CYCLIC SCHEDULING
WITH ACYCLIC JOB
PRECEDENCE CONSTRAINTS:
CONSTRUCTION HEURISTICS

by

Kathryn E. Caggiano
Peter L. Jackson
Technical Report No. 1349
Cyclic Scheduling with Acyclic Job Precedence Constraints: Construction Heuristics*

Kathryn E. Caggiano†
Assistant Professor of Operations and Information Management
University of Wisconsin-Madison School of Business
975 University Avenue
Madison, WI 53706
Email: kcaggiano@bus.wisc.edu
Office: (608)263-6437
Fax: (608)263-3142

Peter L. Jackson
Associate Professor
School of Operations Research and Industrial Engineering
218 Rhodes Hall
Cornell University
Ithaca, NY 14853
Email: pj16@cornell.edu
Office: (607)255-9122
Fax: (607)255-9129

November 2002

*This work was supported in part by the National Science Foundation (Grant DMI0075627)
†Corresponding author
Abstract

Finding minimum flow time cyclic schedules for identical jobs with serial routings is known to be NP-hard. This paper is the first in a sequence of two papers addressing multi-machine, multiple-job cyclic scheduling problems where the job routings are allowed to be arbitrary partial orders, and arbitrary delays may be required between consecutive job operations. The focus of the first paper is schedule construction heuristics for such problems; the latter, improvement heuristics. Two classes of construction heuristics are devised and tested: non-cyclic heuristics, and cyclic dispatch rules. The companion paper describes improvement schemes based on a technique called compression. We demonstrate the superiority of cyclic dispatch rules for schedule construction and show that this favorable performance persists after compression.
1 Introduction

Cyclic schedules are a natural tool for understanding multistage workflow in repetitive manufacturing environments. In particular, these models can be valuable in the areas of facility design, capacity planning, lot sizing, due date quotation, and production scheduling. For multistage cyclic scheduling problems, a critical performance measurement is job flow time. At the production scheduling level, reducing flow time reduces WIP inventory, reduces required buffer sizes, and improves cash flow. At the planning level, flow time reduction decreases response time to changes in the demand process. In this paper, we consider cyclic scheduling problems in which throughput rate targets have already been set and job routings and machine assignments have been chosen. Our task is to sequence operations to minimize an appropriate measure of job flow time. The general problem is known to be NP-hard so we focus on heuristic scheduling procedures. We develop and test a variety of schedule construction algorithms for the broad class of multi-job problems in which the operation precedence constraints form a directed acyclic graph. In a companion paper, we present and evaluate improvement heuristics for the problem.

2 Literature Review

We begin this section by discussing work that is most closely related to the particular problem and objective we are considering. We then discuss cyclic scheduling models and objectives which have been devised for problems in other areas. Additional coverage can be found in a recent survey paper on cyclic scheduling by Yura [62]. Summaries of complexity results for various types of cyclic scheduling problems, almost all of which are NP-hard, can be found in Kamoun and Sriskandarajah [34] and Gallego and Shaw [21].

The problem on which this paper is based was first described by Graves et al. [26], in which the motivation was to develop a scheduling system for integrated-circuit wafer fabrication. The authors consider the problem of scheduling identical serial jobs in a re-entrant flow shop, wherein a job may visit the same machine many times (as a tray of silicon wafers returns many times to the photolithographic process during integrated-circuit manufacturing). As an alternative to using myopic job shop sequencing heuristics to schedule multiple, identical jobs, they propose scheduling a single job a priori in a cyclic manner and using this
fixed schedule to move all jobs through the fabrication facility in a predictable, systematic way. They give a simple, single pass heuristic for constructing a feasible cyclic schedule for a specified cycle length with minimum job flow time. (This problem is subsequently shown to be NP-hard by Roundy [53].) For a variation of this problem in which multi-channel and batch production facilities are allowed (in addition to single channel facilities), Yura [61] determines the minimum lot size that guarantees maximum throughput and gives a simple algorithm to construct a cyclic schedule for this lot size.

