPushCombs**$[i]$ that are pushed out) are found or replaced when all pitch classes in **PotRepCombs**$[i]$ are repeated; and

2. all pitch classes that are duplicates (either in the ordered mosaic to begin with or those from **PotRepCombs**$[i]$ that are repeated) are removed when all pitch classes in **PotPushCombs**$[i]$ are pushed.

We test for these conditions on each iteration by first placing the combination at **PotRepCombs**$[i]$ into a copy of **MissingPcs** called **MissingPcsCopy** and its corresponding combination at **PotPushCombs**$[i]$ into a copy of **DuplicatePcs** called **DuplicatePcsCopy** (lines 6–7).[13] If **DuplicatePcsCopy** and **MissingPcsCopy** are equal in line 8, then **PotRepCombs**$[i]$ is an OARP solution and **PotRepCombs**$[i]$ is saved in **OARPs** in line 9. Upon completion of the loop, **OARPs** is returned (line 10).

[13]Note that **MissingPcsCopy** and **DuplicatePcsCopy** are sorted ordered sets of pcs that represent multisets.

6.3 Ranking candidate compositions

As mentioned briefly above, all candidate compositions found by ComputeCandidateCompositions are ranked in line 24 by calling the RankCandidates function. RankCandidates uses two heuristics for determining the quality of a candidate that we call *zero-gain segments* and *equal-lyne-length*. Candidates are then sorted from best to worst according to the quality of each, and this new order is returned by RankCandidates.

6.3.1 Equal-lyne-length heuristic

Our *equal-lyne-length* heuristic is founded on the hypothesis that sequences of candidate compositions found in a Smalley array (i.e., those that are self-contained) are more likely to contain compositions that collectively result in a WPcMatrix with lynes of approximately equal length. We suggest that a straight-forward way to achieve this is to ensure that compositions *progress* at approximately equal rates in each row. Let's suppose we have a list of candidate compositions, $C_k = \langle c_1, c_2, ...c_n \rangle$, at position k, where $1 \leq k \leq \ell$. Ideally, if all lynes of the WPcMatrix have progressed at the same rate, then after choosing a composition for position k, the lengths of each lyne up to and including position k would be $12k/n$, where n is the number of lynes. Let's suppose that the actual length of a lyne j would be $l_{i,j}$ after choosing c_i from C_k. To measure the *raggedness* or degree of inequality of lyne length, $D_{i,k}$, that results from choosing c_i at position k, we use the following formula, based on city-block distance:

$$D_{i,k} = \sum_{j=1}^{n} \left| l_{i,j} - \frac{12k}{n} \right| \tag{C.1}$$

where $|x|$ denotes the absolute value of x. We adopt a greedy strategy in which, for each k, we choose the c_i that minimizes the raggedness, $D_{i,k}$.

Returning now to the compositions shown in Figure C.2, the raggedness after choosing (a) is 4, while the raggedness after choosing (b) is 14. According to our heuristic, the composition in (a) is thus better than the composition in (b). That is, this composition contributes to producing lynes of more similar length at this position than the other composition. This is, indeed, also the composition found at this position in the sequences underlying all of Babbitt's works based on a Smalley array.

6.3.2 Zero-gain segments heuristic

Using the equal-lyne-length heuristic in the manner just described, it is possible that two compositions will result in the same distance, $D_{i,k}$, defined in Eq. C.1. This does not mean, however, that each of their lynes will have the

same length. We thus propose a second heuristic for further discriminating between compositions, which we call *zero-gain segments*. This heuristic judges a composition better than another based on the qualities it shares with the composition that immediately precedes it in the sequence. Let's suppose we have two consecutive integer compositions, A and B, where $A = \langle s_1, s_2, ..., s_k \rangle$ and $B = \langle t_1, t_2, ..., t_k \rangle$ and B follows A. The *weight*, w, of B is given by $w = \sum_{i=1}^{k} r_i$, where

$$r_i = \begin{cases} 1, \text{if } (s_i = 0 \veebar t_i = 0); \text{ and} \\ 0, \text{otherwise.} \end{cases} \tag{C.2}$$

This weight equals the number of instances where there is a zero summand in the current composition (i.e., B in this example) and a non-zero summand at the corresponding position in the preceding composition (or vice versa). Suppose, for example, that the composition WIntComp(3,3,3,3,0,0) is followed by the composition WIntComp(0,0,0,0,6,6). According to the zero-gain segments heuristic, the latter composition has a weight of 6. Unlike the equal-lyne-length heuristic, the higher the weight found by zero-gain segments heuristic, the better. In our implementation of these heuristics within the RankCandidates function, we obtain a combined weight, $W = D - w$, by subtracting the zero-gains-segment weight, w, from the raggedness, D, that results from choosing a particular composition. At each step in the process of choosing compositions, we then adopt the greedy strategy of minimizing this combined weight, W.

7 Parameter values for generating the WPcMatrices of three of Babbitt's works

As presented above, BacktrackingBabbitt will perform a depth-first search for the first successful sequence of candidate compositions for a given $\mathbf{PcMatrix}_{6,96}$ returned by ComputeOperations and ComputeTranspositions. By providing the correct parameter values for **Choices** and **Trs**, ComputeOperations and ComputeTranspositions will generate the PcMatrix belonging to a specific Babbitt work based on a Smalley array. Likewise, by further providing the correct parameter values for **C** and **R**, BacktrackingBabbitt will generate the WPcMatrix of this PcMatrix's corresponding all-partition array. Table C.1 shows parameter values for all input variables necessary for using our method to generate the all-partition arrays underlying three of Babbitt's works, *Sheer Pluck* (1984), *Joy of More Sextets* (1986) and *None but the Lonely Flute* (1991).

Note in Table C.1 that all three pieces share very closely related parameter values for both **C** and **R**. This is because both *Sheer Pluck* and *Joy of More*

	Sheer Pluck
Row	⟨0, 11, 6, 4, 10, 5, 8, 9, 1, 3, 2, 7⟩
Choices	⟨1, 4, 1, 2, 2, 2, 1, 1⟩
Trs	⟨7, 2, 1, 1, 1, 4, 2, 1, 1, 2, 3, 2, 1, 1, 1⟩
C	⟨0, 61, 55, 7, 15, 28, 20, 21, 46, 185, 75, 5, 9, 21, 4, 28, 0, 70, 1, 79, 35, 38, 29, 45, 68, 31, 32, 24, 12, 0, 65, 74, 58, 13, 6, 7, 110, 24, 59, 12, 7, 7, 12, 20, 33, 0, 1, 2, 32, 3, 12, 5, 23, 1, 20, 3, 6, 0⟩
R	⟨0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0⟩
	Joy of More Sextets
Row	⟨0, 11, 6, 4, 10, 5, 8, 9, 1, 3, 2, 7⟩
Choices	⟨1, 2, 2, 2, 1, 1, 2, 1⟩
Trs	⟨8, 2, 2, 1, 1, 6, 2, 2, 1, 1, 3, 2, 2, 2, 2⟩
C	⟨0, 61, 54, 7, 15, 28, 20, 20, 45, 183, 75, 5, 9, 21, 4, 28, 0, 70, 1, 80, 34, 39, 29, 45, 69, 31, 31, 25, 12, 1, 65, 73, 58, 13, 6, 6, 108, 24, 58, 12, 8, 9, 12, 20, 33, 0, 0, 2, 31, 3, 12, 6, 21, 1, 20, 2, 6, 0⟩
R	⟨0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0⟩
	None but the Lonely Flute
Row	⟨0, 11, 6, 4, 10, 5, 8, 9, 1, 3, 2, 7⟩
Choices	⟨1, 2, 2, 2, 2, 2, 1, 2⟩
Trs	⟨1, 1, 2, 2, 2, 8, 1, 1, 2, 1, 1, 1, 1, 2, 2⟩
C	⟨0, 61, 53, 7, 19, 21, 7, 21, 69, 189, 71, 5, 9, 20, 4, 28, 0, 67, 1, 82, 33, 36, 29, 40, 50, 44, 40, 28, 14, 1, 59, 56, 48, 12, 6, 8, 23, 15, 75, 12, 7, 8, 11, 16, 30, 1, 1, 1, 10, 1, 1, 6, 21, 1, 20, 2, 6, 0⟩
R	⟨0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0⟩

Table C.1: Input variables and their parameter values for generating all-partition arrays of *Sheer Pluck* (1984), *Joy of More Sextets* (1986) and *None but the Lonely Flute* (1991).

Sextets use the same sequence of integer partitions while *None but the Lonely Flute* differs from these by only two partitions. Although the three pieces are based on the same tone row, each uses a different sequence of integer compositions. This is because their PcMatrices differ, as indicated by their parameter values for **Choices** and **Trs**.[14]

Given the parameter values for *Sheer Pluck* in Table C.1(a) as input, BacktrackingBabbitt will return the WPcMatrix shown in Figure C.3 and the sequence of compositions shown in Figure C.18.

8 Evaluation

In this section, we evaluate our model by measuring how well it performs on two tasks: (1) generating WPcMatrices found in works by Babbitt from their PcMatrices; and (2) generating novel WPcMatrices from PcMatrices used in Babbitt's works. In task (1), we measured how well the equal-lyne-length and zero-gain segments heuristics predicted the sequence of integer compositions chosen by Babbitt. In task (2), we analysed the frequencies with which backtracks occur at the different positions in the sequence during the generation of a new array, with and without the proposed heuristics.

[14] *None but the Lonely Flute* has been analysed by Bernstein (2014) and Leong and McNutt (2005). *Joy of More Sextets* has been analysed by Mead (1994, p. 270) and the analysis of *Sheer Pluck* is our own. Note that there is an error in the OARPs in the 22nd aggregate in Mead's analysis, which was discovered using our algorithm.

1)	3	2	1	3	1	2	30)	6	6	0	0	0	0
2)	3	3	3	3	0	0	31)	4	4	0	2	1	1
3)	2	0	0	0	4	6	32)	2	1	2	2	2	3
4)	0	6	2	1	1	2	33)	0	1	0	0	11	0
5)	5	3	1	0	3	0	34)	1	2	2	5	0	2
6)	0	0	4	7	0	1	35)	0	1	9	2	0	0
7)	2	2	0	0	6	2	36)	1	1	0	1	1	8
8)	0	2	4	4	0	2	37)	0	4	0	3	4	1
9)	0	2	1	0	6	3	38)	1	0	1	7	3	0
10)	2	0	1	4	0	5	39)	0	0	0	0	0	12
11)	0	7	3	0	0	2	40)	8	0	2	1	1	0
12)	0	0	7	5	0	0	41)	5	5	1	1	0	0
13)	8	0	2	0	2	0	42)	1	1	7	2	1	0
14)	3	0	0	0	6	3	43)	1	0	0	0	10	1
15)	3	9	0	0	0	0	44)	0	1	1	9	0	1
16)	1	0	5	6	0	0	45)	0	0	0	4	4	4
17)	2	0	2	0	7	1	46)	5	4	0	3	0	0
18)	3	5	1	1	1	1	47)	0	1	3	1	1	6
19)	0	0	3	1	0	8	48)	5	5	2	0	0	0
20)	5	3	0	2	0	2	49)	1	3	0	1	5	2
21)	0	2	2	2	4	2	50)	0	0	10	0	0	2
22)	5	2	1	1	2	1	51)	1	3	3	3	2	0
23)	0	0	4	8	0	0	52)	4	3	0	1	1	3
24)	1	1	1	1	7	1	53)	3	0	1	4	2	2
25)	4	5	0	1	1	1	54)	0	2	3	0	4	3
26)	4	1	0	1	0	6	55)	1	4	1	1	1	4
27)	1	3	1	1	3	3	56)	2	1	4	2	2	1
28)	2	1	6	1	1	1	57)	3	2	1	1	4	1
29)	2	2	2	2	2	2	58)	2	2	2	3	0	3

Fig. C.18: Sequence of compositions found in *Sheer Pluck*.

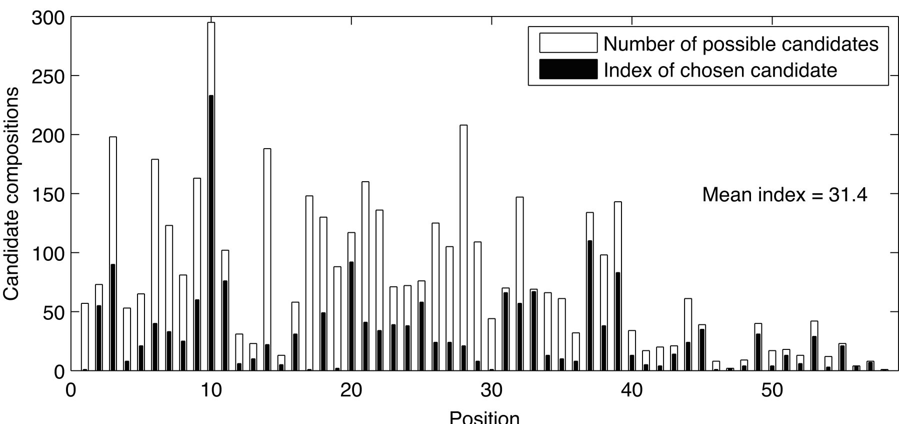
8.1 Task 1: Predicting the sequences of integer compositions in Babbitt's works

In recent years, Babbitt's compositional sketches for many of his works have been made publicly available by the Library of Congress in Washington, DC.[15] In many of these sketches, Babbitt included both the row-form arrays and all-partition arrays that he used when composing his works.[16] There is therefore little debate as to the organisation of tone rows in a piece as found in a PcMatrix, for example, or the sequence of integer compositions found in its corresponding WPcMatrix. What is not as straight-forward, however, is determining *how* such a sequence was discovered, given the large search space of possible sequences.

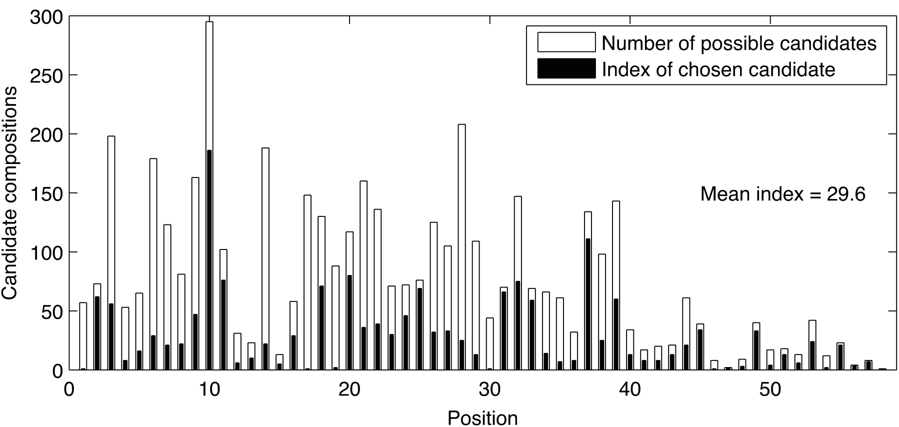
We evaluated our method by measuring how well the equal-lyne-length and zero-gain segments heuristics predict the sequences of integer compositions chosen by Babbitt. By sorting all candidates according to their quality or weight, W (as defined in section 6.3.2 above), we predicted that each integer composition chosen by Babbitt would lie high up in the list of candidate compositions for that integer composition's position in **CList**. In other

[15] `https://lccn.loc.gov/2014565648`.

[16] See Bernstein (2014) for excerpts from several of these sketches.



(a) Equal-lyne-length heuristic in *Sheer Pluck*.



(b) Equal-lyne-length and zero-gain heuristics in *Sheer Pluck*.

Fig. C.19: Results of applying (a) equal-lyne-length and (b) equal-lyne-length and zero-gain segments to select the sequence of integer compositions used in *Sheer Pluck*. Note (1) the less black at each position, the better; and (2) the lower the value of the mean index, the better.

words, the lower the values in **C**, the better our heuristics are predicting the compositions used by Babbitt.

Figure C.19 shows the index of each candidate composition found in the self-contained sequence used in *Sheer Pluck* after applying the equal-lyne-length heuristic (in Figure C.19(a)) and both equal-lyne-length and zero-gain segments (in Figure C.19(b)). We can get a measure of the overall performance of the heuristics simply by finding the mean value of all indices for a particular sequence. As shown in Figure C.19, by applying both heuristics, we achieve a marginally lower value for the mean index.

