

Using Aggregation to Construct Periodic Policies for Routing Jobs to Parallel Servers with Deterministic Service Times

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

Abstract

The problem of routing deterministic arriving jobs to parallel servers with deterministic service times, when the job arrival rate equals the total service capacity, requires finding a periodic routing policy. Because there exist no efficient exact procedures to minimize the long-run average waiting time of arriving jobs, heuristics to construct periodic policies have been proposed. This paper presents an aggregation approach that combines servers with the same service rate, constructs a policy for the aggregated system, and then disaggregates this policy into a feasible policy for the original system. Computational experiments show that using aggregation not only reduces average waiting time but also reduces computational effort.

Introduction

The problem of routing arriving jobs to parallel servers with different service rates is an important problem that occurs in the control of computer systems and in other applications. As discussed in the next section, various approaches for different versions of the problem have been studied. This paper, like van der Laan (2005), focuses on the problem of minimizing the long-run average waiting time of arriving jobs when the arrival of jobs is deterministic with a constant rate and the servers' processing times are also deterministic. Moreover, the job arrival rate equals the total service capacity. We will call this the waiting time problem (WTP).

In the WTP, one job arrives per time unit, and an instance of the problem is defined by the service rates of the servers. We will focus on the case where these service rates are rational. For rational service rates, there is an optimal periodic policy, and this policy has a finite period (van der Laan, 2005). The problem is to allocate the positions in the period to the servers so that each server receives a number of jobs proportional to its capacity. WTP is therefore a problem of generating a *fair sequence*.

van der Laan (2005) presented integer programming approaches for finding a policy that minimizes the long-run average waiting time of arriving jobs. However, these can require excessive time when the number of servers is large. van der Laan (2005) presented some heuristics that construct a periodic policy and used these to construct solutions for a small set of instances. van der Laan (2000) presented the results of using these heuristics on 18 instances.

This paper presents the results of extensive computational results to test the heuristics of van der Laan (and some other relevant heuristics) on 4100 instances of WTP. We also present an aggregation approach that first aggregates an instance, constructs a solution for the aggregate instance, and then disaggregates that solution. The objective of this paper is to show that aggregation is an effective method for generating high-quality policies. The paper precisely defines the aggregation approach and presents computational results that demonstrate its effectiveness.

The remainder of the paper proceeds as follows: we will discuss related work, formulate the WTP, and then present various policy construction heuristics. Then, we discuss the algorithms for aggregating an instance and disaggregating a solution for an aggregate instance. We will also introduce a lower bound that will be used to evaluate the quality of the solutions.

We then discuss the results of computational experiments designed to evaluate the effectiveness of the heuristics and of aggregation before concluding the paper.

Related Work

Besides the analysis of van der Laan (2000, 2003, 2005), Hordijk and van der Laan (2005) studied the WTP and presented lower and upper bounds for the optimal average waiting time for deterministic systems. Sano *et al.* (2004) introduced a generalization of balanced words, applied this concept to the problem of minimizing the maximum waiting time, and showed that, under a policy that is “ m -balanced,” the maximum waiting time is at most m .

For routing of jobs in stochastic systems, Hajek (1985) 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, there exists an optimal routing that is a balanced sequence if the set of rates are balanceable. This is always true if there are two servers; otherwise, it is true only for special cases. They present the special case with only two servers, identical servers, and two sets of identical servers.

If the system is stochastic and real-time information is available about the number of jobs in each server’s queue, then dynamic policies can be used to route jobs to servers. Nevertheless, semidynamic deterministic routing policies are used to route jobs in such systems according to a predetermined pattern. See, for instance, Arian and Levy (1992), Combé and Boxma (1984), and Hordijk *et al.* (1994), and Yum (1981).

As mentioned earlier, for WTP, one seeks a *fair sequence*. As Kubiak (2004) noted, “one mathematically sound and universally accepted definition of fairness in resource allocation and scheduling has not yet emerged.” In general, the resource’s activities should be allocated so that each product (or client or customer) receives a share of the resource that is proportional to its

demand relative to the competing demands. Kubiak (2004) provided 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. For more about the Periodic Maintenance Scheduling Problem, see Wei and Liu (1983); Corominas *et al.* (2007) introduced the RTV problem. Bar-Noy *et al.* (2002) discussed the generalized maintenance scheduling problem. For the problem of scheduling multithreaded computer systems, Waldspurger and Weihl (1995) introduced the stride scheduling algorithm. 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.

Problem Formulation

The queueing system consists of arriving jobs and a set of n parallel servers, each with its own queue. Starting at $t = 0$, one job arrives every time unit. Every arriving job must be routed to one of the servers at the moment that it arrives. If the server is idle, then the job immediately begins processing. If the server is busy, then the job waits in that server's queue.

The servers' processing times are deterministic. Server i has a service rate of a_i jobs per time unit, so the job processing time is a fixed $1/a_i$ time units. The total service capacity equals the arrival rate:

$$a_1 + \dots + a_n = 1$$

We assume that all of the a_i are rational. Therefore, there exists a positive integer T and positive integers x_1, \dots, x_n such that $a_i = x_i/T$ 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 of WTP by the values of (x_1, \dots, x_n) , with $x_1 \geq x_2 \geq \dots \geq x_n$. T is the period of the system

Let $P = P^0 P^1 \dots P^{T-1}$ be a periodic policy for this system. The job arriving at time t is routed to server $s = P^t = P^\tau$, where $\tau \in \{0, \dots, T-1\}$ and t is congruent to τ modulo T . Let b_t be the time at which the job arriving at time t starts. Let W^t be the waiting time of this job. Then, $W^t = b_t - t$. Because the system is empty at $t = 0$, $W^0 = 0$. The job will complete at time $b_t + 1/a_s$. The next job routed to server s cannot start before this completion time.

