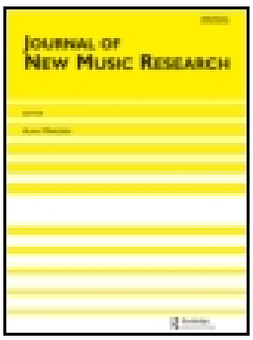

Journal of New Music Research

ISSN: 0929-8215 (Print) 1744-5027 (Online) Journal homepage: http://www.tandfonline.com/loi/nnmr20

Generating Milton Babbitt's All-Partition Arrays

Brian Bemman & David Meredith

To cite this article: Brian Bemman & David Meredith (2016): Generating Milton Babbitt's All-Partition Arrays, Journal of New Music Research, DOI: 10.1080/09298215.2016.1172646

To link to this article: http://dx.doi.org/10.1080/09298215.2016.1172646

Published online: 25 Apr 2016.

Journal of New Music Research, 2016
http://dx.doi.org/10.1080/09298215.2016.1172646

Routledge
Taylor & Francis Group

Generating Milton Babbitt's All-Partition Arrays

Brian Bemman and David Meredith

Aalborg University, Denmark

(Received 29 June 2015; accepted 24 March 2016)

Abstract

In most of Milton Babbitt's (1916–2011) works written since the early 1960s, both the pitch and rhythmic content is organized 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 each integer partition of 12 whose cardinality is no greater than the number of lynes in a piece is expressed by exactly one 'vertical' aggregate. 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.

Keywords: music analysis, Milton Babbitt, all-partition array, pitch class set theory, algorithms, music theory

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 each integer partition of 12 whose cardinality is no greater than the number of lynes in a piece is expressed by exactly one 'vertical' aggregate (see Figure 3, Section 2.2, 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, 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, Leiserson, Rivest, & Stein, 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

Correspondence: Brian Bemman, Aalborg University, Rendsburggade 14, 9000 Aalborg, Denmark. E-mail: bb@create.aau.dk

[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.

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 row form is *hc*-related to the row form $\mathrm{I}_7 = \langle 7, 8, 1, 3, 9, 2, 11, 10, 6, 4, 5, 0\rangle$. 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 $\mathrm{P}_0 = \langle 0, 1, 3, 6, 7, 9, 2, 4, 5, 8, 10, 11\rangle$, then it shares the same disjunct hexachords with the row form $\mathrm{P}_6 = \langle 6, 7, 9, 0, 1, 3, 8, 10, 11, 2, 4, 5\rangle$.

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 formed in such an array is 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

[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.

[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*.

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

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 1 shows an example of such a row-form array.

In Figure 1, tone rows appear in *hc*-related pairs that are grouped into eight 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, 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 1 then corresponds to a 6×96 matrix of pitch classes that we 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 $T_n(S)$ and inversion around 0 followed by transposition by n semitones by $I_n(S)$. We denote the complement of S in the aggregate by $\overline{S}$. Thus, if $S = \{0, 1, 2, 6, 7, 8\}$, then $T_3(S) = T_9(S) = I_{11}(S) = I_5(S) = \overline{S}$ and $T_0(S) = T_6(S) = I_2(S) = I_8(S) = S$. 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 *unordered* aggregate partitioned into *unordered* pitch class sets (Morris, 2003, p. 103). Figure 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 1.

Note in Figure 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 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 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 2(b)). We now define some terms that allow us to use this more informative representation.

An *integer partition*, which we denote by $\text{IntPart}(s_1, s_2, \ldots, s_k)$, 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 & Bourboubi, 2011). For example, if

[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).