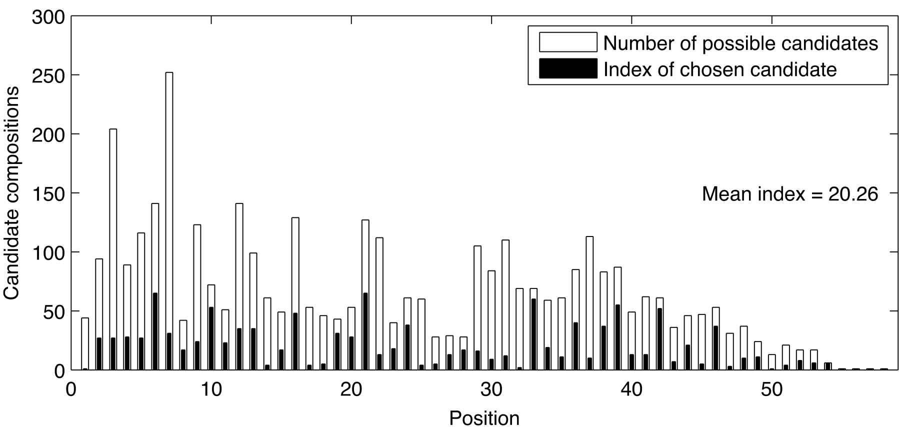
In this paper we have focused on the Smalley array. However, we also explored how well our heuristics predict the sequence of compositions in Babbitt's *About Time*, a piece not based on a Smalley array, that uses a non-self-contained sequence of compositions.[17] Figure C.20 shows the index of each candidate composition found in this piece after applying the equal-lyne-length heuristic (in Figure C.20(a)) and both equal-lyne-length and zero-gain segments (in Figure C.20(b)). As can be seen in Figure C.20, with this piece we achieve substantially lower mean index values than for *Sheer Pluck* (cf. Figure C.19), suggesting that these heuristics better model the structure of *About Time* than they do the structure of *Sheer Pluck*. It should be noted however, that, whereas adding the zero-gain segments heuristic improved the model's performance on *Sheer Pluck*, it actually slightly *reduced* performance on *About Time*. However, using the zero-gain segments heuristic alone resulted in the worst performance on both pieces.

8.2 Task 2: Generating a new Smalley array

We also evaluated the performance of our model when attempting to generate a new WPcMatrix (and sequence of compositions) from a PcMatrix used by Babbitt. This WPcMatrix could be a new Smalley array or, in the event that no other array exists, the array found in one of Babbitt's pieces. We assessed the performance of our algorithm by analysing the frequency at which backtracks occur at each position in the sequence during the generation of a WPcMatrix. In this generation scenario, the parameter values for selecting candidate compositions, **C**, and for selecting the OARP solutions for each candidate composition, **R**, are set to zero. Figure C.21 shows how the first 100000 backtracks were distributed over the 58 positions in the sequence of integer partitions for *Sheer Pluck* under five different conditions.

Note in Figure C.21 the locations of the horizontal line in each box indicating the median backtracking position for each condition (the horizontal line in condition four lies at the top of the box). Each box indicates the interquartile range (IQR). The whiskers indicate the lowest datum still within 1.5 IQR

[17]The all-partition array for *About Time* is given by Mead (1987, pp. 207–209).



(a) Equal-lyne-length heuristic in *About Time*.

(b) Equal-lyne-length and zero-gain heuristics in *About Time*.

Fig. C.20: Results of applying (a) equal-lyne-length and (b) equal-lyne-length and zero-gain segments to select the sequence of integer compositions used in *About Time*. Note (1) the less black at each position, the better; and (2) the lower the value of the mean index, the better.

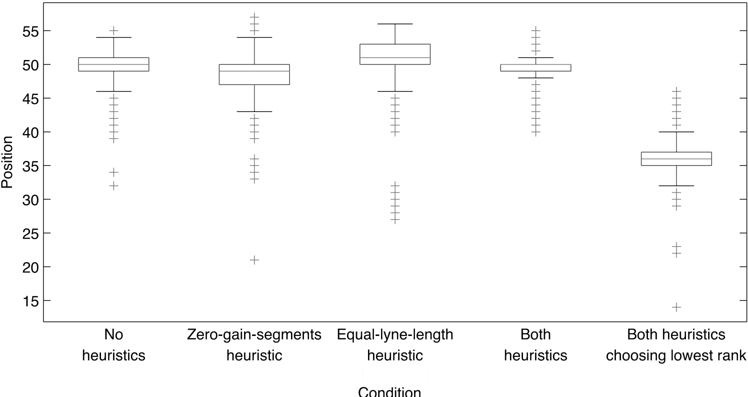


Fig. C.21: Comparison of the distribution of 100000 backtracks over positions in the integer partition sequence for *Sheer Pluck* under five conditions. The horizontal line in each box indicates the median backtracking position for that condition. The median for the "Both heuristics" condition co-incides with the top of the box. See text for details.

of the lower quartile, and the highest datum still within 1.5 IQR of the upper quartile. This means that, for each condition, about 99% of the backtracks lie within the range indicated by the whiskers and 50% of the backtracks lie within the range indicated by the box. Outliers are indicated by crosses. Without knowing where the set of all solutions lie within the search space of all possible sequences of compositions, it is difficult to properly evaluate these findings. However, if we assume a higher median position suggests a stronger likelihood of reaching position 58 and generating a complete solution, then the best performing condition is the equal-lyne-length heuristic alone. Contrary to our findings on the first evaluation task reported above, the results in Figure C.21 suggest that using no heuristics is preferable to using both heuristics or zero-gain-segments alone. It is noticeable that using both heuristics appears to decrease the spread in the positions at which backtracks occur, relative to using no heuristics or zero-gain-segments alone.

9 Conclusion

In this paper, we have proposed an algorithm for generating all-partition arrays of the type used in several of Milton Babbitt's works and shown how it can be used to generate the arrays underlying three of Babbitt's works (*Sheer Pluck* (1984), *Joy of More Sextets* (1986) and *None but the Lonely Flute* (1991)). The arrays found in these pieces belong to a class of six-lyne, horizontally weighted and self-contained all-partition arrays called *Smalley* arrays. Our

proposed method for generating Smalley arrays relies on constructing two intermediary structures we have called the PcMatrix and WPcMatrix. In particular, the functions, ComputeOperations and ComputeTranspositions, encode the decisions required to generate the PcMatrices of a Smalley array while our algorithm BacktrackingBabbitt is designed to take as input a PcMatrix and find in it a compatible sequence of compositions that form a WPcMatrix. As the process of forming a successful WPcMatrix (containing 120 OARPs) is a difficult task, we introduced two heuristics called equal-lyne-length and zero-gain segments for selecting compositions we suggest are more likely to be found in a Smalley array. Our algorithm could, if given enough time, generate all Smalley arrays. However, we have not yet succeeded in using it to generate any all-partition arrays from scratch. This suggests that Smalley likely used some heuristics or techniques for constructing his arrays that are not implemented by our proposed algorithm. Nevertheless, we believe that the work reported here represents a significant first step towards developing a fully specified computational model of the process that Smalley used to construct his arrays. In future research we intend to improve upon this design using better heuristics for guiding the search and by exploring alternative methods for finding a solution in a more reasonable time. We hope to use these improvements to generate new arrays not found in Babbitt's music and explore the possibility of using these novel arrays to create interesting new musical works.

Acknowledgements

The authors would like to thank Zachary Bernstein, Frank Brickle, Andrew Mead, Robert Morris and David Smalley for their kind and informative responses to our queries. The work reported in this paper was carried out as part of the EC-funded collaborative project, "Learning to Create" (Lrn2Cre8). The Lrn2Cre8 project acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET grant number 610859.

References

Babbitt, M. (1955). Some aspects of twelve-tone composition. *The Score and I.M.A. Magazine*, 12:53–61.

Babbitt, M. (1960). Twelve-tone invariants as compositional determinants. *Journal of Music Theory*, 46(2):246–259.

Babbitt, M. (1961). Set structure as a compositional determinant. *Journal of Music Theory*, 5(1):72–94.

Babbitt, M. (1962). Twelve-tone rhythmic structure and the electronic medium. *Perspectives of New Music*, 1(1):49–79.

Babbitt, M. (1973). Since Schoenberg. *Perspectives of New Music*, 12(1/2):3–28.

Bazelow, A. R. and Brickle, F. (1976). A partition problem posed by Milton Babbitt (Part I). *Perspectives of New Music*, 14(2):280–293.

Bazelow, A. R. and Brickle, F. (1979). A combinatorial problem in music theory: Babbitt's partition problem (I). *Annals of the New York Academy of Sciences*, 319(1):47–63.

Bemman, B. and Meredith, D. (2014). From analysis to surface: Generating the surface of Milton Babbitt's *Sheer Pluck* from a parsimonious encoding of an analysis of its pitch-class structure. In *The Music Encoding Conference, 20–23 May 2014*, Charlottesville, VA.

Bemman, B. and Meredith, D. (2015a). Comparison of heuristics for generating all-partition arrays in the style of Milton Babbitt. In *CMMR 11th International Symposium on Computer Music Multidisciplinary Research, 16–19 June 2015*, Plymouth, UK.

Bemman, B. and Meredith, D. (2015b). Exact cover problem in Milton Babbitt's all-partition array. In Collins, T., Meredith, D., and Volk, A., editors, *Mathematics and Computation in Music: Fifth International Conference, MCM 2015, London, UK, June 22–25, 2015, Proceedings*, volume 9110 of *Lecture Notes in Artificial Intelligence*, pages 237–242. Springer, Berlin.

Bernstein, Z. (2014). The problem of completeness in Milton Babbitt's music and thought. In *43rd Annual Meeting of The Music Theory Society of New York State (5–6 April 2014)*, New York University.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition.

Cuciurean, J. (1997). Self-similarity and compositional strategies in the music of Milton Babbitt. *Canadian University Music Review*, 17(2):1–16.

Eger, S. (2013). Restricted weighted integer compositions and extended binomial coefficients. *Journal of Integer Sequences*, 16(13.1.3):1–25.

Kassler, M. (1963). A sketch of the use of formalized languages for the assertion of music. *Perspectives of New Music*, 1(2):83–94.

Kassler, M. (1967). Toward a theory that is the twelve-note-class system. *Perspectives of New Music*, 5(2):1–80.

Kowalski, D. (1985). *The array as a compositional unit: A study of derivational counterpoint as a means of creating hierarchical structures in twelve-tone music*. Ph.D. diss., Princeton University.

Lake, W. E. (1986). The architecture of a superarray composition: Milton Babbitt's String Quartet no. 5. *Perspectives of New Music*, 24(2):88–111.

Leong, D. and McNutt, E. (2005). Virtuosity in Babbitt's *Lonely Flute*. *Music Theory Online*, 11(1).

Martino, D. (1961). The source set and its aggregate formations. *Journal of Music Theory*, 5(2):224–273.

Mead, A. (1987). About about time's time: A survey of Milton Babbitt's recent rhythmic practice. *Perspectives of New Music*, 25:182–235.

Meredith, D. (2006). The ps13 pitch spelling algorithm. *Journal of New Music Research*, 35(2):121–159.

Mead, A. (1994). *An Introduction to the Music of Milton Babbitt*. Princeton University Press, Princeton, NJ.

Morris, R. (1987). *Composition with Pitch-Classes: A Theory of Compositional Design*. Yale University Press, New Haven, CT.

Morris, R. (2003). Pitch-class duplication in serial music: Partitions of the double aggregate. *Perspectives of New Music*, 41(2).

Morris, R. (2010). *The Whistling Blackbird: Essays and Talks on New Music*. The University of Rochester Press, Princeton, NJ.

Page, D. (2013). Parallel algorithm for second-order restricted weak integer composition generation for shared memory machines. *Parallel Processing Letters*, 23(3):1350010.

Starr, D. and Morris, R. (1977). A general theory of combinatoriality and the aggregate, part 1. *Perspectives of New Music*, 16(1):3–35.

Starr, D. and Morris, R. (1978). A general theory of combinatoriality and the aggregate, part 2. *Perspectives of New Music*, 16(2):50–84.

Tani, N. B. and Bouroubi, S. (2011). Enumeration of the partitions of an integer into parts of a specified number of different sizes and especially two sizes. *Journal of Integer Sequences*, 14(11.3.6):1–12.

Winham, G. (1970). Composition with arrays. *Perspectives of New Music*, 9(1):43–67.

Paper D

Integer Programming Formulation of the Problem of Generating Milton Babbitt's All-partition Arrays

Tanaka, T., Bemman, B., and Meredith D.

The paper has been published in the
17th International Society for Music Information Retrieval Conference Proceedings, August 7– 11, New York, NY 2016.

© 2016 International Society for Music Information Retrieval
The layout has been revised.

Abstract

Milton Babbitt (1916–2011) was a composer of twelve-tone serial music noted for creating the all-partition array. The problem of generating an all-partition array involves finding a rectangular array of pitch-class integers that can be partitioned into regions, each of which represents a distinct integer partition of 12. Integer programming (IP) has proven to be effective for solving such combinatorial problems, however, it has never before been applied to the problem addressed in this paper. We introduce a new way of viewing this problem as one in which restricted overlaps between integer partition regions are allowed. This permits us to describe the problem using a set of linear constraints necessary for IP. In particular, we show that this problem can be defined as a special case of the well-known problem of set-covering (SCP), modified with additional constraints. Due to the difficulty of the problem, we have yet to discover a solution. However, we assess the potential practicality of our method by running it on smaller similar problems.

1 Introduction

Milton Babbitt (1916–2011) was a composer of twelve-tone serial music noted for developing complex and highly constrained music. The structures of many of his pieces are governed by a structure known as the *all-partition array*, which consists of a rectangular array of pitch-class integers, partitioned into regions of distinct "shapes", each corresponding to a distinct integer partition of 12. This structure helped Babbitt to achieve *maximal diversity* in his works, that is, the presentation of as many musical parameters in as many different variants as possible (Mead, 1994).

In this paper, we formalize the problem of generating an all-partition array using an integer programming paradigm in which a solution requires solving a special case of the set-covering problem (SCP), where the subsets in the cover are allowed a restricted number of overlaps with one another and where the ways in which these overlaps can occur is constrained. It turns out that this is a hard combinatorial problem. That this problem was solved by Babbitt and one of his students, David Smalley, without the use of a computer is therefore interesting in itself. Moreover, it suggests that there exists an effective procedure for solving the problem.

Construction of an all-partition array begins with an $I \times J$ matrix, A, of pitch-classes, $0, 1, \ldots, 11$, where each row contains $J/12$ twelve-tone rows. In this paper, we only consider matrices where $I = 6$ and $J = 96$, as matrices of this size figure prominently in Babbitt's music (Mead, 1994). This results in a 6×96 matrix of pitch classes, containing 48 twelve-tone rows. In other words, A will contain an approximately uniform distribution of 48 occurrences of each of the integers from 0 to 11. On the musical surface, rows

11	4	3	5	9	10	1	8	2	0	7	6
6	7	0	2	8	1	10	9	5	3	4	11
5	6	11	1	7	0	9	8	4	2	3	10
2	9	10	8	4	3	0	5	11	1	6	7
0	5	4	6	10	11	2	9	3	1	8	7
1	8	9	7	3	2	11	4	10	0	5	6

Fig. D.1: A 6×12 excerpt from a 6×96 pitch-class matrix with the integer composition, $\text{IntComp}_{12}(3,2,1,3,1,2)$ (in dark gray), containing each pitch class exactly once.

of this matrix become expressed as "musical voices", typically distinguished from one another by instrumental register (Mead, 1994). A complete all-partition array is a matrix, A, partitioned into K regions, each of which must contain each of the 12 pitch classes exactly once. Moreover, each of these regions must have a distinct "shape", determined by a distinct *integer partition* of 12 (e.g., $2+2+2+3+3$ or $1+2+3+1+2+3$) that contains I or fewer summands greater than zero (Bemman and Meredith, 2015a). We denote an integer partition of an integer, L, by $\text{IntPart}_L(s_1, s_2, \ldots, s_I)$ and define it to be an ordered set of non-negative integers, $\langle s_1, s_2, \ldots, s_I \rangle$, where $L = \sum_{i=1}^{I} s_i$ and $s_1 \geq s_2 \geq \cdots \geq s_I$. For example, possible integer partitions of 12 when $I = 6$, include $\text{IntPart}_{12}(3,3,2,2,1,1)$ and $\text{IntPart}_{12}(3,3,3,3,0,0)$. We define an *integer composition* of a positive integer, L, denoted by $\text{IntComp}_L(s_1, s_2, \ldots, s_I)$, to also be an ordered set of I non-negative integers, $\langle s_1, s_2, \ldots, s_I \rangle$, where $L = \sum_{i=1}^{I} s_i$, however, unlike an integer partition, the summands are not constrained to being in descending order of size. For example, if $L = 12$ and $I = 6$, then $\text{IntComp}_{12}(3,3,3,3,0,0)$ and $\text{IntComp}_{12}(3,0,3,3,3,0)$ are two distinct integer compositions of 12 defining the same integer *partition*, namely $\text{IntPart}_{12}(3,3,3,3,0,0)$.

Figure D.1 shows a 6×12 excerpt from a 6×96 pitch-class matrix, A, and a region determined by the integer composition, $\text{IntComp}_{12}(3,2,1,3,1,2)$, containing each possible pitch class exactly once. Note, in Figure D.1, that each summand (from left to right) in $\text{IntComp}_{12}(3,2,1,3,1,2)$, gives the number of elements in the corresponding row of the matrix (from top to bottom) in the region determined by the integer composition. We call this part of a region in a given row of the matrix a *summand segment*. For example, in Figure D.1, the summand segment in the first row for the indicated integer partition region contains the pitch classes 11, 4 and 3. On the musical surface, the distinct shape of each integer composition helps contribute to a progression of "musical voices" that vary in textural density, allowing for relatively thick textures in, for example, $\text{IntComp}_{12}(2,2,2,2,2,2)$ (with six participating parts) and comparatively sparse textures in, for example, $\text{IntComp}_{12}(11,0,1,0,0,0)$ (with two participating parts).

