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

Brian Bemman and David Meredith(✉)

Department of Architecture, Design and Media Technology, Aalborg University, Rendsburggade 14, Aalborg, Denmark
{bb,dave}@create.aau.dk
http://www.create.aau.dk

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

Keywords: Babbitt · Knuth · All-partition array · Exact cover · Computational music analysis

1 Introduction

The *exact cover problem* is a constraint satisfaction problem known to be NP-complete [5, p. 2]. It is defined as follows: given a collection of subsets, S, of a set, A, an *exact cover* of A is a sub-collection, s, of S that exhaustively and exclusively partitions A. The classic example of such a problem was provided by Scott and Trotter in 1958 [7]. They found all ways to cover a chessboard with the 12 distinct, non-overlapping pentaminoes while leaving the center four squares uncovered (see Fig. 1).

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

The exact cover problem is typically solved using a greedy backtracking algorithm that performs a depth-first search of the solution space [5, p. 2]. The backtracking process finds a complete solution to a problem by accumulating partial solutions to a set of constraints. It selects the first of these partial solutions until a complete solution is found, or, in the event that the constraints are no longer satisfied by the currently selected partial solution, it returns to the previous point and selects the next partial solution. It continues this process until either a solution is found or it fails.

T. Collins et al. (Eds.): MCM 2015, LNAI 9110, pp. 237–242, 2015.
DOI: 10.1007/978-3-319-20603-5_25

Fig. 1. An exact covering of part of a chessboard using 12 pentaminoes while leaving the center four squares uncovered. As taken from [5, p. 2].

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

Fig. 2. The beginning of the projection of Babbitt's *Sheer Pluck*.

2 All-Partition Array as Exact Cover

A significant number of Milton Babbitt's (1916–2011) works are based on the *all-partition array* [6], which is a sequence of distinct, non-overlapping aggregate-forming regions that completely and exactly partition a matrix of pitch classes called a *projection*. Construction of a *six-part* all-partition array results in an irregular projection of six rows and 696 pitch classes that can be partitioned into 58 aggregate regions. Figure 2 shows the beginning of the projection of Babbitt's *Sheer Pluck*. A projection is not the musical surface but, rather, a framework upon which the surface is based. Figure 3 illustrates the process of defining the first three aggregate regions in this projection.

Note in Fig. 3(b) that the first region contains an aggregate represented as a collection of row segments (from top to bottom) of length $3, 2, 1, 3, 1$, and 2. We define an *integer partition*, denoted by $\text{IntPart}(s_1, s_2, ..., s_k)$, to be a representation of an integer $n = \sum_{i=1}^{k} s_i$, as an *unordered* sum of k positive integers. For example, if $n = 12$ and $k = 6$, then one possible integer partition is $\text{IntPart}(3, 3, 2, 2, 1, 1)$. We define an *integer composition*, denoted by $\text{IntComp}(s_1, s_2, ..., s_k)$, to be a representation of an integer $n = \sum_{i=1}^{k} s_i$, as

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

(a) Area needing to be covered.

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

(b) 1st aggregate region.

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

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

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

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

Fig. 3. Process of forming a sequence of three aggregate regions in an excerpt from the projection shown in Fig. 2.

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

Using our definitions above, the problem we pose with respect to the all-partition array as an exact cover asks, "Given a universe of integer compositions (when $n = 12$ and $k = 6$), denoted by S, and a projection of 696 pitch classes in six rows, denoted by A, does there exist a sequence of 58 partitionally distinct, and aggregate-forming integer compositions, s, that exactly covers A?" We call this the *projection cover problem*. Our efforts to answer this question continue work started by Bazelow and Brickle [2, pp. 282–283], that asked a similar question of all-partition arrays in four parts. However, where their research sought to construct a projection, this paper begins with a completed projection and, as a method for musical analysis, seeks to efficiently reveal its all-partition array structure by discovering how (or if) it can be partitioned.

3 Solving the Projection Cover Problem

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

The algorithm begins in line 1 by computing a $6,188 \times 6$ list of compositions (in six parts), denoted by **compositions**. Lines 2–4 initialize **cList**[*cnt*] to be an empty list of 58 lists, **position** to be a 1×6 vector of indices (one for each row in **projection**), and *cnt* to be 1 (using 1-based indexing). Line 5 begins a **while** loop where *cnt* is less than or equal to the number of required compositions, 58. First, it checks to see whether **cList**[*cnt*] has been computed (line 6). **cList** contains candidate compositions at each *cnt*. Candidate compositions are those compositions that satisfy the constraints of a partial solution (i.e., are partitionally distinct and form a region containing an aggregate).

