Backtracking search heuristics for solving the all-partition array problem

Bemman, Brian; Meredith, David

Published in:
International Society for Music Information Retrieval Conference

Creative Commons License
CC BY 4.0

Publication date:
2019

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

Link to publication from Aalborg University

Citation for published version (APA):
Bemman, B., & Meredith, D. (2019). Backtracking search heuristics for solving the all-partition array problem. In *International Society for Music Information Retrieval Conference* (pp. 391-397). [46] International Society for Music Information Retrieval. http://archives.ismir.net/ismir2019/paper/000046.pdf

BACKTRACKING SEARCH HEURISTICS FOR SOLVING THE ALL-PARTITION ARRAY PROBLEM

Brian Bemman
Aalborg University
Aalborg, Denmark
bb@create.aau.dk

David Meredith
Aalborg University
Aalborg, Denmark
dave@create.aau.dk

ABSTRACT

Recent efforts to model the compositional processes of Milton Babbitt have yielded a number of computationally challenging problems. One of these problems, known as the *all-partition array problem*, is a particularly hard variant of set covering, and several different approaches, including mathematical optimization, constraint satisfaction, and greedy backtracking, have been proposed for solving it. Of these previous approaches, only constraint programming has led to a successful solution. Unfortunately, this solution is expensive in terms of computation time. We present here two new search heuristics and a modification to a previously proposed heuristic, that, when applied to a greedy backtracking algorithm, allow the all-partition array problem to be solved in a practical running time. We demonstrate the success of our heuristics by solving for three different instances of the problem found in Babbitt's music, including one previously solved with constraint programming and one Babbitt himself was unable to solve. Use of the new heuristics allows each instance of the problem to be solved more quickly than was possible with previous approaches.

1. INTRODUCTION

Milton Babbitt (1916–2011) was a composer of serial music, whose work constituted a substantial contribution to 12-tone music theory and composition [2–6]. His works and the compositional techniques he developed have been studied extensively by music theorists and noted for their complexity [10, 13, 15, 18]. Recent computational work has shed further light on this complexity by looking, in particular, at a 12-tone structure and method of composition that Babbitt developed, known as the *all-partition array* [7, 9, 19, 20].

Satisfying all the constraints necessary to construct an all-partition array is challenging, not least because it involves solving a difficult variant of the set-cover problem [7, 8, 12]. In this paper, we present an improvement to a previous method, based on greedy backtracking, that applies two new search heuristics and a modification of a third heuristic that was originally proposed in [9]. These heuristics aim to facilitate the satisfaction of the most challenging of the all-partition array constraints when using a backtracking algorithm (to be discussed in section 4).

In section 2, we review the all-partition array and follow this with an overview in section 3 of recent computational work on solving the all-partition array problem. We focus in particular on the procedural greedy backtracking approach proposed in [9] to which the heuristics proposed in this paper have been applied. In section 4, we present our new heuristics and give pseudo-code for one possible implementation. In section 5, we demonstrate the effectiveness of the new heuristics by using them to discover solutions to three instances of the all-partition array problem. We report and compare the average running times as well as the number of required backtracks for these three solutions. We also provide a complete solution to one of these instances as evidence of correctness. Finally, in section 6, we summarize our results and propose some possible directions for future work.

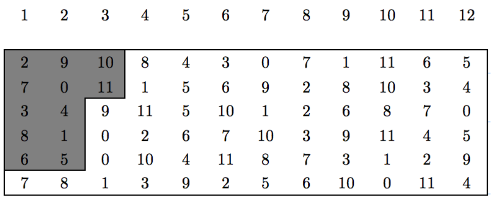
2. THE ALL-PARTITION ARRAY

Constructing an all-partition array starts with an $I \times J$ matrix, A, in which each entry is an integer between 0 and 11, representing a pitch class, and where each row contains $J/12$ twelve-tone rows. In this paper, we focus on matrices where $I = 6$ and $J = 96$ because these figure prominently in Babbitt's music [14], and so far have proved difficult to generate [7, 9]. The resulting set of 48 tone rows in the matrix must be closed under any combination of transposition, inversion and retrograde.[1] Matrix A will therefore contain 48 occurrences of each of the integers from 0 to 11. It is important to note that not all organizations of these pitch classes in A will prove successful in constructing an all-partition array, but a desirable trait is for pitch classes to

[1] The exact instance of the type of 6 × 96 matrix shown throughout this paper has the following form:

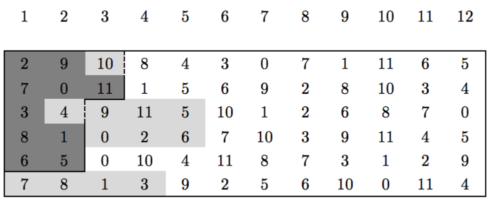
$$A = \begin{bmatrix} \mathrm{R}_5 & \mathrm{I}_4 & \mathrm{RI}_7 & \mathrm{P}_2 & \mathrm{R}_{11} & \mathrm{P}_8 & \mathrm{RI}_1 & \mathrm{I}_{10} \\ \mathrm{RI}_4 & \mathrm{P}_{11} & \mathrm{R}_2 & \mathrm{I}_1 & \mathrm{RI}_{10} & \mathrm{I}_7 & \mathrm{R}_8 & \mathrm{P}_5 \\ \mathrm{P}_3 & \mathrm{R}_0 & \mathrm{I}_{11} & \mathrm{RI}_8 & \mathrm{P}_9 & \mathrm{R}_6 & \mathrm{I}_5 & \mathrm{RI}_2 \\ \mathrm{RI}_5 & \mathrm{I}_2 & \mathrm{R}_3 & \mathrm{P}_0 & \mathrm{RI}_{11} & \mathrm{I}_8 & \mathrm{R}_9 & \mathrm{P}_6 \\ \mathrm{I}_6 & \mathrm{R}_1 & \mathrm{P}_4 & \mathrm{RI}_3 & \mathrm{I}_0 & \mathrm{RI}_9 & \mathrm{P}_{10} & \mathrm{R}_7 \\ \mathrm{P}_7 & \mathrm{R}_{10} & \mathrm{I}_9 & \mathrm{R}_{10} & \mathrm{P}_1 & \mathrm{R}_4 & \mathrm{I}_3 & \mathrm{RI}_6 \end{bmatrix},$$

where $\mathrm{P}_0 = \langle 0, 1, 6, 8, 2, 7, 10, 11, 3, 5, 4, 9 \rangle$.



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

Figure 1: A 6×12 excerpt from a 6×96 pitch-class matrix with a single region defined by a partition (in dark grey) whose "shape" is represented as the integer composition, $\mathrm{IntComp}_{12}(3, 3, 2, 2, 2, 0)$.



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

Figure 2: A 6×12 excerpt from a 6×96 pitch-class matrix with two regions defined by distinct integer partitions. Note that the region formed by the second composition, $\mathrm{IntComp}_{12}(1, 0, 4, 3, 0, 4)$ (in light grey), overlaps two locations (rows 1 and 3) from the first region.

be evenly distributed [7]. However, the exact organizations which prove successful are still unknown [16, p. 284].

As shown in [20], a complete all-partition array is a *covering* of matrix, A, by K sets, each of which is a partition of the set $\{0, 1, \ldots, 11\}$ whose parts (1) contain consecutive row elements from A and (2) have cardinalities equal to the summands in one of the K distinct integer partitions of $L = 12$ (e.g., $6+6$ or $5+4+2+1$) containing I or fewer unordered summands greater than zero. [2] Figure 1 shows a 6×12 excerpt from a 6×96 pitch-class matrix, A, and one set forming a region in A containing every pitch class exactly once and corresponding to an integer partition, whose exact "shape" is more precisely represented as the *integer composition*, $\mathrm{IntComp}_{12}(3, 3, 2, 2, 2, 0)$ [9]. [3]

There are a total of 58 distinct integer partitions of 12 into 6 or fewer non-zero summands [14]. This means that the number of pitch classes required of an all-partition array having K regions, exceeds the number of entries in its matrix by $(K \cdot 12) - (I \cdot J)$. When $I = 6$ and $J = 96$, this difference is 120. On the musical surface, these 120 additionally required pitch classes are found through horizontal repetitions of at most one in each row from any one contiguous region to the next. As shown in [20], these horizontal row repetitions can be found instead through horizontal overlaps. These overlaps greatly simplify any computational model for generating an all-partition array because the matrix can remain fixed in size.

Figure 2 shows the same 6×12 excerpt from Figure 1, now with a second region corresponding to a distinct partition defined by the integer composition, $\mathrm{IntComp}_{12}(1, 0, 4, 3, 0, 4)$ (in light grey), with two overlaps shared with the first region. Note how the second region (in light grey) formed by its composition shares two overlapped locations (in rows 1 and 3) which lie at the rightmost column of the first region. A complete all-partition array of the type considered here is based on an $I = 6$ and $J = 96$ matrix containing $K = 58$ distinct regions with 120 overlaps.

The constraints of the all-partition array are motivated by Babbitt's desire for *maximal diversity* [14], which is the exhaustive presentation of as many musical parameters as possible (e.g., all 12 pitch classes, 48 tone rows, and K partitions). This makes the all-partition array problem appropriate for methods used in combinatorial optimization or constraint satisfaction, which often rely on either maximizing some objective function or strictly satisfying a set of constraints, respectively.