11	4	3	**3**	5	9	10	1	8	2	0	7	6
6	7	**7**	0	2	8	1	10	9	5	3	4	11
5	6	11	1	7	0	9	8	4	2	3	10	
2	9	10	**10**	8	4	3	0	5	11	1	6	7
0	5	4	6	10	11	2	9	3	1	8	7	
1	8	9	7	3	2	11	4	10	0	5	6	

Fig. D.2: A 6×12 excerpt from a 6×96 pitch-class matrix with a region whose shape is determined by the integer composition, $\text{IntComp}_{12}(3,3,3,3,0,0)$ (in light gray), where three elements (in bold) are horizontal insertions of pitch classes from the previous integer partition region. Note that the two indicated regions represent distinct integer partitions.

There exist a total of 58 distinct integer partitions of 12 into 6 or fewer non-zero summands (Mead, 1994). An all-partition array with six rows will thus contain $K = 58$ regions, each containing every pitch class exactly once and each with a distinct shape determined by an integer composition representing a distinct integer partition. However, the number of pitch-class integers required to satisfy this constraint, $58 \times 12 = 696$, exceeds the size of a 6×96 matrix containing 576 elements, by 120. In order to satisfy this constraint, additional pitch-classes therefore have to be inserted into the matrix, with the added constraint that only horizontal insertions of at most one pitch class in each row are allowed for each of the 58 integer partition regions. Each inserted pitch class is identical to its immediate neighbor to the left, this being the right-most element of a summand segment belonging to a previous integer partition region. This constraint ensures that the order of pitch classes in the twelve-tone rows of a given row of A is not altered (Mead, 1994). Figure D.2 shows a second integer partition region, $\text{IntComp}_{12}(3,3,3,3,0,0)$, in the matrix shown in Figure D.1 (indicated in light gray), where three of its elements result from horizontal insertions of pitch classes from the previous integer partition region. Note, in Figure D.2, the three horizontal insertions of pitch-class integers, 3 (in row 1), 7 (in row 2), and 10 (in row 4), required to have each pitch class occur exactly once in the second integer partition region. Not all of the 58 integer partitions must contain one or more of these insertions, however, the total number of insertions must equal the 120 additional pitch classes required to satisfy the constraint that all 58 integer partitions are represented. Note that, in order for each of the resulting integer partition regions to contain every pitch class exactly once, ten occurrences of each of the 12 pitch classes must be inserted into the matrix. This typically results in the resulting matrix being irregular (i.e., "ragged" along its right side).

In this paper, we address the problem of generating an all-partition array by formulating it as a set of linear constraints using the integer programming (IP) paradigm. In section 2, we review previous work on general IP problems and their use in the generation of musical structures. We also review

previous work on the problem of generating all-partition arrays. In section 3, we introduce a way of viewing insertions of elements into the all-partition array as fixed locations in which overlaps occur between contiguous integer partition regions. In this way, our matrix remains regular and we can define the problem as a special case of the well-known IP problem of set-covering (SCP), modified so that certain overlaps are allowed between the subsets. In sections 4 and 5, we present our IP formulation of this problem as a set of linear constraints. Due to the difficulty of the problem, we have yet to discover a solution using our formulation. Nevertheless, in section 6, we present the results of using our implementation to find solutions to smaller versions of the problem and in this way explore the practicality of our proposed method. We conclude in section 7 by mentioning possible extensions to our formulation that could potentially allow it to solve the complete all-partition array generation problem.

2 Previous work

Babbitt himself laid the foundations for the construction of what would become the all-partition array during the 1960s, and he would continue to use the structure in nearly all of his later works (Babbitt, 1960, 1961, 1962, 1973). Subsequent composers made use of the all-partition array in their own music and further developed ways in which its structure could be formed and used (Bazelow and Brickle, 1976, 1979; Kowalski, 1985, 1987; Morris, 1987, 2010; Starr and Morris, 1977, 1978; Winham, 1970). Most of these methods focus on the organization of pitch classes in a twelve-tone row and how their arrangement can make the construction of an all-partition array more likely. We propose here a more general purpose solution that will take any matrix and attempt to generate a successful structure. Furthermore, many of these previous methods were music-theoretical in nature and not explicitly computational. Work by Bazelow and Brickle is one notable exception (Bazelow and Brickle, 1976, 1979). We agree here with their assessment that "partition problems in twelve-tone theory properly belong to the study of combinatorial algorithms" (Bazelow and Brickle, 1979). However, we differ considerably in our approach and how we conceive of the structure of the all-partition array.

More recent efforts to automatically analyze and generate all-partition arrays have been based on backtracking algorithms.(Bemman and Meredith, 2015a,b, 2016a). True to the structure of the all-partition array (as it appears on the musical surface) and the way in which Babbitt and other music theorists conceive of its structure, these attempts to generate an all-partition array form regions of pitch classes according to the process described in section 1, where horizontal repetitions of pitch-classes are added, resulting in an irregular matrix. While these existing methods have further proposed various

heuristics to limit the solution space or allow for incomplete solutions, they were unable to generate a complete all-partition array (Bemman and Meredith, 2015a,b, 2016a).

In general, for difficult combinatorial problems, more efficient solving strategies than backtracking exist. One such example is integer programming (IP). IP is a computationally efficient and practical paradigm for dealing with typically NP-hard problems, such as the traveling salesman, set-covering and set-partitioning problems, where these are expressed using only linear constraints (i.e., equations and inequalities) and a linear objective function (Cormen et al., 2009; Orman, 2006). One benefit of using IP, is that it allows for the separation of the formulation of a problem by users and the development by specialists of an algorithm for solving it. Many of these powerful solvers dedicated to IP problems have been developed and used particularly in the field of operations research. Compared to approximate computational strategies, such as genetic algorithms, IP formulations and their solvers are suitable for searching for solutions that strictly satisfy necessary constraints. For this reason, we expect that the IP paradigm could provide an appropriate method for approaching the problem of generating all-partition arrays.

In recent work, IP has been applied to problems of analysis and generation of music (Tanaka, 2014, 2015). This is of importance to the research presented here as it demonstrates the relevance of these traditional optimization problems of set-covering (SCP) and set-partitioning (SPP), to general problems found in computational musicology, where SPP has been used in the segmentation of melodic motifs and IP has been used in describing global form. In the next section, we address the set-covering problem (SCP) in greater detail and show how it is related to the problem of generating all-partition arrays.

3 Set-covering problem formulation of all-partition array generation

The set-covering (SCP) problem is a well-known problem in computer science and operations research that can be shown to be NP-hard (Cormen et al., 2009). Let E be a set whose elements are $\{E_1, E_2, \cdots, E_{\#E}\}$ (where $\#E$ denotes the number of elements in E), F be a family of subsets of E, $\{F_1, F_2, \cdots, F_{\#F}\}$, and S be a subset of F. By assigning a constant cost, c_s, to each F_s, the objective of the set-covering problem (SCP) is to

$$\begin{aligned} &\mathop{Minimize}_{S \subset F} \sum_{F_s \in S} c_s \\ &subject\ to \bigcup_{F_s \in S} F_s = E. \end{aligned}$$

11	4	3	5	9	10	1	8	2	0	7	6
6	7	0	2	8	1	10	9	5	3	4	11
5	6	11	1	7	0	9	8	4	2	3	10
2	9	10	8	4	3	0	5	11	1	6	7
0	5	4	6	10	11	2	9	3	1	8	7
1	8	9	7	3	2	11	4	10	0	5	6

Fig. D.3: A 6×12 excerpt from a 6×96 pitch-class matrix with two integer compositions, $\text{IntComp}_{12}(3,2,1,3,1,2)$ (in dark gray and outlined) and $\text{IntComp}_{12}(3,3,3,3,0,0)$ (in light gray), that form distinct integer partition regions. Note, that the second composition overlaps three fixed locations in the first.

In other words, a solution S is a *cover* of E that allows for the same elements to appear in more than one subset, F_s. In this section, we suggest that our problem can be viewed as an SCP with additional constraints.

3.1 All-partition array generation as a set-covering problem (SCP) with additional constraints

When viewing the all-partition array in the context of $\bigcup_{F_s \in S} F_s = E$ above, E is the set that consists of all locations (i, j) in the matrix, A, and F_s are the sets of locations (i, j) that correspond to the “shapes" of integer compositions. We call each F_s a *candidate set*. A candidate set F_s is characterized by two conditions that we call *containment* and *consecutiveness*. Containment means that the elements (i.e., locations (i, j)) of F_s correspond to twelve distinct integers, $0, 1, \ldots, 11$, in A. Consecutiveness means that each of its elements belonging to the same row in A are consecutive. In this sense, F includes all sets found in A that satisfy the conditions of consecutiveness and containment.

As the expression $\bigcup_{F_s \in S} F_s = E$ implies, a candidate set is allowed to share elements with another candidate set. Similarly, the pitch classes in A (i.e., corresponding to elements in E) that become insertions in the original problem can be instead regarded as shared elements or *overlaps* between contiguous integer composition regions, with the result that the matrix remains regular. Figure D.3 shows how these overlaps would occur in the two integer composition regions shown in Figure D.2.

Viewed in this way, a solution to the problem of generating an all-partition array thus satisfies the basic criterion of an SCP, namely, the condition for *set-covering*, $\bigcup_{F_s \in S} F_s = E$. However, this criterion alone fails to account for the unique constraints under which such a covering is formed in an all-partition array. In the original SCP, there are no constraints on the order of subsets, the order of their elements or the number of overlaps and the ways in which they can occur. On the other hand, an all-partition array must satisfy such additional conditions. We denote the constraints for satisfying such additional

conditions by *Add. Conditions*.

Add. Conditions includes the conditions in the all-partition array governing (1) the left-to-right order of contiguous candidate sets, (2) permissible overlaps between such sets, and (3) the *distinctness* of sets in S. This last condition of distinctness ensures that the integer compositions used in a cover, S, define every possible integer partition once and only once. On the other hand, the conditions for set-covering, $\bigcup_{F_s \in S} F_s = E$, are conditions of (1) candidate sets (which satisfy containment and consecutiveness) and (2) *covering*, meaning that each element in E is covered no less than once.

We can now state that our problem of generating an all-partition array is to

$$\begin{aligned} \underset{S \subset F}{Minimize} \quad & \sum_{F_s \in S} c_s \\ subject\ to \quad & \bigcup_{F_s \in S} F_s = E, \\ & Add.\ Conditions. \end{aligned}$$

where the associated cost, c_s, of each F_s, can be interpreted as a preference for one integer composition or another. It is likely that, in the interest of musical expression, Babbitt may have preferred the shapes of some integer partition regions over others (Mead, 1994). However, as his preference is unknown, we can regard these costs to have the same value for each F_s.

Due to the condition of distinctness (just described), $|S|$ can be fixed at 58. This feature, combined with the equal costs of each F_s, means that the objective function, $\sum_{F_s \in S} c_s$, for this problem, is constant. For these reasons, the above formulation is a *constraint satisfaction problem*. This motivates our discussions in sections 6 and 7 on possible alternative objective functions.

In the next two sections, we implement the constraint satisfaction problem defined above using integer programming (IP). In particular, section 4 addresses the conditions for set-covering, $\bigcup_{F_s \in S} F_s = E$, and section 5 addresses those in *Add. Conditions*. It is because of our new way of viewing this problem, with a regular matrix and overlaps, that we are able to introduce variables for use in IP to describe these conditions.

4 IP implementation of conditions for set-covering in all-partition array generation

In this section, we introduce our set of linear constraints for satisfying the general conditions for set-covering, $\bigcup_{F_s \in S} F_s = E$, in the generation of an all-partition array. Before we introduce these constraints, we define the necessary variables and constants used in our implementation of the conditions for set-covering. We begin with a given matrix found in one of Babbitt's works based

on the all-partition array. Examples of the matrices used in this paper can be found in Babbitt's *Arie da Capo* (1974) and *None but the Lonely Flute* (1991), among others. Let $(A_{i,j})$ be a $(6, 96)$-matrix whose elements are the pitch-class integers, $0, 1, \ldots, 11$. We denote the number of rows and columns by I and J, respectively.

Let $x_{i,j,k}$ $(1 \leq i \leq I,\ 1 \leq j \leq J)$ be a binary variable corresponding to each location (i, j) in A and a subset (i.e., integer partition) identified by the integer k, where $1 \leq k \leq K$ and $K = 58$. Here, we consider the case where $I = 6$ and $J = 96$, so there are 58 sets of 576 such variables. Each of these variables will indicate whether or not a location (i, j) belongs to a candidate set for the kth position in the sequence of 58 integer partition regions. We denote the set of locations (i, j) whose corresponding value for $x_{i,j,k}$ is 1, to be C_k. Subject to conditions for consecutiveness and containment, C_k will be a candidate set.

Let $(B^p_{i,j})$ $(0 \leq p \leq 11)$ be constant matrices, equal in size to A, where $B^p_{i,j} = 1$ if and only if $A_{i,j} = p$ and $B^p_{i,j} = 0$ otherwise. The locations (i, j) whose values of $B^p_{i,j}$ equal 1, correspond to the locations of pitch-class p in A.

4.1 Conditions for C_k to contain twelve distinct integers in A (condition of containment)

A condition for C_k to satisfy the condition of containment is that its number of elements is 12 and each corresponds to a distinct pitch-class in A. These conditions are expressed by the following two equations:

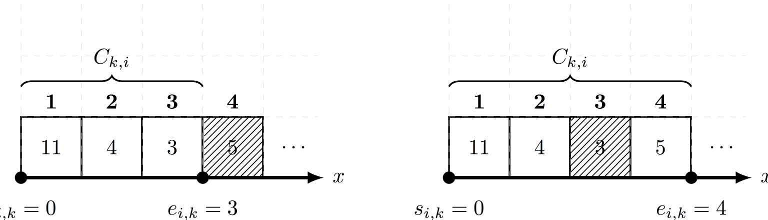
$$\forall k \in [1, K],\ \sum_{i=1}^{I} \sum_{j=1}^{J} x_{i,j,k} = 12, \tag{D.1}$$

$$\forall p \in [0, 11], \forall k \in [1, K],\ \sum_{i=1}^{I} \sum_{j=1}^{J} B^p_{i,j} \cdot x_{i,j,k} = 1. \tag{D.2}$$

Because $x_{i,j,k}$ equals 1 if (i, j) is included in C_k and 0 if it is not, Equation D.1 means that there are 12 elements in C_k. In Equation D.2, we ensure that each corresponding pitch-class integer p for the elements in C_k, appears once and only once.

4.2 Conditions for C_k to be integer compositions in A (condition of consecutiveness)

Let $C_{k,i}$ be the ith-row part of C_k (i.e., the summand segment of composition k for row i). Let $s_{i,k}$ be an integer variable corresponding to the x-coordinate of a "starting point", which lies at the left side of the leftmost component



(a) $C_{k,i}$ contains pitch classes $11, 4, 3$ ($j = 1, 2, 3$) and satisfies consecutiveness.

(b) $C_{k,i}$ contains pitch classes $11, 4, 5$ ($j = 1, 2, 4$) and does not satisfy consecutiveness.

Fig. D.4: Two $C_{k,i}$ and corresponding $s_{i,k}$ and $e_{i,k}$ from Figure D.3 when $k = 1$ and $i = 1$. Shaded elements indicate $x_{i,j,k} = 0$ and unshaded elements indicate $x_{i,j,k} = 1$.

of $C_{k,i}$. The value of $s_{i,k}$ is then equal to the column number of the leftmost component of $C_{k,i}$, minus 1. The origin point of this coordinate lies along the left hand side of the matrix A, and we set the width of each location (i, j) to be 1. Similarly, let $e_{i,k}$ be an integer variable corresponding to the x-coordinate of an "ending point", which lies at the right side of the rightmost component belonging to $C_{k,i}$. The value of $e_{i,k}$ is then equal to the column number of the rightmost component of $C_{k,i}$. Figure D.4 shows an example of two possible $C_{k,i}$ from Figure D.3. If there is no component in $C_{k,i}$ ($k \geq 2$), we define $s_{i,k}$ to be $e_{i,k-1}$ and $e_{i,k}$ to be $s_{i,k}$. If there is no component in $C_{1,i}$, we define $s_{i,k}$ and $e_{i,k}$ to be 0. Then, $s_{i,k}$ and $e_{i,k}$ are subject to the following constraint of range:

$$\forall i \in [1, I], \forall k \in [1, K],\ 0 \leq s_{i,k} \leq e_{i,k} \leq J. \tag{D.3}$$

The condition under which C_k ($k \in [1, K]$) forms an integer composition—that is, satisfies the condition of consecutiveness, is expressed by the following three constraints:

$$\forall i \in [1, I], \forall j \in [1, J], \forall k \in [1, K], \quad j \cdot x_{i,j,k} \leq e_{i,k}, \tag{D.4}$$

$$\forall i \in [1, I], \forall j \in [1, J], \forall k \in [1, K], \quad J - s_{i,k} \geq (J + 1 - j) \cdot x_{i,j,k}, \tag{D.5}$$

$$\forall i \in [1, I], \forall k \in [1, K],\ \sum_{j=1}^{J} x_{i,j,k} = e_{i,k} - s_{i,k}. \tag{D.6}$$

In Equation D.4, each element of $C_{k,i}$ must be located at column $e_{i,k}$ or to the left of column $e_{i,k}$. Equation D.5 states that each element of $C_{k,i}$ must

be located at column $s_{i,k}+1$ or to the right of column $s_{i,k}+1$. Equation D.6, combined with the previous two constraints, states that the length of $C_{k,i}$ must be equal to $e_{i,k}-s_{i,k}$, implying that the column numbers j of the elements in $C_{k,i}$ are consecutive from $s_{i,k}+1$ to $e_{i,k}$, where $C_{k,i}$ contains at least one element.

