THE INSTITUTE FOR SYSTEMS RESEARCH

ISR TECHNICAL REPORT 2007-12

Generating Cyclic Fair Sequences using Aggregation and Stride

Jeffrey W. Herrmann

Generating Cyclic Fair Sequences using Aggregation and Stride Scheduling

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 the problem is known to be NP-hard, we present a heuristic that combines aggregation and parameterized stride scheduling. This novel algorithm 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.

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).

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.

After trying some existing approaches, which created solutions that had excessive variability, we developed a new approach that combines aggregation and parameterized stride scheduling.

This paper will formulate the fair sequencing problem that we consider, present an example, describe and analyze our new algorithm, and present results comparing its performance to that of other approaches.

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 of a feasible sequence is a function 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 can define the total variability *V* of a sequence as follows:

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

This problem is NP-hard (Kubiak, 2004). Note that changes to the absolute positions do not change the variability. The objective function value is invariant under any translations or reflection.

Example

To clarify the problem, we will present a simple, hypothetical example (Example 2 from Miltenburg, 1989). There are n = 3 products with demands d = (6, 6, 1). Therefore, D = 13. One feasible sequence is (1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 3). The variability V = 5/3. (Note that, as long as the sequence iterates between products 1 and 2, the position of product 3 is irrelevant to the variability.)

Another feasible sequence is (1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3). The variability of this sequence is V = 81 2/3, which is 49 times greater than the variability of the previous fair sequence.

Equal Demand

An important special case occurs when all of the products have the same demand. That is, $d_i = d$ for all i. Note that $D = nd$. One can form a sequence with V = 0 by combining d copies of any permutation of {1, …, n}. The intervals for each product are all n.

The aggregation procedure that we present below exploits this special case. The disaggregation is simple because it is solving subproblems that are instances of this special case.

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*).

In the example above, the parameterized stride scheduling algorithm generates the first sequence when δ = 1. If δ = 0.5, product 3 would appear in position 7. As mentioned before, this translation does not affect the variability.

Another Example

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 sequence is shown in Figure 1. The variability *V* = 304.2. (Other values of δ change the absolute position of the low-demand products but not the variability.)

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

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 sequenced. Secondly, because a group will have higher demand, it will be blended better than the individual products.

The aggregation procedure works as follows. If there are *k* products and groups with the same demand *d*, then replace the *k* products by a group that has demand *kd*. The products and groups replaced are called the children of the new group. Note that the total demand remains the same. Repeat until there are no products or groups that have the same demand.

We then apply the parameterized stride scheduling algorithm to the remaining products and groups. After generating a sequence, we disaggregate each group as follows: Consider a group with *k* children that has been assigned *kd* positions in the sequence. Order the children from first to last (any permutation is acceptable). The first position goes to the first child in the group, the second to the second child, and so forth. Position $k+1$ goes to the first child, position $k+2$ position goes to the second child in the group, and this continues until all *kd* positions have been assigned. Each child gets *d* positions. If a child is a group, then the positions assigned to it must go to its children in a similar manner. Repeat until all groups in the original sequence have been disaggregated.

The aggregation procedure requires O(*n*) time. If the aggregation creates a problem with *G* groups (some of which may be individual products), then scheduling algorithm requires O(*GD*) time. Note that the disaggregation does not require stride scheduling but only counting. Let *H* be the maximum number of levels in the aggregation (i.e., the maximum number of

generations). Th disaggregation requires O(*HD*) effort because there are only *D* positions to allocate, and each position is assigned at most *H* times.

In the example above, the aggregation procedure replaces the 13 low-demand products by one group with a demand of 26. The parameterized stride scheduling algorithm (with $\delta = 0.5$) generates the aggregate sequence shown in the top part of Figure 2. Disaggregation yields the aggregate sequence shown in the bottom part of Figure 2. The variability *V* has been reduced from 304.2 to 4.2.