For the case of a single job with a serial line routing, Roundy [53] extends the work of Graves et al. He studies the problem of finding efficient cyclic schedules, those which are not dominated by any other schedule in both cycle length and flow time. He shows that this problem is NP-hard and gives an enumeration scheme based on examining cyclic precedence structures. Roundy shows that for a serial line routing, cycle length and job flow time are minimized simultaneously, and these minimum values can be computed efficiently.

In [51], Rao also examines the single job case and extends Roundy's result for the serial job routing to the assembly job routing under certain dedicated equipment assumptions. Rao constructs a class of problem instances to show that the Graves et al. heuristic can perform arbitrarily badly in the worst case. He analyzes two subproblems, the production timing subproblem (solvable by a polynomial-time algorithm) and the cycle offset problem (NP-hard), and develops solution techniques for them which are subsequently incorporated into heuristics for solving the overall cyclic scheduling problem for a fixed cycle length. Using these heuristics, he obtains solutions for a variety of cycle lengths and empirically evaluates the trade-off between cycle length and flow time.

The cyclic scheduling problem considered in this paper falls in the tradition of Graves et al. [26], Roundy [53], and Rao [51]. We extend the model to allow multiple job types with arbitrary acyclic job routings, and we concentrate on evaluating the relative performance of various classes of construction heuristics. Afentakis [2] proposes a model for cyclic scheduling in flexible manufacturing systems that is similar to ours. He outlines the objective of minimizing flow time within the context of his model, but offers no suggestions for solution.

In flexible manufacturing systems, where setup times are negligible, many applications of cyclic scheduling have been studied at the part level. A minimal part set (MPS) is a set of production parts with the smallest integral quantities that match in proportion the total product sales mix [31]. The usual objective is to find a periodic schedule for a single MPS
that maximizes the throughput (i.e. minimizes the cycle length). Wittrock [60], McCormick et al. [43], Lee [35] [36], Lee and Posner [37], and Song and Lee [57] [56] have all considered this MPS scheduling problem. The model they analyze requires that each machine process all of its assigned operations for the $n^{th}$ MPS before it can process any operation from the $n + 1^{st}$ MPS [60] [43] [37]. By contrast, the model considered in this paper does not have this restriction.

Huh et al. [32] derive a lower bound for the flow time of a cyclic schedule with 100% utilization on the bottleneck machine. Like Roundy and Rao, they consider identical jobs with an acyclic precedence structure. For the numerical problems they considered, their lower bound was between 20% and 175% higher than the typical lower bound used in cyclic scheduling studies.

Several authors have incorporated lot-sizing decisions into cyclic scheduling problems and examined various objectives. Claver and Jackson [13] incorporate lot sizing into the basic model and solve a profit maximization problem. Pinto and Grossmann [48] consider profit maximization for flow shops, where setup times are sequence dependent. Dobson and Yano [19] examine the problem of finding the cyclic schedule and cycle length that jointly minimize that average holding costs for raw materials, WIP, and finished goods in a batch flow shop with setup times. Heuristics are presented for finding non-wrapping permutation schedules, and these are shown to perform well empirically for this class of problem. Campbell [8] considers the problem of determining a fixed ordering frequency for each item type in an assembly schedule where demands vary over time. The objective is minimization of setup and holding costs, and capacity is not considered.

Whybark [59] describes the successful implementation of a production planning and control system based on periodic control. Hall [27] makes qualitative arguments for the use of cyclic scheduling for improvement and synchronization in the production environment. Maxwell, Muckstadt, Jackson, and Roundy [42] give quantitative examples to illustrate how product and process design can affect scheduling in a repetitive manufacturing environment.

Serafini and Ukovich outline a general cyclic scheduling problem in which the cycle length is fixed and pairs of operations are subject to span constraints [54]. A span constraint requires that the difference between the cyclic start times of two operations fall into a particular cyclic time window, or span. Even in the absence of resource constraints, determining whether a feasible cyclic schedule exists is NP-complete. The authors outline an implicit enumeration
scheme. Dauscha, Modrow, and Neumann had previously considered a similar model with applications in traffic control [17].

Another popular application of cyclic scheduling with time window constraints is the problem of scheduling material handling hoists. The usual objective is minimization of the cycle length. Lei and Wang [40], Lei [39], Crama and van de Klundert [15], Lim [41], Ng and Leung [47], and Varnier et al. [58] examine variations of this problem, among others.

