

THE INSTITUTE FOR SYSTEMS RESEARCH

ISR TECHNICAL REPORT 2009-12

---

# Aggregating Alphabets to Construct Balanced Words

Jeffrey W. Herrmann

The  
Institute for  
**Systems**  
Research



A. JAMES CLARK  
SCHOOL OF ENGINEERING

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](http://www.isr.umd.edu)

# Aggregating Alphabets to Construct Balanced Words

Jeffrey W. Herrmann  
A. James Clark School of Engineering  
University of Maryland  
College Park, MD 20742  
jwh2@umd.edu

## Abstract

Balanced words are useful for scheduling mixed-model, just-in-time assembly lines, planning preventive maintenance, managing inventory, and controlling asynchronous transfer mode (ATM) networks. This paper considers the challenging problem of finding a balanced word (a periodic sequence) for a finite set of letters, when the desired densities of the letters in the alphabet are given. We present an aggregation approach that combines letters with the same density, constructs a word for the aggregated alphabet, and then disaggregates this word into a feasible word for the original alphabet. We consider two different measures for evaluating solutions and use the aggregation approach with different heuristics. Computational experiments show that using aggregation not only finds more balanced words but also reduces computational effort.

**Keywords:** balanced words, fair sequences, aggregation, cyclic scheduling

---

## Introduction

Balanced words can be good solutions for problems of finding a fair sequence. The idea of fair sequences occurs in many different areas, including scheduling mixed-model, just-in-time assembly lines, planning preventive maintenance, inventory management, and controlling asynchronous transfer mode (ATM) networks. Kubiak (2004) provides a good overview of the need for fair sequences in different domains and discusses results for multiple related problems, including the product rate variation problem, generalized pinwheel scheduling, the hard real-time periodic scheduling problem, the periodic maintenance scheduling problem, stride scheduling, minimizing response time variability (RTV), and peer-to-peer fair scheduling.

The routing of jobs in stochastic systems also requires fair sequences. Hajek (1985) considered a queueing system where the interarrival times are independent, identically distributed random variables with finite mean and the server has exponentially distributed

processing times. A given fraction of the arriving jobs must be sent to the server, while the rest are sent elsewhere. Hajek showed that a regular admission sequence minimizes the server's expected queue size and the expected waiting time of the admitted jobs. Altman *et al.* (2000) show that, for very general stochastic systems, the optimal routing of jobs to servers is a balanced sequence. Sano *et al.* (2004) introduced a generalization of balanced words and showed that using these policies to route job minimizes the maximum waiting time.

The remainder of the paper proceeds as follows: we will review related work, formulate the balanced word problem, and discuss heuristics for generating sequence. Then, we present the main contribution of this paper: the aggregation approach. We then discuss the results of computational experiments designed to evaluate the effectiveness of using aggregation in combination with the different heuristics before concluding the paper.

## Balanced Words

We are given a finite alphabet and a set of densities for the letters in the alphabet. In scheduling problems, these letters correspond to different types of products that need to be produced at different rates. We wish to construct an infinite sequence (word) over the letters in this alphabet in which each letter occurs at a rate that equals the given density. If the densities are all rational, it is sufficient to construct a cyclic sequence in which each letter occurs the correct number of times in the finite cycle.

Given a density  $p$  in  $(0, 1)$  and a phase  $\theta$  in  $[0, 1)$ , the *regular* sequence  $\sigma(p, \theta)$  has the value  $\lfloor (j+1)p + \theta \rfloor - \lfloor jp + \theta \rfloor$  in position  $j$ . For example,  $\sigma(\frac{2}{7}, 0) = (0001001)^\infty$ .

If we consider just one letter  $a$  in the alphabet, we can derive from any sequence  $S$  an indicator sequence  $I(s, a)$  that has a 1 in position  $j$  if  $S$  has the letter  $a$  in position  $j$ . Otherwise,  $I(s, a)$  has a 0 in position  $j$ .

The regular word problem is to find a sequence  $S$  so that all of the indicator sequences are regular sequences. The complexity of the problem is open (Kubiak, 2009). If the alphabet has at most two distinct densities, then a regular word exists (Altman *et al.*, 2000).

Balanced words are a more general concept than regular words. Two different measures for the degree of balance have been proposed.

Kubiak (2009) gives the following definition: given a finite alphabet  $\{1, \dots, n\}$ , a *c-balanced word* is an infinite sequence  $U$  such that each position in  $U$  is in the alphabet and, if  $x$  and  $y$  are two factors (subsequences) of the same size, then  $||x|_i - |y|_i| \leq c$ , where  $|x|_i$  is the number of times that  $i$  occurs in the factor  $x$ . We will define the *count-balance* of a word  $U$  as the minimal such value of  $c$ . For example, the count-balance of the word  $(1231211321)^\infty$  equals 2 because  $||11|_1 - |23|_1| = 2$  and  $||x|_i - |y|_i| \leq 2$  for all factors  $x$  and  $y$  and all  $i$ .

The count-balance of a regular word equals 1. For any given set of densities, the minimal possible count-balance is less than or equal to 3 (Kubiak, 2009).

Sano *et al.* (2004) gives the following definition for a different measure: given a finite alphabet  $\{1, \dots, n\}$ , a word  $U$  over this alphabet, and a nonnegative integer  $m$ , a letter  $a$  is *m-balanced* in  $U$  if, whenever there exists an  $a$ -chain  $aWa$  in  $U$ , any factor  $W'$  in  $U$  such that  $|W'| = |W| + m + 1$  satisfies  $|W'|_a \geq |W|_a + 1$ . The sequence  $U$  is *m-balanced* if each letter in the alphabet is  $m$ -balanced. We will define the *gap-balance* of a word  $U$  as the minimal such value of  $m$ . For example, in the infinite sequence  $(313132)^\infty$ , the gap-balance of the letters 2 and 3 equals 0, and the gap-balance of the letter 1 equals 2. Note that the factor 3 in the 1-chain 131 is 2 letters shorter than 323, the longest factor with no instance of the letter 1. Therefore, the gap-balance of this word equals 2.

The gap-balance of a word equals 0 if and only if it is a constant gap word, and the gap-balance of a word equals 1 if and only if it is a regular word (Sano *et al.*, 2004). Moreover, the count-balance of any word that has a positive gap-balance is not larger than the gap-balance of that word (Sano *et al.*, 2004).

In other words, the count-balance is the maximum difference in the number of copies of a letter for two factors of the same length. The gap-balance is the maximum difference in length of two factors that contain the same number of copies of a letter. For both measures, a smaller value implies that the occurrences of each letter are distributed more evenly, which is the goal in many applications. Appendix A presents the pseudocode for pseudo-polynomial algorithms that measure the count-balance and gap-balance of a word.

Sano *et al.* (2004) present a search algorithm that randomly generates words and keeps the most balanced one. Otherwise, we know of no algorithms designed specifically to generate balanced words. We will study a number of algorithms used for similar problems.