3. PREVIOUS COMPUTATIONAL WORK ON THE ALL-PARTITION ARRAY PROBLEM

Efforts to solve the all-partition array problem have resulted in a number of different approaches [9, 19, 20]. In [20], an integer programming (IP) model expressed the problem as a set of linear (in)equalities and managed to solve significantly smaller instances of the problem for matrices with six rows and up to 24 columns (requiring $(J + 2)/2$ regions which fix the number of overlaps to be 12). Solving for a matrix of this size required in excess of 30 minutes (Gurobi Optimizer v6.0 solver [1] running on a 2 GHz Intel Core i7 laptop with 8 GB RAM), and the computational time drastically increased with an increase in the number of columns. This suggests that solving for larger matrix sizes, such as the ones considered in this paper, would prove intractable for this IP model.

The first computational method to automatically generate an all-partition array from a pitch class matrix was a constraint programming (CP) model, described in [19]. This model solved a 4×96 matrix (requiring 34 regions and 24 overlaps) by splitting the whole of the matrix in half and solving each smaller sub-matrix before re-joining them to form a complete solution. Solving the second sub-matrix was made easier by discovering certain "easy-to-find" partitions and excluding these from the first sub-matrix. Finding a solution still required over 30 minutes (Sugar v2-1-0 solver [17] running on a 2 GHz Intel Core i7 laptop with 8 GB RAM) and this method of splitting the matrix unfortunately excludes possible solutions. Moreover, it is still unclear whether this model could be used to efficiently solve other instances of the all-partition array problem.

The heuristics proposed in this paper are intended to be used with the procedural greedy backtracking algorithm originally proposed in [9]. Figure 3 gives pseudo-code for a simplified version of the main backtracking procedure originally presented in [9].

[2] As in [20], we denote an integer partition of a positive integer, L, by $\mathrm{IntPart}_L(s_1, s_2, \ldots, s_I)$ and define it to be an ordered set of non-negative integers, $\langle s_1, s_2, \ldots, s_I \rangle$, where $L = \sum_{i=1}^{I} s_i$ and $s_1 \geq s_2 \geq \cdots \geq s_I$.

[3] As in [20], we define an *integer composition* of a positive integer, L, denoted by $\mathrm{IntComp}_L(s_1, s_2, \ldots, s_I)$, to be an ordered set of I non-negative integers, $\langle s_1, s_2, \ldots, s_I \rangle$, where $L = \sum_{i=1}^{I} s_i$. Unlike an integer partition, however, the summands in an integer composition need not be in descending order of size.

```
BACKTRACKINGBABBITT()
1  $\mathbf{C} \leftarrow \bigoplus_{i=1}^{K} \langle\langle\rangle\rangle$      ▶ Lists of candidates
2  $k \leftarrow 1$
3  while $0 < k \leq K$
4    if $\mathbf{C}[k]$ is empty
5      $\mathbf{P} \leftarrow$ FINDUNUSEDPARTITIONS(...)
6      $\mathbf{C}[k] \leftarrow$ FINDCANDIDATES($\mathbf{P}, \ldots$)
7      if $\mathbf{C}[k]$ is empty
8        $k \leftarrow k - 1$      ▶ Backtrack
9      else
10       $k \leftarrow k + 1$      ▶ Proceed
11   else      ▶ Backtracked to previously visited $k$
12     Select next overlaps for current candidate in $\mathbf{C}[k]$
13     if current overlaps for current candidate is nil
14       Current candidate becomes next candidate in $\mathbf{C}[k]$
15       if current candidate is nil
16         $\mathbf{C}[k] \leftarrow \langle\rangle$      ▶ Make empty
17         $k \leftarrow k - 1$
18       else
19         Select first overlaps (if any) for current candidate
20         $k \leftarrow k + 1$
21     else
22       $k \leftarrow k + 1$
23 return $\mathbf{C}$
```

Figure 3: Pseudo-code for a simplified version of the BACKTRACKINGBABBITT algorithm originally presented in [9]. Note that $\bigoplus_{i=1}^{n}\langle x_i\rangle = \langle x_1, x_2, \ldots x_n, \rangle$, the assignment operator is denoted by '$\leftarrow$' and scoping is indicated by indentation.

