THE INSTITUTE FOR SYSTEMS RESEARCH

ISR TECHNICAL REPORT 2008-29

Constructing Perfect Aggregations to Eliminate Response Time Variability in Cyclic Fair Sequences

Jeffrey W. Herrmann

Constructing Perfect Aggregations to Eliminate Response Time Variability in Cyclic Fair Sequences

Jeffrey W. Herrmann
Department of Mechanical Engineering
2181 Martin Hall
University of Maryland
College Park, MD 20742
301-405-5433
jwh2@umd.edu

Abstract

Fair sequences are useful in a variety of manufacturing and computer systems. This paper considers the generation of cyclic fair sequences for a given set of products, each of which must be produced multiple times in each cycle. The objective is to create a sequence so that, for each product, the variability of the time between consecutive completions is minimized. Previous work introduced an aggregation approach that can reduce response time variability (RTV) dramatically. However, in some cases, aggregating more carefully can generate sequences with zero RTV. We call this a "perfect aggregation." This paper discusses properties of instances that have perfect aggregations. Moreover, we present techniques that can find a perfect aggregation if one exists.

Introduction

When a resource must serve many demands simultaneously, it is important to schedule the resource's activities in some fair manner, so that each demand receives a share of the resource that is proportional to its demand relative to the competing demands. A mixed-model assembly line, to mention one standard example, should produce different products at rates that are close to the given demand for each product. Similarly, computer systems must service requests that have different priorities.

Both applications demonstrate the need for a *fair sequence*. Kubiak (2004) provides a good overview of fair sequences and the product rate variation problem and reviews important results. Miltenburg (1989) and Inman and Bulfin (1991) were some of the first to discuss the problem of mixed-model assembly lines. Waldspurger and Weihl (1995) discuss the problem in computer system applications and provide an important stride scheduling heuristic. Kubiak (2004) presents a parameterized stride scheduling heuristic.

In the cyclic situation, an important objective is to minimize the variability in the time between consecutive completions of the same task. Thus, we will use the response time variability (RTV) metric, which was presented and analyzed by Corominas *et al*. (2007). Herrmann (2007) independently studied this measure as well, and Garcia *et al*. (2006) presented metaheuristic procedures for the problem.

Corominas *et al*. (2007) showed that the RTV problem is NP-hard and presented a dynamic program and a mathematical program for finding optimal solutions. Because those approaches required excessive computational effort, they conducted experiments to evaluate the performance of various heuristics. However, the heuristics performed poorly for some classes of problem instances. Independently, Herrmann (2007) presented a heuristic that combined aggregation and parameterized stride scheduling. This aggregation approach (which we will call "natural aggregation") combines products with the same demand into groups, creates a sequence for those groups, and then disaggregates the sequence into a sequence for each product.

Herrmann (2008) extended these last two works by precisely defining the natural aggregation approach and describing the results of extensive computational experiments using it in combination with the heuristics presented by Corominas *et al*. (2007). The results of these

experiments show that the solutions generated using the natural aggregation approach have lower RTV than solutions generated without it.

Waldspurger and Weihl (1995) presented a hierarchical stride scheduling algorithm that combines products into groups. They suggested the use of a binary tree to minimize the maximum absolute deviation. The key distinction between their hierarchical stride scheduling algorithm and the natural aggregation approach is that their algorithm requires using the stride scheduling algorithm to disaggregate each group, since the products in a group may have unequal demands. Also, the placement of products in the tree not specified. Because the natural aggregation approach groups products with equal demand, the disaggregation is much simpler. The limitation, however, is that the problem must have some equal demand products.

In their discussion of the periodic maintenance scheduling problem, a related problem, Wei and Liu (1983) suggested that machines with the same maintenance interval could be replaced by a substitute machine with a smaller maintenance interval and that this replacement would facilitate finding a feasible solution. This concept was not developed into a solution algorithm but is similar to the natural aggregation approach.

The natural aggregation procedure described in Herrmann (2008) is simple and effective. However, in some cases, more sophisticated aggregations can combine all of the products into one group. If there is only one group, the disaggregation leads directly to a sequence with zero RTV. This paper explores this idea in depth.