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

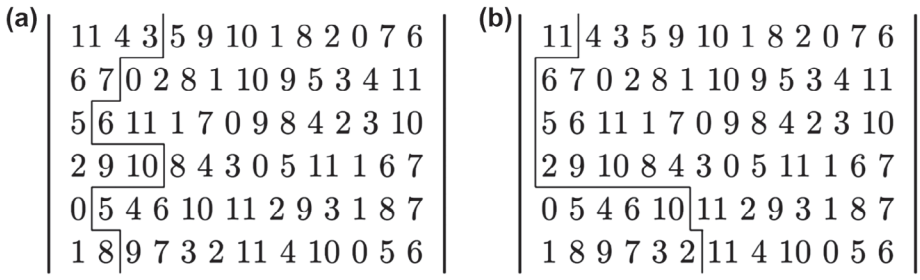

Fig. 2. Two possible ordered mosaics in an excerpt from a $\text{PcMatrix}_{6,96}$ corresponding to the first block of its row-form array (which is shown in Figure 1). Each ordered mosaic consists of the pitch classes on the left of the boundary marked by the ragged line. In (a) the ordered mosaic is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

$n = 12$ and $k = 6$, then one possible integer partition is IntPart(3, 3, 2, 2, 1, 1). An *integer composition*, which we denote by $\text{IntComp}(s_1, s_2, \ldots, s_k)$, 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 $\text{WIntComp}(s_1, s_2, \ldots, s_k)$, 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 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, $\text{IntComp}(s_1, s_2, \ldots, s_k)$, then we define the *integer partition associated with* c, denoted by $[c]$, to be $\text{IntPart}(s_1, s_2, \ldots, s_k)$. 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 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 WPcMatrix can be a regular matrix or an irregular matrix (i.e. with unequal lyne lengths), depending on the distribution of its OARPs. Figure 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.

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

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

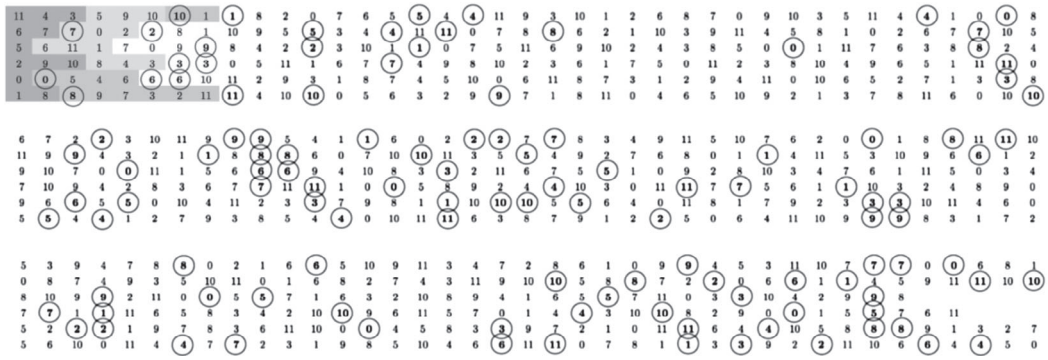

Fig. 3. The complete WPcMatrix for the six-lyne, self-contained and horizontally weighted all-partition array in *Sheer Pluck* (1984), represented as an irregular matrix of pitch classes (note the unequal lyne lengths). The first three of its 58 partitions have been indicated in three different shades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

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. 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 & Morris, 1977, p. 18.)

In particular, we consider those methods presented by Starr and Morris (1977, 1978), Bazelow and Brickle (1976, 1979) and Morris (2010). Writings 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 & Morris, 1977, 1978). In particular, they introduce the notions of *begin-set*, *end-set* and *n-clique*[11] (Starr & 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 & Morris, 1977, p. 8). Figure 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 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 & 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 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 & 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 & 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

[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.

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 straightforward, 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 & 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 organization 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 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 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 5) is that a PcMatrix must contain 48 tone rows belonging to the same row class. The third constraint (labelled (3) in Figure 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-form array while satisfying constraints (2) and (3) above. A Smalley array represents one such way and we discuss this arrangement next.