If **cList**[*cnt*] is empty (line 6), it has not been previously computed and so it calls ParseProjection, which returns **cList** and **currentComp** (line 7). **currentComp** is initialized by ParseProjection to be the first composition in **cList**[*cnt*] if **cList**[*cnt*] is not returned empty. If, after ParseProjection, **cList**[*cnt*] remains empty, there are no candidate compositions at this *cnt*. It must then backtrack, removing the previous composition from **partialSolutions**

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

Fig. 4. Pseudocode for implementation of BacktrackingBabbitt.

(lines 8–11). **partialSolutions** is a 58×6 list of candidate compositions at each *cnt* selected by the algorithm to be a partial solution. If **cList**[*cnt*] is not empty (line 12), then the algorithm has found at least one candidate composition at this *cnt*. The **currentComp** is stored in **partialSolutions**[*cnt*] and both **position** and *cnt* are incremented (lines 13–15). **position** is equivalent to counting from 1 a distance equal to the summation of like parts from each composition in **partialSolutions** from 1 to $cnt - 1$. **position** is incremented at each *cnt* by **currentComp**. In Fig. 3(d), **partialSolutions** currently holds $\langle 3, 2, 1, 3, 1, 2\rangle$ and $\langle 3, 3, 3, 3, 0, 0\rangle$ and so **position** would be $\langle 7, 6, 5, 7, 2, 3\rangle$.

If the first check for whether **cList** is empty (line 6) returns false, **cList**[*cnt*] has already been computed. This means the algorithm has backtracked at some point (line 16). It then attempts to select the next composition in **cList**[*cnt*] by incrementing the index of **currentComp** (line 17). If there is not a next composition here because this index exceeds the size of **cList**[*cnt*] (line 18), it must backtrack (lines 19–21). However, if there is another composition, it can proceed (lines 23–25). It continues this until it returns a complete **partialSolutions** or fails (line 26).

Figure 5 shows pseudocode for the ParseProjection function called in line 7 of BacktrackingBabbitt. ParseProjection begins by creating a copy of **compositions** called **comps** (line 1). Next, it removes from **comps** compositions partitionally equivalent to those already selected as partial solutions (line 2). It then loops through the rows and columns of **comps** (lines 4–6) and initializes *aggregate* to be an empty set (line 5). Next, it finds jth row segments in **projection** parsed by **comps**$[i][j]$ using the distance measured from **position**$[j]$ to the sum of **position**$[j]$ and **comps**$[i][j] - 1$. It stores these

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

Fig. 5. The ParseProjection function.

segments in *aggregate* (lines 7–8). After removing any duplicate integers from *aggregate* (line 10), if *aggregate* is complete, it has found a candidate composition and saves this composition in **cList**[*cnt*][*k*] (lines 10–12). The algorithm then assigns **currentComp** to be the first composition in **cList**[*cnt*] and returns **cList** and **currentComp** (lines 13–14).

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

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

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

(a) Compositions found by human analyst.

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

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

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

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

4 Conclusion

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

Acknowledgments. 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

1. Babbitt, M.: Set structure as a compositional determinant. J. Music Theor. **5**, 72–94 (1987)
2. Bazelow, A.R., Brickle, F.: A partition problem posed by Milton Babbitt. Perspect. New Music **14**(2), 280–293 (1976)
3. Bemman, B., Meredith, D.: From analysis to surface: generating the surface of Milton Babbitt's Sheer Pluck from a parsimonious encoding of an analysis of its pitch-class structure. In: The Music Encoding Conference, Charlottesville, VA, 20–23 May 2014
4. Eger, S.: Restricted weighted integer compositions and extended binomial coefficients. J. Integer Seq. **16**(13.1.3), 1–25 (1997)
5. Donald, K.: Dancing links. 22 February 2000. http://www-cs-faculty.stanford.edu/uno/musings.html
6. Mead, A.: An Introduction to the Music of Milton Babbitt. Princeton University Press, Princeton (1994)
7. Scott, D.S.: Programming a combinatorial puzzle. Technical report No. 1, Princeton University Department of Electrical Engineering, Princeton, NJ, 10 June 1958