The algorithm shown in Figure 3 works from left to right in a given matrix. For each region, indexed by k, it first finds the partitions that have not yet been used in positions 1 to k (line 5) and then finds a list of candidate compositions (i.e., $\mathbf{C}[k]$) from these unused partitions (line 6) that form valid regions in the matrix according to the constraints discussed in section 2. If there are no candidate compositions, the algorithm backtracks by decrementing k by 1 (lines 7–8), otherwise it proceeds by incrementing k by 1 (line 10). In the event that the algorithm has backtracked to a previously visited k containing candidates (line 11), the next set of overlaps for the current candidate in $\mathbf{C}[k]$ is chosen (line 12). If there are no overlaps remaining (line 13) then the next candidate in $\mathbf{C}[k]$ is chosen. The algorithm then backtracks if there are no remaining candidates (lines 16–17) or proceeds after selecting the first set of overlaps (if any are available) for this new current candidate (lines 19–20). Note that we have provided here only those details necessary for understanding how our heuristics have been implemented (see section 4). [4]

Even with the help of the search heuristics described in [9], the algorithm above proved unable to generate a complete solution to the all-partition array problem after $100,000$ backtracks when tested on a 6×96 matrix. In the following section, we discuss why this failure likely occurred.

[4] More detailed pseudo-code for the original backtracking algorithm can be found in [9] and an improved implementation (written in Julia v.1.1.0 [11]) with our proposed heuristics can be found in the following repository: `https://github.com/brianm2b/generate-all-partition-arrays`.

4. PROPOSED BACKTRACKING SEARCH HEURISTICS

As noted in [9], a greedy deterministic backtracking algorithm for solving the all-partition array problem based on a depth-first search procedure which finds candidate regions from left to right in a given matrix will incur considerable computational cost in terms of time without sufficiently good heuristics for limiting the amount of backtracking required. When attempts were made to use this algorithm to solve for a 6×96 matrix, much of the backtracking was concentrated towards the far right of the matrix after the algorithm had already successfully discovered most of the required partitions. This means that the algorithm was unable to use the few remaining unused partitions to find compositions capable of forming successful regions (according to the constraints discussed in section 2) from the remaining matrix elements. If one relaxes the constraint that missing pitch classes from the final region must come from overlaps from previous contiguous regions and, instead, allows these pitch classes to be added to the end of the matrix, then the problem becomes significantly more tractable. As it turns out, this is exactly what Babbitt was forced to do in many of his works [7, 13]. It is these matrices, found for example in Babbitt's *About Time* (1982) and *Arie da Capo* (1974), that are the focus of this paper, as no known solution which satisfies all constraints exists. [5]

Figure 4 illustrates two scenarios for forming a final region in a nearly complete all-partition array where only one partition remains to be used. In Figure 4(b) the final region fails to cover one entry in the matrix due to an overlap of pitch class 9 (row 2) with the previous region. In Figure 4(c), every entry is covered, but pitch class 9 is added to the end (row 3) instead of overlapping with the previous region.

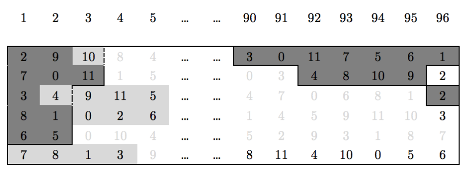
4.1 First Proposed Search Heuristic

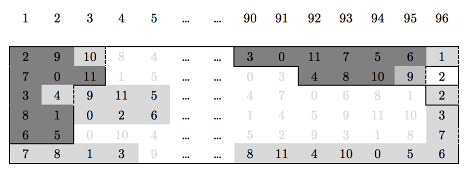
The first of our proposed search heuristics is based on a simple assumption regarding the difficulty of finding a final region noted above. By excluding from the left-to-right search those partitions that successfully form regions in the final position K at the far right-hand side of A, it will be easier not only to form a complete matrix covering (Figure 4(c)), but also to avoid violating the constraint that all missing pitch classes from any one region must be overlaps (Figure 4(b)).

Let us suppose P_k, $1 \leq k \leq K$, is the set of all unused partitions not found in the sequence of selected candidate compositions from 1 up to k and R is the set of partitions that have been found to successfully form regions at K. The modified set of all unused partitions, P'_k, from 1 up to k is given by

$$P'_k = \begin{cases} P_k \setminus R, \text{if } (|P_k \cap R| = r \;\wedge \\ \qquad\qquad\qquad |P_k| > r)\text{; and} \\ P_k, \text{otherwise,} \end{cases} \tag{1}$$

[5] See [13] for an example of an all-partition array of this type that forms an incomplete solution to the all-partition array problem.


(a) A nearly covered matrix and complete all-partition array with 9 uncovered pitch classes and one unused partition, $\text{IntPart}_{12}(7,1,1,1,1,1)$, remaining.


(b) An unsuccessful matrix covering where a single pitch-class 2 remains uncovered by the final region. Note that all missing pitch classes from the uncovered elements in (a) are overlaps with previous contiguous regions.