4.1 Organizational constraints on the row-form array

A number of authors have discussed the various ways in which Babbitt organized the tone rows in his works (Bemman & 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 organizational 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 organizational 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 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 six distinct 2×4 Latin rectangles. Figure 6 shows a row-form array in which the organizational 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 6, arrows labelled T_3 and T_9 indicate relationships by transposition between pairs of consecutive tone rows in one

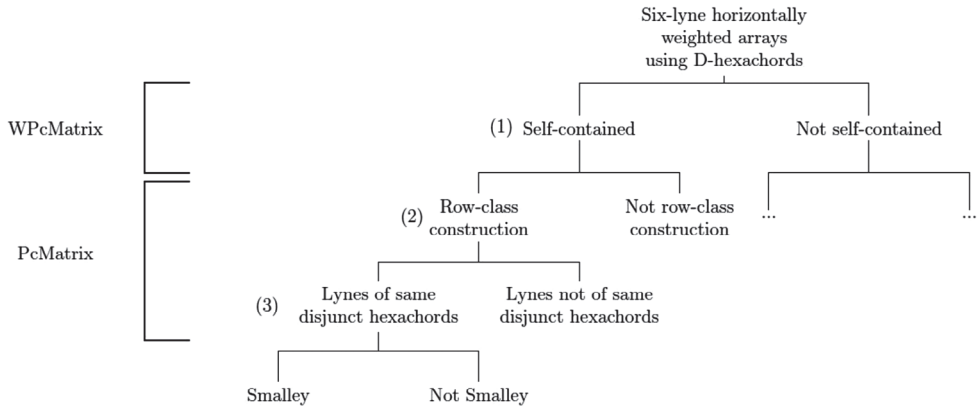

Fig. 5. Classification of six-lyne, horizontally weighted arrays using D-hexachords according to Babbitt's most common practices. Note, in particular, the constraints satisfied by the Smalley arrays.

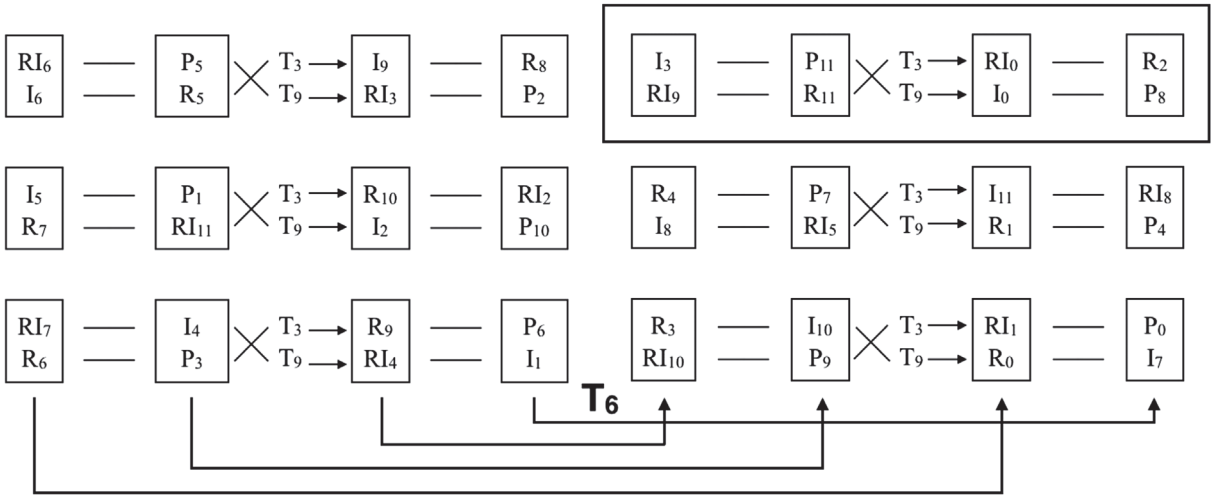

Fig. 6. A Smalley row-form array and organizational constraints. Each box contains an *hc*-related pair of rows. Each horizontal sequence of four *hc*-related pairs forms a distinct 2×4 Latin rectangle of operations. One of these is indicated by the rectangle drawn in the upper right. Note that this 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.

lyne to consecutive tone 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 WPcMatrix of a Smalley array. Figure 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 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 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 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 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 6. The ComputeOperations function, shown in Figure 9, organizes the row form operations in a row-form array. The second function, ComputeTranspositions, shown in Figure 10, then assigns transpositions to these