## Problem Formulation

Let  $A$  be a finite set of letters  $\{1, \dots, n\}$ . Let  $U$  be an infinite word over this set such that  $U_t \in A$  for all  $t \in \mathbb{N}$ . The density of letter  $a \in A$  is  $p_a$  if the following limit exists:

$$p_a = \lim_{n \rightarrow \infty} \frac{U_{[0,n]}(a)}{n}$$

If all of these limits exist, then the sum of the densities must equal 1:

$$\sum_{a \in A} p_a = 1$$

We assume that all of the densities are rational. Therefore, we consider infinite words  $U$  that are the infinite repetition of a finite word  $S$ ; that is,  $U = S^\infty$ . Given an alphabet  $A$  and a set of rational densities, there exists a positive integer  $T$  and positive integers  $x_1, \dots, x_n$  such that

$p_i = x_i / T$  for  $i = 1, \dots, n$  and  $\gcd(x_1, \dots, x_n) = 1$ . Thus,  $x_1 + \dots + x_n = T$ . Hereafter, we will describe an instance by the values of  $(x_1, \dots, x_n)$ , with  $x_1 \geq x_2 \geq \dots \geq x_n$ .

We will study two versions of the balanced word problem (BWP). They differ only in the measure used. BWP-count uses the count-balance measure, and BWP-gap uses the gap-balance measure.

Thus, we can describe BWP-count (and BWP-gap) as follows: Given an instance  $(x_1, \dots, x_n)$ , find a finite word  $S$  of length  $T$  that minimizes the count-balance (gap-balance) of the infinite word  $U$  that is the infinite repetition of  $S$  subject to the constraints that exactly one letter is assigned to each position of  $S$  and each and every letter  $i$  occurs exactly  $x_i$  times in  $S$ .

The complexity of BWP-count appears to be open. Given an instance, finding a word with a count-balance that equals 1 requires finding a regular word. The complexity of this problem is open (Kubiak, 2009). Likewise, the complexity of BWP-gap appears to be open. Given an instance, finding a word with an gap-balance that equals 0 requires finding a constant gap word for  $(x_1, \dots, x_n)$ . The complexity of the constant gap problem is open (Kubiak, 2004). Nevertheless, these problems are related to the Periodic Maintenance Scheduling Problem, which is NP-complete in the strong sense (Kubiak, 2009), and the RTV problem, which is NP-hard (Corominas *et al.*, 2007).

Consider, as an example, the following three-letter instance:  $(x_1, x_2, x_3) = (4, 3, 2)$ . In this system,  $(p_1, p_2, p_3) = (4/9, 1/3, 2/9)$ , and  $T = 9$ . Consider the word  $U = (112231123)^\infty$ . The count-balance of  $U$  equals 2, and the gap-balance of  $U$  equals 3 (because the gap-balance of the letter 1 equals 3). Now, consider the word  $V = (121312123)^\infty$ . The count-balance of  $V$  also equals 2 (because  $|212|_2 - |131|_2 = 2$ ), but the gap-balance of  $V$  equals 2 (because the gap-balance of the letter 2 equals 2).

The count-balance measurement algorithm (Appendix A) uses the following quantities:

$$M_j^A = \max_{k=0, \dots, x_i-1} \left\{ \sum_{q=1}^j \Delta_{i,k+q} \right\} + j - 1$$

$$M_j^B = \min_{k=0, \dots, x_i-1} \left\{ \sum_{q=1}^j \Delta_{i,k+q} \right\} + j + 1$$

If  $M_p^A \geq M_{p+c-1}^B$ , then there is an integer  $m$ , with  $M_p^A \geq m \geq M_{p+c-1}^B$ , such that there is a factor  $x$  of length  $m$  with at most  $p - 1$  copies of  $i$  (because  $M_p^A$  is the length of the longest factor with only  $p - 1$  copies of  $i$ ) and a factor  $y$  of length  $m$  with at least  $p + c$  copies of  $i$  (because  $M_{p+c-1}^B$  is the length of the shortest factor with  $p + c$  copies of  $i$ ). Thus,

$$|y|_i - |x|_i \geq p + c - (p - 1) = c + 1, \text{ so the count-balance must be at least } c + 1.$$

To evaluate the gap-balance, we need the smallest value  $\nu$  such that  $|W'| = |W| + \nu + 1$  satisfies  $|W'|_i \geq |W|_i + 1$ . This is equivalent to  $|W'|_i \leq |W|_i$  implies  $|W'| \leq |W| + \nu$ , which is the same as  $\nu \geq |W'| - |W|$ . So, for the letter  $i$ , we need to look for the shortest factor between two copies of  $i$  and the longest factor so that both have the same number of copies of  $i$ . We define a “gap” as a factor that occurs between two copies of the letter  $i$ . There are exactly  $x_i$  gaps for letter  $i$ . The non-negative length of a gap  $\Delta_{ik}$  is the number of positions between the copies of  $i$ . The shortest factor between two copies of  $i$  that has  $j$  copies of  $i$  and the longest factor that has  $j$  copies of  $i$  will be some  $j+1$  consecutive gaps plus the  $j$  copies of  $i$ . In the gap-balance algorithm,

$$\delta_j = \max_{k=1, \dots, x_i} \left\{ \sum_{q=0}^j \Delta_{i,k+q} \right\} - \min_{k=1, \dots, x_i} \left\{ \sum_{q=0}^j \Delta_{i,k+q} \right\}$$

is therefore the difference between the lengths of the shortest and longest factors that have  $j$  copies of  $i$ . (In the above equation, the second subscript  $k+q$  must be reduced by  $x_i$  if it exceeds  $x_i$ .)

## Heuristics

To construct solutions for the BWP-count (and BWP-gap), we consider a number of heuristics that have been proposed for related problems. The following discussion briefly describes the heuristics. Detailed algorithms and examples are given in Appendix A. We will conduct extensive computational testing to evaluate the performance and computational effort of these heuristics. We will also use these heuristics with the aggregation approach presented later.

**GR.** The greedy regular (GR) algorithm, presented by van der Laan (2005), tries to make the sequence of each letter resemble a regular sequence as much as possible. The highest-density letter will have a regular sequence. The sequences for the other letter are regular with respect to the sequences of the higher-density letters. The GR algorithm generates a periodic policy. The computational effort of the GR algorithm is  $O(nT)$ .

**Stride.** Waldspurger and Weihl (1995) considered the problem of scheduling multithreaded computer systems. In such a system, there are multiple clients, and each client has a number of tickets. A client with twice as many tickets as another client should be allocated twice as many quanta (time slices) in any given time interval. Waldspurger and Weihl introduced the stride scheduling algorithm to solve this problem. They also presented a hierarchical stride scheduling approach that uses a balanced binary tree to group clients, uses stride scheduling to allocate quanta to the groups, and then, within each group, uses stride scheduling to allocate quanta to the clients. Although they note that grouping clients with the same number of tickets would be desirable, their approach does not exploit this. Indeed, the approach does not specify how to create the binary tree. Kubiak (2004) showed that the stride scheduling algorithm is the same as Jefferson's method of apportionment and is an instance of



the more general parametric method of apportionment (Balinski and Young, 1982). Thus, the stride scheduling algorithm can be parameterized.

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

We will use the stride scheduling algorithm with  $\delta = 0.5$  and  $\delta = 1$  to generate periodic policies. The computational effort of the parameterized stride scheduling algorithm is  $O(nT)$ .