Problem Formulation

Given a single server that must produce *n* products, each with a demand d_i that is a positive integer, let $D = d_1 + \cdots + d_n$. A feasible sequence has length *D*, and each product *i* occurs exactly d_i times in the sequence. We assume that each product requires the same amount

of time, so we can ignore time and consider only the positions in the sequence. Moreover, this sequence will be repeated, and we will call each occurrence a cycle. The response time variability (RTV) of a feasible sequence equals the sum of the response time variability for each product. If product *i* occurs at positions $\{p_{i1},\ldots,p_{id_i}\}$, the response time variability is a function of the intervals between each position, which are $\{\Delta_{i1},\ldots,\Delta_{id_i}\}$, where the intervals are measured as follows (with $p_{i0} = p_{id_i} - D$):

$$\Delta_{ik} = p_{ik} - p_{i,k-1}$$

The average interval for product *i* is D/d_i, so we calculate RTV as follows:

$$RTV = \sum_{i=1}^{n}\sum_{k=1}^{d_i}\left(\Delta_{ik} - \frac{D}{d_i}\right)^2$$

Minimizing RTV is NP-hard (Corominas *et al.*, 2007). Note that changes to the absolute positions do not change the variability. The objective function value is invariant under any translations or reflection.

Zero RTV Sequences

In this paper we are concerned with the problem of finding zero RTV sequences. As discussed later, a necessary condition for a zero RTV sequence is that all d_i divide *D*. Although the general RTV problem is NP-hard, the proof by Corominas *et al.* (2007) does not consider instances in which all d_i divide *D*.

Under this condition, it is clear that the problem of finding a zero RTV sequence is closely related to the periodic maintenance scheduling problem (Wei and Liu, 1983). Bar-Noy *et al.* (2002) showed that the periodic maintenance scheduling problem (PMSP) is NP-hard. An

instance of the PMSP specifies, for each of *m* machines, a scheduling interval l_i such that $\frac{1}{l_1}+\cdots+\frac{1}{l_m} < 1$. Let *L* be the least common multiple of the scheduling intervals. The problem is to determine whether there exists a sequence of length *L* in which machine *i* is assigned to L/l_i positions, the positions assigned to machine *i* are all exactly l_i time slots apart (including the distance between the last and first positions in the cycle), and no more than one machine is assigned to each position.

It is easy to transform an instance of PMSP into an instance of the RTV problem. Given an instance $l_1, \ldots, l_m$ of the PMSP, construct an instance of the RTV problem as follows. For $i = 1, \ldots, m$, set $d_i = L/l_i$. Let $P = d_1+\cdots+d_m$. Note that $P < L$. Let $n = m + L - P$, and add $n - m$ new products to the instance. For $i = m + 1, \ldots, n$, set $d_i = 1$. Thus, $D = d_1+\cdots+d_n = L$, and all d_i divide *D*. It is clear that PMSP has a solution if and only if there is a zero RTV sequence for this instance.

The problem with this transformation is that the number of new products that must be added depends upon *L*, which could be as large as $l_1 \cdots l_m$. Thus, the computational complexity of the problem of finding a zero RTV sequence remains open (Kubiak, 2004).

We also note that the problem of finding a zero RTV sequence is the constant gap problem and is equivalent to the problem of finding *n* positive integers $f_1, \ldots, f_n$ such that $\{(f_1-1, \frac{D}{d_1}), \ldots, (f_n-1, \frac{D}{d_n})\}$ is an exact covering sequence (Kubiak, 2004). The complexity of this problem remains open.

Aggregation

A problem instance can be transformed into an equivalent instance with the same total demand *D* but fewer products by combining two or more products that have the same demand. We call this transformation *aggregation*. A sequence for the new instance can be changed into a sequence for the original sequence by reassigning the positions that were assigned to the aggregated product, as discussed below.

Following Herrmann (2008), we describes the natural aggregation approach that iteratively transforms the original instance into one with fewer products. This was first introduced in Herrmann (2007) and is similar to the substitution concept discussed by Wei and Liu (1983).

Let an instance I_k be a set of products P_{kj} for $j = 1,\ldots,n_k$. It will be convenient to represent a product as a set. Each product P_{kj} has demand d_{kj}. We assume that the products are sorted so that $d_{k1} \le d_{k2} \le \cdots \le d_{kn_k}$. Let I_0 be the original instance, and each $P_{0j} = \{j\}$.