A number of authors have considered a cyclic scheduling model that is particularly well-suited for parallel processing applications. In its most basic form, this model has no resource constraints, the precedence constraints between the operations of a job (i.e., vector loop) can be defined across different executions of the job, and the solution need not be periodic (although many authors have limited themselves to studying periodic solutions). The goal is to find a cyclic schedule with minimum cycle length. Chretienne [11] [12] and Cohen et al. [14] consider this problem.

In the presence of various types of resource constraints (identical processors, dedicated processors, etc.), Hanen and Munier show that the loop scheduling problem is NP-hard and outline some special cases which are solvable in polynomial time [45] [28] [29] [44]. Other work on optimal loop scheduling for use in parallelizing compilers can be found in Gasperoni, Schwiegelshohn and Turek [22] [23], Aiken and Nicolau [3], Munshi and Simons [46], Eisenbeis [20], and Cytron [16], to name a few.

Chaar and Davidson [10] use reservation tables to schedule identical, no-wait jobs such that the minimum average latency (maximum average number of job initiations) is realized. Similar ideas are considered by Eisenbeis [20] in developing algorithms for simple pipeline loop scheduling in horizontally microcoded machines.

De Werra and Solot [18] propose an edge coloring model for a special type of no-wait open shop problem in which every machine can have at most one idle interval. They characterize a certain type of graph for which interval cyclic edge colorings (corresponding to feasible cyclic schedules) always exist.

Timed Petri nets have also been a popular analytical tool for studying the behavior of cyclic systems. See, for instance, Hillion and Proth [30], Chretienne [11], and Ramchandani [50].

There are a variety of applications of cyclic scheduling for real-time systems, and the literature is extensive. We mention a few works to suggest the nature of application. Ra-

Finally, we note that a number of authors have investigated stochastic cyclic scheduling models to gain an understanding of their sensitivity to variation in processing times or resource availability: Bowman and Muckstadt [5], Rao [51], Rao and Jackson [52], Lee and Seo [38], and Zhang and Graves [63].

The remainder of this paper is organized as follows. Section 3 develops the problem statement. Section 4 describes a collection of alternative heuristics for generating cyclic schedules using both non-cyclic and cyclic techniques. Section 5 describes how these alternative construction heuristics will be evaluated. Section 6 describes the empirical tests performed and summarizes the results. Section 7 offers conclusions from this study.

3 Problem Formulation

For a fixed cycle length, C, our task is to construct a schedule for executing a given set of multi-stage jobs that repeats itself every C units of time. Since fixing the cycle length is equivalent to fixing the production rate for this problem, our goal is a secondary objective of minimizing a weighted average of the flow times of the individual jobs. By Little’s Law, this is equivalent to minimizing a weighted average of the WIP inventory in the system.

This section is organized into five parts. In the first subsection, we develop the notation to describe the data of the problem. In the next subsection, we follow Roundy’s development in defining a cyclic schedule. Next, we define the objective of weighted flow time and relate it to makespan and inventory measurement. Then, we formally define the weighted flow time cyclic scheduling problem. Finally, we formulate the lower bounding problem and note that it can be solved as a maximum cost network flow problem.
3.1 Machines, Jobs, and Job Routings

The facility consists of a set of machines, \( M \). Let \( J \) denote the set of job types performed on these machines. For each job type \( j \in J \), let \( O_j \) denote the set of operations that must be performed for job type \( j \). Denote by \( O \) the set of all operations: \( O = \bigcup_{j \in J} O_j \).

For each operation \( i \in O \), let \( m_i \in M \) denote the unique machine capable of processing operation \( i \), and let \( p_i \) denote the positive, integral processing time of the operation. For each machine \( m \in M \), let \( O^M_m \) denote the set of operations processed on machine \( m \); that is, \( O^M_m = \{ i \in O \mid m_i = m \} \).