**Bottleneck.** The bottleneck minimization problem (Steiner and Yeomans, 1993) is related to fair sequencing of a mixed-model manufacturing facility. To solve the BWP, we use an algorithm that Steiner and Yeomans (1993) developed. The bottleneck algorithm calculates an earliest and latest start time for each unit of demand (each letter in the alphabet) and then allocates to each position in the word the eligible product (letter) with the smallest latest start time. This heuristic runs in  $O(nT)$  time. Appendix A describes the algorithm in detail.

**Search.** Sano *et al.* (2004) proposed a search algorithm for finding balanced words. The search randomly selects phases  $\{\phi_1, \dots, \phi_n\}$  and then uses these phases to construct a word. If the resulting word has a lower gap-balance than the best one found so far, it is saved.

To generate a word from a set of phases, the algorithm starts at the first position and selects the letter  $i$  with the minimal phase. The algorithm increases  $\phi_i$  by  $T/x_i$  and moves to the next position.

## Aggregation

To improve the performance of these heuristics, we employed an aggregation approach that first aggregates an alphabet, constructs a solution for the aggregate alphabet, and then disaggregates that solution. Aggregation is a well-known and valuable technique for solving optimization problems, especially large-scale mathematical programming problems. Model aggregation replaces a large optimization problem with a smaller, auxiliary problem that is easier to solve (Rogers *et al.*, 1991). The solution to the auxiliary model is then disaggregated to form a solution to the original problem. Model aggregation has been applied to a variety of production and distribution problems, including machine scheduling problems. For example, Rock and Schmidt (1983) and Nowicki and Smutnicki (1989) aggregated the machines in a flow shop scheduling problem to form a two-machine problem.

We previously developed this aggregation scheme, which is similar to the substitution concept discussed by Wei and Liu (1983), to generate solutions for the RTV problem and showed that using aggregation with parameterized stride scheduling and an improvement heuristic generates solutions with lower RTV and reduces the computational effort (Herrmann, 2007, 2009a, b). This paper builds on the previous work but considers a more general problem.

The aggregation approach used here repeatedly aggregates an alphabet until it cannot be aggregated any more. Each aggregation combines letters that have the same density into a group. These letters are removed, and the group becomes a new letter in the new aggregated alphabet. The letters with the smallest densities are combined first. Aggregation reduces the number of letters that need to be considered.

The notation used in the algorithm that follows enables us to keep track of the aggregations in order to describe the disaggregation of a sequence precisely. Let  $I_0$  be the

original instance (alphabet) and  $I_k$  be the  $k$ -th instance generated from  $I_0$ . Let  $n_k$  be the number of letters in instance  $I_k$ . Let  $B_j$  be the set of letters that form the new letter  $j$ , and let  $B_j(i)$  be the  $i$ -th letter in that set. As the aggregation algorithm is presented, we describe its operation on the following five-letter example:  $I_0 = (3, 2, 2, 1, 1)$ ,  $n = 5$ , and  $T = 9$ .

**Aggregation.** Given: an instance  $I_0$  with values  $(x_1, x_2, \dots, x_n)$ .

1. Initialization. Let  $k = 0$  and  $n_0 = n$ .
2. Stopping rule. If all of the letters in  $I_k$  have different values, return  $I_k$  and  $H = k$  because no further aggregation is possible. Otherwise, let  $G$  be the set of letters with the same value such that any smaller value is unique.

Example. With  $k = 0$ ,  $G = \{4, 5\}$  because  $x_4 = x_5$ .

3. Aggregation. Let  $m = |G|$  and let  $i$  be one of the letters in  $G$ . Create a new letter  $n + k + 1$  with value  $x_{n+k+1} = mx_i$ . Create the new instance  $I_{k+1}$  by removing from  $I_k$  all  $m$  letters in  $G$  and adding letter  $n + k + 1$ . Set  $B_{n+k+1} = G$ .  $n_k = n_{k-1} - m + 1$ . Increase  $k$  by 1 and go to Step 2.

Example. With  $k = 0$  and  $G = \{4, 5\}$ , the new letter 6 has value  $x_6 = 2 \times 1 = 2$ .  $B_6 = \{4, 5\}$ .

The letters in  $I_1$  are  $\{1, 2, 3, 6\}$ . When  $k = 1$ ,  $G = \{2, 3, 6\}$ . The new letter 7 has value  $x_7 = 3 \times 2 = 6$ , and  $B_7 = \{2, 3, 6\}$ . The letters in  $I_2$  are  $\{1, 7\}$ , which have different values.

Table 1 describes the instances created for this example.

Table 1. The values for the five original letters in the example instance  $I_0$  and the two new letters in the aggregate instances  $I_1$  and  $I_2$ .

	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$
$I_0$	3	2	2	1	1		
$I_1$	3	2	2			2	
$I_2$	3						6

At any point during the aggregation, the total value in a new instance will equal the total value of the original instance because the value of the new letter equals the sum of the values of the letters that were combined to form it.

The aggregation procedure generates a sequence of instances  $I_0, \dots, I_H$ . ( $H$  is the index of the last aggregation created.) The aggregation can be done at most  $n-1$  times because the number of letters decreases by at least one each time an aggregation occurs. Thus  $H \leq n-1$ . Aggregation runs in  $O(n^2)$  time because each aggregation requires  $O(n)$  time and there are at most  $n-1$  aggregations.

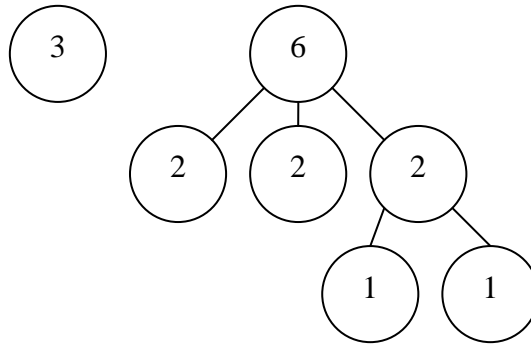


Figure 1. The forest corresponding to the aggregation of the example. The five leaf nodes correspond to the original letters in the example. The two parent nodes correspond to the new letters created during the aggregation. The two root nodes correspond to the letters remaining in the most aggregated instance.

We can represent the aggregation as a forest of weighted trees. There is one tree for each letter in the aggregated instance  $I_H$ . The weight of the root of each tree is the total value of the

letters in  $I_0$  that were aggregated to form the corresponding letter in  $I_H$ . The weight of any node besides the root node is the weight of its parent divided by the number of children of the parent. The leaves of a tree correspond to the letters in  $I_0$  that were aggregated to form the corresponding letter in  $I_H$ , and each one's weight equals the value of that letter. The forest has one parent node for each new letter formed during the aggregation, and the total number of nodes in the forest equals  $n + H < 2n$ . Figure 1 shows the forest corresponding to the aggregation of the (3, 2, 2, 1, 1) instance.

## Disaggregation

When aggregation is complete, we must find a feasible solution for the aggregated instance  $I_H$  and then disaggregate that solution. We will use the heuristics presented earlier to construct a feasible solution. This section presents the disaggregation procedure.