Given an instance I_k, the aggregation procedure transforms I_k into a new problem instance I_{k+1} as follows. First, find the smallest *i* such that $i < n_k$ and $d_{ki} = d_{k,i+1}$. If there exists no such *i*, return with $I_{k+1} = I_k$ because no further aggregation is possible.

If $d_{kn_k} = d_{ki}$, let $m = n_k - i$. Else, find *m* such that $d_{ki} = d_{k,i+m}$ and $d_{k,i+m+1} > d_{k,i+m}$.

Create the new instance I_{k+1} as follows: Assign $n_{k+1} = n_k - m$. Then, $P_{k+1,j} = \{j\}$ and $d_{k+1,j} = d_{kj}$ for $j = 1,\ldots,i-1$. Then, $P_{k+1,i} = \{i,\ldots,i+m\}$ and $d_{k+1,i} = (m+1)d_{ki}$. Finally, $P_{k+1,j} = \{j+m\}$ and $d_{k+1,j} = d_{k,j+m}$ for $j = i+1,\ldots,n_{k+1}$. Renumber as needed so that the products in I_{k+1} are sorted by demand and return I_{k+1}.

The total demand in the new instance will equal the total demand of the original instance because the demand of the new product (which we call a "group") equals the total demand of the products that were removed.

We run the aggregation procedure until no further aggregation is possible to generate a sequence of instances $I_0, \ldots, I_H$. (H is the index of the last aggregation created.) The aggregation can be done at most $n-1$ times because the number of products decreases by at least one each time the aggregation procedure is called (unless no aggregation occurs). Thus $H \le n-1$.

We can then apply a sequence generation algorithm to the most aggregated instance I_H to generate a sequence S_H. We disaggregate S_H to generate S_{H-1} and then continue to disaggregate each sequence in turn to generate $S_{H-2}, \ldots, S_0$. S_0 is a feasible sequence for I_0, the original instance.

The disaggregation of sequence S_k is performed as follows: Let F_k be the set of products j in I_k such that $|P_{kj}| = 1$. F_k will include $n_k - 1$ products. Let g be the remaining product and let $P_{kg} = \{i, \ldots, i+m\}$. To create sequence S_{k-1} for instance I_{k-1}, first let $c = 0$. Then, loop over $a = 1, \ldots, D$. Let $j = S_k(a)$. If $j \in F_k$, then let q be the element of P_{kj}. (There is exactly one.) Assign $S_{k-1}(a) = q$. If j is not in F_k, then $j = g$, the group that needs to be disaggregated. Assign $S_{k-1}(a) = i + c$ and update $c = c + 1 \pmod{m+1}$.

Consider the group g, which is formed from $m+1$ products. It has been assigned d_{kg} positions in the sequence. According to the aggregation scheme, $d_{kg} = (m+1)\, d_{k-1,i}$. When creating S_{k-1}, the first position assigned to g in S_k goes to i (the first product in the group), the

second position assigned to *g* goes to product $i+1$, and so forth. This continues until all d_{kg} positions have been assigned. Each product in the group gets $d_{k-1,i}$ positions.

Aggregation runs in O(n^2) time because each aggregation requires O(n) time and there are at most $n-1$ aggregations. Likewise, because each sequence disaggregation requires O(*D*) effort, disaggregation runs in O(nD) time in total.

Tables 1 and 2 present an 8-product example that is aggregated three times. S_2 is a feasible sequence for I_2. Note that, at each step of disaggregating the sequence, copies of product *j* are replaced by the product(s) in P_{kj}. Also, note that, in the completely disaggregated sequence, the RTV of product 8 does not equal zero.

Table 1. An 8-product instance and the aggregate instances formed from it. (The aggregation combines the double underlined products at each step.)

		d_{kj}, j =							
k	n_k	1	2	3	4	5	6	7	8
0	8	1	1	1	1	1	2	2	3
1	4	2	2	3	5				
2	3	3	4	5					

Table 2. The disaggregation of sequence S_2 for instance I_2.

S_2	3	2	1	3	2	1	3	2	1	3	2	3
S_1	4	1	3	4	2	3	4	1	3	4	2	4
S_0	1	6	8	2	7	8	3	6	8	4	7	5

Perfect Aggregation

Aggregation simplifies sequencing. Ideally, if only one product remains in the final instance, then it gets all of the positions in the sequence, and one can immediately proceed to disaggregation, which creates a sequence with zero RTV. We call this a *perfect aggregation*.