For each job type \( j \in J \), let \( A_j \) denote the job routing for job type \( j \), defined as the set of precedence relations on the operations in \( O_j \). That is, if \( i, i' \in O_j \) and \( (i, i') \in A_j \), then operation \( i \) must be performed before operation \( i' \). We require that the directed graph \( (O_j, A_j) \) be connected and acyclic. For \( i \in O_j \), let \( S_i = \{ i' \in O_j : (i, i') \in A_j \} \) denote the set of immediate job successors of operation \( i \), and let \( P_i = \{ i' \in O_j : i \in S_i \} \) denote the set of immediate job predecessors of operation \( i \). Let \( U_i \) denote the set of all job successors of operation \( i : U_i = \bigcup_{i' \in S_i} (\{ i' \} \cup U_{i'}) \). Associated with each pair (\( i, i' \)) in \( A_j \) is an integer, \( d_{ii'} \), representing the required delay (possibly negative) between the end of operation \( i \) and the beginning of operation \( i' \).

Let \( A = \bigcup_{j \in J} A_j \) denote the set of precedence relations from all job routings. Note that requiring the directed graph induced by \( A \) to be acyclic does exclude some problems of interest. For example, cyclic scheduling problems that require synchronization of operations (i.e., time window constraints) can be modeled in our framework by using appropriately defined delays on the arcs. However, such instances are not considered in this paper because the synchronization induces cycles in the underlying directed graph.

3.2 Cyclic Schedules

Our definition of cyclic schedules follows Roundy [53]. Let \( C \in \mathbb{Z}^+ \) denote the cycle length of interest. (Throughout, \( \mathbb{Z}^+ \) denotes the positive integers and \( \mathbb{Z}_0^+ \) the nonnegative integers.) For feasibility, we assume \( C \geq \sum_{i \in O} p_i \) for all \( m \in M \). For each \( i \in O \), the cyclic schedule time for operation \( i \) can be uniquely represented by a pair \( (T_i, D_i) \in \{0, 1, 2, \ldots, C-1\} \times \mathbb{Z} \), where \( T_i \) denotes the start time of the operation on its designated machine, modulo \( C \), and \( D_i \) represents the integer cycle offset of this operation relative to other operations in the
same job. The cyclic schedule \((T, D)\) is feasible if and only if, for all \(i \in O\), and all \(i' \in S_i\),

\[
T_{i'} \geq T_i + (D_i - D_{i'})C + p_i + d_{i'i'},
\]

and, for all machines \(m \in M\),

\[
i, i' \in O_m^M \text{ and } T_{i'} \geq T_i \implies T_{i'} \geq T_i + p_i,
\]

and

\[
\min_{i \in O_m^M} T_i + C \geq \max_{i \in O_m^M} (T_i + p_i).
\]

Constraints (3.1) ensure that the precedence relations established by \(A\) are not violated; constraints (3.2) and (3.3) ensure that no two operations overlap in time on the same machine. Figure 1 gives an example of a problem instance and a feasible cyclic schedule.

### 3.3 Weighted Average Flow Time

We are interested in finding feasible cyclic schedules that minimize some measure of flow time for a given cycle length, \(C\). Flow time for a serial job routing is straightforward to compute. Let \(N_j = |O_j^D|\), \(O_j^f = \{i_1, i_2, \ldots, i_{N_j}\}\), and \(A_j = \{(i_h, i_{h+1}) : h = 1, 2, \ldots, N_j - 1\}\) describe the serial routing for job type \(j\). The flow time of job type \(j\) in a feasible cyclic schedule is the total time that elapses between the start time of the first operation and the finish time of the last operation:

\[
(T_{i_{N_j}} + D_{i_{N_j}}C + p_{i_{N_j}}) - (T_{i_1} - D_{i_1}C) = T_{i_{N_j}} - T_{i_1} + p_{i_{N_j}} + (D_{i_{N_j}} - D_{i_1})C.
\]

For a job type \(j\) that is not necessarily a serial routing, let

\[
I_j = \{i \in O_j^D : \{i' \in O_j^f : i \in S_{i'}\} = \emptyset\}
\]

be the set of initial operations for the job, and let

\[
F_j = \{i \in O_j^f : \{i' \in O_j^f : i' \in S_i\} = \emptyset\}
\]

be the set of final operations for the job. Since the graph induced by \(A_j\) is acyclic, \(|I_j| \geq 1\) and \(|F_j| \geq 1\). Given a cyclic schedule \((T, D)\) the weighted average flow time of job \(j\) is defined as:

\[
WFT_j(T, D) = \sum_{i \in I_j} \sum_{f \in F_j} w_{if} (T_f - T_i + p_f + (D_f - D_i)C),
\]