Let  $S_H$  be a feasible solution for the instance  $I_H$ . In particular,  $S_H$  is a sequence of length  $T$ . Each position in  $S_H$  is a letter in the instance  $I_H$ . Disaggregating  $S_H$  requires  $H$  steps that correspond to the aggregations that generated the instances  $I_1$  to  $I_H$ , but they will, naturally, be considered in reverse order. We disaggregate  $S_H$  to generate  $S_{H-1}$  and then continue to disaggregate each solution in turn to generate  $S_{H-2}, \dots, S_0$ .  $S_0$  is a feasible solution for  $I_0$ , the original instance.

The basic idea of disaggregating a solution  $S_k$  is to replace each new letter with the letters used to form it. Letter  $n+k$  was formed to create instance  $I_k$  from the letters in  $B_{n+k}$ , which were in  $I_{k-1}$ . It has  $x_{n+k}$  positions in  $S_k$ . According to the aggregation scheme,  $x_{n+k} = mx_i$ , where  $m = |B_{n+k}|$  and  $i$  is one of the letters in  $B_{n+k}$ . The first position in  $S_k$  assigned

to letter  $n+k$  will, in the new solution  $S_{k-1}$ , go to the first letter in  $B_{n+k}$ , the second position assigned to letter  $n+k$  will go to the second letter in  $B_{n+k}$ , and so forth. This will continue until all  $x_{n+k}$  positions have been assigned. Each letter in  $B_{n+k}$  will get  $x_{n+k}/m$  positions in  $S_{k-1}$ .

In the following algorithm,  $j = S_k(a)$  means that letter  $j$  is in position  $a$  in solution  $S_k$ , and  $B_{n+k}(i)$  is the  $i$ -th letter in  $B_{n+k}$ .

**Disaggregation.** Given: The instances  $I_0, \dots, I_H$  and the solution  $S_H$ , a feasible solution for the instance  $I_H$ .

1. Initialization. Let  $k = H$ .
2. Set  $m = |B_{n+k}|$  and  $i = 1$ .
3. For  $a = 0, \dots, T-1$ , perform the following step:
  - a. If  $S_k(a) < n+k$ , assign  $S_{k-1}(a) = S_k(a)$ . Otherwise, assign  $S_{k-1}(a) = B_{n+k}(i)$ , increase  $i$  by 1, and, if  $i > m$ , set  $i = 1$ .
4. Decrease  $k$  by 1. If  $k > 0$ , go to Step 2. Otherwise, stop and return  $S_0$ .

Example. Consider the aggregation of the instance  $(3, 2, 2, 1, 1)$  presented earlier and the solution  $S_2 = 7-7-1-7-7-1-7-7-1$ , which is a feasible solution for the aggregated instance  $I_2$ .

When  $k = 2$ ,  $n+k = 7$ , and  $B_7 = \{2, 3, 6\}$ . The positions in  $S_2$  that are assigned to letter 7 will be reassigned to letters 2, 3, and 6. The resulting solution  $S_1 = 2-3-1-6-2-1-3-6-1$ .

When  $k = 1$ ,  $n+k = 6$ , and  $B_6 = \{4, 5\}$ . The positions in  $S_1$  that are assigned to letter 6 will be reassigned to letters 4 and 5. The resulting solution  $S_0 = 2-3-1-4-2-1-3-5-1$ . Table 2 lists these three solutions.

Table 2. The disaggregation of solution  $S_2$  for instance  $I_2$  in the example. The first row is  $S_2$ , a feasible solution for instance  $I_2$ . The second row is  $S_1$ , a feasible solution for instance  $I_1$ . The third row is  $S_0$ , a feasible solution for instance  $I_0$ .

$a$	0	1	2	3	4	5	6	7	8
$S_2(a)$	7	7	1	7	7	1	7	7	1
$S_1(a)$	2	3	1	6	2	1	3	6	1
$S_0(a)$	2	3	1	4	2	1	3	5	1

As noted earlier, there are at most  $n-1$  aggregations. Because each solution disaggregation requires  $O(T)$  effort, disaggregation runs in  $O(nT)$  time in total.

### Disaggregating Balanced Words

How does disaggregating a word affect its count-balance or its gap-balance? That is, is the count-balance (or gap-balance) of the disaggregated word equal to the count-balance of the aggregated word?

**Theorem 1.** Disaggregating a word does not increase its count-balance.

**Proof.** Consider the aggregated word  $U+$  and the disaggregated word  $U-$ . The disaggregation replaces  $mx$  copies of the letter  $j$  in  $U+$  by  $x$  copies of the  $m$  letters in  $B_j$  in a round-robin manner.

Let  $c$  be the count-balance of  $U+$  and let  $y$  and  $z$  be factors of  $U$  such that  $c = |z|_j - |y|_j$ .

Let  $a$  be one of the letters that replaces letter  $j$ . Because the copies of  $j$  are replaced in a round-

robin manner,  $|y|_a = \lfloor |y|_j / m \rfloor$  or  $|y|_a = \lceil |y|_j / m \rceil$ . Likewise,  $|z|_a = \lfloor |z|_j / m \rfloor$  or  $|z|_a = \lceil |z|_j / m \rceil$ .

Therefore,  $|z|_a - |y|_a \leq \lceil |z|_j / m \rceil - \lfloor |y|_j / m \rfloor$

If  $c = 1$ , we consider three cases. First, if  $|z|_j \equiv 0 \pmod{m}$ , then  $|z|_a = |z|_j / m$  and  $\lfloor |y|_j / m \rfloor = |z|_j / m - 1$ . Thus,  $|z|_a - |y|_a \leq 1$ . Second, if  $|y|_j \equiv 0 \pmod{m}$ , then  $|y|_a = |y|_j / m$  and  $\lceil |z|_j / m \rceil = |y|_j / m + 1$ . Thus,  $|z|_a - |y|_a \leq 1$ . Otherwise,  $\lceil |z|_j / m \rceil = \lfloor |y|_j / m \rfloor + 1$ , so  $|z|_a - |y|_a \leq 1$ .

If  $c = 2$ , we consider the following three cases. First, if  $|y|_j \equiv 0 \pmod{m}$ , then  $|y|_a = |y|_j / m$  and  $\lceil |z|_j / m \rceil = |y|_j / m + 1$ . Thus,  $|z|_a - |y|_a \leq 1$ . Second, if  $|y|_j \equiv m - 1 \pmod{m}$ , then  $|z|_j \equiv 1 \pmod{m}$  and  $\lceil |z|_j / m \rceil = \lfloor |y|_j / m \rfloor + 2$ . Therefore,  $|z|_a - |y|_a \leq 2$ . Otherwise,  $\lceil |z|_j / m \rceil = \lfloor |y|_j / m \rfloor + 1$ , so  $|z|_a - |y|_a \leq 1$ .

If  $c \geq 3$ , then we note that  $\lceil |z|_j / m \rceil < |z|_j / m + 1$  and  $\lfloor |y|_j / m \rfloor > |y|_j / m - 1$ . Therefore,  $|z|_a - |y|_a \leq (|z|_a - |y|_a) / m + 2 = c / m + 2 \leq c / 2 + 2$ . This difference must be an integer, so, when  $c = 3$ ,  $|z|_a - |y|_a \leq 3$ . For  $c \geq 4$ , it is clear that  $c / 2 + 2 \leq c$ .

These cases all show that, in the disaggregated word,  $|z|_a - |y|_a \leq c$ , so the count-balance of the disaggregated word is not larger than the count-balance of the aggregated word. Q.E.D.