Hereafter, we will assume that the greatest common divisor *g* of all of the demands is 1. If $g > 1$, then we can divide all of the demands by *g*, find a sequence for the reduced instance,

and then concatenate *g* copies of this sequence to form a solution to the original instance. If the solution to the reduced instance has zero RTV, the solution to the original instance also has zero RTV.

In some cases, the natural aggregation approach will yield a perfect aggregation. For instance, if there are *m* products that have demand *a*, *n-m* products with demand *b*, and $ma = (n-m)b$, then the first aggregation combines the first *m* products into one with a demand of *ma*, the second aggregation combines the other *n-m* products into one with a demand of $(n-m)b = ma$, and the third aggregation combines the two remaining products into one product with a demand of $2ma$.

In some cases, no perfect aggregation is possible. Consider an instance with demands of 1, 1, 4, and 6. Because aggregation can combine products only when their demand is equal, the only possible aggregation leads to an instance with demands of 2, 4, and 6. This cannot be aggregated further.

In other cases, a perfect aggregation is possible even if the natural aggregation is not perfect. Consider again the 8-product example from the previous section. Table 3 shows a perfect aggregation that first combines products 1 and 2 to create a product with demand of 2. After that, the aggregation proceeds as in the natural aggregation approach. Table 4 shows the resulting sequences. The RTV of the last sequence equals zero.

Table 3. A perfect aggregation of the 8-product instance.

		d_{kj}, j =							
k	n_k	1	2	3	4	5	6	7	8
0	8	1	1	1	1	1	2	2	3
1	7	1	1	1	2	2	2	3	
2	5	2	2	2	3	3			
3	3	3	3	6					
4	2	6	6						
5	1	12							

Table 4. The disaggregation of the sequences corresponding to the perfect aggregation.

S_5	1	1	1	1	1	1	1	1	1	1	1	1
S_4	1	2	1	2	1	2	1	2	1	2	1	2
S_3	1	3	2	3	1	3	2	3	1	3	2	3
S_2	4	1	5	2	4	3	5	1	4	2	5	3
S_1	1	4	7	5	2	6	7	4	3	5	7	6
S_0	3	1	8	6	4	7	8	2	5	6	8	7

Necessary Conditions

Given an instance of the RTV problem, it is useful to know whether a perfect aggregation is possible.

The first condition is necessary for a zero RTV sequence. If an instance does not meet this condition, there can be no zero RTV sequence and thus there can be no perfect aggregation.

(1) All $d_i \mid D$.

Next we will consider conditions that are necessary to form a perfect aggregation. Note: $lcm(d_i, d_j)$ is the least common multiple of d_i and d_j. $N(a)$ is the number of products that have demand a.

(2) Let a be the smallest demand and let b be the smallest demand greater than a. $N(a) \geq b/a$.

(3) There exists $e < D$ such that all $d_i \mid e$ and $e \mid D$. In other words, $lcm(d_1, \ldots, d_n) < D$.

If condition (2) is not satisfied, then there is no way to aggregate the products with the minimum demand into a group that can then be combined with any other products. We note that condition (2) is similar to the condition that $d_1 = d_2$, which is necessary for a zero RTV sequence (cf. Lemma 19.4, Kubiak, 2004).

Condition (3) is needed because, in a perfect aggregation, there is, at the end, an aggregation of products (some might be groups, but all have the same demand) into one product with demand of D. The demand of these products, which we denote as e, must be less than D

and must divide *D*. Every product in the original instance either has been aggregated into one of these products or is one of these products. Therefore, each product's demand must divide *e*. The smallest such *e* is the least common multiple of the demands.

Sufficient Conditions

The following special cases yield a perfect aggregation immediately.

(1) All $d_i = D/n$.

(2) The *n* products include *m* products that have demand $d_i = a$, *n*-*m* products with demand $d_i = b$, and both $a \mid D$ and $b \mid D$.

(3) There exists $e < D$ such that all $d_i \mid e$, $e \mid D$, and $e \mid d_i N(d_i)$.