The aggregation may be more complex. For instance, consider a problem with $n = 14$ products with demands $d = (1, 1, 1, 1, 1, 2, 2, 2, 3, 4, 5, 6, 7, 10)$. The five products with demand equal to 1 are replaced with a group with demand equal to 5. Because $d_{11} = 5$, this group and product 11 can be replaced with a group with demand equal to 10. Because $d_{14} = 10$, this group and product 14 can be replaced with a group with demand equal to 20. In a similar manner, products 6, 7, and 8 (with demand equal to 2) are replaced with a group, and this group and the product 12 can be replaced by a second supergroup. The aggregation, which has $H = 4$ levels, is shown in Figure 3.

When a sequence for the five remaining groups is generated, every other position allocated to the first supergroup will go to product 14. Then, every other of the 10 remaining positions will go to product 11. This leaves 5 positions that can be assigned to products 1 through 5 in any way.

The aggregation can greatly simplify sequencing. If only one supergroup remains, 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 $d = (100, 1, 1, \ldots, 1)$. 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 supergroup 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 variability.

We note that 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.

Finally, after developing this aggregation procedure, we discovered that Wei and Liu (1983) had 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. However, their insight was not developed into a solution algorithm.

Better Aggregation

The aggregation procedure specified above creates an effective set of groups. However, it is not guaranteed to create an minimal variability sequence. Consider, for instance, a problem with $n = 17$ products with the following demands: $d = (1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 7, 14)$. $D = 42$. The standard aggregation groups the nine products with demand equal to 1 into a

group and the six products with demand equal to 2 into another group. After using the parameterized stride scheduling algorithm and disaggregating, the resulting sequence has a variability of $V = 20$.

However, it is possible to create a sequence with no variability as follows. First, replace seven of the products with demand equal to 1 by a group with demand equal to 7, and then replace the other two products with demand equal to 1 by a second group with demand equal to 2. Replace these first of these groups and product 16 (whose demand equals 7) by a group with demand equal to 14. Replace the second group and the six products with demand equal to 2 by a group with demand equal to 14. Finally, because $d_{17} = 14$, replace the two supergroups and the product 17 by a supergroup with demand equal to 42. We can then disaggregate the sequence to form a sequence with no variability (see Figure 6).

This example also illustrate that it may be possible, by combining the products in a different way, to create a better aggregation (one that leads to a sequence with lower variability).

Skewed Demand Distributions

We can show that the aggregation procedure significantly reduces variability for instances with skewed demand distributions.

In this case, there are n products. The demand of product 1 equals d, where $d \geq n-1$. The demand of the remaining $n-1$ products equals 1. Therefore, $D = d + n - 1$.

If $\delta = 1$, then the parameterized stride scheduling algorithm creates a sequence with product 1 in the first d positions. This occurs because the priority for any low-demand product equals 1, but the priority for product 1 when considering position $k+1$ equals $d/(k+1)$, which is greater than or equal to one until that product has received d positions. The remaining $n-1$ positions are occupied by the $n-1$ low-demand products.

If $\delta < 1$, then the parameterized stride scheduling algorithm will first assign k positions to product 1, where k is the smallest positive integer such that $d/(k+\delta) < 1/\delta$. Then, the algorithm will assign the low-demand products (which now have the larger priority) to the next $n-1$ positions. The remaining $d-k$ positions will be assigned to product 1.

In either case, product 1 has $d-1$ intervals that equal 1 and one interval that equals n. The intervals for the low-demand products are all D. Therefore, the variability $V = \frac{1}{d}(d-1)(n-1)^2 = \frac{n-1}{d}(d(n-1)-n+1)$.

Now, we aggregate the low-demand products into a single group. Let $m = n-1$ be the demand for this group. (Thus, $d \geq m$.) We will show that the parameterized stride scheduling algorithm will assign $n-1$ positions for the group such that the group does not have any two consecutive positions. Consider the first position. The priority of product 1 equals d/δ, which is greater than or equal to the priority of the group, which equals m/δ. Thus, product 1 receives the first position.

Now, assume that product 1 received position $k+p$ (giving product 1 a total of k positions and the group a total of p positions) and the group received position $k+p+1$ (giving it a total of $p+1$ positions). The following conditions must hold:

$$\frac{d}{k-1+\delta} \geq \frac{m}{p+\delta} \quad (1)$$

$$\frac{d}{k+\delta} < \frac{m}{p+\delta} \quad (2)$$

Condition (1) holds because product 1 received position $k+p$, and condition (2) holds because the group received position $k+p+1$.