4.3 Condition for covering A

As every location (i, j) in A (i.e., E in our SCP) must be covered at least once, we pose the following condition of covering:

$$\forall i \in [1, I], \forall j \in [1, J],\ \sum_{k=1}^{K} x_{i,j,k} \geq 1. \tag{D.7}$$

Equation D.1 states that for all $K = 58$ integer partitions, there are $12 \cdot K = 696$ variables, $x_{i,j,k}$, that will equal 1. A successful cover of A by Equation D.7, however, states that all of $I \cdot J = 576$ places (i, j) in A, are covered once or more than once. Collectively, these imply that there are 120 or less than 120 places (i.e., combinations of (i, j)) that are covered twice or more than twice. These 120 overlaps correspond to the 120 insertions of pitch-class integers used when constructing an all-partition array in its original form. By satisfying all of the constraints above, each C_k forms a candidate set (i.e., a member of F in our SCP) and the condition for set-covering, $\bigcup_{F_s \in S} F_s = E$, is satisfied.

5 IP implementation of additional conditions in all-partition array generation

In this section, we introduce our set of additional linear constraints beyond those required for satisfying the condition of set-covering in the SCP.

5.1 Left-to-right order of C_k and permissible overlaps

C_k must be located immediately to the right of C_{k-1}. This is expressed by

$$\forall i \in [1, I], \forall k \in [2, K],\ e_{i,k-1} \leq e_{i,k}, \tag{D.8}$$

$C_{k-1,i}$ and $C_{k,i}$ may overlap by no more than one element. This is expressed by the following inequality:

$$\forall i \in [1, I], \forall k \in [2, K],\ e_{i,k-1} - 1 \leq s_{i,k} \leq e_{i,k-1}, \tag{D.9}$$

meaning that $s_{i,k}$ will be equal to $e_{i,k-1}$ if there is no overlap and $s_{i,k}$ will be equal to $e_{i,k-1}-1$ if there is an overlap.

5.2 Conditions for C_k to be integer compositions defining distinct integer partitions (condition of distinctness)

Let $y_{i,k,l}$ be a binary variable that indicates whether or not the length of $C_{k,i}$ is greater than or equal to l $(1 \leq l \leq L, L = 12)$, by introducing the following constraints:

$$\forall i \in [1, I], \forall k \in [1, K],\ e_{i,k} - s_{i,k} = \sum_{l=1}^{L} y_{i,k,l}, \tag{D.10}$$

$$\forall i \in [1, I], \forall k \in [1, K], \forall l \in [2, L], \quad y_{i,k,l-1} \geq y_{i,k,l}. \tag{D.11}$$

Equation D.10 states that the sum of all elements in $\langle y_{i,k,1}, y_{i,k,2}, \ldots, y_{i,k,L} \rangle$ is equal to the length of $C_{k,i}$, while Equation D.11 states that its elements equal to 1 begin in the first position and are consecutive (e.g., $\langle 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$, when the length of $C_{k,i}$ is 3.)

The number of the lengths of $C_{k,i}$ $(1 \leq i \leq I)$ that are greater than or equal to l is given by $\sum_{i=1}^{I} y_{i,k,l}$. The twelve values of $\sum_{i=1}^{I} y_{i,k,l}$ $(1 \leq l \leq L)$ then, will precisely represent the type of partition. For example, if C_k is $\mathrm{IntComp}_{12}(3, 2, 1, 3, 1, 2)$, then $y_{i,k,l}$ $(\forall i \in [1, I], \forall l \in [1, L])$ would be

1,1,1,0,0,0,0,0,0,0,0,0
1,1,0,0,0,0,0,0,0,0,0,0
1,0,0,0,0,0,0,0,0,0,0,0
1,1,1,0,0,0,0,0,0,0,0,0
1,0,0,0,0,0,0,0,0,0,0,0
1,1,0,0,0,0,0,0,0,0,0,0

and $\sum_{i=1}^{I} y_{i,k,l}$ would be $[6, 4, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0]$.

We denote the number of all integer partitions by N $(N = K = 58)$ and denote a single integer partition n $(1 \leq n \leq N)$ by P_n. We can express P_n as $[P_{n,1}, P_{n,2}, \ldots, P_{n,L}]$ $(1 \leq n \leq N)$, where $P_{n,l}$ corresponds to the twelve values $\sum_{i=1}^{I} y_{i,k,l}$ $(1 \leq l \leq L)$ described above.

Then, by implementing the following expression:

$$\forall k \in [1, K], \forall n \in [1, N], \forall l \in [1, L], \quad P_{n,l} \cdot z_{k,n} \leq \sum_{i=1}^{I} y_{i,k,l}. \tag{D.12}$$

we can express whether C_k defines the integer partition n or not by the binary variable $z_{k,n}$. For example, if $z_{k,n} = 0$, the value of $P_{n,l} \cdot z_{k,n} = 0$ constrains nothing, and thus C_k cannot be the integer partition n (because of the next equation). On the other hand, if $z_{k,n} = 1$, C_k must be the integer partition n.

Accordingly, $z_{k,n}$ will equal 1 only if the twelve values $\sum_{i=1}^{I} y_{i,k,l}$ correspond to P_n. From this, determining whether or not all different partitions are present can be expressed by the following equation:

$$\forall n \in [1, N], \sum_{k=1}^{K} z_{k,n} = 1. \tag{D.13}$$

6 Experiments

In order to determine whether or not our formulation works as intended, we implemented the constraints described in sections 4 and 5 and supplied these to an IP solver based on branch-and-bound (Gurobi Optimizer). As the objective function in our formulation amounts to a constant-cost function (described in section 3), we replaced it with a non-constant objective function, $\sum_{i,j,k} c_{i,j,k} \cdot x_{i,j,k}$, where $c_{i,j,k}$ assumes a randomly generated integer for promoting this process of branch and bound. When the first feasible solution is found, we stop the search.

Although we first attempted to find a complete all-partition array, we were unable to discover a solution after one day of calculation. This highlights the difficulty of the problem and reinforces those findings by previous methods that were similarly unable to find a complete all-partition array (Bemman and Meredith, 2015a). As the target of our current formulation is only solutions which strictly satisfy all constraints, we opted to try finding complete solutions to smaller-sized problems, using the first j columns of the original matrix. Because we cannot use all 58 integer partitions in the case $K < N$, a slight modification to Equation D.13 was needed for this change. Its equality was replaced by $\leq$ and an additional constraint, $\forall k \in [1, K]$, $\sum_{n=1}^{N} z_{k,n} = 1$, for allocating one partition to each C_k, was added.

Figure D.5 shows the duration (vertical axis) of time spent on finding a solution in matrices of varying size. The number of integer compositions, K, was set to $(J + 2)/2$, where J is an even number. This ensures that a given solution will always contain 12 overlaps. These findings suggest that the necessary computational time in finding a solution tends to dramatically increase with an increase in J. However, this increase fluctuates, suggesting that each small matrix represents a unique problem space with different sets of difficulties (e.g., the case $J = 14$ was unfeasible). For this reason, finding a solution in a complete matrix (6,96) within a realistic limitation of time would be difficult for our current method, even using a fast IP solver. This strongly motivates future improvements as well as the possibility of an altogether different strategy.

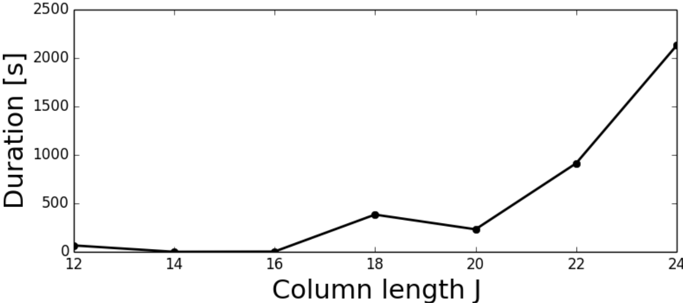


Fig. D.5: Duration of time spent on finding the first solution for each small matrix, whose column length is $J (12 \leq J \leq 24, J \in 2\mathbb{N})$. K is set to $(J+2)/2$, resulting in 12 overlaps. Note, that no feasible solution exists when $J = 14$.

7 Conclusion and future work

In this paper, we have introduced a novel integer-programming-based perspective on the problem of generating Milton Babbitt's all-partition arrays. We have shown that insertions and the irregular matrix that results can be replaced with restricted overlaps, leaving the regular matrix unchanged. This view allows us to formulate the problem as a set-covering problem (SCP) with additional constraints and then implement it using integer programming. Due to the difficulty of the problem, we have so far been unable to find a solution. However, we have been able to produce solutions in a practical running time (< 2500 seconds) when the matrix is reduced in size to 24 columns or less. These results motivate possible extensions to our formulation. First, a relaxation of the problem is possible, for example, by using an objective function that measures the degree of incompleteness of a solution. This could allow for approximate solutions to be discovered, such as those found in previous work (Bemman and Meredith, 2015a). Second, it may be the case that a solution to the full problem may be achievable by combining solutions to smaller subproblems that we have shown to be solvable in a practical running time.

8 Acknowledgements

The work of Tsubasa Tanaka reported in this paper was supported by JSPS Postdoctoral Fellowships for Research Abroad. The work of Brian Bemman and David Meredith was carried out as part of the project Lrn2Cre8, which acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET grant number 610859.

References

Babbitt, M. (1960). Twelve-tone invariants as compositional determinants. *Journal of Music Theory*, 46(2):246–259.

Babbitt, M. (1961). Set structure as a compositional determinant. *Journal of Music Theory*, 5(1):72–94.

Babbitt, M. (1962). Twelve-tone rhythmic structure and the electronic medium. *Perspectives of New Music*, 1(1):49–79.

Babbitt, M. (1973). Since Schoenberg. *Perspectives of New Music*, 12(1/2):3–28.

Bazelow, A. R. and Brickle, F. (1976). A partition problem posed by Milton Babbitt (Part I). *Perspectives of New Music*, 14(2):280–293.

Bazelow, A. R. and Brickle, F. (1979). A combinatorial problem in music theory: Babbitt's partition problem (I). *Annals of the New York Academy of Sciences*, 319(1):47–63.

Bemman, B. and Meredith, D. (2015a). Comparison of heuristics for generating all-partition arrays in the style of Milton Babbitt. In *CMMR 11th International Symposium on Computer Music Multidisciplinary Research, 16–19 June 2015*, Plymouth, UK.

Bemman, B. and Meredith, D. (2015b). Exact cover problem in Milton Babbitt's all-partition array. In Collins, T., Meredith, D., and Volk, A., editors, *Mathematics and Computation in Music: Fifth International Conference, MCM 2015, London, UK, June 22–25, 2015, Proceedings*, volume 9110 of *Lecture Notes in Artificial Intelligence*, pages 237–242. Springer, Berlin.

Bemman, B. and Meredith, D. (2016a). Generating Milton Babbitt's all-partition arrays. *Journal of New Music Research*, 45(2):1–21.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition.

Kowalski, D. (1985). *The array as a compositional unit: A study of derivational counterpoint as a means of creating hierarchical structures in twelve-tone music*. Ph.D. diss., Princeton University.

Kowalski, D. (1987). The construction and use of self-deriving arrays. *Perspectives of New Music*, 24(1/2):296–361.

Mead, A. (1994). *An Introduction to the Music of Milton Babbitt*. Princeton University Press, Princeton, NJ.

Morris, R. (1987). *Composition with Pitch-Classes: A Theory of Compositional Design*. Yale University Press, New Haven, CT.

Morris, R. (2010). *The Whistling Blackbird: Essays and Talks on New Music*. The University of Rochester Press, Princeton, NJ.

Orman, A. J. and Williams, H. Paul. (2006). A survey of different integer programming formulations of the travelling salesman problem. In Erricos John Kontoghiorghes and Cristian Gatu, editors, *Optimisation, Econometric and Financial Analysis*, volume 9 of *Advances in computational management science*, pages 93–108. Springer-Verlag Berlin Heidelberg, Berlin, 2006.

Starr, D. and Morris, R. (1977). A general theory of combinatoriality and the aggregate, part 1. *Perspectives of New Music*, 16(1):3–35.

Starr, D. and Morris, R. (1978). A general theory of combinatoriality and the aggregate, part 2. *Perspectives of New Music*, 16(2):50–84.

Tanaka, T. and Fujii, K. (2014). Melodic pattern segmentation of polyphonic music as a set partitioning problem. In *International Congress on Music and Mathematics, Puerto Vallarta, November 26–29, 2014, Proceedings*. Springer, Berlin, to be published.

Tanaka, T. and Fujii, K. (2015). Describing global musical structures by integer programming. In Tom Collins, David Meredith, and Anja Volk, editors, *Mathematics and Computation in Music: Fifth International Conference, MCM 2015, London, UK, June 22–25, 2015, Proceedings*, volume 9110 of *Lecture Notes in Artificial Intelligence*, pages 52–63. Springer, Berlin, 2015.

Winham, G. (1970). Composition with arrays. *Perspectives of New Music*, 9(1):43–67.

Paper E

Constraint Programming Approach to the Problem of Generating Milton Babbitt's All-partition Arrays

Tanaka, T., Bemman, B., and Meredith D.

The paper has been published in
M. Rueher (ed.) 22nd International Conference on Principles and Practice of Constraint Programming, CP2016, Toulouse, France, September 5–9, 2016 Proceedings, Lecture Notes in Computer Science, Vol. 9892, pp. 802–810, 2016.

© 2016 Springer, Berlin
The layout has been revised.

Abstract

Milton Babbitt (1916–2011) was a composer of twelve-tone serial music noted for creating the all-partition array. One part of the problem in generating an all-partition array requires finding a covering of a pitch-class matrix by a collection of sets, each forming a region containing 12 distinct elements and corresponding to a distinct integer partition of 12. Constraint programming (CP) is a tool for solving such combinatorial and constraint satisfaction problems. In this paper, we use CP for the first time to formalize this problem in generating an all-partition array. Solving the whole of this problem is difficult and few known solutions exist. Therefore, we propose solving two sub-problems and joining these to form a complete solution. We conclude by presenting a solution found using this method. Our solution is the first we are aware of to be discovered automatically using a computer and differs from those found by composers.

1 Introduction

Milton Babbitt (1916–2011) was a composer of twelve-tone serial music noted for developing highly constrained and often complex musical structures. Many of his pieces are organized according to one such structure known as the *all-partition array* (Babbitt, 1973). An all-partition array is a covering of a matrix of pitch-class integers by a collection of sets, each of which forms a region in this matrix containing 12 distinct pitch classes from consecutive elements in its rows and that corresponds to a distinct integer partition of 12 (to be clarified in the next section). This unique structure imposes a strict organization on the pitch classes in his works, and it serves as both a method of musical composition and musical form. Moreover, the all-partition array allowed Babbitt one of many ways to achieve *maximal diversity* in his music.[1]

In this paper, we formulate one part of the problem in generating an all-partition array, beginning from a given matrix of pitch-class integers, using constraint programming (CP) and with a particular focus on its mathematical aspects. Using our model and a method for dividing this matrix into smaller, sub-problems, we obtained a solution, which, we believe, is the first to be discovered automatically using a computer and differs from those found by composers. CP is a programming paradigm that has been successfully applied to the solving of various constraint satisfaction problems in music (Anders et al., 2005; Carpentier et al., 2010; Chemillier and Truchet, 2001; Laurson and Kuuskankare, 2001; Puget and Régin, 2007). It seems natural then, that CP could be used in the problem we address here. Moreover, having such

[1]Maximal diversity is the presentation of as many musical parameters in as many different ways as possible (Mead, 1994).

11	10	3	7	6	2	4	5	8	1	9	0
8	9	4	0	1	5	3	2	11	6	10	7
2	5	1	6	9	10	0	8	7	11	4	3
7	4	8	3	0	11	9	1	2	10	5	6

Fig. E.1: A 4×12 excerpt from a 4×96 pitch-class matrix with two distinct integer partition regions represented precisely by the integer compositions, $\text{IntComp}_{12}(4,4,4,0)$ (in dark gray) and $\text{IntComp}_{12}(0,6,3,3)$ (in light gray), each containing every pitch class exactly once.

a model could, for example, be used as a basis for generating new musical works.

1.1 The structure of an all-partition array

In this section, we describe the structure of an all-partition array in a way that assumes the reader has a basic understanding of pitch class set theory. Constructing an all-partition array begins with the construction of an $I \times J$ matrix, A, whose elements are pitch-class integers, $0, 1, \ldots, 11$, where each row contains $J/12$ twelve-tone rows. The dimensions of this matrix constrain the most important requirement of the structure of an all-partition array, however, Babbitt generally limited himself to sizes of 4×96, 6×96, and 12×72 (Mead, 1994).