**Theorem 2.** Disaggregating a word does not increase its gap-balance.

**Proof.** Consider the aggregated word  $U_+$  and the disaggregated word  $U_-$ . The disaggregation replaces  $mx$  copies of the letter  $j$  in  $U_+$  by  $x$  copies of the  $m$  letters in  $B_j$  in a round-robin manner.

Let  $v$  be the gap-balance of a letter  $i$  in  $U_-$  that replaced the letter  $j$  in  $U_+$ . Then there exist factors  $W'$  and  $W$  such that  $|W'|_i = |W|_i$  and  $|W'| = |W| + v$ . Moreover, the positions immediately before and after  $W'$  and  $W$  contain the letter  $i$ . Let  $t = |W'|_i = |W|_i$ . Therefore,



factors  $W'$  and  $W$  contain  $t$  copies of  $i$  and  $t+1$  copies of each of the  $m-1$  other letters than replaced the letter  $j$ . Therefore, in the word  $U_+$ , the positions corresponding to  $W'$  contain  $t+(m-1)(t+1)$  copies of  $j$ . Likewise, the positions corresponding to  $W$  contain  $t+(m-1)(t+1)$  copies of  $j$ . Moreover, the positions immediately before and after  $W'$  and  $W$  contain the letter  $j$ . Because  $|W'| = |W| + v$ , the gap-balance of the letter  $j$  in  $U_+$  must be at least  $v$ . Therefore, the gap-balance of any letter that replaced  $j$  is less than or equal to the gap-balance of  $j$ . The gap-balance of no other letter changes because of the disaggregation, so the gap-balance of the disaggregated word is less than or equal to the gap-balance of the aggregated word. Q.E.D.

## Computational Experiments

The purpose of the computational experiments was to compare the performance of the heuristics and to show how the aggregation technique performs in combination with these heuristics to find balanced words. All of the algorithms were implemented in Matlab and executed using Matlab R2006b on a Dell Optiplex GX745 with Intel Core2Duo CPU 6600 @ 2.40 GHz and 2.00 GB RAM running Microsoft Windows XP Professional Version 2002 Service Pack 3.

We generated 1,800 instances as follows. First, we set the value of  $T$  and the number of letters  $n$ . To generate an instance, we generated  $T-n$  random numbers from a discrete uniform distribution over  $\{1, \dots, n\}$ . We then let  $x_i$  equal one plus the number of copies of  $i$  in the set of  $T-n$  random numbers (this avoided the possibility that any  $x_i = 0$ ). We generated 100 instances for each of the combinations of  $T$  and  $n$  shown in Table 3.

Table 3. Combinations of  $T$  and  $n$  used to generate instances.

$T$	$n$								
100	10	20	30	40	50	60	70	80	90
500	50	100	150	200	250	300	350	400	450

All of these instances can be aggregated. For each instance, we constructed solutions as follows. First, we applied one of the basic heuristics to the instance (we call this the H solution). Next, we aggregated the instance. For the aggregate instance, we applied the heuristic to construct an aggregated solution. We disaggregated this solution to construct the AHD solution. This makes two policies using one basic heuristic. We repeated this for the remaining basic heuristics for a total of 10 policies.

Before discussing the results of the heuristics, we consider first how many times that an instance could be aggregated. Table 4 shows that the average number of aggregations decreases steadily as  $n$  increases. For instance, the average number of aggregations per instance is near six for  $T = 100$  and  $n = 20$ , but, as  $n$  increases, this decreases to just over two.

Table 4. Average number of aggregations for the instances in each problem set.

$T$	$n$	Average number of aggregations
100	10	2.66
	20	6.00
	30	5.71
	40	5.11
	50	4.03
	60	3.68
	70	3.23
	80	2.64
	90	2.07
500	50	10.77
	100	9.20
	150	7.39
	200	6.09
	250	5.10
	300	4.34
	350	3.84
	400	3.20
	450	2.69

As  $n$  approaches  $T$ , the average number of letters in the aggregated instances also decreases because the aggregation depends upon the number of distinct values of values. Each distinct value leads to an aggregation of multiple letters and generates a letter in the aggregated instance. Thus, the number of letters in the aggregated instance generally equals the number of aggregations needed to create it. Of course, there are some cases in which two groups can be combined, which increases the number of aggregations and reduces the number of letters, and some letters may have unique values, but this occurred less often as  $n$  increased. As  $n$  approaches  $T$ , the number of distinct values decreases, so there are fewer aggregations and fewer letters in the aggregated instances.

We will first consider the results for minimizing the count-balance. As shown in Table 5, the stride scheduling and bottleneck heuristics generated words with larger count-balances. The performance of the GR heuristic improved as  $n$  increased (and approached  $T$ ). Using aggregation led to the best solutions with the stride, bottleneck, and search algorithms. Aggregation was not as useful with the GR heuristic.

As shown in Table 6, with the gap-balance, the general trend is similar, but the differences are greater because the gap-balance can be quite large for some words. The bottleneck heuristic generated words with larger gap-balances. Using aggregation with the stride and search heuristics consistently generated the best solutions. The GR heuristic generated poor-quality solutions when  $n$  was small, but, as  $n$  approached  $T$ , the solution quality dramatically improved. Interestingly, using aggregation with the GR heuristic generated better solutions when  $n$  was small, but constructed more unbalanced solutions as  $n$  approached  $T$ .

Table 5. Average values of the count-balance for the H and AHD solutions generated using five basic heuristics.

$D$	$N$	Search		Bottleneck		GR		Stride 0.5		Stride 1.0	
		H	AHD	H	AHD	H	AHD	H	AHD	H	AHD
100	10	2	2	2.85	2	2.89	2.62	2.70	2.01	2.91	2.02
	20	2	2	2.99	2	2.24	2.43	2.85	2	3.15	2
	30	2	2	3	2	2	2.26	3	2	3.60	2
	40	2	2	2.98	2	2	2.03	3.04	1.99	3.22	1.99
	50	2	1.97	3	1.97	2	2	2.80	1.95	3.75	1.99
	60	2	1.81	3	1.86	2	1.94	3.24	1.82	4.12	1.86
	70	2	1.65	3	1.71	1.99	1.71	3.37	1.63	3.57	1.68
	80	2	1.68	2.9	1.85	1.9	1.85	3.08	1.58	3.08	1.85
	90	2	1.36	2.43	1.36	1.43	1.36	2.43	1.36	2.43	1.36
500	50	2	2	3	2	3.31	2.82	3	2	3.05	2
	100	2	2	3	2	2.16	2.47	3	2	3.75	2
	150	2	2	3	2	2	2.19	3.02	2	4.34	2
	200	2	2	3	2	2	2.02	3.54	2	4.19	2
	250	2	2	3	2	2	2	3.71	2	4.71	2
	300	2	2	3	2	2	2	3.68	1.95	4.81	1.99
	350	2	1.95	3	2	2	1.98	3.79	1.92	4.19	1.96
	400	2	1.76	3	2	2	1.93	3.65	1.71	3.65	1.81
	450	2	1.43	2.92	1.75	1.92	1.75	2.96	1.42	2.96	1.43

Table 6. Average values of the gap-balance for the H and AHD solutions generated using five basic heuristics.