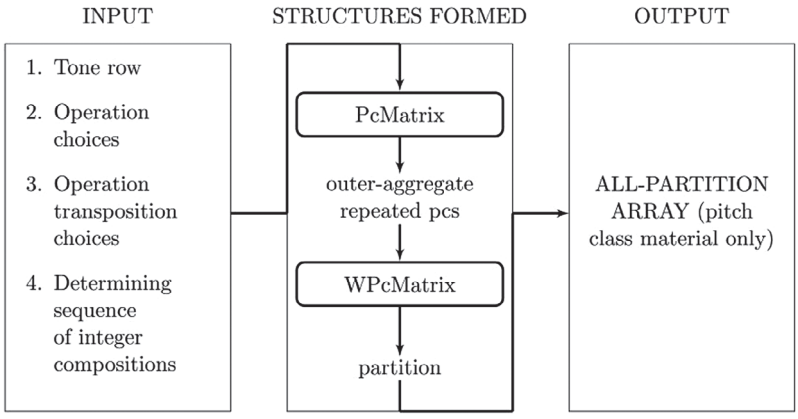

Fig. 7. Basic flow of constructing an all-partition array using our proposed method. Arrows indicate one or more functions necessary for generating one structure from a previous one.

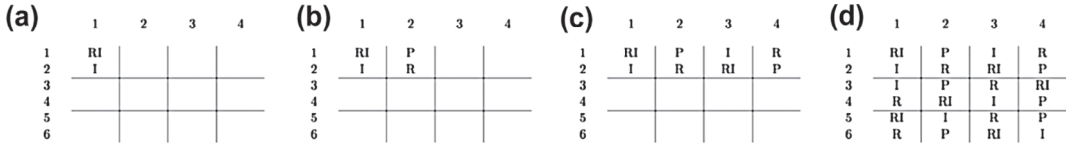

Fig. 8. Process of filling columns 1–4 of a row-form array with operations according to the constraints shown in Figure 6. (a) Step 1. Place an ordered pair of operations in rows 1–2 of column 1 (in this case, ⟨RI, I⟩). (b) Step 2. Choose an ordering for the other two operations and place in rows 1–2 of column 2. (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. (d) Repeat Steps 1–3 for rows 3–4 and 5–6.

```
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. 9. Pseudocode for the ComputeOperations function.

operations. Figure 8 shows how the process of filling a row-form array with operations unfolds in ComputeOperations. Note in Figure 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 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 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 ⟨1–4, 1–6, 1–2, 1–2, 1–2, 1–2, 1–2⟩. 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 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).

```
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. 10. The ComputeTranspositions function.

In this pseudocode, '←' is the assignment operator and block structure is indicated by indentation alone. Ordered sets are notated with angle brackets ('⟨·⟩') and their names are rendered in bold font. The length of an ordered set, **A**, is denoted by |**A**|. The ith element of an ordered set, **A**, is denoted by **A**[i]. Thus, **A**[1] is the first element in **A** and **A**[|**A**|] is the last element in **A**. If each element in an ordered set, **A**, is itself an ordered set, then **A**[i][j] denotes the jth element of the ith element of **A**. **A**[$i..j$] denotes the segment of the ordered set, **A**, containing the ith to the jth elements in **A**. Similarly, **A**[$i..j$][$k..\ell$] denotes the rectangular region of the two-dimensional array, **A**, containing the elements that occur in both the ith to jth rows and the kth to ℓth columns. If **A** and **B** are two ordered sets, then **A** ⊕ **B** is the concatenation of **A** and **B**—that is, **A** ⊕ **B** = ⟨**A**[1], . . . **A**[|**A**|], **B**[1] . . . **B**[|**B**|]⟩. If ⟨**A**$_1$, **A**$_2$, . . . **A**$_n$⟩ 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 9, ComputeOperations begins by initializing an empty 6 × 8 row-form array called **RFA** (line 1). **RFA** is a two-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 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 **RFA**[$i..i+1$][1]. m is then incremented in line 12. Figure 8(a) illustrates the process in lines 10–12 for $i = 1$. The algorithm then computes the complement 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 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 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 1).

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

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

In Figure 10, 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 11(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 **RFA**[i][1] (for which the transposition level has just been computed), and stores these in **HexTrans**. In line 10, the transposition indicated by **Trs**[k] is selected and stored in **RFA**[$i+1$][1].trans, and then k is incremented in line 11.