The proof that condition (1) yields a perfect aggregation is trivial: just aggregate the *n* products immediately. If condition (2) holds, without loss of generality assume $a < b$. Note that $ma + (n-m)b = D$, so $b \mid ma$. Let *e* be the least common multiple of *a* and *b*. Because $b \mid ma$, $e \mid ma$, and $e < D$. Let $r = e/a$. We can combine the *m* products with demand of *a* into m/r products with demand of *e*. Likewise, $e \mid (n-m)b$. Let $s = e/b$. We can combine the $n - m$ products with demand of *b* into $(n-m)/s$ products with demand of *e*. The resulting instance has $m/r + (n-m)/s$ products with demand of *e*.

If condition (3) holds, let $r_i = e/d_i$. Because $e \mid d_i N(d_i)$, $r \mid N(d_i)$. We can combine the $N(d_i)$ products with demand d_i into $N(d_i)/r_i$ products with demand of *e*. After doing this for all of the different demands, the resulting instance has $\sum N(d_i)/r_i = \sum d_i N(d_i)/e = D/e$ products with demand of *e*. (The summations are over the distinct values of demand.)

Zero RTV and Perfect Aggregation

It is clear that finding a perfect aggregation is sufficient to construct a zero RTV sequence. However, a perfect aggregation is not necessary to generate a zero RTV sequence, as we can see in the following example.

Consider a 48-product instance in which 45 products have a demand of 1, $d_{46} = 4$, $d_{47} = 5$, and $d_{48} = 6$. Note that $D = 60$. Construct a sequence S as follows: product 46 is in positions 2, 17, 32, and 47; product 47 is in positions 1, 13, 25, 37, and 49; and product 48 is in positions 10, 20, 30, 40, 50, and 60. The remaining 45 positions are assigned to products 1 to 45 in any way. The RTV of S equals zero. However, there is no perfect aggregation because the least common multiple of 1, 4, 5, and 6 equals 60. Thus, the instance does not satisfy a necessary condition for perfect aggregation.

Also, we note that an instance may have more than one perfect aggregation. Consider a 13-product instance in which nine products (d_1 to d_9) have a demand of 1, three products (d_{10} to d_{12}) have a demand of 3, and $d_{13} = 6$. $D = 24$. One perfect aggregation proceeds by combining d_1 through d_6, and then d_{10} and d_{11} (this creates two new products with demand of 6). Then, combine d_7, d_8, and d_9 to form a new product with demand of 3. Combine this new product with d_{12} to get a fourth product with demand of 6.

A second perfect aggregation proceeds by combining d_1 through d_3, d_4 through d_6, and d_7 through d_9 to create three new products with demand of 3. Then, combine two of the products with demand of 3 to create a new product with demand of 6. Repeat this last step twice to yield an instance with four products with demand of 6.

Finding a Perfect Aggregation

For instances that satisfy all the necessary conditions but do not satisfy any of the sufficient conditions, we must search for a perfect aggregation in order to determine if one exists. This section presents an enumerative technique that tries to match products to factors of *D*, which we will call "openings."

To be sure that we exhaustively search all possibilities, we will maintain a sequence of openings and always operate on the first one. There are two key operations. The first operation matches the first opening to a product whose demand equals the size of the opening. The second operation replaces the first opening by a number of equally-sized openings whose combined size equals the size of the opening being replaced.

For instance, consider an instance with three products with demand of 1 and a fourth product with demand of 3. In this case, $D = 6$. Then, we can divide *D* into two openings of 3. Then we match the first of those openings to the fourth product, because the demand of the fourth product equals the size of the opening. This leaves three unmatched products. The remaining opening is 3, and we can divide that opening into three openings of 1. These can be matched to the three products with demand of 1.

Let *O* be a sequence of *p* openings with sizes $s_1, \ldots, s_p$. Let *U* be a set of *m* unmatched products with demands $d_1, \ldots, d_m$.

The procedure *match*(*O*, *U*) finds a product *k* in *U* such that $d_k = s_1$, removes product *k* from *U*, and removes the first opening from *O*. For example, if $O = (3, 3)$ and $U = \{1, 1, 1, 3\}$, as in the example above, *match*(*O*, *U*) returns $O = (3)$ and $U = \{1, 1, 1\}$.

The procedure *split*(*O*, *r*) replaces the first opening with *r* openings of size s_1 / r. (This assumes that *r* is a factor of s_1.) For example, if *O* = (3), as in the example above, *split*(*O*, 3) returns the openings (1, 1, 1).