The long-run average waiting time under policy P is W :

$$W = \limsup_{\tau \rightarrow \infty} \frac{1}{\tau} \sum_{t=1}^{\tau} W^t$$

There exist periodic policies that minimize W , and T is the smallest period of such optimal periodic policies (van der Laan, 2005).

Thus, we can describe WTP as follows: Given an instance (x_1, \dots, x_n) , find a periodic policy P of length T that minimizes W subject to the constraints that exactly every arriving job is routed to exactly one server in each and every one of the T positions and each and every server i occurs exactly x_i times in P .

The complexity of WTP appears to be open. Given an instance of WTP, the question of finding a periodic policy with $W = 0$ requires finding a constant gap word for (x_1, \dots, x_n) . A constant gap word is a periodic infinite word on the alphabet $\{1, \dots, n\}$ such that each letter i occurs at positions $f_i + kT/x_i$ for all $k \in \{0, 1, 2, \dots\}$, where f_i is the position of the first occurrence of i . That is, the gap between consecutive positions of letter i is always T/x_i . A

constant gap word exists only if the quantities T/x_i are all integers. The complexity of this constant gap problem is open (Kubiak, 2004). Nevertheless, WTP is related to the Periodic Maintenance Scheduling Problem (see Wei and Liu, 1983), which is NP-complete in the strong sense (Kubiak, 2009), and the RTV problem, which is NP-hard (Corominas *et al.*, 2007).

To simplify the evaluation of a policy, define u_i^s as the total amount of time units that server i has been idle between time $t = 0$ and $t = s$. This depends upon how many jobs were routed to server i , when they arrived, and the server's processing time. Then, S^t , the total unused work capacity at time t , and S , the total unused work capacity, can be determined as follows:

$$S^t = \sum_{i=1}^n a_i u_i^t$$

$$d_i = \lim_{t \rightarrow \infty} a_i u_i^t$$

$$S = \lim_{t \rightarrow \infty} S^t = \sum_{i=1}^n d_i$$

When a periodic policy is applied, $W = S^t - (n-1)/2$ for $t \geq T-1$ (van der Laan, 2005).

Heuristics

To construct periodic policies for the WTP, we consider the basic heuristics and the improvement heuristic of van der Laan (2005). We will also consider the parameterized stride scheduling heuristic. The following discussion introduces the heuristics. Detailed algorithms 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.

OSSM. The one step S minimization (OSSM) algorithm is a greedy algorithm that attempts to minimize W by minimizing S . At time t an OSSM algorithm routes the arriving job

to the server such that S^{t+1} is minimized. To break ties, two different tie-breaking rules are used, which yield two different versions of the algorithm: OSSM1 routes the arriving job to the fastest server, and OSSM2 routes the arriving job to the slowest server. The computational effort of the OSSM algorithm is $O(nT)$.

SWT. The shortest waiting time (SWT) algorithm is a greedy algorithm that routes the arriving job to the first server that can start processing the job. The computational effort of the SWT algorithm is $O(nT)$.

GR. The greedy regular (GR) algorithm tries to make the routing of jobs to each server resemble a regular sequence as much as possible (see Kubiak, 2009, for more about regular sequences). The computational effort of the GR algorithm is $O(nT)$.

Stride. The parameterized stride scheduling algorithm builds a fair sequence (Kubiak, 2004). The algorithm has a single parameter δ that can range from 0 to 1. 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 $O(nT)$.

SG. The special greedy (SG) algorithm is used to improve an existing policy. The computational effort of the SG algorithm is $O(nT)$. Given a policy generated by one of the heuristics and the corresponding input values (d_1, \dots, d_n) given by $d_i = \lim_{\tau \rightarrow \infty} a_i u_i^\tau$, we apply the SG algorithm repeatedly until it converges, i.e., when the generated policy and the input values (d_1, \dots, d_n) do not change anymore. Under some conditions, the SG algorithm will generate an optimal solution (Van der Laan, 2005).

Aggregation

To improve the performance of these heuristics, we employed an aggregation approach that first aggregates an instance, constructs a solution for the aggregate instance, 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.

This aggregation scheme was developed to generate better solutions for the RTV problem (Herrmann, 2007). Subsequently, the author discovered some similar ideas in the literature, though no one had previously studied the benefit of aggregation systematically. Wei and Liu (1983) suggested substituting a single machine for multiple machines that have the same maintenance period. Altman *et al.* (2000) observed that a problem with only two distinct service rates can be solved by transforming it into a problem with exactly two servers. Van der Laan (2003) suggested combining servers with identical rates and demonstrated this idea on an example.

Testing aggregation on a large number of problem instances showed that using aggregation with parameterized stride scheduling and an improvement heuristic generates solutions with lower RTV than those generated by parameterized stride scheduling and an

improvement heuristic (Herrmann, 2009a, b). Using aggregation also reduces the computational effort.

The aggregation approach used here repeatedly aggregates an instance until it cannot be aggregated any more. Each aggregation combines servers that have the same service rate into a group. These servers are removed, and the group becomes a new server in the new aggregated instance. The servers with the smallest service rate are combined first. Aggregation reduces the number of servers 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 and I_k be the k -th instance generated from I_0 . Let n_k be the number of servers in instance I_k . Let B_j be the set of servers that form the new server j , and let $B_j(i)$ be the i -th server in that set. As the aggregation algorithm is presented, we describe its operation on the following five-server example: $I_0 = (3, 2, 2, 1, 1)$, $n = 5$, and $T = 9$.