In lines 12–17, the value in **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 6. k is then incremented again in line 18. An example of the process in lines 12–18 is shown in Figure 11(b). Lines 19–22 carry out the process exemplified in Figure 11(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 **RFA**[i][2] are computed using TranspositionsFromRowClass. In line 20, the FindHexContent function returns a list of possible transpositions for **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 **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 **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 11(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 11(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 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

(i) 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
(ii) is partitionally distinct from all other compositions found in the sequence before it.

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

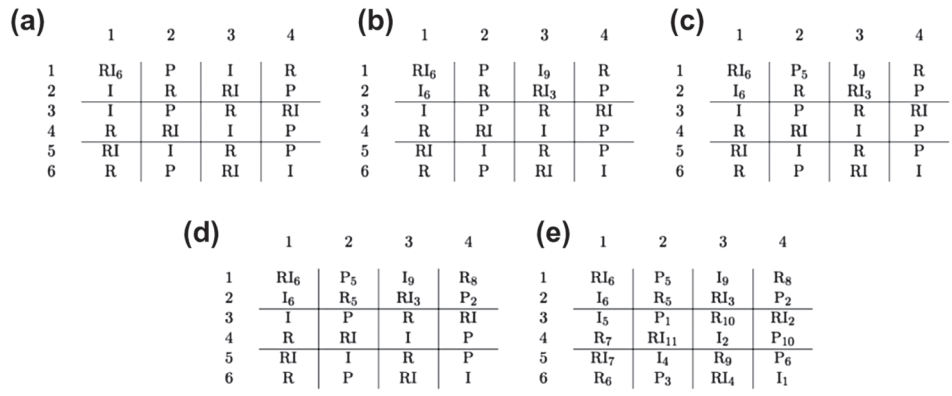

Fig. 11. Process of assigning transpositions to operations in columns 1–4 in order to complete a row-form array according to the constraints shown in Figure 6. (a) Step 1. Place a transposition in **RFA**[1][1]. (b) Steps 2 and 3. Place a transposition in **RFA**[2][1] *hc*-related to **RFA**[1][1] and place T_3 and T_9 transpositions of these in **RFA**[1][3] and **RFA**[2][3], respectively. (c) Step 4. Place a transposition in **RFA**[1][2], such that the row form shares disjunct hexachords with **RFA**[1][1]. (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. (e) Repeat steps 1–6 for remaining rows until complete.

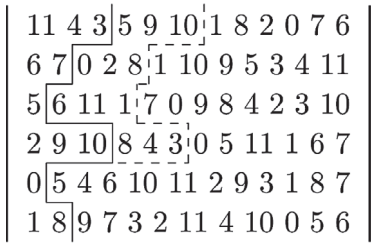

Fig. 12. IntComp(3, 2, 1, 3, 1, 2) (indicated by the ragged solid line boundary), forming an ordered mosaic containing a complete aggregate, followed by WIntComp(3, 3, 3, 3, 0, 0) (indicated by the dashed line), forming an ordered mosaic containing a sufficiently complete aggregate, in an excerpt from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

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 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 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 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 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 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. Figures 13(a) and (b) show two possible solutions to completing the sufficiently complete aggregate formed by the current composition in Figure 12. Figure 13(c) provides an example of a sufficiently complete aggregate that does not have a solution.

In Figure 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 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

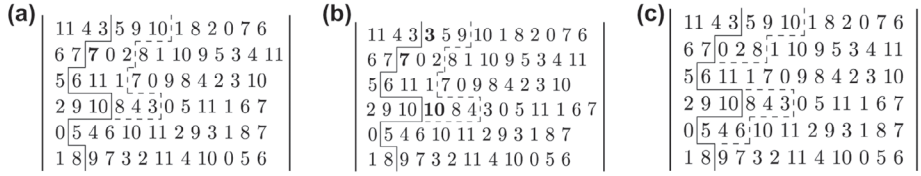

Fig. 13. (a) Simple and (b) complex solutions (in bold) to completing the sufficiently complete aggregate in the region formed by the candidate composition, WIntComp(3, 3, 3, 3, 0, 0), shown in Figure 12. (c) No solution exists for the sufficiently complete aggregate in the region formed by the composition,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