The procedure *solve*(*O*, *U*) returns "true" if all of the products in *U* can be matched to the openings in *O*, possibly after splitting them, and "false" otherwise. For example, if *O* = (3, 3) and *U* = {1, 1, 1, 3}, *solve*(*O*, *U*) returns "true." However, if *U* = {2, 2, 2}, *solve*(*O*, *U*) returns "false." Procedure *solve* uses some rules to check necessary and sufficient conditions for openings and works as follows.

1. Check the necessary conditions for openings. If any are violated, return "false."

2. Check the sufficient condition for openings. If it is true, return "true."

3. If there exists no product *k* in *U* such that $d_k = s_1$, go to step 4. Otherwise, match(*O*, *U*). If there are no more products in *U*, then return "true"; otherwise repeat step 3.

4. If any products were removed from *U* by step 3, check the necessary conditions for openings again. If any are violated, return "false."

5. If any products were removed from *U* by step 3, check the sufficient condition for openings again. If it is true, return "true."

6. Let $\{a_1, \ldots, a_q\}$ be the set of prime factors of s_1. Let *j* = 1.

7. Let O_j = split(*O*, a_j) and then perform *solve*(O_j, *U*). If this is "true," then return "true."

8. If *j* < *q*, then increase *j* by 1 and return to step 7.

9. Return "false."

There are three necessary conditions for openings: (a) $\min\{s_1,\ldots,s_p\} \geq \min\{d_1,\ldots,d_m\}$. If this is not true, then there is no demand small enough to go into the smallest opening. (b) $\max\{s_1,\ldots,s_p\} \geq \max\{d_1,\ldots,d_m\}$. If this is not true, then there is no opening large enough for the largest demand. (c) For each $k = 1, \ldots, m$, there exists j such that $d_k \mid s_j$. Otherwise, there is at least one product that cannot be aggregated into these openings.

To illustrate the third necessary condition for openings, suppose $O = (3, 3)$ and $U = \{2, 2, 2\}$. The products cannot be aggregated because an opening of size 3 cannot be split into openings of size 2.

The only sufficient condition checked in the *solve* procedure is the condition that all of the demands in U are equal and their sum equals the size of the first opening. For example, this condition is true if $O = (3)$ and $U = \{1, 1, 1\}$.

To use this procedure on an instance, we set $U = \{ d_1,\ldots,d_m \}$, let e be the least common multiple of $d_1,\ldots,d_m$, and set $O = (e, \ldots, e)$. That is, there are D/e openings of size e.

For example, consider the 8-product instance described earlier. $U = \{1, 1, 1, 1, 1, 2, 2, 3\}$, and $O = (6, 6)$. The indentation of the paragraphs refers to nested calls of the procedure *solve*.

Calling *solve*(O, U) first checks the necessary conditions (which are satisfied) and the sufficient condition (which is not). No products have demand equal to 6. The prime factors of 6 are 2 and 3. The procedure splits the first opening to make $O_1 = (3, 3, 6)$ and calls *solve*(O_1, U).

This procedure first checks the necessary conditions (which are satisfied) and the sufficient condition (which is not). One product has demand of 3, so O_1 is reduced to

(3, 6) and U is reduced to {1, 1, 1, 1, 1, 2, 2}. The necessary conditions are still satisfied and the sufficient condition is not. The only prime factor of 3 is 3, so the procedure splits the first opening and calls *solve*((1,1,1,6), {1, 1, 1, 1, 1, 2, 2}).

This procedure first checks the necessary conditions (which are satisfied) and the sufficient condition (which is not). The first three openings can be matched to products, which leaves one opening of 6 and the demands {1, 1, 2, 2}. The necessary conditions are still satisfied and the sufficient condition is not. The prime factors of 6 are 2 and 3. The procedure first calls *split*((6), 2) to get the openings (3, 3) and then calls *solve*((3, 3), {1, 1, 2, 2}).

This procedure returns a "false" because the third necessary condition is not true.

Then the procedure calls *split*((6), 3) to get the openings (2, 2, 2) and then calls *solve*((2, 2, 2), {1, 1, 2, 2}).

This procedure first checks the necessary conditions (which are satisfied) and the sufficient condition (which is not). The first two openings can be matched to products, which leaves one opening of 2 and the demands {1, 1}. The necessary conditions are still satisfied. Moreover, the sufficient condition is satisfied, so the procedure returns "true."

