THE INSTITUTE FOR SYSTEMS RESEARCH

ISR TECHNICAL REPORT 2008-20

Using Aggregation to Reduce Response Time Variability in Cyclic Fair Sequences

Jeffrey W. Herrmann

ISR develops, applies and teaches advanced methodologies of design and analysis to solve complex, hierarchical, heterogeneous and dynamic problems of engineering technology and systems for industry and government.

ISR is a permanent institute of the University of Maryland, within the A. James Clark School of Engineering. It is a graduated National Science Foundation Engineering Research Center.

www.isr.umd.edu

Using Aggregation to Reduce 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. Because minimizing response time variability is known to be NP-hard and the performance of existing heuristics is poor for certain classes of problems, we present an aggregation approach that 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. Computational experiments show that using aggregation can reduce response time variability dramatically.

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) discusses a parameterized stride scheduling heuristic that we will adapt in our work.

We were motivated to consider the fair sequencing problem while working with a healthcare facility that needed to schedule the collection of waste from waste collection rooms throughout the building. Given data about how often a trash handler needs to visit each room (to take away a cart of waste), the facilities manager wanted these visits to occur as regularly as possible so that excessive waste would not collect in any room. For instance, if a room needs four visits per eight-hour shift, then, ideally, it would be visited every two hours. Given a schedule for one shift, the same schedule can be repeated every shift. The time to visit each room and return with the trash cart varies slightly depending on the room location and other factors. However, the variation is small and can be ignored. The problem is difficult because different rooms require a different number of visits per shift.

This problem is clearly one of creating a fair sequence. In the product rate variation problem, the typical objective is to minimize the maximum absolute deviation (over each product and each position in the finite sequence) between the actual cumulative production and the ideal cumulative production. However, a more appropriate objective in a cyclic situation is to minimize the variability in the time between consecutive completions of the same task (consecutive visits to the same room in our waste collection problem). 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.

If the intervals between consecutive completions of the same task had to be equal to a predetermined quantity, we would have the periodic maintenance scheduling problem (Wei and Liu, 1983). However, in our case, we don't require this and instead seek to keep the intervals nearly the same. 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, which was not developed into a solution algorithm, is similar to the aggregation proposed here.

Waldspurger and Weihl (1995) present a hierarchical stride scheduling algorithm that combines products into groups. They suggest the use of a binary tree to minimize the maximum absolute deviation. The key distinction between their hierarchical stride scheduling algorithm and the aggregation approach presented here 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 our aggregation scheme groups products with equal demand, the disaggregation is much simpler. The limitation, however, is that the problem must have some equal demand products.

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) described the RTV problem and presented a heuristic that combined aggregation and parameterized stride scheduling. The aggregation

approach 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.

The current paper builds on these last two works. After formulating the problem, it precisely defines the aggregation approach of Herrmann (2007) and describes the results of extensive computational experiments using the aggregation approach in combination with the heuristics presented by Corominas *et al*. (2007). The goal of these experiments is to determine if the solutions generated using the aggregation approach have lower RTV than solutions generated without using the aggregation approach.

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 $\left\{p_{i1},\ldots,p_{id_i}\right\}$, the response time variability is a function of the intervals between each position, which are $\left\{\Delta_{i1},\ldots,\Delta_{id_i}\right\}$, 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:

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

This problem 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.

Parameterized Stride Scheduling

The parameterized stride scheduling algorithm builds a fair sequence and performs well at minimizing the maximum absolute deviation (Kubiak, 2004). The algorithm has a single parameter δ that can range from 0 to 1. This parameter affects the relative priority of low-demand products and their absolute position within the sequence. When δ is near 0, low-demand products will be positioned earlier in the sequence. When δ is near 1, low-demand products will be positioned later in the sequence.

The algorithm starts with an empty sequence. Given a partial sequence, with the first k positions filled, let x_{ik} be the number of times that product i occurs in those k positions. Then, position $k+1$ is allocated to customer i^* where

$$i^* = \arg\max_i \left\{ \frac{d_i}{x_{ik} + \delta} \right\}$$

Of course, there may be ties, so a tie-breaking procedure is needed. We always select the lowest-numbered product to break a tie. The computational effort of the algorithm is O(nD).