& Brickle, 1979, p. 55). The weak integer composition, WIntComp(3, 3, 0, 3, 3, 0), shown in Figure 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 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 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 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 to first try any remaining untried OARP solution for the current candidate composition. **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. **C**[Cnt] $>$ |**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, **CList**[Cnt][**C**[Cnt]], will only be empty in line 22 if this candidate composition is complete. If the current composition is complete, then **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.

```
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. 14. The BacktrackingBabbitt algorithm.

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 15 shows pseudocode for the Advance and Backtrack functions called by the BacktrackingBabbitt algorithm. As shown in Figure 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 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 16 and 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 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

```
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. 15. The (a) ADVANCE and (b) BACKTRACK functions called by BACKTRACKINGBABBITT.

```
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. 16. The COMPUTECANDIDATECOMPOSITIONS algorithm.

```
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. 17. The COMPUTEOARPS function.

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*. 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

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. 18. Sequence of compositions found in *Sheer Pluck*.

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 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. 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, **PostRepCombs** and **PostPushCombs**, respectively. The **for** loop that begins in line 5 then iterates over all the combinations in **PostRepCombs**. For an OARP solution to be found, two conditions must be met:

(i) all pitch classes that are missing (either from the ordered mosaic to begin with or those from **PostPushCombs**[i] that are pushed out) are found or replaced when all pitch classes in **PostRepCombs**[i] are repeated; and
(ii) all pitch classes that are duplicates (either in the ordered mosaic to begin with or those from **PostRepCombs**[i] that are repeated) are removed when all pitch classes in **PostPushCombs**[i] are pushed.

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

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

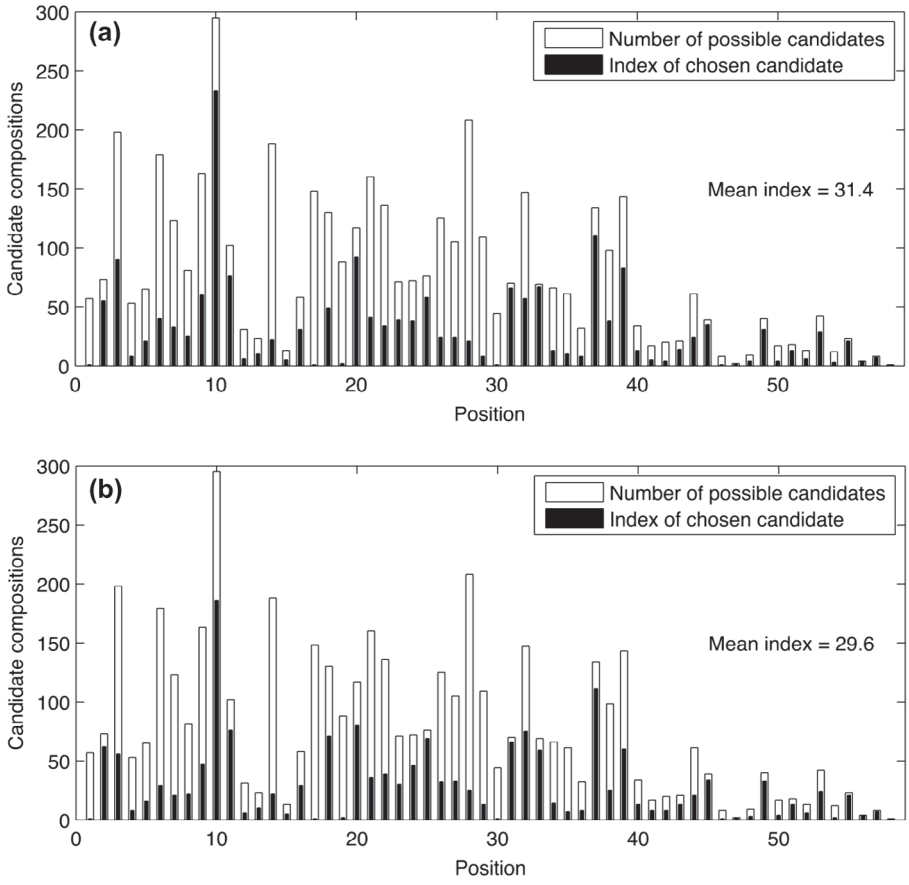

Fig. 19. Results of applying (a) equal-lyne-length and (b) equal-lyne-length and zero-gain segments to select the sequence of integer compositions used in *Sheer Pluck*. Note (1) the less black at each position, the better; and (2) the lower the value of the mean index, the better.