Aggregation. Given: an instance I_0 with service rates (x_1, x_2, \dots, x_n) .

1. Initialization. Let $k = 0$ and $n_0 = n$.
2. Stopping rule. If all of the servers in I_k have different service rates, return I_k and $H = k$ because no further aggregation is possible. Otherwise, let G be the set of servers with the same service rate such that any smaller service rate 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 servers in G . Create a new server $n + k + 1$ with service rate $x_{n+k+1} = mx_i$. Create the new instance I_{k+1}

by removing from I_k all m servers in G and adding server $n + k + 1$. Set

$$B_{n+k+1} = G. \quad n_k = n_{k-1} - m + 1. \quad \text{Increase } k \text{ by 1 and go to Step 2.}$$

Example. With $k = 0$ and $G = \{4, 5\}$, the new server 6 has service rate $x_6 = 2 \times 1 = 2$. $B_6 = \{4, 5\}$. The servers in I_1 are $\{1, 2, 3, 6\}$. When $k = 1$, $G = \{2, 3, 6\}$. The new server 7 has service rate $x_7 = 3 \times 2 = 6$, and $B_7 = \{2, 3, 6\}$. The servers in I_2 are $\{1, 7\}$, which have different service rates.

Table 2. The service rates for the five original servers in the example instance I_0 and the two new servers 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

Table 2 describes the instances created for this example.

At any point during the aggregation, the total service rate in a new instance will equal the total service rate of the original instance because the service rate of the new server equals the sum of the service rates of the servers 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 servers 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.

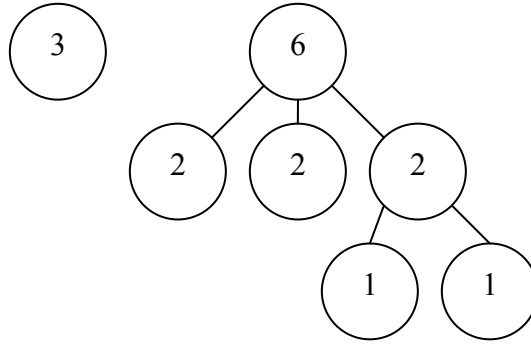


Figure 2. The forest corresponding to the aggregation of the example. The five leaf nodes correspond to the original servers in the example. The two parent nodes correspond to the new servers created during the aggregation. The two root nodes correspond to the servers remaining in the most aggregated instance.

We can represent the aggregation as a forest of weighted trees. There is one tree for each server in the aggregated instance I_H . The weight of the root of each tree is the total service rate of the servers in I_0 that were aggregated to form the corresponding server 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 servers in I_0 that were aggregated to form the corresponding server in I_H , and each one's weight equals the service rate of that server. The forest has one parent node for each new server formed during the aggregation, and the total number of nodes in the forest equals $n + H < 2n$. Figure 2 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 periodic policy for the aggregated instance I_H and then disaggregate that policy. We will use the heuristics presented earlier to construct a feasible policy. This section presents the disaggregation procedure.

Let S_H be a feasible policy for the instance I_H . In particular, S_H is a sequence of length T . Each position in S_H is a server 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 policy in turn to generate S_{H-2}, \dots, S_0 . S_0 is a feasible policy for I_0 , the original instance.

The basic idea of disaggregating a policy S_k is to replace each new server with the servers used to form it. Server $n+k$ was formed to create instance I_k from the servers 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 servers in B_{n+k} . The first position in S_k assigned to server $n+k$ will, in the new policy S_{k-1} , go to the first server in B_{n+k} , the second position assigned to server $n+k$ will go to the second server in B_{n+k} , and so forth. This will continue until all x_{n+k} positions have been assigned. Each server 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 server j is in position a in policy S_k , and $B_{n+k}(i)$ is the i -th server in B_{n+k} .

Disaggregation. Given: The instances I_0, \dots, I_H and the policy S_H , a feasible policy 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 policy $S_2 = 7-7-1-7-7-1-7-7-1$, which is a feasible policy 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 server 7 will be reassigned to servers 2, 3, and 6. The resulting policy $S_1 = 2-3-1-6-2-1-3-6-1$.

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

Table 3. The disaggregation of policy S_2 for instance I_2 in the example. The first row is S_2 , a feasible policy for instance I_2 . The second row is S_1 , a feasible policy for instance I_1 . The third row is S_0 , a feasible policy 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 policy disaggregation requires $O(T)$ effort, disaggregation runs in $O(nT)$ time in total.

Lower Bound

To provide some idea of the quality of the solutions generated, we will compare their long-term average waiting time W to a lower bound. van der Laan (2005) provides the following lower bound for W :

$$W \geq LB = \frac{1}{2} - \frac{C(x_1, \dots, x_n)}{2T}$$

$$C(x_1, \dots, x_n) = \sum_{i=1}^n \gcd(x_i, T)$$

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 minimize W . 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 4,100 instances as follows. First, we set the value of T and the number of servers 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 combination of T and n shown in Table 4.

All of these instances can be aggregated. For each instance, we constructed 24 policies as follows. First, we applied one of the basic heuristics to the instance (we call this the H policy). Then, we applied the SG algorithm repeatedly to the H policy to construct the HE policy.

Table 4. 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						
1000	100	200	300	400	500	600	700	800	900						
1500	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	

Table 5. Average number of aggregations for the instances in each problem set.