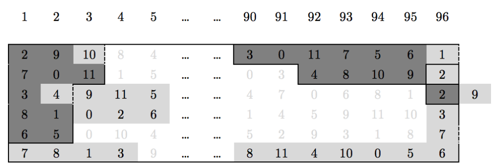

(c) A successful matrix covering by the final region where pitch-class 2 from (b) has been covered, requiring that the missing pitch class 9 is added to the end of the matrix instead of overlapping with the previous region (as in (b)).

Figure 4: A 6×96 pitch-class matrix with 57 of its $K = 58$ partitions in (a) and two possible ways in (b) and (c) to ensure that the final unused partition, $\text{IntComp}_{12}(1,1,1,1,1,7)$ (in light grey), forms a region containing every pitch class exactly once—both of which result in an incomplete solution to the all-partition array problem. For clarity, greyed out pitch classes belong to regions formed by partitions that have not been shown.

where r, $0 \leq r \leq |R|$, denotes a specified number of partitions in R to exclude from P_k. The first case in Eqn (1) states that the modified set of all unused partitions, P'_k, is the set difference of all unused partitions from 1 up to k and those found in R when (1) the number of partitions at the intersection of P_k and R is equal to r, and (2) the number of unused partitions is greater than r. When either of these two conditions are not met, P'_k is simply all unused partitions remaining (i.e., P_k). Collectively, these cases allow for partitions from R to be freely chosen so long as at least r partitions from R remain unused until there are r regions left to be found.

4.2 Second Proposed Search Heuristic

The second of our proposed heuristics is based in part on a modification to one originally proposed in [9]. A central feature of both heuristics, however, is that they work on the assumption that, since the matrix from which an all-partition array is constructed is regular (i.e., not ragged), regions should be chosen at each k so as to minimize the "raggedness" of their right hand column locations in each row. We make two significant improvements to this original heuristic which allow us to (1) work with a modification of the problem in which additionally required pitch classes appear as overlaps and not horizontal repetitions [20] and (2) minimize the raggedness of regions at each k in a way which takes into account both how far off from and in which direction their right hand column locations are from "ideal" locations specified in (1) while correcting for this same error found in the previous $k-1$ region.

For the first of our improvements, let us suppose we have a list of candidate compositions, $C_k = \langle c_{k,1}, c_{k,2}, ...c_{k,N} \rangle$ (corresponding to e.g., line 6 in Figure 3), at position k, where $1 \leq k \leq K$. Ideally, after choosing a composition for k, the rightmost column location of each row in this composition's region would be $J \cdot k/K$. This rightmost column location *after* choosing $c_{k,n}$ from C_k we denote $l'_{k,n,i}$ for a matrix row, i. For example, if we let $L'_{k,n}$ be equal to $\langle l'_{k,n,1}, \ldots, l'_{k,n,I} \rangle$, then the second region shown in Figure 2 would be $L'_{k,n} = \langle 3,3,5,5,2,4 \rangle$. To measure the raggedness or degree of difference in a region's rightmost column locations, $D_{k,n}$, that results from choosing $c_{k,n}$ in a fixed-size matrix, we use the following formula, based on city-block distance:

$$D_{k,n} = \sum_{i=1}^{I} \left| l'_{k,n,i} - \tfrac{J \cdot k}{K} \right|, \qquad (2)$$

where $|x|$ denotes the absolute value of x. The term, $J \cdot k/K$, specifies for any given region and row at k an "ideal" column location in a fixed-size matrix containing overlaps rather than in a potentially ragged matrix containing horizontal repetitions as in the original construction of an all-partition array. [6] This modification simplifies the modeling of the problem and aligns it with other proposed models [19, 20].

Our second improvement is based on the observation that Eqn (2) computes the magnitude and not the *direction* of the difference in each row between a region's right hand column location and its ideal location. This means, for example, that it is not possible to distinguish between two regions that are equally ragged according to $D_{k,n}$ but where one may be short of its ideal column locations and the other is longer. For this reason, we propose the use of an *adjustment* which captures for a given k how far off and in which direction the region chosen at $k-1$ is from its ideal column locations defined by the right hand term in Eqn (2). We can express this adjustment by defining for a region at k the rightmost column location for a matrix row, i, *before* choosing $c_{k,n}$ from C_k, which we denote $l_{k,i}$. For example, if we let $L_{k,i}$ be equal to $\langle l_{k,1}, \ldots, l_{k,I} \rangle$, then, for the second region shown in Figure 2, $L_{k,i} = \langle 3,3,2,2,2,0 \rangle$. Our second heuristic then is given by

$$S_{k,n} = \sum_{i=1}^{I} \left| \left(l'_{k,n,i} - \tfrac{J \cdot k}{K} \right) + \left(l_{k,i} - \tfrac{J \cdot (k-1)}{K} \right) \right|, \qquad (3)$$

where the adjustment, expressed as the difference $l_{k,i} - \frac{J \cdot (k-1)}{K}$, has been added to the difference shown in Eqn (2).