In this paper, we consider only matrices where $I = 4$ and $J = 96$, as matrices of this size figure prominently in Babbitt's music (Mead, 1994). This results in a 4×96 matrix of pitch classes, containing 32 twelve-tone rows from the possible 48 related by any combination of transposition, inversion and retrograde (i.e., reversal). In other words, A will contain an approximately uniform distribution of 32 occurrences of each of the integers from 0 to 11.[2] On the musical surface, rows of this matrix become expressed as "musical voices", typically distinguished from one another by instrumental register (Mead, 1994).

A complete all-partition array is a covering of matrix, A, by K sets, each of which is itself a partition of the set $\{0, 1, \ldots, 11\}$ whose parts (1) contain consecutive row elements from A and (2) have cardinalities equal to the summands in one of the K distinct integer partitions of 12 (e.g., $6 + 6$ or $5 + 4 + 2 + 1$) containing I or fewer summands greater than zero.[3] Figure E.1 shows a 4×12 excerpt from a 4×96 pitch-class matrix, A, and two such sets forming *regions* in A each containing every pitch class exactly once and corresponding to two distinct integer partitions, whose exact "shapes" are

[2]For a more detailed description of the constraints governing the organization of matrices in Babbitt's music, see Bemman and Meredith (2016); Mead (1994).

[3]We denote an integer partition of an integer, L, by $\text{IntPart}_L(s_1, s_2, \ldots, s_I)$ and define it to be an ordered set of non-negative integers, $\langle s_1, s_2, \ldots, s_I \rangle$, where $L = \sum_{i=1}^{I} s_i$ and $s_1 \geq s_2 \geq \cdots \geq s_I$.

11	10	3	7	6	2	4	5	8	1	9	0
8	9	4	0	1	5	3	2	11	6	10	7
2	5	1	6	9	10	0	8	7	11	4	3
7	4	8	3	0	11	9	1	2	10	5	6

Fig. E.2: A 4×12 excerpt from a 4×96 pitch-class matrix with a third integer composition, $\text{IntComp}_{12}(5,1,0,6)$ (in medium gray), sharing one pitch class from each of the two previous regions.

more precisely represented as the *integer compositions*, $\text{IntComp}_{12}(4,4,4,0)$ and $\text{IntComp}_{12}(0,6,3,3)$.[4] Note, in Figure E.1, that each summand (from left to right) in $\text{IntComp}_{12}(4,4,4,0)$, gives the number of elements in the corresponding row of the matrix (from top to bottom) in this region. Unlike common tiling problems using, for example, polyominoes, these regions need not have connected interiors, as demonstrated by the second region in Figure E.1 between rows 3 and 4. On the musical surface, the distinct shape of each region helps contribute to a progression of “musical voices" that vary in textural density, allowing for relatively thick textures in, e.g., $\text{IntComp}_{12}(3,3,3,3)$ (with four participating parts) and comparatively sparse textures in, e.g., $\text{IntComp}_{12}(11,0,1,0)$ (with two participating parts).

There exist a total of 34 distinct integer partitions of 12 into 4 or fewer non-zero summands (Mead, 1994). An all-partition array with four rows will thus contain $K = 34$ regions, each containing every pitch class exactly once and each with a distinct “shape" determined by an integer composition defining a distinct integer partition. However, the number of pitch classes required to satisfy this constraint, $34 \times 12 = 408$, exceeds the size of a 4×96 matrix containing 384 elements, by 24. In order to satisfy this constraint, contiguous regions may share pitch classes, with the added constraint that only horizontal *overlaps* of at most one pitch class in each row are allowed for each of the 34 integer partition regions.

Figure E.2 shows a third region, $\text{IntComp}_{12}(5,1,0,6)$ (in medium gray), in the matrix shown in Figure E.1, where two of its elements result from overlapped pitch classes from previous regions. Note, in Figure E.2, the two horizontal overlaps of pitch class, 7 (in row 1 and belonging to the first region) and 8 (in row 4 and belonging to the second region), required to have each pitch class occur exactly once in the third integer partition region. This means that while contiguous regions may share pitch classes, such regions need not be necessarily adjacent in sequence.

Composers have primarily relied on constructing all-partition arrays by hand and at least some of their methods have been published (Babbitt, 1973;

[4] We define an *integer composition* of a positive integer, L, denoted by $\text{IntComp}_L(s_1, s_2, \ldots, s_I)$, to also be an ordered set of I non-negative integers, $\langle s_1, s_2, \ldots, s_I \rangle$, where $L = \sum_{i=1}^{I} s_i$.

Starr and Morris, 1977, 1978). Algorithms for automating this task have also been proposed (Bazelow and Brickle, 1979; Bemman and Meredith, 2016). However, generating an all-partition array is a large combinatorial problem and satisfying the constraints of its structure is difficult. To date, none of these algorithms have been able to solve this problem automatically. This observation motivates our decision here to look for alternative programming paradigms and methods for possibly better addressing this problem. In section 2, we present our CP constraints for implementing the problem of generating an all-partition array from a given matrix. As solving for the entire matrix directly is difficult, in section 3, we present a method of dividing this matrix into two smaller matrices, choosing integer partitions based on how frequently they appear in solutions to one of these smaller matrices, and rejoining them to form a complete solution. We conclude here with a solution discovered using this method.

2 CP constraints for the problem of generating an all-partition array from a given matrix

We begin the discussion of our CP constraints for generating an all-partition array, with a given matrix found in one of Babbitt's works based on the all-partition array.[5]

Let $(A_{i,j})$ be this $(4, 96)$-matrix whose elements are the pitch-class integers, $0, 1, \ldots, 11$. We denote the number of rows and columns by I and J, respectively. Let $x_{i,j,k}$ $(1 \leq i \leq I, 1 \leq j \leq J)$ be a binary variable corresponding to each location (i, j) in A and a subset (i.e., a region) identified by the integer k, where $1 \leq k \leq K$ and $K = 34$. There are then 34 sets of 384 such variables. Each of these variables will indicate whether or not a location (i, j) belongs to a *candidate set*, which we denote, C_k, for the kth position in the sequence of 34 regions.

For C_k to be a candidate set, it must form a region in A (as described in section 1), by satisfying two conditions, *consecutiveness* and *containment*, which we will introduce below. Having satisfied these conditions, C_k will be a candidate set in a possible solution to our problem, in which its elements correspond to 12 distinct pitch classes in A and whose "shape" is defined by an integer composition. Additional constraints e.g., ensuring that each of these candidate sets is then a distinct integer partition and that their overlaps do not exceed one in each row, will then complete our formulation of this problem.

[5]Examples of this matrix can be found in Babbitt's *My Ends are My Beginnings* (1978) and *Beaten Paths* (1988), among others.

2.1 Consecutiveness

The condition of consecutiveness states that pitch classes belonging to the same region and row in A must lie adjacent to one another with no gaps between. We ensure this is the case by placing constraints on the strings of 0's and 1's that are allowed in the rows formed by $\langle x_{i,1,k}, x_{i,2,k}, \ldots, x_{i,J,k} \rangle$ for each (i,k). If, for example, the string $\langle \ldots, 0, 1, \ldots \rangle$ appears in the ith row for some k, then there can be no 1 occurring before 0. This is expressed by the following:

$$\forall i \in [1,I], \forall j \in [3,J], \forall k \in [1,K],$$
$$(x_{i,j-1,k} = 0 \ \wedge \ x_{i,j,k} = 1) \Longrightarrow \bigwedge_{j'=1}^{j-2} (x_{i,j',k} = 0). \tag{E.1}$$

On the other hand, if $\langle \ldots, 1, 0, \ldots \rangle$ appears in this row, then there can be no 1 after 0. This is expressed by the following:

$$\forall i \in [1,I], \forall j \in [1,J-2], \forall k \in [1,K],$$
$$(x_{i,j,k} = 1 \ \wedge \ x_{i,j+1,k} = 0) \Longrightarrow \bigwedge_{j'=j+2}^{J} (x_{i,j',k} = 0). \tag{E.2}$$

In other words, all 1's in $\langle x_{i,1,k}, x_{i,2,k}, \ldots, x_{i,J,k} \rangle$ for each (i,k) must be consecutive, with any 0's lying to the left or right end points of this string.

2.2 Containment

The condition of containment states that regions in A must contain 12 distinct pitch classes. Let B_p $(0 \le p \le 11)$ be the set of all locations (i,j) of pitch class p in matrix A. From this, we can express the condition of containment by the following:

$$\forall p \in [0,11], \forall k \in [1,K], \sum_{(i,j) \in B_p} x_{i,j,k} = 1, \tag{E.3}$$

where for each k, $x_{i,j,k}$ is equal to 1 at one and only one location (i,j) whose pitch class is p in A. When this is the case, C_k will contain one of each pitch class.

2.3 Covering all (i,j) in A

A solution to our problem requires that every one element in A is covered by at least one of the regions, C_k. We can express this condition by the following constraint:

$$\forall i \in [1,I], \forall j \in [1,J], \bigvee_{k=1}^{K} (x_{i,j,k} = 1). \tag{E.4}$$

2.4 Restrictions on the left-to-right order of candidate sets and their overlaps

As discussed in section 1, adjacent regions need not be contiguous in each row in A, however, there are restrictions on their left-to-right order and allowed overlaps. The number of overlaps in each row between these regions must not exceed 1. We can express this restriction by the following constraint:

$$\forall i \in [1, I], \forall j \in [2, J], \forall k \in [1, K-1],$$
$$(x_{i,j,k} = 1) \implies \bigwedge_{k'=k+1}^{K} (x_{i,j-1,k'} = 0). \quad \text{(E.5)}$$

When combined with the constraint of consecutiveness, constraint E.5 means that if $x_{i,j,k}$ is equal to 1, the ith row of $C_{k'}$, whose k' is greater than k, is either (1) located at the right-hand side of (i, j) without overlapping the ith row of C_k or (2) has only one overlap at the right-most element of the ith row of C_k.

2.5 Candidate sets as all different integer partitions

In order to determine that the integer composition "shape" of C_k is a distinct integer partition, we introduce two variables, $y_{i,k,l}$ and $z_{k,l}$. Let $y_{i,k,l}$ be a binary variable that indicates whether or not the length of the ith row of C_k is greater than or equal to l $(1 \leq l \leq L, L = 12)$, by introducing the following two constraints:

$$\forall i \in [1, I], \forall k \in [1, K], \sum_{j=1}^{J} x_{i,j,k} = \sum_{l=1}^{L} y_{i,k,l} \quad \text{(E.6)}$$

$$\forall i \in [1, I], \forall k \in [1, K], \forall l \in [2, L], (y_{i,k,l} = 1) \implies (y_{i,k,l-1} = 1). \quad \text{(E.7)}$$

Equation E.6 states that the sum of all elements in $\langle y_{i,k,1}, y_{i,k,2}, \ldots, y_{i,k,L} \rangle$ is equal to the length of the ith row of C_k while Equation E.7 states that its elements equal to 1 begin in the first position and are consecutive. For example, when the length of the ith row of C_k is 3, $\langle y_{i,k,1}, y_{i,k,2}, \ldots, y_{i,k,L} \rangle$ is $\langle 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$. The total number of rows in C_k whose lengths are greater than or equal to l is given by $\sum_{i=1}^{I} y_{i,k,l}$. Let $z_{k,l}$ $(0 \leq z_{k,l} \leq I)$ be an integer variable that is equal to $\sum_{i=1}^{I} y_{i,k,l}$ $(1 \leq l \leq L)$ with the following constraint:

$$\forall k \in [1, K], \forall l \in [1, L], z_{k,l} = \sum_{i=1}^{I} y_{i,k,l}. \quad \text{(E.8)}$$

The ordered set of twelve values $z_{k,l}$ $(1 \leq l \leq L)$ will then identify the type of integer partition corresponding to C_k. For example, when $z_{k,l}$ is $\langle 4, 4, 2, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$, C_k is $\text{IntPart}_{12}(5, 3, 2, 2)$. We denote this set $z_{k,l}$ corresponding to an integer partition n by $P_n = \langle P_{n,1}, P_{n,2}, \ldots P_{n,L} \rangle$ $(1 \leq n \leq$

$N, N = 34$), where integer partitions appear in reverse lexicographical order, meaning that those containing the fewest parts and largest part lengths appear first. For example, P_1 is $\text{IntPart}_{12}(12, 0, 0, 0)$ and P_{34} is $\text{IntPart}_{12}(3, 3, 3, 3)$. From this, we determine the integer composition shape of C_k to be the integer partition n by the following constraint:

$$\forall k \in [1, K], \forall n \in [1, N], \ (w_k = n) \iff \bigwedge_{l=1}^{L} (z_{k,l} = P_{n,l}), \tag{E.9}$$

where w_k ($1 \leq w_k \leq N$) is an integer variable that indicates to which P_n C_k corresponds. We can now express the condition that all integer partitions are distinct by the constraint, AllDifferent$(w_1, w_2, \ldots, w_K)$.

3 Solution

In order to confirm that our formulation of this problem is accurate, we implemented our constraints described in section 2 and supplied these to a CP solver (Sugar v2-1-0 Tamura and Banbara, 2008). We first tried to solve for the whole matrix directly, however, we were unable to obtain a solution after a day of calculation. We decided instead, to divide the matrix into two, equally-sized halves and try solving for each in such a way that their rejoining would form a complete solution to the original problem. We made this division of the original matrix at $[1, I] \times [1, J/2]$. Columns 1 to $(J/2)$ then correspond to the first smaller matrix we denote by A_1 and columns $(J/2) + 1$ to J correspond to the second smaller matrix we denote by A_2. We allocated $34/2 = 17$ integer partitions to be found in each.

With little modification, our constraints can be adapted to the solving of these sub-problems. These changes include modifying B_p (in equation E.3) to contain only the locations of pitch classes in either A_1 or A_2, setting K to be the new number of partitions in each (i.e., 17) and J to be their new column lengths $96/2 = 48$. Solutions to A_1 and A_2 in which no integer partition is used more than once and contains only pitch classes from one or the other matrix (but not both), collectively form a solution to the original problem. Due to its smaller size, we were able to find solutions beginning with A_1 over the course of a day, in which 506 were found. Naturally, solving for A_1 makes finding a solution in A_2 more difficult as the number of available partitions is now fewer, and in fact, all 506 solutions to A_1 made A_2 unsatisfiable. We noticed, however, that certain partitions in these 506 solutions e.g., $\text{IntPart}_{12}(3, 3, 3, 3)$ and $\text{IntPart}_{12}(4, 3, 3, 2)$ occurred far less frequently than others. It would be reasonable then to conclude that solutions in A_1 which contain the greatest number of these less frequently occurring partitions will make solving for A_2 more likely, as the fewer available partitions in A_2 now consist of a proportionally greater number of frequently occurring partitions.

Therefore, we solved again for A_1, this time by arbitrarily restricting the domain of w_k to exclude the top 6 most frequently occurring partitions and include the top 5 least frequently occurring partitions.

If we denote the subset of integers from $[1, 34]$ corresponding to the partitions found in this solution to A_1, S, then the domain of w_k for possible solutions to A_2 becomes $[1, 34] \setminus S$. We then tried solving for A_2, under the assumption that its proportionally greater number of more frequently occurring partitions would make finding a solution easier. While this means we exclude possible solutions e.g., ones in which a rarely occurring partition occurs in A_2 or where a partition contains pitch classes from both A_1 and A_2, we were able to generate a complete solution in this way. Solving for A_1 took approx. 4 minutes while solving for A_2 took approx. 28 minutes. Table E.1 shows the complete solution found using this method of re-joining A_1 and A_2.

et37	62	4581	90		7	t6e	23510498	-867	2t	e31	-1094
8940	15	32e6t7			859410t23		e67	549	10	-0	8te27365
2516	9t	0	87e4365t	219	e	03847		1t2	9653	784	
	74830e	9	12	t56e07348	6	5291	t	03e	-e478	t6592	
4^3	62^3	641^2	82^2	93	91^3	543	831	3^4	$4^2 2^2$	$53^2 1$	84

85637		-72e	t8	01945	32		7et6890	514	-4e2	-2t36795481
		-58	49013	-3et276	450891et7	26	3		-30	-0
-4e09t	-t5126	430	7e	8	6	95t1	-124	0e3872	16t9578	e
-21	4380e79	-9t16	-625			304e87	5	9t6		(vacant)
$5^2 2$	75	$43^2 2$	532^2	651	921	642	731^2	63^2	732	$10\ 1^2$

023t67e		-e	9		8504	1	2e3t76	48	9501	
9185	42673te10598	-84	67		-7	t3e29	-908145	73	26et	
4		0396t5	-5	21	e3	40785		619t20e	837	4
		127	83e0421t	-t5964738e0	2t916	-6		5	4	380e79t1625
741	12	6321	821^2	10 2	5421	$5^2 1^2$	6^2	$72^2 1$	$4^2 31$	11 1

Table E.1: A generated all-partition array corresponding to a complete solution to our problem, represented in the way used by music theorists (Mead, 1994). Each column contains the elements in A belonging to C_k, where a dash indicates those that overlap. Note, that partitions are denoted using a shorthand notation, e.g., 4^3, where the base indicates the length of a part and the exponent denotes its number of occurrences. For clarity, the integers 10 and 11 have been replaced by the letters t and e, respectively.