Now consider position $k+p+2$. From (1) we have that $dp+d\delta \geq mk-m+m\delta$. Because *m* is not greater than *d*, we can derive the following:

$$\begin{aligned} m(k+\delta) &\leq dp+d\delta+m \\ &\leq dp+d\delta+d \\ &= d(p+1+\delta) \\ \frac{m}{p+1+\delta} &\leq \frac{d}{k+\delta} \end{aligned}$$

Therefore, product 1 will receive position $k+p+2$. We can conclude that the group never receives two consecutive positions.

Then, for product 1, there are $n-1$ intervals that equal 2 and $d-n+1$ intervals that equal 1. After disaggregation, the intervals for the products in the group are all *D*. In this case, the variability $V=\frac{n-1}{d}(d-n+1)$, which is clearly smaller than the variability of the sequence obtained without aggregation.

A Waste Collection Example

We now consider an instance from the waste collection setting that motivated this work. There are $n = 14$ products with demands $d = (2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5)$. Therefore, $D = 46$. Using the parameterized stride scheduling algorithm with any δ creates a sequence with variability $V = 344.267$, as shown in Figure 4.

The sequence created by the hierarchical stride scheduling algorithm depends upon how the products are placed into the binary tree. If the products with equal demand are grouped at the lowest levels of the tree, then the variability of the resulting sequence is $V = 40.267$. However, other placements lead to sequences with variability over ten times greater.

The aggregation procedure creates four groups with the following demands: (8, 12, 16, 10). After running the parameterized stride scheduling algorithm and disaggregating, the variability is reduced to $V = 8.267$. The sequences are shown in Figure 5.

Summary and Conclusions

Unlike most other work on fair sequences, we have considered the response time variability of cyclic sequences. Because absolute position is not important, our novel aggregation scheme, combined with parameterized stride scheduling, creates sequences with low variability. We have proven its superiority in the case of skewed demand distributions and demonstrated its performance on a variety of examples, including one from the real-world problem that motivated this work. The aggregation scheme has been used to create sequences for the waste management problem.

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.

Figures

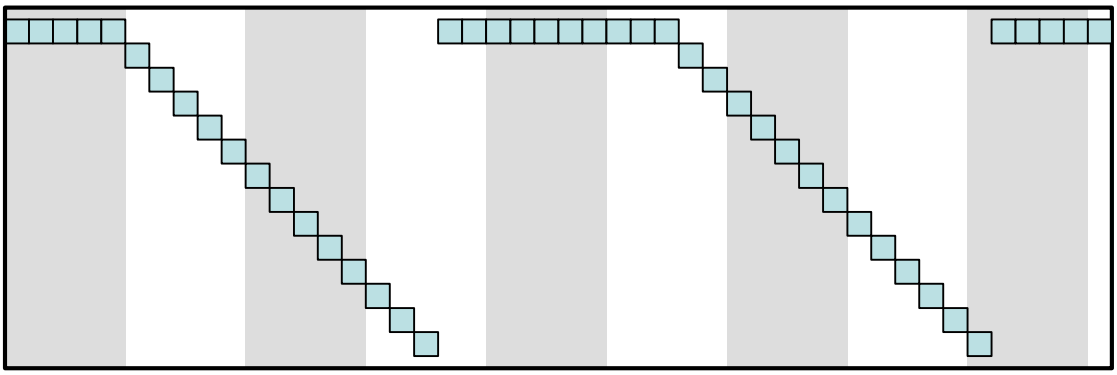

Figure 1. The sequence created by stride scheduling for a problem with one product whose demand equals 20 and thirteen products whose demand equals 2. Each row represents a different product. Each box represents a position allocated to that product.