OARPs in line 9. Upon completion of the loop, **OARPs** is returned (line 10).

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 straightforward 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, $C_k = \langle c_1, c_2, \ldots c_n \rangle$, 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:

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

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 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 Equation (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 $A = \langle s_1, s_2, \ldots, s_k \rangle$ and $B = \langle t_1, t_2, \ldots, t_k \rangle$ and B follows A. The *weight*, w, of B is given by $w = \sum_{i=1}^{k} r_i$, where

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

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 the 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 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 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 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 1, as input, BacktrackingBabbitt will return the WPcMatrix shown in Figure 3 and the sequence of compositions shown in Figure 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.

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

Figure 19 shows the index of each candidate composition found in the self-contained sequence used in *Sheer Pluck*

[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.

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

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

Table 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).

	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, 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, 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, 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, 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, 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, 0, 0, 0, 0⟩

after applying the equal-lyne-length heuristic (in Figure 19(a)) and both equal-lyne-length and zero-gain segments (in Figure 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 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 20 shows the index of each candidate composition found in this piece after applying the equal-lyne-length heuristic (in Figure 20(a)) and both equal-lyne-length and zero-gain segments (in Figure 20(b)). As can be seen in Figure 20, with this piece we achieve substantially lower mean index values than for *Sheer Pluck* (cf. Figure 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.

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

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 21 shows how the first 100,000 backtracks were distributed over the 58 positions in the sequence of integer partitions for *Sheer Pluck* under five different conditions.

Note in Figure 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 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,