The parameterized stride scheduling algorithm can generate sequences with large variability. Consider the following example. There are n = 14 products with demands d = (20, 2, 2, …, 2). Therefore, D = 46. The parameterized stride scheduling algorithm (with δ = 0.5) generates the following sequence is (1, 1, 1, 1, 1, 2, 3, …, 14, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3, …,

14, 1, 1, 1, 1, 1). The RTV equals 304.2. (Other values of δ change the absolute position of the low-demand products but not the RTV.)

The high variability of this sequence occurs because the algorithm positions all of the low-demand products together, which creates a lumpy pattern for the high-demand product. A more fair sequence would blend the two types of products more evenly.

Aggregation

This insight led to the development of an 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).

To do this blending, we will combine products that have the same demand into groups. This has two benefits. First, it reduces the number of products that need to be sequenced. Secondly, because a group will have higher demand, it will be blended better than the individual products.

Let an instance I_k be a sequence 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, …, 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}, …, 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, 2, and 3 present a 10-product example that is aggregated three times. S_3 is a feasible sequence for I_3. Note that, at each step of disaggregating the sequence, copies of product j are replaced by the product(s) in P_{kj}.

Table 1. A 10-product instance and the aggregate instances formed from it.

		d_{kj}, j =									
k	n_k	1	2	3	4	5	6	7	8	9	10
0	10	1	1	1	1	1	2	2	2	3	5
1	6	2	2	2	3	5	5				
2	4	3	5	5	6						
3	3	3	6	10							

Table 2. The product sets for the instances in Table 1.

		P_{kj}, j =									
k	n_k	1	2	3	4	5	6	7	8	9	10
0	10	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}	{9}	{10}
1	6	{6}	{7}	{8}	{9}	{1,2,3,4,5}	{10}				
2	4	{4}	{5}	{6}	{1,2,3}						
3	3	{1}	{4}	{2,3}							

Table 3. The disaggregation of sequence S_3 for instance I_3.

S_3	3	2	3	2	3	1	3	2	3	2	3	1	3	2	3	2	3	1	3
S_2	2	4	3	4	2	1	3	4	2	4	3	1	2	4	3	4	2	1	3
S_1	5	1	6	2	5	4	6	3	5	1	6	4	5	2	6	3	5	4	6
S_0	1	6	10	7	2	9	10	8	3	6	10	9	4	7	10	8	5	9	10

The aggregation can greatly simplify sequencing. 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. For instance, consider the following example from Waldspurger and Weihl (1995). There are n = 101 products with demands d = (100, 1, 1, …, 1). Therefore, D = 200. To solve this problem, we first aggregate the one hundred low-demand products into one group with a total demand of 100. Then we aggregate the group and the high-demand product into a larger group with a total demand of 200. Now that we have only one group, we disaggregate the 200 positions by allocating them alternately to the high-demand product and the group of 100. Then, we disaggregate the group's 100 positions by giving one to each low-demand product. The resulting sequence has zero RTV.

In the 14-product example presented earlier in this paper, the aggregation procedure replaces the 13 low-demand products by one group with a demand of 26 to create an instance with only two products. Applying the parameterized stride scheduling algorithm (with $\delta = 0.5$) to generate an aggregate sequence and then disaggregating it reduces the RTV from 304.2 to 4.2. (The aggregate and disaggregated sequences are shown in Figure 1.)

Heuristics

The purpose of this paper is to investigate how much the aggregation approach reduces RTV when used with various heuristics. To that end we will consider five basic heuristics and the exchange heuristic of Corominas *et al.* (2007).

Webster. This heuristic uses the parametric stride scheduling algorithm with $\delta = 0.5$.

Jefferson. This heuristic uses the parametric stride scheduling algorithm with $\delta = 1$. The Webster and Jefferson heuristics run in O(*nD*) time.

Bottleneck. This heuristic is the due date algorithm that Steiner and Yeomans (1993) developed for solving the mixed-model production problem. This heuristic runs in O(*D* log *D*) time.