$D$	$N$	Search		Bottleneck		GR		Stride 0.5		Stride 1.0	
		H	AHD	H	AHD	H	AHD	H	AHD	H	AHD
100	10	4.19	3.98	8.84	6.14	23.22	17.49	8.55	4.57	8.84	5.45
	20	5.00	3.73	18.02	7.98	50.12	16.98	16.04	3.97	18.66	4.70
	30	5.37	3.37	26.04	5.71	37.42	12.22	23.70	3.32	28.39	3.69
	40	5.37	3.05	33.41	4.89	22.23	10.70	29.29	2.89	38.17	3.40
	50	5.40	2.70	41.07	5.15	15.38	9.10	34.20	2.65	47.80	2.95
	60	5.18	2.19	47.24	4.83	9.65	7.26	49.46	2.16	57.15	2.31
	70	4.77	1.80	55.33	4.16	5.02	5.46	64.16	1.80	67.11	1.88
	80	4.27	1.62	61.66	3.03	2.48	4.96	75.74	1.58	76.18	1.78
	90	3.15	0.79	80.00	2.23	0.86	2.23	83.80	0.79	83.80	0.79
500	50	11.20	6.93	48.41	16.88	244.27	78.62	45.31	7.28	48.52	9.60
	100	13.32	5.88	94.61	18.47	296.16	48.92	84.33	5.89	98.47	6.91
	150	14.36	5.17	133.22	19.14	197.40	41.10	117.60	5.09	147.92	5.67
	200	14.50	4.32	171.85	22.88	118.38	32.25	150.47	4.16	197.91	4.53
	250	14.82	3.67	203.50	21.70	80.88	25.55	179.02	3.49	247.60	3.90
	300	14.86	3.01	237.93	20.71	48.40	19.53	249.33	2.91	296.20	3.17
	350	14.79	2.50	281.61	15.74	25.88	16.22	322.94	2.52	345.97	2.63
	400	13.99	1.92	311.46	13.51	10.08	13.19	388.71	1.92	395.38	2.06
	450	12.87	1.36	400.08	13.71	2.66	6.38	443.66	1.34	443.82	1.37

We also measured the clock time needed to generate these policies. Table 7 summarizes these results, and Figure 2 shows the average time needed to generate the different policies for different heuristics and different values of  $T$ . These are averages over all of the corresponding problem sets.

As  $T$  increased, the time required increased for all heuristics and policies. The search heuristic took the most time, and using aggregation further increased the time required. This occurs because the search heuristic repeatedly evaluates the gap-balance of the solutions generated, aggregating increases the densities, and evaluating the gap-balance requires more effort as the number of large densities increases. The other heuristics (with or without aggregation) took much less time. For these heuristics, the time required increased when  $T$  increased, but increasing  $n$  made little no difference, except for the stride scheduling heuristic, which, when  $T = 500$ , required more time as  $n$  increased. Using aggregation reduced the time required for the bottleneck heuristic for all values of  $n$  and  $T$ . Using aggregation with the GR heuristic did not affect the time required. Using aggregation with the stride scheduling heuristic increased the time required slightly when  $T = 100$  but reduced the time required when  $T = 500$ . For both values of  $T$ , using aggregation with stride scheduling heuristic required less time than the GR heuristic.

These results show that using the stride scheduling heuristic with aggregation generates the best solutions with the least computational effort (compared to the other heuristics).

Table 7. Average values of the gap-balance for the H and AHD solutions generated using five basic heuristics.

$D$	$N$	Search		Bottleneck		GR		Stride 0.5		Stride 1.0	
		H	AHD	H	AHD	H	AHD	H	AHD	H	HE
100	10	2.9596	4.9964	0.0015	0.0014	0.0020	0.0023	0.0009	0.0012	0.0005	0.0012
	20	2.1294	5.5540	0.0011	0.0009	0.0016	0.0019	0.0006	0.0008	0.0006	0.0008
	30	1.8855	5.8556	0.0012	0.0009	0.0016	0.0019	0.0006	0.0008	0.0006	0.0008
	40	1.7923	6.1272	0.0012	0.0008	0.0017	0.0019	0.0007	0.0008	0.0006	0.0008
	50	1.7395	6.6604	0.0011	0.0008	0.0017	0.0019	0.0007	0.0008	0.0007	0.0008
	60	1.6999	7.7378	0.0011	0.0007	0.0017	0.0018	0.0007	0.0008	0.0007	0.0008
	70	1.6655	9.7657	0.0011	0.0006	0.0018	0.0019	0.0008	0.0008	0.0008	0.0008
	80	1.6308	11.1628	0.0011	0.0006	0.0018	0.0018	0.0008	0.0008	0.0008	0.0008
	90	1.5928	15.1616	0.0010	0.0006	0.0019	0.0018	0.0009	0.0008	0.0009	0.0008
500	50	14.4718	74.3008	0.0135	0.0063	0.0080	0.0085	0.0035	0.0034	0.0034	0.0034
	100	10.7971	93.3957	0.0175	0.0050	0.0083	0.0085	0.0046	0.0034	0.0045	0.0034
	150	10.2488	112.3142	0.0192	0.0043	0.0088	0.0086	0.0057	0.0035	0.0056	0.0034
	200	10.3019	136.1293	0.0207	0.0037	0.0092	0.0087	0.0068	0.0036	0.0066	0.0036
	250	10.5470	166.3376	0.0215	0.0033	0.0096	0.0089	0.0079	0.0038	0.0077	0.0038
	300	10.8360	195.6134	0.0217	0.0030	0.0100	0.0091	0.0090	0.0040	0.0087	0.0040
	350	11.1434	245.6855	0.0220	0.0032	0.0105	0.0095	0.0101	0.0044	0.0098	0.0044
	400	11.4403	341.7110	0.0219	0.0034	0.0110	0.0097	0.0111	0.0046	0.0108	0.0046
	450	11.7140	528.2417	0.0205	0.0037	0.0115	0.0100	0.0122	0.0050	0.0118	0.0050