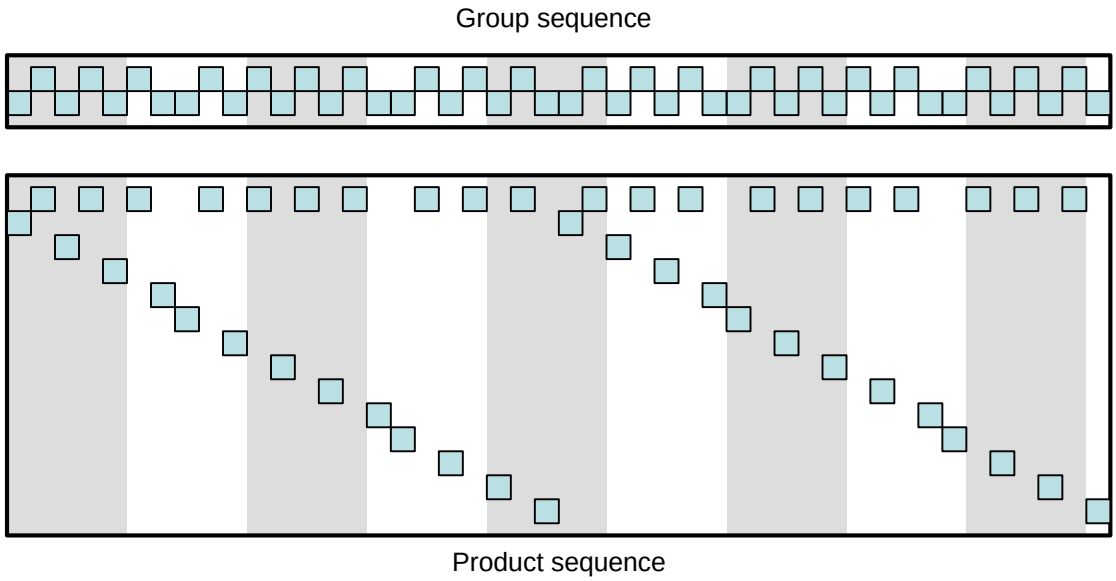

Figure 2. The sequence created by aggregation and stride scheduling for a problem with one product whose demand equals 20 and thirteen products whose demand equals 2. In the top sequence, the first row is the high-demand product and the second row is the group of low-demand products. In the bottom sequence, each row represents a different product. Each box represents a position allocated to that product.

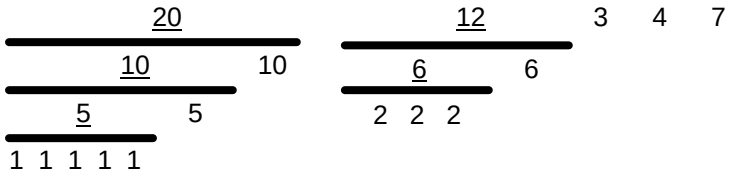

Figure 3. Aggregation of products into groups and supergroups.
The numbers that are not underlined represent the demand of different products.
The underlined numbers represent the demand of groups or supergroups.

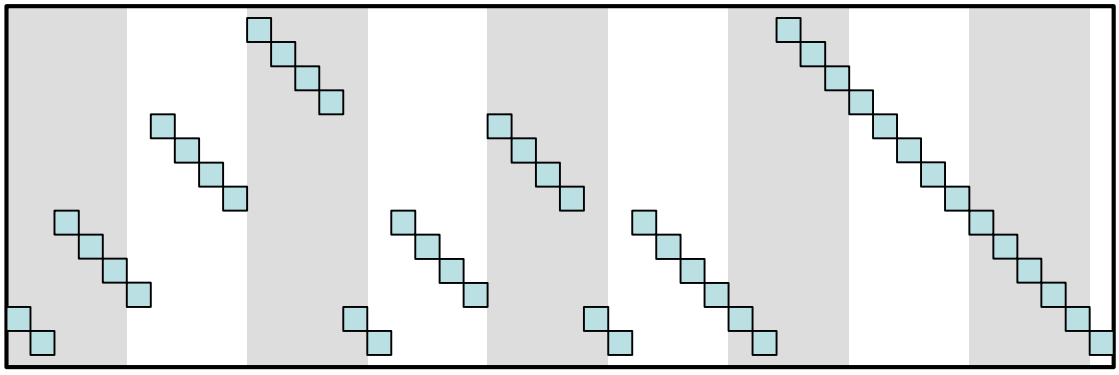

Figure 4. The sequence created by stride scheduling for a problem with fourteen products. Each row represents a different product. Each box represents a position allocated to that product.