Random swap. This heuristic (presented in Corominas *et al.*, 2007) starts with the sequence that the bottleneck heuristic generates. Then, for each position $j = 1, \ldots, D$, it generates a random number $k \in \{1, \ldots, D\}$ and swaps the tasks in positions *j* and *k*. We use a discrete uniform distribution over $\{1, \ldots, D\}$ for choosing *k*. Because the swapping runs in O(*D*), this heuristic runs in O(*D* log *D*) time.

Insertion. This heuristic, introduced by Corominas *et al.* (2007), solves a sequence of two-product problems in order to generate a sequence recursively. It runs in O(*D* log *D*) time. To solve the two-product problem with $d_1 < d_2$, we use the following procedure: let $m = D \bmod d_1$ (or $m = d_1$ if d_1 divides D) and $R = \lceil D / d_1 \rceil$ Note that $mR + (d_1 - m)(R - 1) = D$. Assign *m* copies of product 1 to positions R, ..., mR and $d_1 - m$ copies of product 1 to the following positions:

$$mR + R - 1, mR + 2(R-1), \ldots, mR + (d_1 - m - 1)(R-1), D.$$

Assign d_2 copies of product 2 to all of the remaining positions.

Exchange. Corominas *et al.* (2007) developed the exchange heuristic as a technique for improving any given sequence. Unlike the other heuristics presented, it treats the sequence as a cycle, not a finite sequence. It repeatedly loops through the positions, exchanging a task with the

task immediately following it if that exchange reduces the RTV or reduces the maximum distance between two tasks for a product. It runs in O(nD^4) time.

Computational Experiments

The purpose of the computational experiments was to show how the aggregation technique performs in combination with a variety of heuristics on the metric of RTV.

We generated 4700 instances as follows. First, we set the total number of tasks *D* and the number of products *n*. (We followed Corominas *et al.*, 2007, in choosing values of *D* and *n*, but these are not the same instances.) To generate an instance, we generated $D-n$ random numbers from a discrete uniform distribution over $\{1,\ldots,n\}$. We then let d_i equal one plus the number of copies of *i* in the set of $D-n$ random numbers (this avoided the possibility that any $d_i = 0$). We generated 100 instances for each of the following combinations of *D* and *n*:

$D = 100$ and $n = 3$, 10, 20, 30, 40, 50, 60, 70, 80, and 90;

$D = 500$ and $n = 3$, 5, 10, 50, 100, 150, 200, 250, 300, 350, 400, and 450;

$D = 1000$ and $n = 10$, 50, 100, 200, 300, 400, 500, 600, 700, 800, and 900; and

$D = 1500$ and $n = 100$, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, and 1400.

For each instance that can be aggregated, we constructed 30 sequences as follows. First, we applied one of the basic heuristics to the instance (we call this the H sequence). Then, we applied the exchange heuristic to the H sequence to construct the HE sequence. (This essentially repeats what Corominas *et al.*, 2007, did.)

Next, we aggregated the instance if possible. For the aggregate instance, we applied the heuristic to construct an aggregated solution. We disaggregated this solution to construct the AHD sequence. Then, we applied the exchange heuristic to the AHD sequence to construct the

AHDE sequence. We also applied the exchange heuristic to the aggregated solution and then disaggregated to construct the AHED sequence. Finally, we applied the exchange heuristic to the AHED sequence to construct the AHEDE sequence. This makes six sequences using one basic heuristic and combinations of aggregation-disaggregation and the exchange heuristic. We repeated this for the remaining basic heuristics for a total of 30 solutions.

The clock time needed to generate the six sequences for one instance and one basic heuristic increased as D increased. These times ranged from 1 to 15 seconds for instances with $D = 1500$. The longest times occurred when $n = 400$, and the shortest times occurred when $n = 1400$.