[6] In [9], this "ideal" column location was expressed as $12k/n$, where n is the number of matrix rows, due to the use of horizontal repetitions.

The absolute value of this sum is then taken and the resulting positive value is summed over all rows of i to form the final measure of adjusted raggedness, $S_{k,n}$. Use of Eqn (3) has the effect of preventing the accumulation of regions having the same directional error, either too short or too far past the ideal column locations for each row.

4.3 Modified backtracking algorithm with improved search heuristics

Whether only the magnitude of difference from an ideal location (Eqn (2)), or the magnitude and direction of this difference (Eqn (3)), is used, the goal of both heuristics is to minimize the degree of raggedness in the column locations of any one region formed by a candidate composition at k. We propose using a greedy strategy, for implementation with the backtracking algorithm proposed in [9], in which, for each k, we choose the candidate composition, c_n, in C_k that minimizes either $D_{k,n}$ or $S_{k,n}$. As a given region may have more than one possible set of overlaps (or none), we sort each region's sets of overlaps for each c_n according to whichever set results in the smallest value for the single heuristic, either $D_{k,n}$ or $S_{k,n}$, used globally throughout the search. The first of our heuristics shown in Eqn (1) can be implemented in two parts: one which occurs before the backtracking search begins and the other during the search when unused partitions from 1 up to k are found. Our two other heuristics shown in Eqn (2) and Eqn (3) can be implemented simply during the search when sorting the list of discovered candidate compositions in each C_k.

In Figure 5, asterisks indicate our modifications to the original backtracking algorithm in Figure 3 required to implement the new heuristics. Prior to the start of the backtracking search, the FINDCANDIDATES function finds the set of partitions, $\mathbf{R}$, that prove successful in forming regions at K, as required in Eqn (1) (see line 2 in Figure 5). These partitions are then passed to the function for finding unused partitions from 1 up to k in line 6, which returns a sorted set of lexicographically ordered compositions grouped by partition, $\mathbf{P}'$. In line 7, the pool of candidates for the kth region is chosen from this returned set. In line 11, the set of candidates found at k is sorted according to the value assigned to each candidate by either Eqn (2) or (3) (but not both).

```
BACKTRACKINGBABBITT*()
1    $\mathbf{C} \leftarrow \bigoplus_{i=1}^{K} \langle\langle\rangle\rangle$        ▶ Lists of candidates
2*   $\mathbf{R} \leftarrow$ FINDCANDIDATES(...)
3    $k \leftarrow 1$
4    while $0 < k \leq K$
5      if $\mathbf{C}[k]$ is empty
6*       $\mathbf{P}' \leftarrow$ FINDUNUSEDPARTITIONS($\mathbf{R}, \ldots$)
7*       $\mathbf{C}[k] \leftarrow$ FINDCANDIDATES($\mathbf{P}', \ldots$)
8        if $\mathbf{C}[k]$ is empty
9          $k \leftarrow k - 1$        ▶ Backtrack
10       else
11*        $\mathbf{C}[k] \leftarrow$ SORTBYHEURISTICS($\mathbf{C}[k]$)
12         $k \leftarrow k + 1$        ▶ Proceed
13     else        ▶ Backtracked to previously visited k
14       Select next overlaps for current candidate in $\mathbf{C}[k]$
15       if current overlaps for current candidate is nil
16         Current candidate becomes next candidate in $\mathbf{C}[k]$
17         if current candidate is nil
18           $\mathbf{C}[k] \leftarrow \langle\rangle$        ▶ Make empty
19           $k \leftarrow k - 1$
20         else
21           Select first overlaps (if any) for current candidate
22           $k \leftarrow k + 1$
23       else
24         $k \leftarrow k + 1$
25   return C
```

Figure 5: Pseudo-code for a simplified version of modifications to the BACKTRACKINGBABBITT algorithm originally posed in [9] required to implement our new heuristics, expressed in Eqn (1), Eqn (2), and Eqn (3). The '*' denotes modified lines to the original implementation shown in Figure 3.

5. SOLUTIONS

As evidence of the correctness for the modified backtracking algorithm in Figure 5 and a demonstration of its performance using our proposed heuristics, we solved for three different instances of matrices of two sizes found in Babbitt's works. Table 2 shows the approximate times in seconds and fewest number of backtracks required by our heuristics to solve these three matrices (Julia v.1.1.0 [11] running on a 2.4 GHz Intel Core i5 laptop with 8 GB RAM).