T	n	Average number of aggregations	Average number of aggregate servers
100	10	2.66	6.45
	20	6.00	6.16
	30	5.71	5.37
	40	5.11	5.10
	50	4.03	4.58
	60	3.68	3.92
	70	3.23	3.43
	80	2.64	3.06
	90	2.07	2.43
500	50	10.77	11.36
	100	9.20	9.14
	150	7.39	7.83
	200	6.09	6.57
	250	5.10	5.59
	300	4.34	4.77
	350	3.84	4.16
	400	3.20	3.62
	450	2.69	2.96
1000	100	12.81	12.76
	200	10.09	9.84
	300	7.87	8.40
	400	6.46	7.01
	500	5.57	6.08
	600	4.82	5.22
	700	4.12	4.55
	800	3.52	4.00
	900	2.97	3.15
1500	100	14.85	16.17
	200	12.53	12.84
	300	10.69	10.29
	400	8.87	9.48
	500	7.63	8.25
	600	6.72	7.39
	700	6.11	6.55
	800	5.45	5.94
	900	5.03	5.50
	1000	4.50	4.88
	1100	4.06	4.46
	1200	3.74	4.10
	1300	3.13	3.46
	1400	2.86	3.05

Next, we aggregated the instance. For the aggregate instance, we applied the heuristic to construct an aggregated policy. We disaggregated this policy to construct the AHD policy. Then, we applied the SG algorithm repeatedly to the AHD policy to construct the AHDE policy. This makes four policies using one basic heuristic and combinations of aggregation-disaggregation and the SG algorithm. We repeated this for the remaining basic heuristics for a total of 24 policies.

Before discussing the results of the heuristics, we consider first how many times that an instance could be aggregated and the number of aggregate servers remaining in the most aggregated instances. Table 5 shows that these values are generally correlated and that, as n increases, the average number of aggregations and the number of aggregate servers remaining decreases steadily for all values of T . (The only exception is for $T = 100$ when n increases from 10 to 20.) 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. The average number of aggregations per instance is 14.85 for $T = 1500$ and $n = 100$ but decreases to 2.86 for $T = 1500$ and $n = 1400$. The largest number of aggregations was 18, which occurred in five instances (one with $n = 100$ and $T = 1000$ and four with $n = 100$ and $T = 1500$). Among the instances within a single problem set, the variation in the number of aggregations and the number of aggregate servers was low. For instance, for $T = 1500$ and $n = 500$, the number of aggregate servers ranged from 7 to 10.

As n approaches T , the average number of servers in the aggregated instances also decreases because the aggregation depends upon the number of distinct values of service rates. Each distinct value leads to an aggregation of multiple servers and generates a server in the aggregated instance. Thus, the number of servers 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 servers, and some servers may have unique service rates, 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 servers in the aggregated instances. For instance, with $T = 1500$, the

average number of servers in the aggregated instances went from 16.17 when $n = 100$ to 3.05 when $n = 1400$.

The results of using the parameterized stride scheduling algorithm with different values of δ were nearly identical. Hereafter, we report only the results with $\delta = 0.5$.

Table 6. Average values of the lower bound (LB) and the W for the H and HE policies generated using five basic heuristics.

T	n	LB	OSSM1		OSSM2		SWT		GR		Stride 0.5	
			H	HE	H	HE	H	HE	H	HE	H	HE
100	10	0.33	1.08	1.04	1.09	1.05	1.13	1.07	1.72	1.50	1.70	1.25
	20	0.21	1.30	1.17	1.31	1.15	1.38	1.16	2.22	1.86	2.65	1.59
	30	0.14	1.50	1.17	1.50	1.16	1.46	1.15	2.19	1.89	3.56	1.88
	40	0.11	1.90	1.61	1.96	1.64	1.94	1.66	1.79	1.51	4.58	1.91
	50	0.10	1.52	1.18	1.61	1.16	1.63	1.26	1.37	1.20	5.65	1.99
	60	0.07	0.77	0.54	0.77	0.54	0.76	0.51	0.90	0.68	6.72	1.28
	70	0.04	0.25	0.12	0.25	0.12	0.26	0.15	0.47	0.42	7.10	0.60
	80	0.02	0.03	0.03	0.03	0.03	0.04	0.04	0.20	0.15	6.34	0.22
	90	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.04	0.03	4.04	0.10
500	50	0.33	3.80	3.03	3.83	3.07	3.88	3.07	7.79	5.43	7.49	4.91
	100	0.21	5.89	4.96	6.02	5.01	6.08	4.89	10.77	7.34	12.88	6.23
	150	0.14	7.10	5.11	7.07	5.06	7.04	4.80	10.34	7.68	17.16	8.60
	200	0.12	10.00	7.52	9.97	7.55	10.00	7.59	8.72	6.41	22.64	8.90
	250	0.09	6.98	3.92	7.03	3.97	7.07	4.01	6.61	5.40	28.64	7.59
	300	0.07	3.66	2.15	3.57	2.01	3.58	2.02	4.46	2.71	33.85	4.95
	350	0.04	1.28	0.14	1.24	0.14	1.24	0.14	2.27	1.68	35.57	1.96
	400	0.02	0.06	0.03	0.05	0.03	0.05	0.03	0.80	0.57	31.73	0.34
	450	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.09	20.21	0.10
1000	100	0.30	7.35	5.56	7.36	5.56	7.43	5.54	15.61	10.16	14.85	9.57
	200	0.19	11.83	9.86	11.83	9.87	11.96	9.85	21.51	13.84	25.45	11.71
	300	0.13	13.90	9.83	13.88	9.83	13.85	9.80	20.83	15.44	34.13	16.81
	400	0.12	19.98	15.05	20.07	15.21	20.05	15.23	17.41	12.42	45.10	17.71
	500	0.10	13.90	7.41	14.07	7.39	14.23	7.52	13.20	10.63	57.55	14.97
	600	0.07	7.00	3.70	7.00	3.70	6.98	3.59	8.83	5.04	67.85	9.48
	700	0.04	2.42	0.16	2.42	0.16	2.42	0.20	4.50	3.22	71.18	3.63
	800	0.02	0.07	0.04	0.07	0.04	0.08	0.05	1.63	1.11	63.61	0.52
	900	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.25	0.09	40.49	0.12
1500	100	0.31	8.31	6.02	8.29	6.02	8.28	6.08	17.75	12.09	16.04	10.76
	200	0.21	13.02	10.48	13.07	10.50	13.31	10.58	27.59	17.46	27.40	16.23
	300	0.11	17.84	14.47	17.87	14.46	17.95	14.46	32.18	20.18	38.00	16.83
	400	0.04	21.02	14.33	21.01	14.22	20.98	14.04	31.93	22.81	49.13	21.81
	500	0.02	22.49	14.11	22.49	14.12	22.48	14.16	29.76	20.87	56.68	21.99
	600	0.00	30.09	22.42	30.10	22.42	30.11	22.43	26.00	18.10	67.57	26.45
	700	0.00	26.39	18.90	26.41	18.91	26.45	18.94	21.70	16.02	80.16	26.50
	800	0.00	16.41	11.83	16.41	11.83	16.42	11.82	17.80	13.90	92.28	18.92
	900	0.00	10.63	5.81	10.63	5.80	10.63	5.81	13.30	7.57	101.68	14.24
	1000	0.00	5.47	0.16	5.47	0.16	5.47	0.16	8.67	5.57	106.22	7.72
	1100	0.00	1.91	0.09	1.91	0.09	1.91	0.09	4.97	3.96	105.35	3.36
	1200	0.00	0.08	0.02	0.08	0.02	0.08	0.02	2.40	1.76	95.36	0.68
	1300	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.80	0.12	75.01	0.21
	1400	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.06	43.56	0.08