Before discussing the results of the heuristics, we note that the number of times that an instance was aggregated depended greatly upon the ratio of n/D. When this ratio was very small (less than 0.05), many instances were not aggregated. For small values of n/D, all of the instances can be aggregated, and the average number of aggregations jumps to near 6 for $D = 100$ and is near 15 for $D = 1500$. The largest number of aggregations was 18, which occurred in five instances (one with $n = 100$ and $D = 1000$ and four with $n = 100$ and $D = 1500$). As n/D increases further, the average number of aggregations decreases steadily for all values of D. When n/D is near 0.9, the average number of aggregations is between 2 and 3.

Because the exchange heuristic is known to reduce RTV, significantly in some cases, our discussion will focus on the HE and AHDE sequences. As shown by Corominas *et al.* (2007), the exchange heuristic significantly reduces the RTV of the solutions generated by the heuristics (without aggregation). Thus, the HE sequences are more interesting that the H sequences.

The results of these experiments show that the AHD sequences may perform poorly (compared to the other sequences) for large values of n and in combination with the random

swap and insertion heuristics. The AHED sequences also perform poorly. Finally, the RTV of the AHEDE sequences was only slightly better than that for the AHDE sequences.

The notable exception to this last result occurred for the random swap and insertion heuristics when D = 1000 and D = 1500. In those cases, the AHEDE sequences were significantly better than the AHDE sequences for medium values of n. However, these sequences were still not as good as the sequences generated using the Webster, Jefferson, and bottleneck heuristics.

Tables 4 to 7 present the results for the HE and AHDE sequences for each of the basic heuristics. The results are averaged over the instances where aggregation was performed. The number of instances (out of 100) where any aggregation was performed is also shown.

Using aggregation with the Webster, Jefferson, and bottleneck heuristics generates the best sequences. Compared to the HE sequences, the AHDE sequences reduce RTV dramatically when n is moderate (not at the extremes). For instance, when n = 300 and D = 1000, the average RTV of the AHDE sequences is less than 4% of the average RTV of the HE sequences. When n = 400 and D = 1500, the average RTV of the AHDE sequences is between 1% and 2% of the average RTV of the HE sequences.

We can see why if we consider a set of products with the same demand. The Webster, Jefferson, and bottleneck heuristics tend to put copies of these products next to each other, which increases the RTV of products with higher demand. Aggregation spreads out the copies these products, leaving space for the products with higher demand.

When n is very small and D = 100, 500, or 1000, the AHDE sequences are about the same as the HE sequences because the heuristic sequences are already good. When n is near D,

the AHDE sequences are about the same as the HE sequences because the exchange heuristic is very good at constructing low-RTV sequences in those cases.

The random swap and insertion heuristics generate sequences that are not as good. Using aggregation improves the random swap sequences less than it improves the sequences generated with other heuristics. In sequences generated by the random swap heuristic, the advantages that occur with aggregation are lost as the products are shuffled in the aggregated sequence. In sequences generated by the insertion heuristic, products with low demand usually have higher RTV than the other products in that sequence because the low demand products are assigned last and get the "leftover" spots, which are not spread out evenly. However, the heuristic works well if there are many products with the same demand, because it will allocate them fairly. The aggregation deliberately creates an instance in which no products have the same demand, which degrades the performance of the insertion heuristic. The products that are not aggregated will be assigned last and will have large RTV.

Table 4. Comparison of the HE and AHDE sequences across five basic heuristics for $D = 100$.

		Webster		Jefferson		Bottleneck		Random Swap		Insertion	
n	aggregation	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
3	17	15.5	15.5	15.5	15.5	15.5	15.5	113.9	52.4	26.0	15.5
10	100	98.3	73.0	97.6	75.5	96.9	72.6	165.8	134.6	94.2	104.7
20	100	165.7	59.1	160.0	62.8	163.9	58.0	118.3	88.3	116.3	78.2
30	100	180.2	39.0	155.3	38.7	156.4	38.2	83.8	61.8	105.4	48.8
40	100	217.6	26.1	260.6	27.3	205.9	28.5	66.1	57.4	96.6	38.4
50	100	85.6	18.3	79.6	19.6	90.1	18.5	32.9	33.9	52.1	23.8
60	100	36.3	9.0	35.8	8.9	31.8	8.8	14.1	14.2	66.6	13.3
70	100	5.3	3.7	6.9	3.4	6.0	3.8	5.7	5.7	51.1	5.4
80	100	1.4	1.3	1.7	1.4	1.4	1.5	2.0	1.9	21.2	1.7
90	100	0.3	0.3	0.3	0.3	0.3	0.3	0.4	0.4	0.3	0.3

Table 5. Comparison of the HE and AHDE sequences across five basic heuristics for $D = 500$.