4 Conclusion

In this paper, we have introduced a novel formulation of one part of the problem of generating an all-partition array, beginning from a given matrix, using constraint programming (CP). Solving for the whole of this matrix directly proved too difficult using our constraints. Therefore, we introduced a method

of dividing the matrix into two halves, solving for each and then re-joining them to form a complete solution. Using this method, we were able to discover a solution. This solution is the first we are aware of to be automatically generated by a computer. Moreover, it is an all-together new all-partition array from those previously discovered by Babbitt and other composers. In future work, we hope to examine in more detail how to make finding solutions in larger matrices possible and without excluding potential solutions.

Acknowledgments.

The work of Tsubasa Tanaka reported in this paper was supported by JSPS Postdoctoral Fellowships for Research Abroad. The work of Brian Bemman and David Meredith was carried out as part of the project Lrn2Cre8, which acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET grant number 610859.

References

Anders, T., Anagnostopoulou, C., and Alcorn, M. (2005). Strasheela: Design and usage of a music composition environment based on the OZ programming model. In Roy, P., editors, *Second International Conference, MOZ 2004*, volume 3389 of *Multiparadigm Programming in Mozart/OZ*. Springer-Verlag, Berlin.

Babbitt, M. (1960). Twelve-tone invariants as compositional determinants. *Journal of Music Theory*, 46(2):246–259.

Babbitt, M. (1961). Set structure as a compositional determinant. *Journal of Music Theory*, 5(1):72–94.

Babbitt, M. (1962). Twelve-tone rhythmic structure and the electronic medium. *Perspectives of New Music*, 1(1):49–79.

Babbitt, M. (1973). Since Schoenberg. *Perspectives of New Music*, 12(1/2):3–28.

Babbitt, M. (2003). *The Collected Essays of Milton Babbitt*. Princeton University Press, Princeton, NJ.

Bazelow, A. R. and Brickle, F. (1976). A partition problem posed by Milton Babbitt (Part I). *Perspectives of New Music*, 14(2):280–293.

Bazelow, A. R. and Brickle, F. (1979). A combinatorial problem in music theory: Babbitt's partition problem (I). *Annals of the New York Academy of Sciences*, 319(1):47–63.

Bemman, B. and Meredith, D. (2015a). Comparison of heuristics for generating all-partition arrays in the style of Milton Babbitt. In *CMMR 11th International Symposium on Computer Music Multidisciplinary Research, 16–19 June 2015*, Plymouth, UK.

Bemman, B. and Meredith, D. (2015b). Exact cover problem in Milton Babbitt's all-partition array. In Collins, T., Meredith, D., and Volk, A., editors, *Mathematics and Computation in Music: Fifth International Conference, MCM 2015, London, UK, June 22–25, 2015, Proceedings*, volume 9110 of *Lecture Notes in Artificial Intelligence*, pages 237–242. Springer, Berlin.

Bemman, B. and Meredith, D. (2016). Generating Milton Babbitt's all-partition arrays. *Journal of New Music Research*, 45(2):1–21.

Bernstein, Z. (2014). The problem of completeness in Milton Babbitt's music and thought. In *43rd Annual Meeting of The Music Theory Society of New York State (5–6 April 2014)*, New York, NY.

Carpentier, G., Assayag, G., and Saint-James, E. (2010). Solving the musical orchestration problem using multiobjective constrained optimization with a genetic local search approach. *Heuristics*, 16(5):681–714.

Chemillier, M. and Truchet, C. (2001). Two musical CSPs. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition.

Eger, S. (2013). Restricted weighted integer compositions and extended binomial coefficients. *Journal of Integer Sequences*, 16(13.1.3):1–25.

Kowalski, D. (1985). *The array as a compositional unit: A study of derivational counterpoint as a means of creating hierarchical structures in twelve-tone music*. Ph.D. diss., Princeton University.

Kowalski, D. (1987). The construction and use of self-deriving arrays. *Perspectives of New Music*, 24(1/2):296–361.

Laurson, M. and Kuuskankare, M. (2001). A constraint based approach to musical textures and instrumental writing. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*.

Morris, R. (1987). *Composition with Pitch-Classes: A Theory of Compositional Design*. Yale University Press, New Haven, CT.

Mead, A. (1994). *An Introduction to the Music of Milton Babbitt*. Princeton University Press, Princeton, NJ.

Morris, R. (2010). *The Whistling Blackbird: Essays and Talks on New Music*. The University of Rochester Press, Princeton, NJ.

Orman, A. J. and Williams, H. Paul. (2006). A survey of different integer programming formulations of the travelling salesman problem. In Kontoghiorghes, Erricos John and Gatu, Cristian, editors, *Optimisation, Econometric and Financial Analysis*, volume 9110 of *Advances in computational management science*, pages 93-108. Springer, Berlin.

Page, D. (2013). Parallel algorithm for second-order restricted weak integer composition generation for shared memory machines. *Parallel Processing Letters*, 23(3):1350010.

Puget, J. F. and Régin, J C. (2007). Solving the all interval problem. Technical report. Available online at `https://ianm.host.cs.st-andrews.ac.uk/CSPLib/prob/prob007/puget.pdf`.

Starr, D. and Morris, R. (1977). A general theory of combinatoriality and the aggregate, part 1. *Perspectives of New Music*, 16(1):3–35.

Starr, D. and Morris, R. (1978). A general theory of combinatoriality and the aggregate, part 2. *Perspectives of New Music*, 16(2):50–84.

Tamura, N. and Banbara, M. (2008). Sugar: A CSP to SAT Translator Based on Order Encoding, in *Proceedings of the 2nd International CSP Solver Competition*, 65–69.

Tanaka, T. and Fujii, K. (2015). Describing global musical structures by integer programming. In Collins, T., Meredith, D., and Volk, A., editors, *Mathematics and Computation in Music: Fifth International Conference, MCM 2015, London, UK, June 22–25, 2015, Proceedings*, volume 9110 of *Lecture Notes in Artificial Intelligence*, pages 52–63. Springer, Berlin.

Tanaka, T. and Fujii, K. (2014). Melodic Pattern Segmentation of Polyphonic Music as a Set Partitioning Problem. *International Congress on Music and Mathematics, ICMM 2014, Puerto Vallarta, Mexico, June 22–25, 2015, Proceedings*, Springer, Berlin (to be published).

Tani, N. B. and Bouroubi, S. (2011). Enumeration of the partitions of an integer into parts of a specified number of different sizes and especially two sizes. *Journal of Integer Sequences*, 14(11.3.6):1–12.

Winham, G. (1970). Composition with arrays. *Perspectives of New Music*, 9(1):43–67.

Paper F

Generating New Musical Works in the Style of Milton Babbitt

Bemman, B. and Meredith, D.

The paper has been submitted to the
Computer Music Journal, 2016.

Currently in review
The layout has been revised.

Abstract

Milton Babbitt (1916–2011) was a composer of twelve-tone and serial music whose works and theoretical writings had a profound impact on the composition of modern academic music. In this paper, we first review in detail his compositional process and the techniques he developed, focusing in particular on the all-partition array, time-point rows, and the equal-note-value-strings found in his later works. Next, we describe our proposed procedure for automating his compositional process using these techniques. We conclude by using our procedure to automatically generate an entirely new musical work that we argue is in the style of Babbitt.

1 Introduction

Milton Babbitt (1916–2011) was a composer of twelve-tone and serial music whose works and theoretical writings had a profound impact on modern academic music. Beginning in the 1950s and over the course of two decades, Babbitt formalized the 12-tone system and established techniques, such as the time-point row and the all-partition array (Babbitt, 1955, 1960, 1961, 1962, 1973). Many of these techniques remain of interest to composition and music research today (Bemman and Meredith, 2015a,b; Bernstein, 2014). Indeed, music theorists have written at length on these techniques and the highly constrained and often complex structures in Babbitt's music that result (Bemman and Meredith, 2016; Mead, 1994). In recent years, the sketches for many of his works have been made publicly available by the Library of Congress in Washington, DC.[1] Researchers are now able to examine Babbitt's compositional process in much greater detail than before.

In this paper, we describe the process Babbitt devised in composing his later works (approx. 1980–2011), focusing in particular on the *all-partition array*, *time-point row*, and *equal-note-value string*. The constraints under which he composed his music at this time are strict and his application of these techniques follows an often well-defined procedure. For this reason, we suggest that Babbitt's compositional process is inherently algorithmic in nature and that, therefore, many of these techniques can be modeled by machine. This line of reasoning follows years of research in the field of computational music analysis and generation, with similar efforts having been made to, for example, harmonize chorales in the style of Bach (Ebcioğlu, 1987) and more recently, generate the structures found in Babbitt's own music (Bemman and Meredith, 2016; Tanaka et al., 2016a,b).

In the remainder of this paper, we present our proposed procedure for automating Babbitt's compositional process in his later works. First, we introduce a method for generating the time-point and pitch-class rows of a

[1] `https://lccn.loc.gov/2014565648`. Last accessed 2016-11-15.

1.	7 2		2 3 1 9 8 5
2.	6		6 11 10 0
3.	11	0 5 7	
4.	8 3 4	2 9 10	
5.	5 10 9	11 3 4	4 7
6.	0 1	1 6 8	

Fig. F.1: Excerpt containing three ordered mosaics from the all-partition array found in Babbitt's *None but the Lonely Flute*.

piece from its all-partition array. Then, we show how this same method can be used to generate pitch-class rows containing repetitions. Next, we describe a method for determining the rhythms in a piece as well as the placement of rests and ties. We conclude by using this procedure to automatically generate a novel musical work that we believe to be in the style of Babbitt.

1.1 All-partition array

Both the pitch and rhythmic content in Babbitt's later works are organized according to a structure known as the *all-partition array* (Bemman and Meredith, 2016; Mead, 1994). In an all-partition array, Babbitt constructed aggregates (i.e., collections of the 12 musical pitch classes) so that each was a distinct set of partitioned pitch-class segments called an *ordered mosaic*. For example, $\langle\langle 2,0,1,3\rangle,\langle 5,4,6,7\rangle,\langle 8,9,11,10\rangle\rangle$ is one ordered mosaic made up of three pitch-class segments of length four.[2] A second ordered mosaic might contain three pitch-class segments in which two are of length five and one is of length two, e.g., $\langle\langle 6,1,11,4,2\rangle,\langle 5,7\rangle,\langle 0,9,8,3,10\rangle\rangle$. Each pitch-class segment in an ordered mosaic is then assigned to a "voice" on the musical surface, forming what is known as a *lyne* (Mead, 1994). Collectively, these pitch-class segments in each lyne for all ordered mosaics form a concatenation of 12-tone rows that may or may not contain repetitions. Figure F.1 shows an excerpt from the all-partition array in Babbitt's *None but the Lonely Flute* with six such lynes and the first three of its ordered mosaics.

Note, in Figure F.1, that the pitch-class segments in the first ordered mosaic are distributed across each of the six possible lynes but that in the third ordered mosaic, these pitch-class segments appear in only three lynes, 1, 2 and 5. In the third ordered mosaic, the first pitch class in each segment is a repetition from the previous ordered mosaics: pitch-classes 2 and 6 from the first ordered mosaic and pitch-class 4 from the second ordered mosaic. An all-partition array must contain a number of ordered mosaics equal to

[2] We denote ordered sets using angle brackets, $\langle . \rangle$, and unordered sets with braces, $\{.\}$.

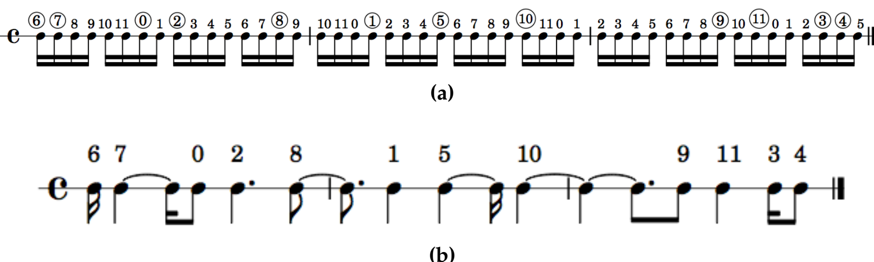


Fig. F.2: Members of a time-point row in (a) set against a grid of sixteenth-note units and one possible rhythmic interpretation of this row in (b) where duration is equal to inter-onset interval.

the number of distinct ways in which 12 can be partitioned into *k* parts (i.e., lynes) or fewer, with each of these partitions represented by a distinct ordered mosaic. Babbitt used 4, 6, and 12-part all-partition arrays, containing, respectively, 34, 58, and 77 ordered mosaics.

1.2 Time-point row

Babbitt himself laid the foundations for his time-point system and outlined general principles for applying it to composition (Babbitt, 1962). In a time-point row, Babbitt sought to derive a correspondence between the 12-tone row and time. By replacing the interval of a half-step in a 12-tone row with a fixed period of time called a *unit*, the time-point intervals (analogous to directed pitch-class intervals) between adjacent members become lengths of time measured in units rather than pitch intervals measured in semitones. In his later works, Babbitt typically used a sixteenth note as the unit (Bernstein, 2014). Figure F.2 shows an example of a time-point row using a sixteenth note as the unit and one possible rhythmic representation.

Note, in Figure F.2, how time points in a row denote onsets in time corresponding to new rhythmic events. In Figure F.2(b), these rhythmic events have a duration equal to the inter-onset intervals of each time point. However, Babbitt often sought less straight-forward rhythmic interpretations for his time-point rows than that shown in Figure F.2(b), and in the following sections we will see in greater detail how his use of additional techniques gave his later works their characteristically varied and often complex rhythms.

2 Babbitt's compositional process in his later works

In this section, we describe in detail Babbitt's compositional process as found in his later works. Many of the techniques he used in this process have

$$\text{C}_6 - \text{B}_6 \left\{ \begin{array}{|lll|} 7 & 2 & \\ 6 & & \\ \end{array} \right.$$
$$\text{C}_5 - \text{B}_5 \left\{ \begin{array}{|lll|} 11 & & \\ 8 & 3 & 4 \\ \end{array} \right.$$
$$\text{C}_4 - \text{B}_4 \left\{ \begin{array}{|lll|} 5 & 10 & 9 \\ 0 & 1 & \\ \end{array} \right.$$

(a)

ff	7	2	
f	6		
mf	11		
mp	8	3	4
p	5	10	9
pp	0	1	

(b)

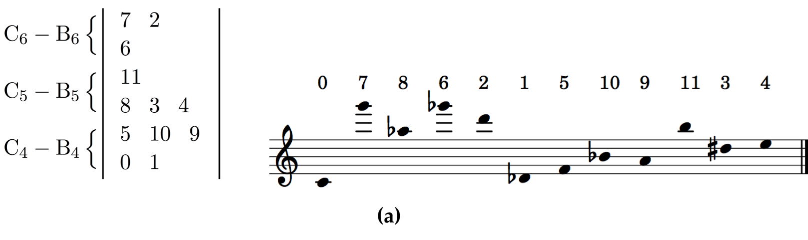
Fig. F.3: Pitch-class ordered mosaic (PcOM) distinguished by register in (a) and time-point ordered mosaic (TpOM) distinguished by dynamic level in (b).

been described elsewhere (Bernstein, 2014; Mead, 1994), however, a thorough understanding of how exactly this process comes to form the musical surface of his works is essential in explaining how we have automated this process.

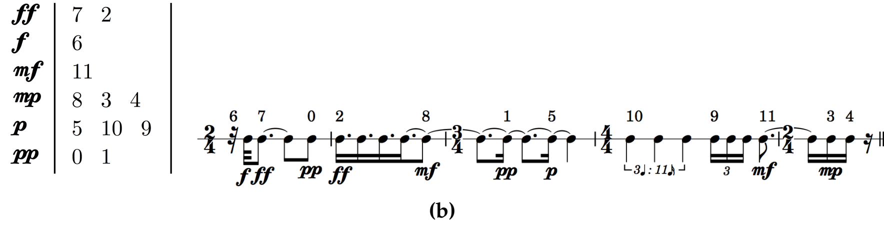
2.1 Rows from ordered mosaics

In many of Babbitt's later works based on the all-partition array, the available time-point rows are the same as the available pitch-class rows because they are constructed from the same ordered mosaics. Typically, Babbitt used register to distinguish pitch-class segments and dynamic level to distinguish time-point segments. Figure F.3 shows a pitch-class ordered mosaic (hereafter abbreviated "PcOM") in (a) and a time-point ordered mosaic (hereafter abbreviated "TpOM") in (b) from an all-partition array. An ordered mosaic places constraints on the possible rows that can be constructed from it. Each segment in a mosaic is ordered, meaning that elements, when "linearized" to form a row, e.g., in Figure F.2(b), must remain in the left-to-right order in which they occur in their segment. For example, in Figure F.3, possible rows from both mosaics may begin with $\langle 7, 2, \ldots \rangle$ but not $\langle 2, 7, \ldots \rangle$. Similarly, elements from other segments may intervene, so long as their left-to-right order is not violated. For example, $\langle 7, 8, 2, 3 \ldots \rangle$ is allowed but not $\langle 7, 3, 2, 8 \ldots \rangle$. Figure F.4 shows a possible pitch-class row, (a), and a possible time-point row, (b) taken from the ordered mosaics in Figure F.3(a) and (b), respectively.