Table 7. Average values of W for the AHD and AHDE policies generated using five basic heuristics. (The average value of W for the H policies generated using OSSM1 is shown for comparison.)

T	n	OSSM1			OSSM2		SWT		GR		Stride 0.5	
		H	AHD	AHDE	AHD	AHDE	AHD	AHDE	AHD	AHDE	AHD	AHDE
100	10	1.08	0.93	0.91	0.94	0.92	0.96	0.93	1.28	1.11	1.00	0.94
	20	1.30	0.68	0.64	0.68	0.64	0.69	0.64	0.98	0.74	0.70	0.65
	30	1.50	0.47	0.43	0.47	0.43	0.48	0.44	0.74	0.52	0.48	0.44
	40	1.90	0.35	0.32	0.36	0.32	0.37	0.33	0.54	0.39	0.36	0.32
	50	1.52	0.27	0.24	0.27	0.24	0.28	0.24	0.45	0.33	0.27	0.25
	60	0.77	0.18	0.16	0.18	0.16	0.18	0.16	0.35	0.21	0.18	0.16
	70	0.25	0.10	0.09	0.09	0.08	0.10	0.10	0.26	0.14	0.10	0.09
	80	0.03	0.05	0.05	0.05	0.05	0.07	0.06	0.21	0.12	0.06	0.06
	90	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.10	0.04	0.01	0.01
500	50	3.80	1.31	1.19	1.30	1.19	1.33	1.21	2.14	1.40	1.35	1.20
	100	5.89	0.84	0.71	0.84	0.72	0.85	0.72	1.38	0.81	0.85	0.71
	150	7.10	0.59	0.50	0.60	0.51	0.60	0.51	0.98	0.58	0.59	0.50
	200	10.00	0.43	0.37	0.43	0.37	0.43	0.38	0.81	0.45	0.42	0.37
	250	6.98	0.29	0.25	0.29	0.25	0.29	0.25	0.65	0.31	0.29	0.25
	300	3.66	0.20	0.18	0.21	0.18	0.20	0.18	0.47	0.21	0.20	0.18
	350	1.28	0.13	0.12	0.13	0.12	0.13	0.12	0.40	0.16	0.13	0.12
	400	0.06	0.07	0.06	0.07	0.06	0.06	0.06	0.30	0.09	0.06	0.06
	450	0.00	0.01	0.01	0.01	0.01	0.02	0.02	0.12	0.05	0.02	0.02
1000	100	7.35	1.37	1.20	1.38	1.21	1.38	1.21	2.29	1.37	1.41	1.22
	200	11.83	0.87	0.73	0.87	0.73	0.88	0.73	1.50	0.82	0.87	0.73
	300	13.90	0.63	0.53	0.63	0.53	0.63	0.53	1.09	0.60	0.63	0.53
	400	19.98	0.45	0.39	0.45	0.39	0.45	0.39	0.90	0.45	0.45	0.39
	500	13.90	0.33	0.28	0.33	0.28	0.33	0.28	0.76	0.33	0.33	0.28
	600	7.00	0.21	0.18	0.21	0.18	0.21	0.18	0.56	0.22	0.21	0.18
	700	2.42	0.14	0.12	0.14	0.12	0.13	0.12	0.49	0.16	0.13	0.12
	800	0.07	0.08	0.07	0.08	0.07	0.08	0.07	0.39	0.11	0.08	0.07
	900	0.01	0.02	0.02	0.02	0.02	0.03	0.03	0.19	0.05	0.03	0.03
1500	100	8.31	1.79	1.61	1.79	1.62	1.80	1.62	3.03	1.84	1.84	1.62
	200	13.02	1.23	1.05	1.23	1.05	1.25	1.06	2.14	1.15	1.24	1.05
	300	17.84	0.89	0.76	0.90	0.76	0.90	0.76	1.57	0.84	0.90	0.76
	400	21.02	0.71	0.59	0.71	0.59	0.71	0.60	1.25	0.68	0.71	0.60
	500	22.49	0.55	0.45	0.55	0.45	0.55	0.46	1.05	0.54	0.55	0.46
	600	30.09	0.43	0.37	0.43	0.37	0.43	0.37	0.91	0.44	0.43	0.37
	700	26.39	0.33	0.28	0.33	0.29	0.33	0.29	0.81	0.35	0.33	0.29
	800	16.41	0.25	0.22	0.25	0.22	0.26	0.22	0.66	0.26	0.26	0.22
	900	10.63	0.20	0.17	0.20	0.17	0.20	0.17	0.65	0.22	0.20	0.17
	1000	5.47	0.16	0.14	0.16	0.14	0.16	0.14	0.55	0.18	0.16	0.14
	1100	1.91	0.10	0.09	0.10	0.09	0.11	0.09	0.48	0.15	0.11	0.09
	1200	0.08	0.07	0.06	0.07	0.06	0.07	0.06	0.40	0.10	0.07	0.06
	1300	0.00	0.02	0.02	0.02	0.02	0.03	0.03	0.24	0.06	0.03	0.03
	1400	0.00	0.01	0.01	0.01	0.01	0.02	0.02	0.18	0.04	0.02	0.02