		Webster		Jefferson		Bottleneck		Random Swap		Insertion	
n	aggregation	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
3	11	44.5	44.5	44.5	44.5	44.5	44.5	644.0	288.9	65.6	44.5
5	35	130.3	128.2	130.6	132.2	130.2	129.1	1391.0	1062.8	220.2	567.9
10	89	348.8	344.7	349.1	345.7	344.6	313.2	3472.1	2931.7	712.9	974.4
50	100	2006.0	513.6	1971.3	537.9	1967.9	515.1	1699.9	1214.1	1565.4	987.4
100	100	2099.7	306.3	1755.6	306.3	1801.1	303.3	818.7	659.2	1290.6	531.2
150	100	6778.6	211.6	5728.4	210.4	5926.2	215.8	722.0	677.0	1915.4	453.9
200	100	1850.0	153.0	3478.8	151.9	2889.5	156.3	474.2	449.4	658.7	401.0
250	100	650.9	83.0	744.3	83.4	595.8	91.5	192.5	185.7	552.7	176.6
300	100	415.9	42.1	255.6	42.4	239.0	46.5	77.2	76.7	454.9	65.2
350	100	25.1	17.7	105.0	18.6	26.9	21.4	28.3	27.5	359.1	24.4
400	100	7.7	6.5	12.6	8.4	9.9	7.6	9.5	9.3	164.2	9.4
450	100	1.5	1.5	1.6	1.5	1.6	1.5	1.6	1.6	1.5	1.6

Table 6. Comparison of the HE and AHDE sequences across five basic heuristics for D = 1000.

		Webster		Jefferson		Bottleneck		Random Swap		Insertion	
n	aggregation	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
10	79	572.2	568.5	570.7	575.4	550.5	521.6	8078.4	6745.1	1190.6	1723.5
50	100	4126.4	1454.1	3951.8	1552.2	3962.9	1452.6	8641.4	6146.0	3358.2	3937.4
100	100	5657.1	984.0	5814.2	983.5	5949.2	969.7	3672.0	2627.7	4741.4	2170.3
200	100	4322.4	612.1	4092.0	614.0	4143.0	618.5	1867.1	1549.5	3941.2	1287.3
300	100	11213.1	433.7	11525.5	432.9	12582.6	448.1	1788.9	1755.1	3482.1	1485.8
400	100	4840.7	312.2	8903.6	309.7	5948.0	318.5	957.6	913.7	1826.3	936.0
500	100	1218.8	185.5	2097.2	187.8	1451.3	200.5	373.1	368.0	1706.1	366.4
600	100	1030.3	93.4	581.8	94.8	532.1	107.5	155.3	149.3	911.3	135.1
700	100	49.0	40.0	271.9	40.4	67.8	47.7	58.0	58.8	769.2	53.0
800	100	17.2	15.9	24.3	16.6	19.8	18.8	20.9	19.3	314.0	20.5
900	100	3.8	3.6	3.8	3.5	3.9	3.5	3.9	4.0	3.4	3.8

Table 7. Comparison of the HE and AHDE sequences across five basic heuristics for D = 1500.