The procedure *solve*((1,1,1,6), {1, 1, 1, 1, 1, 2, 2}) returns "true."

The procedure *solve*(O_1, U) returns "true."

The procedure *solve*(O, U) returns "true."

While not discussed here, by keeping track of which procedure calls return a "true," the procedure can be implemented to return the information needed to construct the perfect aggregation.

Other Techniques

Experimentation with the enumerative procedure above showed that it used excessive computational effort for some instances with a large number of products (as discussed below). These instances have some products with demand greater than 1 but also many products with demand of 1. To find perfect aggregations more quickly, we developed two preprocessing procedures. Both aggregate products into groups with a combined demand of *e*, where *e* is the least common multiple of all the demands, divides *D*, and is less than *D*.

The procedure P1 aggregates products with the same demand into groups with demand of *e*. That is, if $N(d_i) \geq e/d_i$ and $d_i > 1$, then e/d_i products can be aggregated into a group with demand of *e*; therefore remove them from the instance. This is repeated until no more groups can be formed this way.

The procedure P2 uses products with demand of 1 to form groups with products with larger demand. Consider some demand $d_i > 1$ and let $m_i = N(d_i)$. (We assume that $m_i < e/d_i$; otherwise, some could have been removed with P1.) If $N(1) \geq e - d_i m_i$, then we can form $e/d_i - m_i$ new groups with demand of d_i by repeatedly aggregating d_i products with demand of 1, and then we can combine these new products with the m_i products with demand equal to d_i to form a new product with demand of *e*. Thus, remove from the instance the m_i products with demand equal to d_i and the $e - d_i m_i$ products with demand of 1. Repeat this with all demands that are greater than 1.

Running procedures P1 and P2 on an instance creates a reduced instance. If the reduced instance has no products at all, the procedures show that there is a perfect aggregation. If we can aggregate the reduced instance into products with demand of e, then we have a perfect aggregation for the original instance. However, either procedure may reduce an instance with a perfect aggregation into an instance without one, as we can see in the two following examples. Thus, both procedures must be considered as heuristics.

First, consider an instance with the following demands: (2, 2, 2, 2, 2, 2, 2, 3, 3, 4). In this case, $e = 12$, and there is a perfect aggregation. Combine products 1 and 2 and then combine products 3 and 4 to form two new products with demand of 4. Combine products 5, 6, and 7 into a new product with demand of 6. This yields an instance with demands of (3, 3, 4, 4, 4, 6), for which the perfect aggregation is clear. However, running P1 reduces the original instance into (2, 3, 3, 4), which cannot be aggregated into a group with demand of 12.

Second, consider an instance with the following demands: (1, 1, 1, 1, 1, 2, 3, 4, 4, 6). In this case, $e = 12$, and there is a perfect aggregation. Combine products 1 and 2 into a product with demand of 2, and combine products 3, 4, and 5 into a product with demand of 3. This yields an instance with demands of (2, 2, 3, 3, 4, 4, 6), for which the perfect aggregation is clear. However, running P2 reduces the original instance into (1, 2, 3, 6), which cannot be aggregated at all.

Computational Experiments

In order to determine when perfect aggregations may occur, we considered 4700 instances of the RTV problem. These are the same instances considered in Herrmann (2008).

In these instances, necessary condition 2 was satisfied most often (in 3765 instances). Only 796 instances satisfied both necessary conditions 1 and 3. In general, for a given D, as n

increases, the number of instances (in a set of 100) that satisfied the necessary conditions and sufficient conditions increases. This occurs primarily because the product demands become very small (near 1) as *n* increases. For *D* = 1500 and *n* = 1000, 1200, 1300, or 1400, all of the instances satisfied all three necessary conditions.

Many sets had no instances that satisfied any sufficient conditions. However, with *D* = 100 and *n* = 90, 57 out of the 100 instances satisfied sufficient conditions 2 and 3, the most of any set of these instances. Table 5 shows the problem sets with instances that did satisfy at least one sufficient condition.

Table 5. Problem sets with instances that satisfied sufficient conditions.