9
Figure 1: Feasible Cyclic Schedule Example
and the weighted average flow time of the schedule \((T, D)\) is

\[
WFT(T, D) \equiv \sum_{j \in J} WFT_j(T, D).
\]

(3.5)

We assume the weights \(w_{ij}\) are nonnegative integers and that \(w_{ij} = 0\) if there is no directed path from \(i\) to \(f\). (If the weights are rationals then the problem can be scaled to have integral weights.) Note that this objective weights every maximal path in the routing, not the makespan of the job.

3.4 Problem Statement

The weighted average flow time reduction cyclic scheduling problem (WFTCSP) is the problem of choosing \((T, D) = \{(T_i, D_i) \in \{0, 1, 2, \ldots, C - 1\} \times \mathbb{Z} : i \in O\}\) to minimize \(WFT(T, D)\) subject to (3.1), (3.2), and (3.3). The data of the problem are \(M, J, C, O, \{(p_i, m_i) : i \in O\},\)
\[
\{O_j^f, A_j\} : j \in J,\}
\{d_{w'} : (i, i') \in A\},\] and \\(
\{w_{ij} : (i, f) \in \bigcup_{j \in J} I_j \times F_j\}\). Other equivalent formulations of this problem are possible. See Caggiano [6] for a mixed-integer linear programming formulation and a disjunctive graph interpretation.

3.5 Lower Bound

A lower bound on the \(WFT\) objective is obtained easily by relaxing the machine sequence constraints (3.2) and (3.3), thereby allowing operations on the same machine to overlap in time.

The weighted average flow time lower bound problem (WFTLBP) is the problem of choosing \((T, D) = \{(T_i, D_i) \in \{0, 1, 2, \ldots, C - 1\} \times \mathbb{Z} : i \in O\}\) to minimize \(WFT(T, D)\) subject to (3.1). This problem is decomposable by job. Let \(WFT_j\) denote the minimum value of the weighted flow time for the WFTLBP defined on routing graph \((O_j^f, A_j)\), for job \(j \in J\). Then, \(WFT = \Sigma_{j \in J} WFT_j\) is a lower bound for the objective value of WFTCSP.

Sometimes, a tighter lower bound on the \(WFT\) objective can be obtained by relaxing the machine sequence constraints for all but one of the machines and by solving the resulting single-machine cyclic sequencing problem using branch and bound. This approach is described in Huh et al. [32] and is easily extended to the general problem considered in this paper. In spite of the success reported in [32], we were unable to find instances in our test set for which this approach led to a lower bound that was significantly higher than \(WFT\).
Furthermore, the computation time of the branch and bound procedure was prohibitively high for the size of problems considered in this paper. Accordingly, we report only $WFT$ for the problem instances considered and leave the search for better lower bounds to future research.

4 Construction Heuristics

There are a wide variety of construction heuristics that may be used to generate cyclic schedules. A straightforward approach is to use traditional job-shop scheduling heuristics in a non-cyclic setting (i.e., $C = \infty$) and convert the resulting schedule into a feasible cyclic schedule. A second approach is to develop a heuristic that naturally yields a feasible cyclic schedule. In this paper, we investigate both of these approaches and compare the resulting schedules. In section 4.1, we outline several makespan heuristics and job sequencing heuristics typically used in a non-cyclic setting, and in 4.2 we describe how the resulting non-cyclic schedules are converted into feasible cyclic schedules. Finally, in 4.3 we develop a general framework for implementing a cyclic version of priority dispatch rules. In describing the implementation of these heuristics, we will reference numerous scheduling subroutines that perform specific tasks. These functions are summarized in Table 1 in the order in which they are discussed.