		Webster		Jefferson		Bottleneck		Random Swap		Insertion	
n	aggregation	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
100	100	11389.9	2045.1	11224.6	2155.2	11233.4	2041.8	9793.5	6692.2	9239.1	5212.0
200	100	8232.5	1275.7	8338.3	1334.4	8181.8	1300.2	4426.9	3348.3	6926.3	2702.9
300	100	8672.9	982.0	7610.1	1009.2	7796.3	988.6	3152.5	2442.9	8325.6	2338.0
400	100	67875.4	779.4	52338.9	776.5	45754.6	799.9	3286.9	3072.0	8023.1	3147.0
500	100	28090.6	583.9	21590.5	579.0	21642.3	605.8	2738.8	2440.9	5148.5	2678.5
600	100	8165.7	478.1	18664.8	471.4	7449.3	496.6	1534.7	1450.3	3766.7	1513.0
700	100	2690.3	324.8	4462.3	328.1	4978.0	349.7	778.3	737.3	3338.9	748.2
800	100	1638.5	211.8	1890.3	213.3	1365.6	241.6	412.2	414.7	2461.2	402.7
900	100	2301.3	124.1	1252.9	125.2	1139.8	150.2	213.4	216.3	2018.7	211.3
1000	100	198.7	64.8	292.5	65.0	232.5	81.9	108.6	108.3	2013.9	108.8
1100	100	37.7	32.9	67.6	34.0	73.9	39.5	53.6	50.1	1531.7	45.5
1200	100	25.0	14.4	30.0	16.4	38.9	16.9	22.6	22.4	1257.0	17.5
1300	100	1.3	5.0	21.6	5.6	12.9	6.4	7.8	7.2	22.5	8.3
1400	100	0.1	0.0	3.5	2.6	4.5	0.0	1.0	1.1	1.3	0.2

Summary and Conclusions

This paper presents an aggregation approach for the problem of minimizing RTV. We combined this approach with various heuristics for the RTV problem in order to determine when aggregation is useful. The computational effort of the aggregation and disaggregation procedures is comparable to that of the heuristics themselves. Thus, it is reasonable to consider aggregation and disaggregation as part of the solution approach.

The results show that, when the number of products is very small or when the number of products is near the total demand (and each product has very small demand), combining aggregation with other heuristics does not reduce RTV compared to using the heuristics without aggregation.

In all other cases, combining aggregation with other heuristics does dramatically reduce RTV compared to using the heuristics without aggregation. In these cases, the solutions

generated by the heuristics have large values of RTV, and aggregation provides a way to find much better solutions. Aggregation is particularly effective in combination with the Webster, Jefferson, and bottleneck heuristics. Our results also confirm that the exchange heuristic plays a valuable role in reducing RTV.

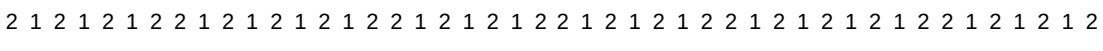
The aggregation procedure described in this paper is simple but effective. However, it is possible to create more sophisticated aggregations that more intelligently combine products in order to minimize the number of products in the highest level of aggregation, with the goal of aggregating all of the products into one group. If there is only one group, the disaggregation leads directly to a solution with zero RTV. Future work will consider algorithms for systematically creating more sophisticated aggregations.

Other future work should consider problems with multiple servers. In the case of multiple servers, it would be interesting to look at the problem under the constraint that, for each product, all of its demand must be satisfied by exactly one of the servers. That is, the products are first assigned to servers, and then we seek to find a low variability sequence for each server.

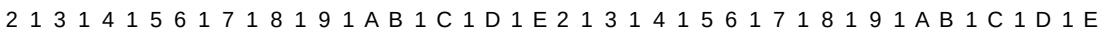
Acknowledgements

This work was motivated by a collaboration with the University of Maryland Medical Center. The author appreciates the help of Leonard Taylor, who introduced the problem, provided useful data, and recognized the value of the work presented here. The suggestions made by anonymous reviewers on an earlier version of this paper are also appreciated.

Figures



(a) Aggregate Sequence



(b) Disaggregated Sequence

Figure 1. (a) An aggregate sequence for the 14-product problem after aggregation into a 2-product problem. (b) The disaggregated sequence (the letters A, B, C, D, and E represent products 10, 11, 12, 13, and 14).

References

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

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.

Steiner, George, and Scott Yeomans (1993) "Level Schedules for Mixed-Model, Just-in-Time Processes," *Management Science*, 39(6):728-735

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.