For the purposes of bench marking, all solving times shown in Table 2 were averaged over three runs and terminated after 2×10^7 backtracks if no solution was found. The reported solutions were found by running the algorithm with the given set of heuristics for all values of the parameter, r, from 1 to $|R|$ (where $|x|$ denotes cardinality) and selecting the one which resulted in the fewest number of respective backtracks. The first of these matrices was constructed manually by Babbitt and no known complete solution has existed prior to our solving it here using P' ($r = 11$) and D, and P' ($r = 13 = |R|$) and S—the latter of which resulted in a fewer number of backtracks. The complete solution to this matrix using the best combination of heuristics appears in Table 1 below. The second matrix in Table 2 was constructed manually by a student of Babbitt named David Smalley [9] and at least one known solution (discovered by Smalley) existed prior to our solving it here using P' ($r = 22 = |R|$) and D, and P' ($r = 18$) and S. The final matrix was previously solved in [19] using constraint programming in ≈ 30 minutes, however, the combination of P' ($r = 1$ where $|R| = 7$) and S discovered a solution in ≈ 22 minutes. Overall, this matrix required the highest number of backtracks (over 5 million in the best case) of all matrices tested here. In all cases, we have used P', as solving using either D or S alone proved infeasible within the specified backtracking limit. Similarly, using P' alone also proved unsuccessful.

The significant difference in solving times and number of required backtracks in Table 2 for each of the matrices using one set of heuristics or another is interesting to note. The only difference between the first and second matrices, for example, is their organization of pitch classes, as both are the same size and require the same number of regions and overlaps. However, using P' and D solves the second matrix in approximately 1 second and 2377 backtracks but

	29A843		-30	-071	-1		-1B6	-6	5	43	
	70B	-B	15	-56	9		-92	-28A34B0			
349B5A	1	2	-2		-2	6870	-0		9		
8102	6	-6	-67A	-A39B4	-4	-45	-5	-5	21860743B	-B9A50786	21A
6	5	-50A4	-4,B,8	-8	-873	-3129A	-A		-A		5640B8397
7		-78139	-9	2	56A0B	-B	4387	-791		-12	
641^2	631^3	541^3	$3^2 2^2 1^2$	5321^2	531^4	5421	4321^3	731^2	91^3	82^2	93

-3		A82		9	651		-1B07	-7	-7A32489		
	-0	-05716	9		-9A2438B		6		-6	-67	51094A8
	-9		453BA72	861		-10BA	-10539	4	1	08	67
	5	B943	0168	27AB3		-35492768	-8		0	14935A	B
-721	4						-4	5A06B	-B	-B2	-23
5A46B098	-83172BA6			450	-07		2	-23198	5		
831	81^4	543	741	$53^2 1$	732	84	$4^2 1^4$	$5^2 1^2$	71^5	62^3	$72^2 1$

	-9	-9			05B1	67	-72	38A	-A490		157
321	075	-5		B632A	-A8	-894	-4	165	-5	7B03	824
	2B43	-3			-3		59A160	-027	8	-89	-9A3
		87	2061A	95			3	4B	61	-12	
798	-816	-6BA04	-45	817	-792	-230B	-B			64A5	
-5064BA	-A	12	7938B	04	-46	5A1	8	9	732B		-B60
63^2	$43^2 1^2$	$52^2 1^3$	$5^2 2$	532^2	432^2	$43^2 2$	621^4	$3^3 21$	$4^2 21^2$	$4^2 2^2$	3^4

6B	8342	-2A	96	175	0B8	924	-4A36	-67	-7	-7	
9A	76	-61	-1B	-B		-B5	-5098				
-35	-5	B4780	-02	-2	1	63		AB9	5	-5	
0874	B	53	-3A	-A96		70		-02		-2	
21	-19	-9	78	-830	546A	-A	B271	-13	-38	-89AB460	58913274B0A6
	-0A		54	-4	3297	18		-854	-40AB6921	-13	
42^4	$42^3 1^2$	$52^3 1$	2^6	$3^3 1^3$	$4^2 31$	$32^4 1$	4^3	$3^2 2^3$	821^2	721^3	12

B105498A23	-36B5701A9	-9	-94283		0			-0B7561	
	42	3A	-A501B76		3A429856B17	-70	34	8A92	
		418207		6		54B93A	7620185A9B	34	706812
						-281			459BA3
-6		-65		293187					
7	8	B		-B4A05		6			
$10\ 1^2$	921	$62^2 1^2$	75	651	$11\ 1$	6321	$10\ 2$	642	6^2

Table 1: A generated all-partition array which solves one instance of the all-partition array problem based on a 6×96 matrix having 58 distinct regions and 120 overlaps. Each box contains the elements in A belonging to a region formed by a distinct integer partition, where a dash indicates those that overlap. Note that partitions are denoted using a shorthand notation, e.g., 4^3, where the base indicates the length of a part and the exponent denotes its number of occurrences. The integers 10 and 11 are the letters A and B, respectively.