4.1 Non-cyclic Heuristics

4.1.1 Priority Dispatch Rules

Priority dispatch rules for scheduling problems are described in detail in [4]. We review some relevant definitions here. Given a feasible non-cyclic schedule, a local left shift is a scheduling change in which an operation is moved earlier in time without violating any job precedence constraints and without changing the sequence of operations on its designated machine. A global left shift is a scheduling change in which an operation is moved earlier in time without violating any job precedence constraints and without causing any operation overlap on its designated machine (the sequence of operations, however, may be changed). A semiactive schedule is one in which no local left shifts are possible. An active schedule is one in which no global left shifts are possible.
<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnforceMachineSequence()</td>
<td>Takes a set of cyclic start times for a particular machine, and beginning from a specified operation shifts operation start times successively to the right (i.e., later), modulo the cycle length, just enough to enforce the machine sequence constraints for that machine.</td>
</tr>
<tr>
<td>WrapAndSmooth()</td>
<td>Takes an arbitrary set of operation start times, expresses them modulo the cycle length, and smoothes out the operation overlaps through calls to EnforceMachineSequence()</td>
</tr>
<tr>
<td>CyclicActiveSchedule()</td>
<td>Takes as input a non-cyclic schedule and makes it cyclically active.</td>
</tr>
<tr>
<td>Priority()</td>
<td>Returns the priority value of an operation execution using a specified criterion, given a list of recently scheduled operations.</td>
</tr>
<tr>
<td>SimulatePDR()</td>
<td>Implements the basic simulation step of a priority dispatch algorithm.</td>
</tr>
<tr>
<td>CyclicPDR()</td>
<td>Implements a cycle-counting, cycle-fitting, and anti-aging priority dispatch rule to generate a feasible cyclic schedule.</td>
</tr>
<tr>
<td>Evaluate()</td>
<td>Begins with an arbitrary schedule of start times, and returns a feasible, possibly improved, cyclic schedule and the corresponding value of the weighted flow time objective.</td>
</tr>
<tr>
<td>CoarseCompression()</td>
<td>Improves the weighted flow time of a feasible cyclic schedule based on the concept of compression. Coarse compression does not undo compression steps and uses quick techniques to select constraints for compression.</td>
</tr>
<tr>
<td>FineCompression()</td>
<td>Improves the weighted flow time of a feasible cyclic schedule based on the concept of compression. Fine compression will undo compression steps that result in degraded schedules. It uses conservative techniques to select constraints for compression.</td>
</tr>
</tbody>
</table>

Table 1: Scheduling Function Summary
In our computational studies, we implemented two priority dispatch rules (PDR-A and PDR-SA) and tested each with three different selection criteria (Most Work Remaining, Least Work Remaining, and Shortest Processing Time) for a total of six different priority dispatch construction heuristics. The rule PDR-A maintains an active schedule throughout the dispatch process, and the rule PDR-SA maintains a semiactive schedule.

4.1.2 The Shifting Bottleneck Procedure

The shifting bottleneck procedure was developed by Adams, Balas, and Zawack [1] as a heuristic solution technique for the job shop problem. It is an iterative scheme in which machines are sequenced one by one. In each iteration, every machine which has not yet been sequenced gives rise to a one-machine subproblem. Each of these subproblems is solved using an algorithm of Carlier [9], and the machine whose induced one-machine subproblem has the longest makespan (i.e., the bottleneck machine) is selected to be sequenced. The sequence used is the one found in the corresponding one-machine subproblem. Each time a machine is sequenced, a local reoptimization scheme is used to resequence each previously sequenced machine for which improvement is possible. Our implementation of the shifting bottleneck procedure differs from the original version in two ways:

- We omit the local reoptimization scheme. Solving the required $O(|M|^2)$ subproblems is already quite expensive in terms of running time. Given that the cyclicity constraints are being ignored during the construction process, the additional effort of reoptimization is not likely to be justified.