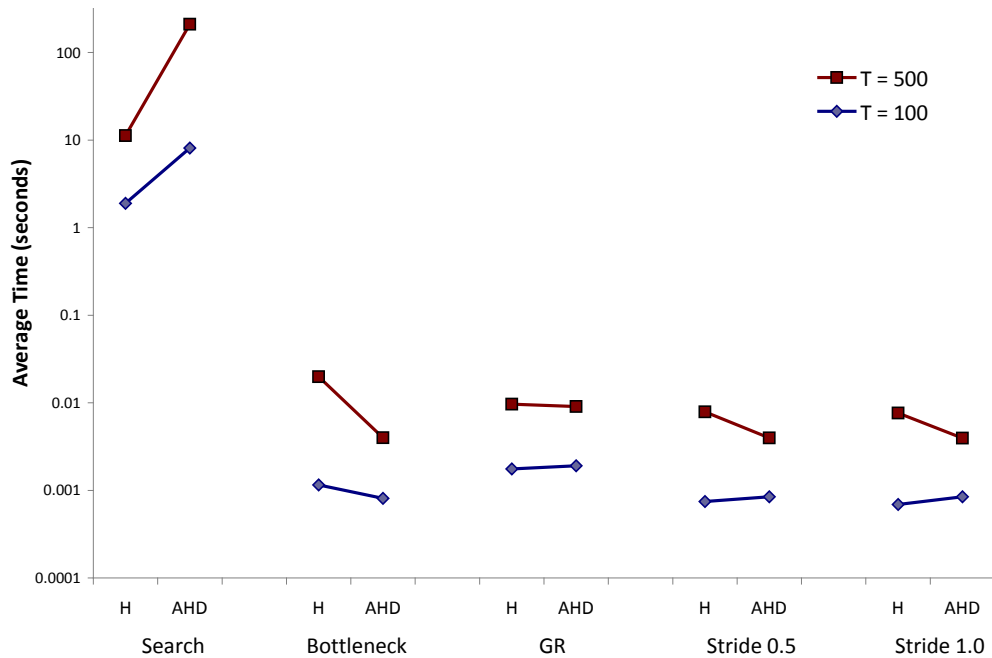


Figure 2. Average time required to generate policies for different heuristics, solutions, and values of  $T$ . Times are averaged over the corresponding problem sets and instances within those sets. Note that the vertical scale is logarithmic in order to improve the clarity of the figure.

## Summary and Conclusions

This paper presents an aggregation approach for the problem of finding balanced words, which have applications in many sequencing problems. We used two different measures to evaluate words. We combined this approach with various heuristics in order to determine when aggregation is useful. The aggregation algorithm runs in polynomial time, but the solution generation and disaggregation algorithms require pseudo-polynomial time.

The results show that using aggregation can generate more balanced solutions. Moreover, using aggregation can reduce the computational effort needed to construct a solution.

Among the heuristics, the results of our experiments show that the GR heuristic generates balanced words without aggregation. When combined with aggregation, stride scheduling generates the best solutions. The bottleneck heuristic does not perform as well. The search

algorithm of Sano *et al.* (2004) generates good solutions, but it requires additional computational effort.

For the BWP, we recommend using aggregation with stride scheduling. These techniques generate the best policies and require little computational effort.

The results here, along with the results of Herrmann (2009a, b) on using aggregation for the RTV problem, indicate that this type of aggregation approach is a powerful technique for problems that require generating a fair sequence. Unlike the previous work, which focused on specific scheduling problems, the work presented in this paper considers the more general problem of finding balanced words, which have applications in numerous domains.

The aggregation procedure presented here cannot aggregate an instance if all of the letters have different values. For such cases, the results here indicate which heuristics perform well without aggregation. In general, it may be useful to develop and test other types of aggregation. Future work will consider systematic approaches along this line.

## References

- Altman, E., B. Gaujal, and A. Hordijk (2000) “Balanced sequences and optimal routing,” *Journal of the ACM*, Volume 47, Number 4, pages 752–775.
- Balinski, M.L, and H.P. Young (1982) *Fair Representation*. Yale University Press, New Haven, Connecticut.
- Corominas, Albert, Wieslaw Kubiak, and Natalia Moreno Palli (2007) “Response time variability,” *Journal of Scheduling*, 10:97-110.
- Hajek, B. (1985) “Extremal Splittings of Point Processes,” *Mathematics of Operations Research*, Volume 10, pages 543-556.

- 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) "Constructing Perfect Aggregations to Eliminate Response Time Variability in Cyclic Fair Sequences," Technical Report 2008-29, Institute for Systems Research, University of Maryland, College Park. Available online at <http://hdl.handle.net/1903/8643>
- Herrmann, Jeffrey W. (2009a) "Generating Cyclic Fair Sequences for Multiple Servers," MISTA 2009, Dublin, Ireland, August 10-12, 2009.
- Herrmann, Jeffrey W. (2009b) "Using Aggregation to Reduce Response Time Variability in Cyclic Fair Sequences," to appear in *Journal of Scheduling*, 2009.
- 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.
- Kubiak, W. (2009) *Proportional Optimization and Fairness*, Springer, New York.
- Nowicki, E., and C. Smutnicki (1989) "Worst-case analysis of an approximation algorithm for flow shop scheduling," *Operations Research Letters*, 8:171-177.
- Rock, H., and G. Schmidt (1983) "Machine Aggregation Heuristics in Shop Scheduling," *Methods of Operations Research*, 45:303-314.
- Rogers, David F., Robert D. Plante, Richard T. Wong, and James R. Evans (1991) "Aggregation and Disaggregation Techniques and Methodology in Optimization," *Operations Research*, 39(4):553-582.



- Sano, Shinya, Naoto Miyoshi, and Ryohei Kataoka (2004) “*gap-balanced* words: A generalization of balanced words,” *Theoretical Computer Science*, Volume 314, Issues 1-2, 25 February 2004, pages 97-120.
- van der Laan, D.A. (2000) “Routing Jobs to Servers with Deterministic Service Times,” Technical report 2000-20, Leiden University.
- van der Laan, Dinard (2005) “Routing Jobs to Servers with Deterministic Service Times,” *Mathematics of Operations Research*, Volume 30, Number 1, pages 195-224.
- 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.

## Appendix A. Algorithms for the Heuristics.

### count-balance algorithm

The count-balance algorithm determines the count-balance of the infinite word generate from a finite word and can be described as follows. The input is an instance  $(x_1, \dots, x_n)$  with  $x_1 \geq x_2 \geq \dots \geq x_n$ . Let  $T = x_1 + \dots + x_n$ . In this algorithm,  $S_t$  refers to the letter in position  $t$  of  $S$ .

Note that, if  $n = 1$ , then the count-balance of  $S$  equals 0.

1. Set  $c = 1$ . Set  $L_i = \max_{t=1, \dots, T} \{t : S_t = i\}$  and  $g_i = 0$  for all  $i = 1, \dots, n$ .
2. For  $t = 1, \dots, T$ , perform the following steps:
  - a. Let  $i = S_t$ . Increase  $g_i$  by 1. If  $L_i < t$ , set  $\Delta_{ig_i} = t - L_i - 1$ ; else set

$$\Delta_{ig_i} = T + t - L_i - 1.$$

3. For  $i = 1, \dots, n$ , perform the following steps:
  - a. If  $x_i \geq c + 1$  and  $\min_{j=1, \dots, x_i} \{\Delta_{ij}\} < \max_{j=1, \dots, x_i} \{\Delta_{ij}\}$ , go to step *b*. Else, go to next *i*.
  - b. For  $j = 1, \dots, x_i$ , perform the following step:
    - i. Set  $M_j^A = \max_{k=0, \dots, x_i-1} \left\{ \sum_{q=1}^j \Delta_{i, k+q} \right\} + j - 1$  and  $M_j^B = \min_{k=0, \dots, x_i-1} \left\{ \sum_{q=1}^j \Delta_{i, k+q} \right\} + j + 1$

(in these summations,  $\Delta_{i, k+q} = \Delta_{i, k+q-x_i}$  if  $k+q > x_i$ ).

- c. Set  $p = 1$ .
  - d. If  $M_p^A \geq M_{p+c-1}^B$ , increase  $c$  by 1 and go back to step *c*.
  - e. Increase  $p$  by 1. If  $p \leq x_i - c + 1$ , then go back to step *d*.
4. Return  $c$  as the count-balance.