The results of these experiments show that the OSSM1, OSSM2, and SWT heuristics generate policies that are better than those that the GR and stride scheduling heuristics generate. As shown in Table 6, the H policies generated by OSSM1, OSSM2, and SWT have practically no waiting for problem sets with n near T . The H policies generated by the GR heuristic are better only for moderate values of n (that is, when n is near $T/2$). The H policies generated by stride scheduling perform quite poorly for all values of n . The SG algorithm, which constructs

the HE policies, reduces W dramatically when improving the H policies generated by stride scheduling because the H policies perform quite poorly. For example, with $n = 1100$ and $T = 1500$, the average W of the HE policies is 102 time units less than the average W of the H policies, which is a 96.8% reduction. With the other H policies, the improvement after using the SG algorithm is significant but not as dramatic.

As shown in Table 7, the quality of the AHD policies and AHDE policies is very good compared to the H and HE policies. For instance, when using the OSSM1 heuristic on the instances with $n = 600$ and $T = 1500$, the average W of the H policies is 30.09, the average W of the HE policies is 22.42, the average W of the AHD policies is 0.43, and the average W of the AHDE policies is 0.37. For these instances, aggregation (without using the SG algorithm) reduces W by 29.66 time units, which is a 98.5% reduction in average waiting time. The differences between the heuristics are quite small. Even the stride scheduling algorithm, which generates poor-quality policies by itself, performs well when used with aggregation. When n is near T , all of the AHD policies have practically no waiting. Compared to the AHD policies, the AHDE policies are slightly better. In general, there is little room for improvement because the AHD policies are quite good.

We also measured the clock time needed to generate these policies. (Due to space constraints, detailed results are not included in this paper.) Figure 3 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 time needed to run only the heuristic, which generates the H policies, increased with n . For $T = 100$, the stride scheduling algorithm took the least time. For $T = 500, 1000, \text{ and } 1500$,

the stride scheduling algorithm took the least time for small n , but the GR algorithm took the least time for large n . The OSSM1, OSSM2, and SWT algorithms took more time.

Naturally, more time is needed to generate the HE policies, and the time needed is dominated by the effort spent running the SG algorithm one or more times to find better policies. In general, more effort was needed for medium values of n (those near $T/2$) than for small or large values of n (those near $T/10$ or near T). The quality of the HE policies is significantly better than the quality of the H policies for those medium values of n , which shows that the extra effort generated some benefit.

The time required for the AHD policies increased as n increased, but the time remained approximately the same for different heuristics. The time required for the AHD policies was generally much less than the time required for the H policies. For instance, consider the problem set with $T = 1500$ and $n = 1000$. The average time required to construct the AHD policies with the GR algorithm was 80% of the time required to construct the H policies. For the stride scheduling algorithm, the average time required to construct the AHD policies was 28% of the time required to construct the H policies. For the OSSM1, OSSM2, and SWT algorithms, which were slower than the GR algorithm, the average time required to construct the AHD policies was 13% to 17% of the time required to construct the H policies.

The time required for the AHDE policies increased slowly as n increased, and the time remained approximately the same for different heuristics. The time required for the AHDE policies was generally much more than the time required for the AHD policies. For instance, consider the problem set with $T = 1500$ and $n = 1000$. The average time required to construct the AHDE policies with the OSSM1, OSSM2, SWT, and GR algorithms was about 7 to 8 times more than the time required to construct the AHD policies. For the stride scheduling algorithm,

the average time required to construct the AHDE policies was over 10 times greater than the time required to construct the AHD policies.

The time required to construct the AHD and AHDE policies is roughly the same for the different heuristics used because the aggregated instance has a smaller value of n (though the value of T remains the same). For smaller values of n , the heuristics require about the same amount of time to construct a policy. Moreover, the time for aggregation and disaggregation does not depend, of course, on the heuristic.