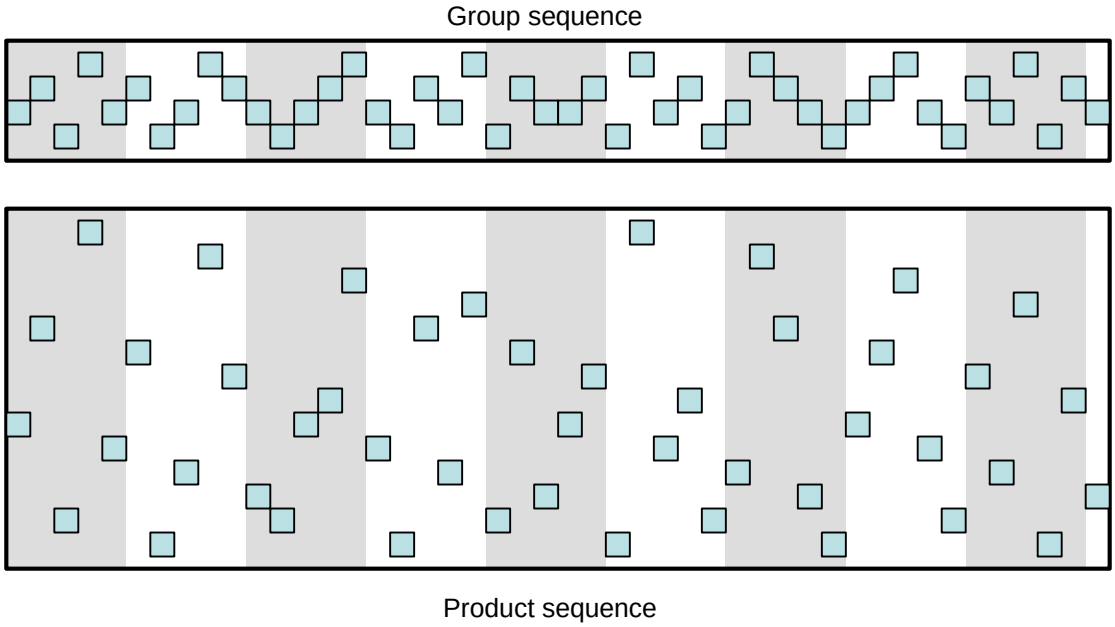

Figure 5. The sequence created by aggregation and stride scheduling for a problem with fourteen products.
In the top sequence, each row represents one of the groups.
In the bottom sequence, each row represents a different product.
Each box represents a position allocated to the group or product.

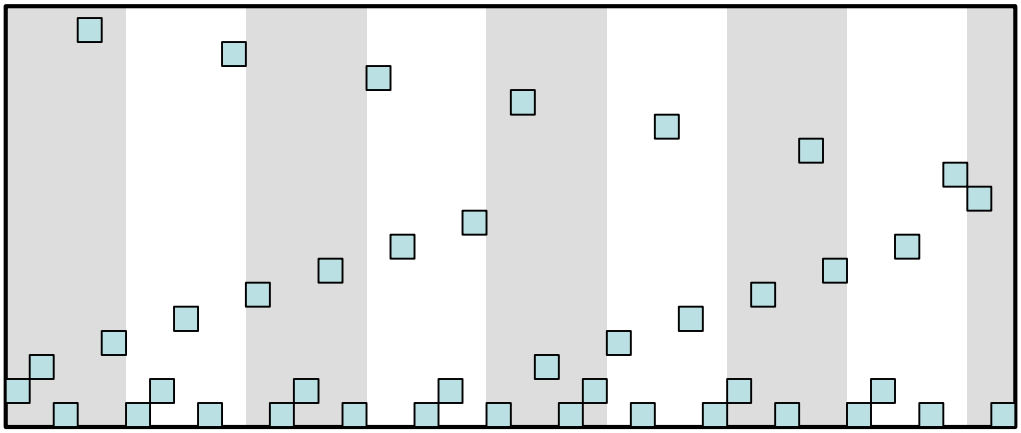

Figure 6. The disaggregated sequence with no variability for a problem with seventeen products.
Each row represents a different product.
Each box represents a position allocated to the group or product.

References

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.

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

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.