		Sufficient Conditions		
D	*n*	**1**	**2**	**3**
100	70	0	0	1
	80	0	10	10
	90	0	57	57
500	450	0	8	8
1000	900	0	1	1
1500	1100	0	0	1
	1300	0	0	9
	1400	0	4	17

For *D* = 100, 500, and 1000, those instances that satisfied all three necessary conditions also satisfied at least one sufficient condition. Only for *D* = 1500 were there instances that satisfied all three necessary conditions but none of the sufficient conditions.

The techniques for finding perfect aggregations were tested on the 100 instances with *D* = 1500 and *n* = 1000 (all of these satisfied all three necessary conditions). For 68 of these instances, procedure *solve* was able to find a perfect aggregation in less than 0.5 seconds. For 19 of these instances, procedure *solve* reached the maximum recursion limit of 500. The other 13 required excessive computation time.

However, using procedures P1 and P2 on these 100 instances was more successful. The remaining instance was always a set of products with demand equal to 1, for which constructing a perfect aggregation into groups of the appropriate size is trivial. Therefore, there is a perfect aggregation and a zero RTV sequence for all 100 of these instances. In Herrmann (2008), the best combination of the natural aggregation approach and heuristic yielded sequences with an average RTV of 64.8 for these instances.

Summary and Conclusions

This paper discussed the properties of perfect aggregations that can be used to construct zero RTV sequences. A perfect aggregation is a series of aggregations that transforms an instance with multiple products into an instance with only one product. Each aggregation replaces a set of products with a new product whose demand equals the total demand of the products in that set. Unlike the natural aggregation approach of Herrmann (2008), each aggregation may combine only some of the products that have the same demand.

Given an instance of the RTV problem, constructing a perfect aggregation for that instance leads immediately to a zero RTV sequence. However, there exist instances that have zero RTV solutions but do not have a perfect aggregation. Thus, a perfect aggregation is sufficient but not necessary for having a zero RTV sequence.

The paper presented necessary conditions and sufficient conditions for a perfect aggregation. The paper also presented an enumerative technique that searches for a perfect aggregation and heuristic procedures that can be used to simplify an instance before beginning the search. Computational experiments show that these techniques can find perfect aggregations for instances that satisfy the necessary conditions but not the sufficient conditions.

In the future, it will be useful to identify more necessary and sufficient conditions for perfect aggregations and to study further the complexity of finding a zero RTV sequence. It will also be interesting to apply these ideas to other problems of creating fair sequences.

References

Bar-Noy, Amotz, Randeep Bhatia, Joseph Naor, and Baruch Schieber, "Minimizing service an operation costs of periodic scheduling," *Mathematics of Operations Research*, Volume 27, Number 3, pages 518-544, 2002.

Corominas, Albert, Wieslaw Kubiak, and Natalia Moreno Palli (2007) "Response time variability," *Journal of Scheduling*, 10:97-110.

Garcia, A., R. Pastor, and A. Corominas (2006) "Solving the Response Time Variability Problem by Means of Metaheuristics," in *Artificial Intelligence Research and Development*, edited by Monique Polit, T. Talbert, and B. Lopez, pages 187-194, IOS Press, 2006.

Herrmann, Jeffrey W. (2007) "Generating Cyclic Fair Sequences using Aggregation and Stride Scheduling," Technical Report 2007-12, Institute for Systems Research, University of Maryland, College Park. Available online at http://hdl.handle.net/1903/7082

Herrmann, Jeffrey W. (2008) "Using Aggregation to Reduce Response Time Variability in Cyclic Fair Sequences," Technical Report, Institute for Systems Research, University of Maryland, College Park. Available online at http://hdl.handle.net/1903/8382

Inman, R.R., and Bulfin, R.L. (1991) Sequencing JIT Mixed-Model Assembly Lines. *Management Science*, 37(7):901-904.

Kubiak, W. (2004) Fair sequences. In *Handbook of Scheduling: Algorithms, Models and Performance Analysis*, Leung, J.Y-T., editor, Chapman & Hall/CRC, Boca Raton, Florida.

Miltenburg, J. (1989) Level Schedules for Mixed-Model Assembly Lines in Just-in-Time Production Systems. *Management Science*, 35(2):192-207.

Waldspurger, C.A., and Weihl, W.E. (1995) Stride scheduling: Deterministic proportional-share resource management. Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, Cambridge, Massachusetts.

Wei, W.D., and Liu, C.L. (1983) On a periodic maintenance problem. *Operations Research Letters*, 2(2):90-93.