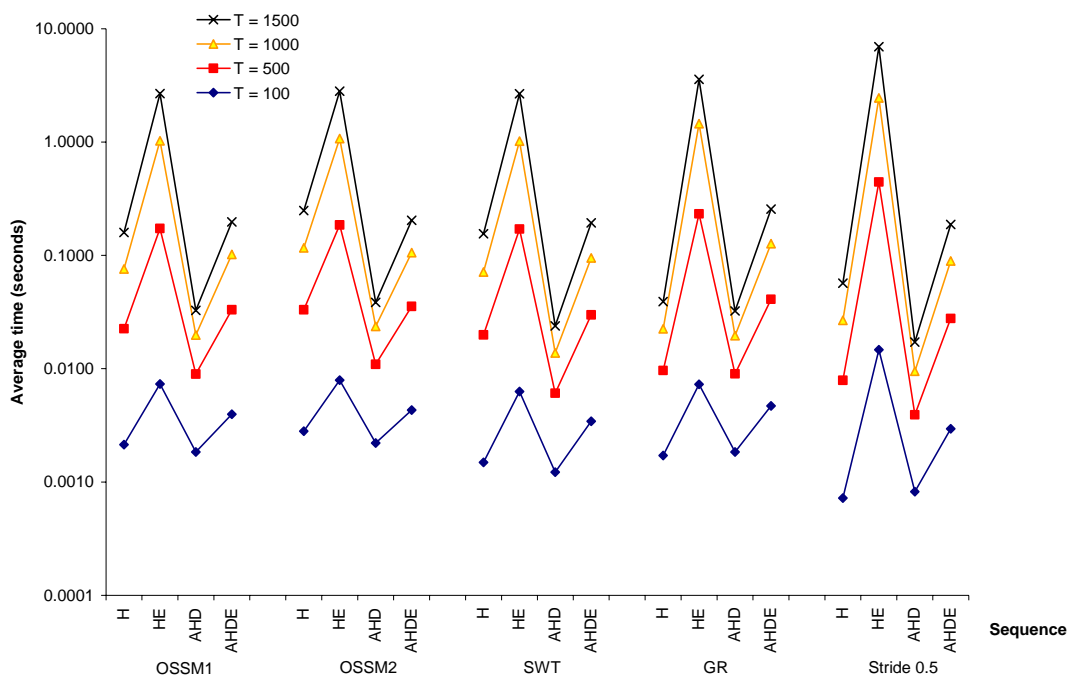


Figure 3. Average time required to generate policies for different heuristics, policies, and values of T . Times are averaged over the corresponding problem sets and instances within those sets. Note that the vertical scale is logarithmic in order to improve the clarity of the figure.

Summary and Conclusions

This paper presents an aggregation approach for the problem of minimizing long-run average waiting time. We combined this approach with various heuristics in order to determine when aggregation is useful. The aggregation algorithm runs in polynomial time, but the policy generation and disaggregation algorithms require pseudo-polynomial time.

The results show that using aggregation can generate policies that have much lower average waiting time. Moreover, using aggregation reduces the computational effort needed to construct a policy.

Among the heuristics, the results of our experiments show that the OSSM1, OSSM2, and SWT heuristics generate policies that are better than those that the GR and stride scheduling heuristics generate. Using the SG algorithm to improve policies reduces average waiting time to some degree, but it requires additional computational effort. When aggregation is used, the additional improvement is small compared to the cost of the computational effort.

For the WTP, we recommend using aggregation with the OSSM1, OSSM2, and SWT. 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. Additional work will continue investigating the usefulness of aggregation on these types of problems.

The aggregation procedure described in this paper is simple but effective. It is possible to create more sophisticated aggregations that more intelligently combine servers in order to minimize the number of servers in the highest level of aggregation, with the goal of aggregating all of the servers into one group. If there is only one group, the disaggregation leads directly to a solution with zero waiting time. Future work will consider algorithms for systematically creating more sophisticated aggregations (see, for example, Herrmann, 2008).

The aggregation procedure presented here cannot aggregate an instance if all of the servers have different service rates. 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.

Acknowledgements

The author would like to thank the anonymous reviewers whose suggestions helped me improve the presentation of the results.

References

- Altman, E., Gaujal, B., Hordijk, A. (2000). Balanced sequences and optimal routing. *Journal of the ACM*, 47(4), 752–775.
- Arian, Y., Levy, Y. (1992). Algorithms for generalized round robin. *Operations Research Letters*, 12, 313-319.
- Balinski, M.L, Young, H.P. (1982). *Fair Representation*. New Haven, Connecticut: Yale University Press.
- Bar-Noy, A., Bhatia, R., Naor, J., Schieber, B. (2002). Minimizing service and operation costs of periodic scheduling. *Mathematics of Operations Research*, 27, 518-544.
- Combé, M.B., Boxma, O.J. (1994). Optimization of static traffic allocation policies. *Theoretical Computer Science*, 125, 17-43.
- Corominas, A., Kubiak, W., Palli, N.M. (2007). Response time variability. *Journal of Scheduling*, 10, 97-110.
- Hajek, B. (1985). Extremal splittings of point processes. *Mathematics of Operations Research*, 10, 543-556.
- Herrmann, J.W. (2007). Generating cyclic fair sequences using aggregation and stride scheduling. Technical Report 2007-12, Institute for Systems Research, University of Maryland, College Park. <http://hdl.handle.net/1903/7082>. Accessed 1 November 2010.