		P', D			P', S		
	Matrix A	time (s)	backtracks	r	time (s)	backtracks	r
1.	Babbitt$(6, 96)$	176.92	583724	11	3.88	5909	13
2.	Smalley$(6, 96)$	1.25	2377	22	96.03	231933	18
3.	Babbitt$(4, 96)$	3791.94	14224645	1	1321.09	5390388	1

Table 2: Approximate times and fewest number of required backtracks for solving three different matrices using the search heuristics, P', D, and S.

using P' and S is significantly more costly. The reverse is true of these sets of heuristics in the first matrix. This result appears to support the findings reported in [20], that each matrix represents a unique problem space, which, in our case, may require the use of heuristics with different considerations. Contrary to what one might expect, however, the smaller third matrix actually proved more difficult than the larger and more combinatorially expansive matrices (i.e., requiring more regions) with a best-case solving time and number of backtracks roughly 3 orders of magnitude greater. Finally, it is important to note that while the best solving times and number of backtracks for the first two matrices are low, their use of values greater than 1 for the parameter, r, in P' means that some potential solutions are excluded, namely, those in which one or more of these r partitions in P' appear at positions, k, where $k \leq K - r$. This is not true, however, in the third matrix, where only one of the possible partitions in P' was excluded (i.e., $r = 1$).

6. CONCLUSION

In this paper, we provided improvements to an existing backtracking algorithm in the form of three search heuristics which proved successful in solving the all-partition array problem for three different instances found in Babbitt's music. Our findings demonstrate that, when used together, our proposed heuristics allow the backtracking approach to outperform other approaches. However, it is also apparent from our results that the solving time and number of required backtracks required is highly dependent on the specific matrix given as input. In future work, it would prove useful to investigate methods of analyzing the organization of pitch classes in a matrix prior to searching and using these findings to modify the ideal column locations accordingly during the search.

7. REFERENCES

[1] *Gurobi Optimizer Reference Manual version 6.0*. Gurobi Optimization, Inc., Houston, TX, 2015.

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

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

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

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

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

[7] Brian Bemman. *Computational Problems in Modeling the Compositional Process of Milton Babbitt*. Ph.d. diss., Aalborg University, 2017.

[8] Brian Bemman and David Meredith. Exact cover problem in Milton Babbitt's all-partition array. In Tom Collins, David Meredith, and Anja Volk, editors, *Mathematics and Computation in Music: Fifth International Conference, MCM 2015, London, UK, June 22–25, 2015, Proceedings*, volume 9110 of *Lecture Notes in Artificial Intelligence*, pages 237–242. Springer, Berlin, 2015.

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

[10] Zachary Bernstein. The problem of completeness in Milton Babbitt's music and thought. *Music Theory Spectrum*, 38(2):241–264, 2017.

[11] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[12] Richard M. Karp. Reducibility among combinatorial problems. In Miller R.E., Thatcher J.W., and Bohlinger J.D., editors, *Complexity of Computer Computations*, The IBM Research Symposia Series. Springer, 1972.

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

[14] Andrew Mead. *An Introduction to the Music of Milton Babbitt*. Princeton University Press, Princeton, NJ., 1994.

[15] Andrew Mead. Still being an american composer: Milton Babbitt's at eighty. *Perspectives of New Music*, 35(2):101–126, 1997.

[16] Robert Morris. Mathematics and the twelve-tone system: Past, present, and future. In Noll T. (eds) In: Klouche T., editor, *Mathematics and Computation in Music, MCM 2009 Proceedings*, volume 37 of *Communications in Computer and Information Science*, pages 266–288. Springer, Berlin, Heidelberg, 2009.

[17] Tamura. Naoyuki and Mutsunori Banbara. Sugar: A csp to sat translator based on order encoding. In *Proceedings of the 2nd International CSP Solver Competition*, Communications in Computer and Information Science, pages 65–69. 2008.

[18] Daniel Starr and Robert Morris. A general theory of combinatoriality and the aggregate, part 2. *Perspectives of New Music*, 16(2):50–84, 1978.

[19] Tsubasa Tanaka, Brian Bemman, and David Meredith. Constraint programming approach to the problem of generating Milton Babbitt's all-partition arrays. In Michael Rueher, editor, *International Conference on Principles and Practice of Constraint Programming, CP2016, Toulouse, France, September 5–9, 2016 Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 802–810. Springer, Berlin.

[20] Tsubasa Tanaka, Brian Bemman, and David Meredith. Integer programming formulation of the problem of generating Milton Babbitt's all-partition arrays. In *17th International Society for Music Information Retrieval Conference, August 7–11, 2016, New York, NY.*