Note, in Figure F.4(a), how segments of pitch classes are distinguished from each other by pitch-register and in (b), how the changes in dynamic level in a time-point row mark the arrival of a time point belonging to a different segment. Perceptually, these new time points act as both temporal boundaries of local events and reminders of a global stream of temporal events unfolding in each time-point segment, made clear by differing dynamic levels.



(a)



(b)

Fig. F.4: Opening pitch-class row in (a) and time-point row in (b) from Babbitt's *None but the Lonely Flute* (right) taken from their respective ordered mosaics (left). Note, in (a) that no time information is specified and in (b) that no pitch information is specified.

The musical surface of Babbitt's later works is formed by uniting the pitch information specified by a row constructed from a PcOM with the timing information specified by some rhythmic interpretation of a row constructed from a TpOM. Figure F.5 shows the opening of Babbitt's *None but the Lonely Flute* and how the pitch-class row shown in Figure F.4(a) and the time-point row shown in Figure F.4(b) have been united to form the musical surface.
Note, in Figure F.5, that on the musical surface, depending on the chosen rhythmic interpretation of its time-point row, pitch classes in a row generally proceed faster than the time points in a row (Mead, 1994). For example, at time point 2 there are four corresponding pitch classes, 8, 6, 2, and 1. However, Babbitt is often careful to ensure that time points are allowed to "catch up" as, for example, at time points 8, 1 and 5 where there is only a single pitch class, 1. Indeed, Mead has noted that Babbitt has had a "longtime predilection for manifesting similar sorts of distributions of events in different domains over different spans of time" (Mead, 1994, p. 49). It is clear then that the chosen rhythmic interpretation of a time-point row, as shown, for example, in Figure F.2(b), is crucial to maintaining such a uniform distribution of pitch and time events.

2.2 Equal-note-value strings and rhythm

A rhythmic interpretation of a time-point row, as shown, for example in Figure F.2(b), is determined by Babbitt's choice of *equal-note-value strings* taken

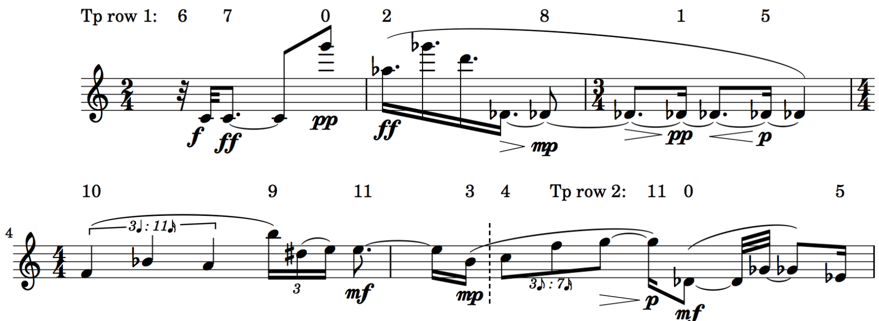


Fig. F.5: Opening of Babbitt's *None but the Lonely Flute* corresponding to the pitch-class and time-point rows shown in Figure F.4. Note, that the vertical dashed line in bar 5 marks the boundary between the first and second pitch-class rows.

from a PcOM to form a pitch-class row. An equal-note-value string is a string of n pitch classes that subdivide a time-point interval into n equal durations (i.e., note values) (Bernstein, 2014; Mead, 1994). Figure F.6 provides examples of Babbitt's use of equal-note-value strings from the opening of *None but the Lonely Flute*.

As shown by the dotted line in Figure F.6, an equal-note-value string containing the pitch classes 8, 6, 2 and 1 equally subdivides the time-point interval of 6, occurring between time points 2 and 8. Because the unit in this time-point row is equal to a sixteenth note, this time-point interval of 6 is equal to a dotted quarter note. This interval has then been equally divided by each of its equal-note-value string's four pitch classes into four durations equal to a dotted sixteenth note. In fact, each time-point interval in the example shown in Figure F.6 contains an equal-note-value string. For example, there is an equal-note-value string of length 1 between time points 0 and 2 (pitch-class 7) and between time points 7 and 0 (pitch-class 0). Between time points 6 and 7 there is an equal-note-value string of length 2 in which one member is a rest. We will see in the following sections, in our discussion of automating Babbitt's compositional process, how exactly Babbitt determined the rests, ties and repetitions of pitch classes that appear frequently in his music.

3 Generating rows from ordered mosaics

In this section, we begin the discussion of our proposed procedure for modeling Babbitt's compositional process described above. In computer science, a *stack* is an abstract data type that stores a collection of elements using a "last-in-first-out" protocol (Cormen et al., 2009, pp. 232–236). A stack has two

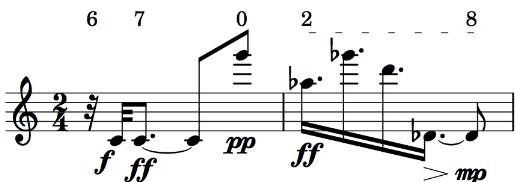


Fig. F.6: Equal-note-value strings in an excerpt from the opening of Babbitt's None but the Lonely Flute shown in Figure F.5.

associated operations, `pop()` and `push()`, that, respectively, remove its top element and insert an element at the top of the stack. By representing each segment of an ordered mosaic as a stack, such that the left-most element lies on the top of each stack, we can account for the left-to-right order in which elements in a time-point or pitch-class row must be taken from each segment. Figure F.7 shows how a time-point or pitch-class row can be generated from an ordered mosaic represented as a sequence of stacks.

Note, in Figure F.7, that at each step from (a) to (c), the top element of each indicated stack is popped and stored from left-to-right to form the sequence of elements above. As this process unfolds, the stack number is simultaneously stored below its corresponding element. In (d), this process concludes with the completed time-point row from Figure F.4(b) and an empty sequence of stacks. Because only the top element of each stack after every chosen element is a possible next choice, the sequence of stack numbers shown in Figure F.7, $\langle 2,1,6,1,4,6,5,5,5,3,4,4 \rangle$, uniquely encodes its corresponding row above. Computing all distinct permutations of such a sequence of stack numbers then corresponds to all possible rows that can be generated from a given ordered mosaic.

3.1 Pitch-class repetitions in rows containing equal-note-value strings

That there are pitch-class repetitions in Babbitt's music distinguishes it from the works of several other 12-tone composers. In looking at Babbitt's later works, we find two sorts of repetition, those in which a pitch class is immediately repeated, e.g., $\langle 6,6 \rangle$, and those in which the most recently chosen pitch class from a stack is repeated, e.g., $\langle 6,7,6 \rangle$ in the ordered mosaics shown in Figure F.3. When and where these repetitions occur in a pitch-class row are determined, in part, by the equal-note-value strings used to construct the row. Figure F.8 shows the process of constructing a pitch-class row with repetitions, this time, by grouping pitch classes from its PcOM into equal-note-value strings.

Note, in Figure F.8, that pitch classes in a PcOM can be in any one of three

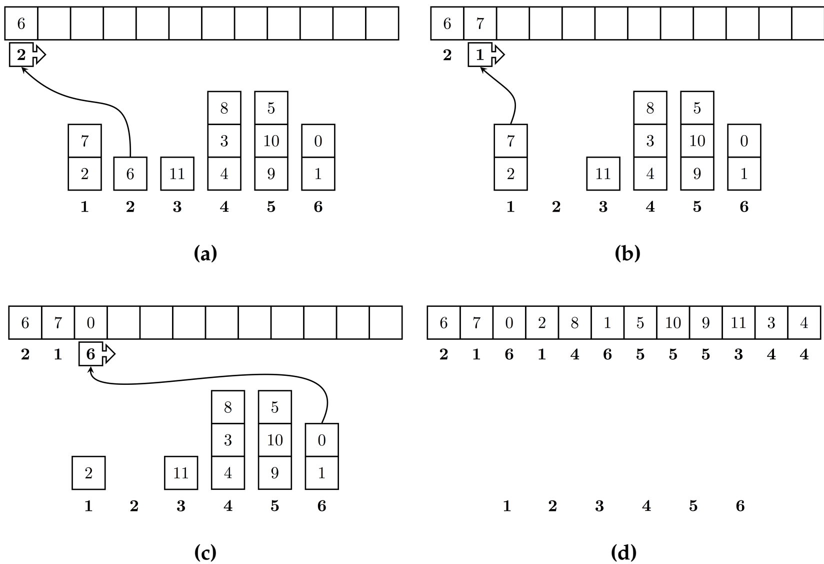


Fig. F.7: Process of selecting elements from stacks (a–c), numbered 1–6, in an ordered mosaic to generate a pitch-class or time-point row until all stacks are empty (d). Note that the generated row corresponds to the time-point row shown in Figure F.4(b).

states in the process of constructing a pitch-class row: (1) unused (indicated by the white boxes), (2) repeatable (indicated by the shaded boxes) or (3) used (indicated by their removal from a stack). In Figure F.8(a), after the first equal-note-value string is generated, the repeatable pitch classes are 6 and 2, but not 7, as 2 and 7 belong to the same segment and 7 is not the most recent pitch class to be taken from this segment. In Figure F.8(b), this equal-note-value string contains two repeatable pitch classes, one of which is an immediate repetition, 6 (from Figure F.8(a)). In Figure F.8(c), this equal-note-value string contains only a single repeatable pitch class, 0, that is not immediate. In Figure F.8(d), we have a completed pitch-class row containing four equal-note-value strings and 16 total pitch classes due to its four repetitions.

3.2 When and where pitch-class repetitions occur

The problem of determining where exactly pitch-class repetitions may occur in Babbitt's music is addressed by Babbitt himself, who states that "pitch repetition is not a pitch procedure, but a temporal procedure, independent of the considerations of the pitch system, and, if a time-point system is assumed, the temporal placements of such pitch repetitions are determined by the time-point structure, not by pitch considerations" (Babbitt, 1962, p. 65). It

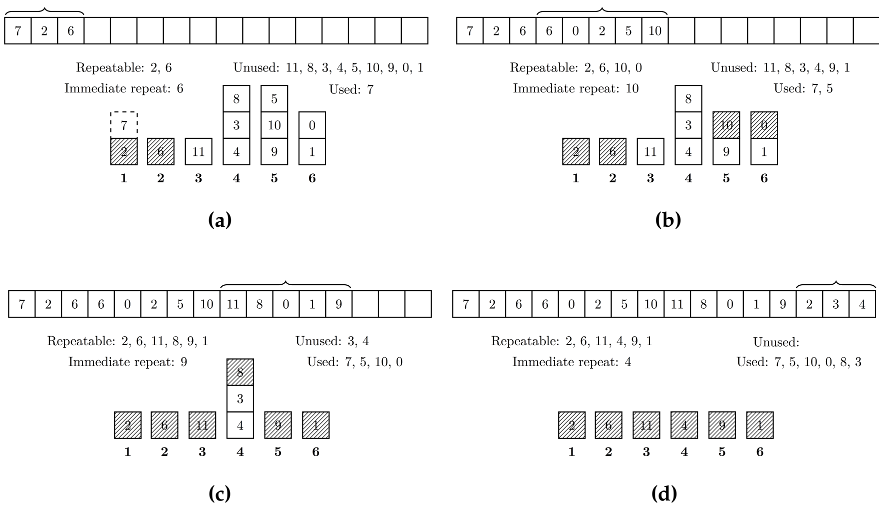


Fig. F.8: Process of generating a pitch-class row of equal-note-value strings containing repetitions from stacks (a–d), numbered 1–6, in a PcOM. Shaded boxes indicate pitch classes that are repeatable while white boxes indicate unused pitch classes.

is likely that Babbitt used repetitions to articulate any number of temporal events, including beat, meter and syncopation, among others. In our model, however, immediate repetitions are those that occur predominantly on the beat while all other repeatable pitch classes will primarily occur off the beat.

Let's suppose in a piece we have a sequence of time points, $P = \langle p_1, p_2, \ldots, p_n \rangle$. The time-point intervals of n time points form the sequence, $T = \langle t_1, t_2, \ldots, t_{n-1} \rangle$, where $t_i = p_{i+1} - p_i \pmod{12}$. Whether or not a time point, p_i, falls on the beat in an implied meter, is determined by the *prefix sum* of time-point interval, t_{i-1}, modulo the number of time-point units, u, between consecutive beats. We therefore define o_i to be 1 if and only if p_i falls on a beat, as follows:

$$o_i = \begin{cases} 1, & \text{if } \sum_{j=1}^{i-1} t_j \equiv 0 \pmod{u}\text{; and} \\ 0, & \text{otherwise.} \end{cases} \tag{F.1}$$

In Equation F.1, when $o_i = 1$, the equal-note-value string at time point, p_i, for time-point interval, t_{i-1}, falls on the beat. For example, in a common-time meter with four sixteenth note units to the beat and where $\sum_{j=1}^{i-1} t_j = 4$, a time-point interval, t_{i-1}, modulo 4 would equal 0, and thus its time point p_i falls on the beat.

As shown in Figure F.8, possible pitch-class events in an equal note-value

		ir	or	np
$o_i =$	1	50%	25%	25%
	0	0%	25%	75%

Table F.1: Probabilities of pitch-class events occurring in the music generated by our model.

string are (1) an immediate repetition (ir), (2) some other repetition (or), and (3) a new pitch class (np). In order to promote a variety of musically interesting events in the music generated by our model, we assign varying probabilities to these possible events at each given time-point interval. We have chosen these probabilities based on observations in Babbitt's music of the approximate number of times these musical events occur. These probabilities are summarized in Table F.1. In the case that the chosen pitch-class event is either some other repetition or a new pitch class and there is more than one available in an ordered mosaic (as there is, for example, in Figure F.8(b)), we ensure that each available pitch class has an equal probability of being chosen. Naturally, there is only ever one available repetition that can be immediate. In Babbitt's later works, all the pitch classes in a given equal-note-value string must be distinct, implying that only the first pitch class in such a string can be an immediate repeat. The remaining pitch classes in such a string can be either other repeats or new pitch classes.

In general, however, Babbitt constructed strings containing typically only two repetitions. We suspect this ensured that the lengths of pitch-class rows on the musical surface do not grow exceedingly long with repetitions. Accordingly, the strings generated by our model are constrained to contain only two repetitions.

3.3 Maximum length of an equal-note-value string

As illustrated by the opening of Babbitt's *None but the Lonely flute*, shown in Figure F.5, the juxtaposition of different length equal-note-value strings and time-point intervals contribute to making a musically interesting result in which the distributions of time points and pitch classes can remain approximately uniform. It seems, however, that Babbitt did not consider all combinations of string length and time-point intervals to be musically meaningful and, indeed, avoided certain combinations altogether. Consider, for example, how difficult it might be for a human to perform a 12-note tuplet in the time of a sixteenth note at even a moderate tempo. To avoid such problems, for any given time-point interval we constrain the maximum length of its corresponding equal-note-value string.

The maximum allowable length for a given equal-note-value string cannot (1) exceed the number of unused pitch classes in its PcOM at any given point in constructing a pitch-class row (i.e., the white boxes shown in the process

described in Figure F.8) and (2) result in subdivisions with a duration smaller than some fixed note value. The maximum length of an equal-note-value string, s_i, for a time-point interval, t_i, is given by

$$|s_i|_{max} = \min(r, t_i d) \quad (F.2)$$

tp unit	d
32nd note	1
sixteenth note	2
eighth note	4

where d is the length of the time-point unit in 32nd notes and r is the number of unused time points in the TpOM before constructing s_i. For example, when the unit in a time-point row is equal to a sixteenth note, meaning $d = 2$, and the current time-point interval is 3, then an equal-note-value string of length $3 \cdot 2 = 6$ is acceptable in a PcOM with 6 or more unused pitch classes. Each of the durations in this equal-note-value string would then be equal to a 32nd note.

4 Generating rests and ties using equal-note-value strings

As illustrated in Figure F.1, pitch classes in a lyne and, by extension, their corresponding segments from all PcOMs in an all-partition array are ordered as a result of the 12-tone rows to which they belong. However, in Figure F.8, we saw how equal-note-value strings generated from a PcOM can contain pitch classes not necessarily belonging to a single segment or lyne. It is therefore possible that such strings may contain an ordering of pitch classes not found in any of the 12-tone rows in the lynes of an all-partition array. Generally speaking, Babbitt found this undesirable as it violates an essential principle of 12-tone composition—namely, that pitch classes from a 12-tone row appear in their given order.

In Babbitt's later works, he ensured that the ordering of pitch classes in an equal-note-value string corresponds to the orderings found in these 12-tone rows by checking that either the string in its entirety or its substrings belong to one or more segments from other PcOMs in its all-partition array. In this way, Babbitt was able to use equal-note-value strings to create a dense network of motivic ideas across a piece by linking different ordered mosaics in an all-partition array (Bernstein, 2014; Mead, 1994). This process of constructing equal-note-value strings that can also be constructed from other PcOMs has been called *referenced array segments* (Bernstein, 2014; Mead, 1994).