- Herrmann, J.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. <http://hdl.handle.net/1903/8643>. Accessed 1 November 2010.
- Herrmann, J.W. (2009a). Generating cyclic fair sequences for multiple servers. MISTA 2009, Dublin, Ireland, August 10-12, 2009.
- Herrmann, J.W. (2009b). Using aggregation to reduce response time variability in cyclic fair sequences. *Journal of Scheduling*, doi:10.1007/s10951-009-0127-7
- Hordijk, A., Koole, G.M., Loeve, J.A. (1994). Analysis of a customer assignment model with no state information. *Probability in the Engineering and Informational Sciences*, 8, 419-429.
- Hordijk, A., van der Laan, D. (2005). On the average waiting time for regular routing to deterministic queues. *Mathematics of Operations Research*, 30(2), 521–544.
- Kubiak, W. (2004). Fair sequences. In J.Y-T. Leung (Ed.), *Handbook of Scheduling: Algorithms, Models and Performance Analysis* (pp. 1-21). Boca Raton, Florida: Chapman & Hall/CRC.
- Kubiak, W. (2009). *Proportional Optimization and Fairness*, New York: Springer.
- Nowicki, E., Smutnicki, C. (1989). Worst-case analysis of an approximation algorithm for flow shop scheduling. *Operations Research Letters*, 8, 171-177.
- Rock, H., Schmidt, G. (1983). Machine aggregation heuristics in shop scheduling. *Methods of Operations Research*, 45, 303-314.
- Rogers, D.F., Plante, R.D., Wong, R.T., Evans, J.R. (1991). Aggregation and disaggregation techniques and methodology in optimization. *Operations Research*, 39(4), 553-582.

- Sano, S., Miyoshi, N., Kataoka, R. (2004). “ m -balanced words: a generalization of balanced words. *Theoretical Computer Science*, 314(1-2), 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, D.A. (2003). The structure and performance of optimal routing sequences,” Ph.D. thesis, Leiden University, Leiden, The Netherlands.
- van der Laan, D.A. (2005). Routing jobs to servers with deterministic service times. *Mathematics of Operations Research*, 30(1), 195-224.
- Waldspurger, C.A., 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., Liu, C.L. (1983). On a periodic maintenance problem. *Operations Research Letters*, 2(2), 90-93.
- Yum, T.P. (1981). The design and analysis of a semi-dynamic deterministic routing rule. *IEEE Transactions on Communications*, 29, 498-504.

Appendix A. Algorithms for the Heuristics.

Recall that the values of x_1, \dots, x_n are positive integers.

OSSM algorithm

The OSSM algorithm can be described as follows. The only difference between OSSM1 and OSSM2 is the tie-breaking rule in step 2d. The input is an instance (x_1, \dots, x_n) . Let

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

The quantity w_i equals the product of v_i , the time needed by the server to finish the jobs that are currently in the system, and x_i . Note that $x_i / a_i = T$. The quantity V is used to look ahead to a point in time when some value of r_i is positive.

1. Set $w_i = 0$ for all $i = 1, \dots, n$.
2. For $t = 0, \dots, 2T - 1$, perform the following steps:
 - a. Set $v_i = w_i / x_i$ for all $i = 1, \dots, n$.
 - b. Set $V = \lfloor \min \{v_1, \dots, v_n\} \rfloor + 1$.
 - c. Set $r_i = \max \{Vx_i - w_i, 0\}$ for all $i = 1, \dots, n$.
 - d. Set P^t to the server s that has the largest value of r_i . If multiple servers have the same value: For OSSM1, select the fastest one (the one with the largest x_i).
For OSSM2, select the slowest one (the one with the smallest x_i).
 - e. Increase w_s by T .
 - f. Decrease w_i to $\max \{w_i - x_i, 0\}$ for all $i = 1, \dots, n$.
3. Return P^T, \dots, P^{2T-1} as the policy.

SWT algorithm

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

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

The quantity w_i equals the product of v_i , the time needed by the server to finish the jobs that are currently in the system, and x_i . Note that $x_i / a_i = T$.

1. Set $w_i = 0$ for all $i = 1, \dots, n$.
2. For $t = 0, \dots, 2T - 1$, perform the following steps:
 - a. Set $v_i = w_i / x_i$ for all $i = 1, \dots, n$.
 - b. Set P^t to the server s that has the smallest value of v_i . If multiple servers have the same value: select the fastest one (the one with the largest x_i).
 - c. Increase w_s by T .
 - d. Decrease w_i to $\max\{w_i - x_i, 0\}$ for all $i = 1, \dots, n$.
3. Return P^T, \dots, P^{2T-1} as the policy.

GR algorithm

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

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

Note that N_s records how many jobs have been routed to server s , and R_i records how many jobs have been routed to servers i, \dots, n . The quantity Δ_s reflects the relative balance between server s and the servers that are slower than s . $\Delta_s > 0$ indicates that the number of jobs routed to server s so far is smaller than the number that should be routed to the server when it is compared to the slower servers.

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 fastest server s with $\Delta_s > 0$. (That is, $s = \min\{i : \Delta_i > 0\}$.)
 - 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 policy.

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 δ . 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 server s that has the largest value of $\frac{x_i}{N_i + \delta}$. In case of a tie, select the server with smallest x_i .
 - b. Increase N_s by 1.
3. Return P^0, \dots, P^{T-1} as the policy.

SG algorithm

The SG algorithm can be described as follows. The input is an instance (x_1, \dots, x_n) with $x_1 \geq x_2 \geq \dots \geq x_n$ and the non-negative real values (d_1, \dots, d_n) . Let $T = x_1 + \dots + x_n$.

1. Set $A = \{1, \dots, n\}$. Set $g_i = \frac{d_i T}{x_i}$ and $N_i = 0$ for all $i = 1, \dots, n$.
2. For $t = 0, \dots, T-1$, perform the following steps:
 - a. Set P^t to the server s in A that has the smallest value of g_i . If multiple servers have the same value select the fastest one (the one with the largest x_i).
 - b. Increase g_s by T/x_s and increase N_s by 1. If $N_s = x_s$, then remove s from A .
3. Return P^0, \dots, P^{T-1} as the policy.