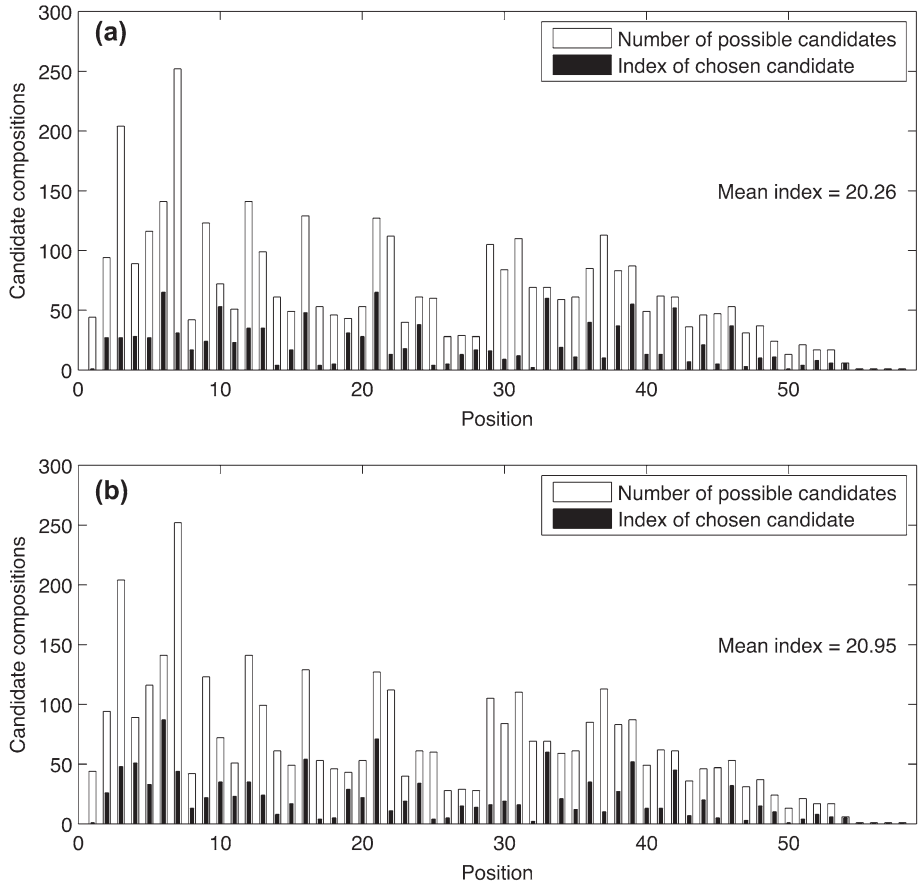

Fig. 20. Results of applying (a) equal-lyne-length and (b) equal-lyne-length and zero-gain segments to select the sequence of integer compositions used in *About Time*. Note (1) the less black at each position, the better; and (2) the lower the value of the mean index, the better.

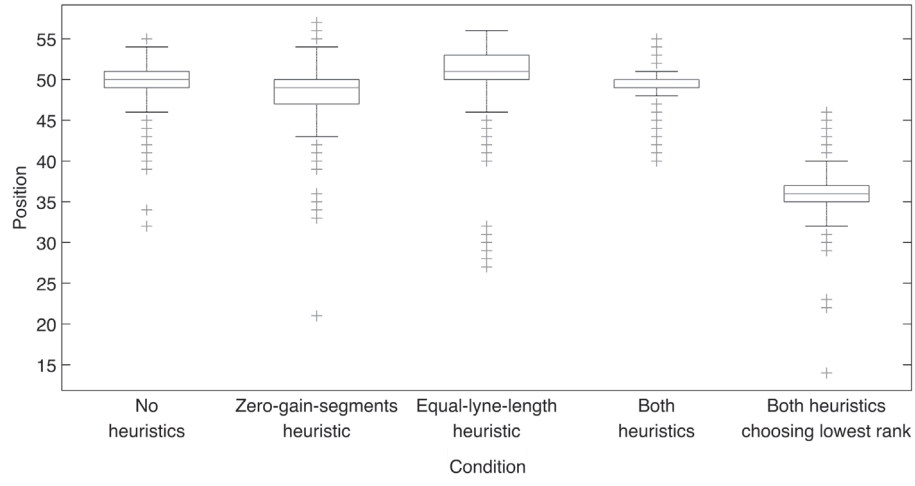

Fig. 21. Comparison of the distribution of 100,000 backtracks over positions in the integer partition sequence for *Sheer Pluck* under five conditions. The horizontal line in each box indicates the median backtracking position for that condition. The median for the 'Both heuristics' condition coincides with the top of the box. See text for details.

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 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, Robert Morris and David Smalley for their kind and informative responses to our queries.

Funding

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

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

Bazelow, A.R., & 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., & 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* (4pp). Paper presented at the The Music Encoding Conference, 20–23 May 2014, Charlottesville, VA.

Bemman, B., & Meredith, D. (2015a). Comparison of heuristics for generating all-partition arrays in the style of Milton Babbitt. In M. Aramaki, R. Kronland-Martinet, & S. Ystad (Eds.), *Proceedings of the 11th International Symposium on Computer Music Multidisciplinary Research, Plymouth, UK* (pp. 770–777). Marseille: LMA Publications.

Bemman, B., & Meredith, D. (2015b). Exact cover problem in Milton Babbitt's all-partition array. In T. Collins, D. Meredith, & 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, Vol. 9110, pp. 237–242) Cham, Switzerland: Springer.

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., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). Cambridge, MA: MIT Press.

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* (PhD thesis). Princeton University, Princeton, NJ, USA.

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., & McNutt, E. (2005). Virtuosity in Babbitt's *Lonely Flute*. *Music Theory Online*, *11(1)*, http://www.mtosmt.org/issues/mto.05.11.1/mto.05.11.1.leong_mcnutt.html.

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.

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

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

Morris, R. (1987). *Composition with pitch-classes: A theory of compositional design*. New Haven, CT: Yale University Press.

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*. Princeton, NJ: The University of Rochester Press.

Page, D. (2013). Parallel algorithm for second-order restricted weak integer composition generation for shared memory machines. *Parallel Processing Letters, 23*(3), 1350010. doi: 10.1142/S0129626413500102

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

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

Tani, N. & 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.

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