The *array segments* of an all-partition array are the pitch-class segments found in all the PcOMs in an all-partition array. For example, the excerpt from Figure F.1 contains one pitch-class segment of length 6, one segment of length 4, six segments of length 3, three segments of length 2 and two

1.	7 2		2 3 1 9 8 5
2.	6		6 11 10 0
3.	11	0 5 7	
4.	8 3 4	2 9 10	
5.	5 10 9	11 3 4	4 7
6.	0 1	1 6 8	

Fig. F.9: An equal-note-value string, $\langle 6, 11 \rangle$ constructed from two segments in the first PcOM that references a single segment, $\langle 6, 11, 10, 0 \rangle$, belonging to the third PcOM.

segments of length 1. A complete all-partition array, containing 34, 58 or 77 ordered mosaics, will have many more segments. An equal-note-value string, s, is said to *reference* an array segment, a, (typically in another PcOM) if a substring of s is a substring of a.

For example, Figure F.9 shows the excerpt from the all-partition array shown in Figure F.1 with an equal-note-value string constructed from two segments in its first PcOM that references a segment in its third PcOM. Note, in Figure F.9, how the referenced array segment, $\langle 6, 11, 10, 0 \rangle$ in the third PcOM contains a substring equal to the equal-note-value string, $\langle 6, 11 \rangle$, constructed from the first PcOM. Moreover, in both PcOMs, the process of selecting elements (shown in Figure F.7) to form this substring is not violated.

When constructing an equal-note-value string in this way, we argue Babbitt sought to minimize the number of referenced array segments required to account for all its pitch classes. More formally, we propose that Babbitt desired a minimum cardinality set, C, of substrings from an equal-note-value string, s, that (1) covers s and (2) whose members, c_i, are substrings of array segments other than those used to construct s. If we return again to Figure F.9, the 6 in our equal-note-value string, $\langle 6, 11 \rangle$, could have referred to the segment, $\langle 6, 11, 10, 0 \rangle$ in the third PcOM and the 11 could have referred to the segment, $\langle 11, 3, 4 \rangle$, in the second. However, this would require a set, $C = \langle \langle 6 \rangle, \langle 11 \rangle \rangle$, having a cardinality of 2, which would therefore be considered less optimal than our original reference to $\langle 6, 11, 10, 0 \rangle$.

4.1 Rests and ties

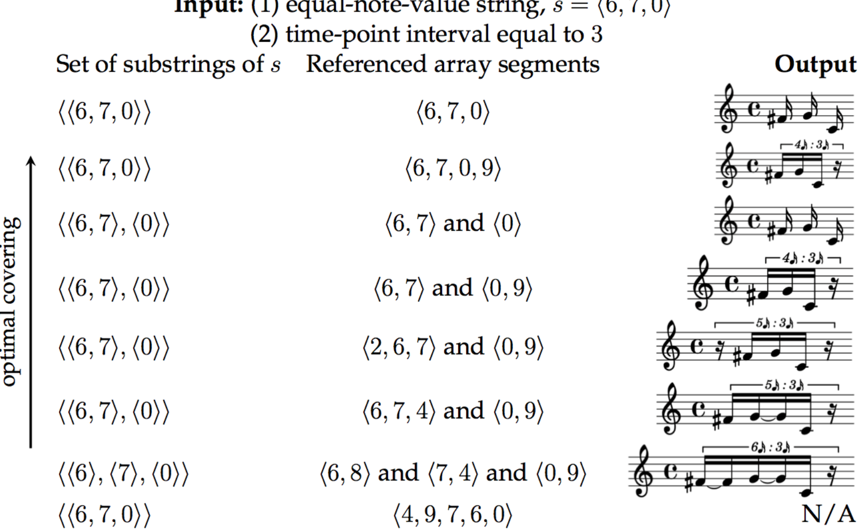
It is possible that many equal-note-value strings may not have a minimum cardinality set. It is also possible that a referenced array segment may contain more pitch classes than its equal-note-value string (as shown in Figure F.9) whether it has a minimum cardinality set or not. It is in these cases that rests and ties arise. As Bernstein (2014) has noted, Babbitt indicated in his sketches the ordinal positions of pitch classes from a string in their referenced array

segments. Because each reference must be a substring, numbers belonging to the same array segment must be sequential in ascending order. For example, an equal-note value string, $\langle 6,11,7\rangle$, would have the ordinal positions $1,2,3$ in a reference to a single array segment $\langle\langle 6,11,7\rangle\rangle$ or $1,2,1$ in a reference to two array segments, e.g., $\langle\langle 6,11\rangle,\langle 7,9,2,10\rangle\rangle$. Babbitt indicated the lengths of each referenced array segment by noting their final ordinal positions with either an underscore or parentheses (Bernstein, 2014). In the example using two array segments just provided, these lengths could be indicated by the following: $1,\underline{2},(1)$.

As Bernstein (2014) notes, an underscore indicates that this final pitch class of an array segment should sound on the musical surface while parentheses indicate that this pitch class should not. We take this to mean that parentheses indicate rests and underscores, in general, indicate ties. We've observed that Babbitt does not typically embed rests in the middle of an equal-note-value string in his later works, opting instead to append or prepend them to a string. On the other hand, while ties do appear at the ends of equal-note-value strings, we only permit them to appear inside a string in our model. For example, an equal-note-value string, $\langle 6,11\rangle$, that references the array segment, $\langle 6,11,2\rangle$, would contain a rest in its final position. This then transforms the original string of length 2 into one of length 3, $\langle 6,11,\text{rest}\rangle$. On the other hand, a tie in our model can only occur from a referenced array segment containing more pitch classes than the substring from its equal-note-value string. For example, an equal-note-value string, $\langle 6,11\rangle$, that references the two array segments, $\langle 6,7\rangle,\langle 11\rangle$ would contain a tie in its second position, $\langle 6,\text{rest},11\rangle$. Table F.2 shows various referenced array segments and the musical output for a given input of a string, $\langle 6,7,0\rangle$, and a time-point interval equal to 3 sixteenth notes.

Note, in Table F.2, how the minimum cardinality set, $\langle\langle 6,7,0\rangle\rangle$, for the referenced array segment, $\langle 6,7,0\rangle$, in the first row, is optimal and its corresponding musical output contains no rests or ties. In the second row, its referenced array segment, while forming a minimum cardinality set, nonetheless contains more pitch classes than its equal-note-value string. As such, the corresponding output contains a single rest at the end and we consider this slightly less optimal. In the sixth row, note how the presence of pitch-class 4 in the referenced array segment causes a tie to appear in the output. We believe this is least optimal. Finally, note that the reference in the eighth row is not possible, as it does not contain substrings that belong to this equal-note-value string. In our model, the covers $\langle\langle 6,7\rangle,\langle 0\rangle\rangle$ and $\langle\langle 6\rangle,\langle 7,0\rangle\rangle$ would be considered equally good.

It is important to note that equal-note-value strings in our model differ slightly from those in Babbitt's practice. In Babbitt's later works, he ensured that the referenced array segments in a piece form an exhaustive partition of all array segments found in its all-partition array, with no one substring



Input: (1) equal-note-value string, $s = \langle 6, 7, 0 \rangle$
(2) time-point interval equal to 3

optimal covering

Set of substrings of s	Referenced array segments	**Output**
$\langle\langle 6,7,0\rangle\rangle$	$\langle 6,7,0\rangle$	
$\langle\langle 6,7,0\rangle\rangle$	$\langle 6,7,0,9\rangle$	
$\langle\langle 6,7\rangle, \langle 0\rangle\rangle$	$\langle 6,7\rangle$ and $\langle 0\rangle$	
$\langle\langle 6,7\rangle, \langle 0\rangle\rangle$	$\langle 6,7\rangle$ and $\langle 0,9\rangle$	
$\langle\langle 6,7\rangle, \langle 0\rangle\rangle$	$\langle 2,6,7\rangle$ and $\langle 0,9\rangle$	
$\langle\langle 6,7\rangle, \langle 0\rangle\rangle$	$\langle 6,7,4\rangle$ and $\langle 0,9\rangle$	
$\langle\langle 6\rangle, \langle 7\rangle, \langle 0\rangle\rangle$	$\langle 6,8\rangle$ and $\langle 7,4\rangle$ and $\langle 0,9\rangle$	
$\langle\langle 6,7,0\rangle\rangle$	$\langle 4,9,7,6,0\rangle$	N/A

Table F.2: Various referenced array segments and the musical output for a given input equal-note-value string, $s = \langle 6, 7, 0 \rangle$, and time-point interval equal to 3 (unit equal to a sixteenth note). Note that an optimal cover is the minimum cardinality set of substrings that covers s.

of pitch classes in a segment referenced more than once. Finding such a partition of an all-partition array by equal-note-value strings is a difficult problem. Presently, we do not have a method for solving this problem. For this reason, we have adopted a greedy approach in which equal-note-value strings are constructed according to the first possible reference and where a single array segment may be referenced more than once.

5 Automating Babbitt's compositional process in his later works

Our proposed procedure for automating Babbitt's compositional process in his later works is shown in Figure F.10. We begin with an all-partition array as input. For each of the n ordered mosaics in this array, we first generate a time-point row. The result is a string of $n \times 12$ time points. For each of the $(n \times 12) - 1$ time-point intervals, we then compute from its corresponding PcOM, a pseudo-random equal-note-value string (abbreviated in Figure F.10 as "ENVS") according to the probabilities of immediate repeats, other repeats and new pitch classes (described in Table F.1) and ensuring its maximum length is not exceeded (as described in Equation F.2). If this equal-note-value string has a referenced array segment, we adopt a "greedy" approach in

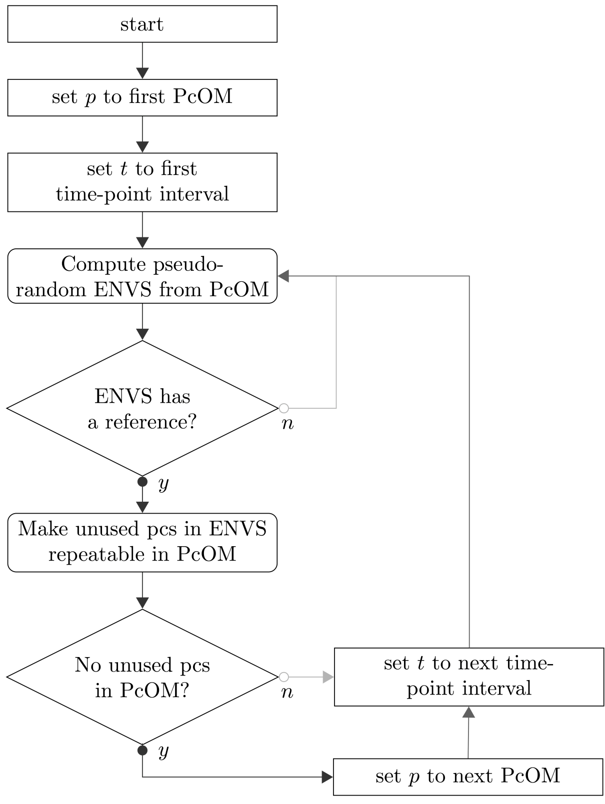


Fig. F.10: Proposed procedure for modeling Babbitt's compositional process in his late practice pieces based on the all-partition array and time-point system.

which we select the one which produces a minimum cardinality set cover (or as optimal as possible, as shown in Table F.2), retaining the appropriate rests and ties. If this particular equal-note-value string does not have a reference, we generate a new string containing the same number of pitch classes and try again to find a reference. As references can be difficult to find for larger lengths, attempting this process more than once ensures that smaller lengths are not favored and that a variety of string lengths will occur in the music. If still no reference is found after 5 attempts, we choose a new length and string of pitch classes, repeating this entire process of attempting to find a referenced array segment.

6 Generated piece

In this section, we present a novel piece automatically generated using the procedure proposed in this paper. The piece, shown in Figure F.11, is a work for flute and string quartet where each instrument contains an all-partition array of 34 ordered mosaics in four lynes. In the flute part, its all-partition array has four voices distributed across two registers from C4–B5. In the quartet, each instrument similarly contains four voices distributed across two registers with the cello from C2–B2, the viola from C3–B4, violin II from G3–F#4, and violin I from C4–B5. Both the flute and quartet use the same sequence of time points so that each change in dynamic level aligns for every instrument. For the sake of length, we have chosen here to present only the first 17 of this all-partition array's 34 ordered mosaics. It is important to note of this piece, however, that pitch, onset, duration, voice, dynamic level and meter have all been automatically generated. Meters have been chosen by dividing the prefix sum of time-point intervals by the number of units to a beat, from left-to-right until this value lies between 2 and 6 beats, inclusive. The fractional meters that result are more common in Babbitt's earlier practice, however, they do appear in his later works. To ensure that not all instruments play together at all times, we have randomly chosen for a given time-point interval whether or not an equal-note-value string will occur for each instrument. Similarly, we have randomly chosen whether or not simultaneities will occur in an equal-note-value string of the stringed instruments. Such simultaneities are constrained to not exceed 2 and must belong to the same voice, features which we have also found in Babbitt's own works.

Fig. F.11: Excerpt from a novel piece automatically generated in the style of Babbitt for flute and string quartet.

Fig. F.11 (cont.)**:** Excerpt from a novel piece automatically generated in the style of Babbitt for flute and string quartet.

Fig. F.11 (cont.)**:** Excerpt from a novel piece automatically generated in the style of Babbitt for flute and string quartet.

Fig. F.11 (cont.)**:** Excerpt from a novel piece automatically generated in the style of Babbitt for flute and string quartet.

Fig. F.11 (cont.)**:** Excerpt from a novel piece automatically generated in the style of Babbitt for flute and string quartet.

Fig. F.11 (cont.)**:** Excerpt from a novel piece automatically generated in the style of Babbitt for flute and string quartet.

Fig. F.11 (cont.)**:** Excerpt from a novel piece automatically generated in the style of Babbitt for flute and string quartet.

Fig. F.11 (cont.)**:** Excerpt from a novel piece automatically generated in the style of Babbitt for flute and string quartet.

7 Conclusion

In this paper, we have proposed a procedure for automating Milton Babbitt's compositional process in his later works. This process includes his use of techniques such as the all-partition array, time-point row and equal-note-value strings. As our generated piece has demonstrated, these techniques alone are sufficient for generating a number of musical parameters that appear on the musical surface, including pitch, onset, duration, voice, and dynamic level. Additional parameters, such as articulation, phrase markings, tempo indication, and form, have not been included in our procedure. However, we are confident that further analysis of his sketches could reveal similarly algorithmic techniques for determining these additional parameters.

Acknowledgements

The work reported in this paper was carried out as part of the EC-funded collaborative project, "Learning to Create" (Lrn2Cre8). The Lrn2Cre8 project acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET grant number 610859.

References

Babbitt, M. (1955). Some aspects of twelve-tone composition. *The Score and I.M.A. Magazine* 12:53–61.

Babbitt, M. (1960). Twelve-tone invariants as compositional determinants. *Journal of Music Theory* 46(2):246–259.

Babbitt, M. (1961). Set structure as a compositional determinant. *Journal of Music Theory* 5(1):72–94.

Babbitt, M. (1962). Twelve-tone rhythmic structure and the electronic medium. *Perspectives of New Music* 1(1):49–79.

Babbitt, M. (1973). Since Schöenberg. *Perspectives of New Music* 12(1/2):3–28.

Bemman, B., and Meredith, D. (2015a). Comparison of heuristics for generating all-partition arrays in the style of Milton Babbitt. In *CMMR 11th International Symposium on Computer Music Multidisciplinary Research, 16–19 June 2015*. Plymouth, UK.

Bemman, B., and Meredith, D. (2015b). Exact cover problem in Milton Babbitt's all-partition array. In *T. Collins, D. Meredith, and A. Volk, (eds) Mathematics and Computation in Music: Fifth International Conference, MCM 2015, London, UK, June 22–25, 2015, Proceedings*, Lecture Notes in Artificial Intelligence, volume 9110. Berlin: Springer, pp. 237–242.

Bemman, B., and Meredith, D. (2016). Generating Milton Babbitt's all-partition arrays. *Journal of New Music Research* 45(2):184–204.

Bernstein, Z. (2014). The problem of completeness in Milton Babbitt's music and thought. In *43rd Annual Meeting of The Music Theory Society of New York State, April 5–6, 2014*. New York University.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition.

Ebcioğlu, K. (1987). Report on the CHORAL project: An expert system for harmonizing four-part chorales. Technical Report RC12628. `http://global-supercomputing.com/people/kemal.ebcioglu/pdf/RC12628.pdf`.

Mead, A. (1994). *An Introduction to the Music of Milton Babbitt*. Princeton, NJ.: Princeton University Press.

Tanaka, T., Bemman, B. and Meredith, D. (2016a). Integer Programming Formulation of the Problem of Generating Milton Babbitt's All-partition Arrays. In *17th International Society for Music Information Retrieval (ISMIR2016), August 7–11, 2016*, New York, USA.

Tanaka, T., Bemman, B. and Meredith, D. (2016b). Constraint Programming Approach to the Problem of Generating Milton Babbitt's All-partition Arrays. In *M. Rueher, (ed) 22nd International Conference Principles and Practice of Constraint Programming, CP 2016, Toulouse, France, September 5–9, 2016, Proceedings*, Lecture Notes in Computer Science, volume 9892. Berlin: Springer, pp. 802–810.

ISSN (online): 2446-1628
ISBN (online): 978-87-7112-916-8

AALBORG **UNIVERSITY PRESS**