## gap-balance algorithm

The gap-balance algorithm determines the gap-balance of the infinite word generate from a finite word and can be described as follows. The input is an instance  $(x_1, \dots, x_n)$  with  $x_1 \geq x_2 \geq \dots \geq x_n$ . Let  $T = x_1 + \dots + x_n$ . In this algorithm,  $S_t$  refers to the letter in position  $t$  of  $S$ . Note that, if  $n = 1$ , then the count-balance of  $S$  equals 0.

1. Set  $c = 1$ . Set  $L_i = \max_{t=1, \dots, T} \{t : S_t = i\}$  and  $g_i = 0$  for all  $i = 1, \dots, n$ .

2. For  $t = 1, \dots, T$ , perform the following steps:

a. Let  $i = S_t$ . Increase  $g_i$  by 1. If  $L_i < t$ , set  $\Delta_{ig_i} = t - L_i - 1$ ; else set

$$\Delta_{ig_i} = T + t - L_i - 1.$$

3. For  $i = 1, \dots, n$ , perform the following steps:

a. If  $x_i \geq 2$  and  $\min_{j=1, \dots, x_i} \{\Delta_{ij}\} < \max_{j=1, \dots, x_i} \{\Delta_{ij}\}$ , go to step b.

Else, set  $b_i = 0$  and go to next  $i$ .

b. For  $j = 0, \dots, x_i - 2$ , set  $\delta_j = \max_{k=1, \dots, x_i} \left\{ \sum_{q=0}^j \Delta_{i, k+q} \right\} - \min_{k=1, \dots, x_i} \left\{ \sum_{q=0}^j \Delta_{i, k+q} \right\}$  (in these

summations,  $\Delta_{i, k+q} = \Delta_{i, k+q-x_i}$  if  $k+q > x_i$ ).

c. Set  $b_i = \max_{j=0, \dots, x_i-2} \{\delta_j\}$ .

4. Return  $m = \max_{i=1, \dots, n} \{b_i\}$  as the gap-balance of  $S$ .

## GR algorithm

The GR algorithm can be described as follows. The input is an instance  $(x_1, \dots, x_n)$  with

$x_1 \geq x_2 \geq \dots \geq x_n$ . Let  $T = x_1 + \dots + x_n$ .

1. Set  $X_i = \sum_{k=i}^n x_k$ ,  $N_i = 0$ , and  $R_i = 0$  for all  $i = 1, \dots, n$ .
2. For  $t = 0, \dots, T-1$ , perform the following steps:
  - a. Set  $\Delta_i = x_i(1 + R_i) - N_i X_i$  for all  $i = 1, \dots, n$ .
  - b. Set  $P^t$  to the letter  $s$  where  $s = \min\{i : \Delta_i > 0\}$ .
  - c. Increase  $N_s$  by 1.
  - d. Increase  $R_i$  by 1 for all  $i = 1, \dots, s$ .
3. Return  $P^0, \dots, P^{T-1}$  as the solution.

Table A.1. The construction of a periodic solution for the instance (4, 3, 2) using the GR heuristic.

$t$	0	1	2	3	4	5	6	7	8
$N_1$	0	1	1	2	2	3	3	4	4
$N_2$	0	0	1	1	2	2	2	2	3
$N_3$	0	0	0	0	0	0	1	1	1
$R_1$	0	1	2	3	4	5	6	7	8
$R_2$	0	0	1	1	2	2	3	3	4
$R_3$	0	0	0	0	0	0	1	1	1
$\Delta_1$	4	-1	3	-2	2	-3	1	-4	0
$\Delta_2$	3	3	1	1	-1	-1	2	2	0
$\Delta_3$	2	2	2	2	2	2	2	2	2
$P^t$	1	2	1	2	1	3	1	2	3

### Parameterized stride scheduling algorithm

The parameterized stride scheduling algorithm can be described as follows. The inputs are an instance  $(x_1, \dots, x_n)$  and the parameter  $\delta$ . Let  $T = x_1 + \dots + x_n$ .

1. Initialization.  $N_i = 0$  for  $i = 1, \dots, n$ .
2. For  $t = 0, \dots, T-1$ , perform the following steps:
  - a. Set  $P^t$  to the letter  $s$  that has the largest value of  $\frac{x_i}{N_i + \delta}$ . In case of a tie, select the letter with smallest  $x_i$ .
  - b. Increase  $N_s$  by 1.
3. Return  $P^0, \dots, P^{T-1}$  as the solution.

Table A.2. The construction of a solution for the instance (4, 3, 2) using the stride scheduling heuristic with  $\delta = 0.5$ .

$t$	0	1	2	3	4	5	6	7	8
$N_1$	0	1	1	2	2	3	3	4	4
$N_2$	0	0	1	1	2	2	2	2	3
$N_3$	0	0	0	0	0	0	1	1	1
$x_1 / (N_1 + \delta)$	8	2.67	2.67	2.67	1.6	1.6	1.14	1.14	1.14
$x_2 / (N_2 + \delta)$	6	6	2	2	2	1.2	1.2	1.2	0.86
$x_3 / (N_3 + \delta)$	4	4	4	1.33	1.33	1.33	1.33	0.8	0.8
$P^t$	1	2	3	1	2	1	3	2	1

### Bottleneck algorithm

The bottleneck algorithm can be described as follows. The input is an instance  $(x_1, \dots, x_n)$ . Let

$$T = x_1 + \dots + x_n.$$

1. Set  $w = 0$  and  $y_i = 1$  for all  $i = 1, \dots, n$ .

2. For  $i = 1, \dots, n$  and  $j = 1, \dots, x_i$ , calculate the following quantities:

$$EST_{ij} = \left\lceil \frac{T(j-1) + w}{x_i} - 1 \right\rceil$$

$$LST_{ij} = \left\lfloor \frac{Tj - w}{x_i} \right\rfloor$$

3. For  $k = 0, \dots, T-1$ , perform the following steps:

- a. Let  $R = \{i : y_i \leq x_i, EST_{iy_i} \leq k, LST_{iy_i} \geq k\}$ . If  $R$  is empty, go to Step 5.
  - b. Let  $i$  be the product in  $R$  that has the smallest  $LST_{iy_i}$ .
  - c. Assign product  $i$  to position  $k + 1$ , and increase  $y_i$  by 1.
4. Save the current sequence. If  $w < \max\{x_i\}$ , then set  $w$  to the value of the smallest  $x_i$  that is greater than  $w$ , and go to Step 2. Otherwise, go to Step 5.
5. Return the last saved sequence.

### Search algorithm

The Search algorithm can be described as follows. The input is an instance  $(x_1, \dots, x_n)$  with  $x_1 \geq x_2 \geq \dots \geq x_n$ . Let  $T = x_1 + \dots + x_n$ . Let  $M$  be the total number of samples.

1. For  $a = 1, \dots, M$ , perform the following steps:
  - a. Randomly select  $\phi_i \in [0, T/x_i]$  for all  $i = 1, \dots, n$ .
  - b. For  $t = 1, \dots, T$ , set  $P^t$  to the letter  $i$  with  $\phi_i = \min\{\phi_1, \dots, \phi_n\}$  and then increase  $\phi_i$  by  $T/x_i$ .
  - c. Determine the gap-balance of  $P$ . If this is the best gap-balance so far, save  $P$ .
2. Return the best word found.