- Instead of using Carlier's branch and bound algorithm to solve the one-machine subproblems, we apply the Earliest Due Date (EDD) rule of Jackson [33]. As with local reoptimization, the computational effort of branch and bound during the construction process is not likely to be justified. The EDD rule is fast, easy to implement, and the makespan value resulting from the EDD sequence is guaranteed to be no more than twice the optimal makespan value. We use the EDD rule within the shifting bottleneck framework to both rank and sequence machines.
4.1.3 Job Sequencing Heuristics

Instead of scheduling operations individually, job sequencing heuristics schedule all of the operations of a job as a group. Each job is first scheduled using a priority dispatch rule as though it were the only job in the facility. The jobs are then added, one at a time, to the end of a joint schedule as early as possible without violating machine sequence constraints. Figure 2 illustrates the result of scheduling five serial jobs, each with five operations, on five machines using this approach. All inter-operation delays are zero in this example. The vertical lines depict the precedence constraints. Observe that while this approach minimizes flow time in the non-cyclic schedule, it can result in idle time between operations on each machine, including the bottleneck machine (i.e., the machine with the largest workload). One variation of the job sequencing heuristic sequences jobs in descending order of weighted flow time lower bound, $WFT_j$, and another variation uses the ascending order. In Figure 2, the jobs have been scheduled in descending order.

![Job Sequencing Heuristic Example](image)

Figure 2: Job Sequencing Heuristic Example

4.2 Cyclic Feasibility

The start times $T = \{T_i : i \in O\}$ are said to be cyclic start times if $T_i \in \{0, 1, 2, \ldots, C - 1\}$, $\forall i \in O$. If, in addition, these times satisfy (3.2) and (3.3), they are said to be cyclically feasible. As we show in the companion paper [7], given a vector, $T$, of cyclically feasible start times, an optimal vector of cycle offsets $D = \{D_i : i \in O\}$ can always be found such that $(T, D)$ is a feasible cyclic schedule. Let $\mathbf{T}$ denote the set of all cyclically feasible start time vectors, and note that $\mathbf{T} \equiv \times_{m \in M} \mathbf{T}^m$, where $\mathbf{T}^m$ denotes the set of cyclically feasible start time vectors for machine $m$. 
The schedules created by the non-cyclic construction heuristics described in the previous section must be converted into feasible cyclic schedules before they can be evaluated. We use two functions to accomplish this conversion. Function \texttt{EnforceMachineSequence()} takes a vector of cyclic start times for a particular machine, \( m \), and makes these times cyclically feasible by "smoothing" the times successively to the right (i.e. later), modulo \( \mathcal{C} \), just enough to remove any operation overlap on the machine. (The operation from which the smoothing starts is a function parameter.) Function \texttt{WrapAndSmooth()} takes a vector of arbitrary start times, \( T \in \mathbb{Z}^{\mathcal{O}} \), wraps them around a cylinder of circumference \( \mathcal{C} \), and smoothes out all operation overlaps by calling \texttt{EnforceMachineSequence()} for each machine \( m \in M \). Hence, \texttt{WrapAndSmooth()} can be used to convert any set of start times into a set of cyclically feasible start times. Observe that if \( T \in \mathcal{T} \), then function \texttt{WrapAndSmooth()} has no effect (i.e., \( T = \texttt{WrapAndSmooth}(C, T) \)). Both of these functions are described in greater detail in the companion paper [7].

4.2.1 Cyclic Global Left Shifts

Because of the smoothing performed by the \texttt{EnforceMachineSchedule()} function, \texttt{WrapAndSmooth()} can destroy much of the structure of a non-cyclic schedule. To help preserve some of this structure, we preprocess each non-cyclic schedule using the function \texttt{CyclicActiveSchedule()} before calling \texttt{WrapAndSmooth}(). The function \texttt{CyclicActiveSchedule()} takes as input a non-cyclic schedule \( \tau \) and repeatedly makes cyclic global left shifts until no more can be made.

Like its non-cyclic counterpart, a \textit{cyclic global left shift} is a scheduling change in which an operation is moved earlier in time without violating any job precedence constraints and without causing any operation overlap on its designated machine. The difference for a cyclic global left shift is that when computing the earliest possible start time for an operation relative to its immediate job predecessor operations, we make this computation modulo \( \mathcal{C} \). Figure 3 illustrates the result of performing a cyclic global left shift on operation 4 of a serial job with precedence structure \( 1 \to 2 \to 3 \to 4 \to 5 \). Repeated application of such shifts makes it more likely that the resulting schedule will fit within a given required cycle length. A schedule is called \textit{cyclically active} if no cyclic global left shift can be made. Hence, the function \texttt{CyclicActiveSchedule()} makes a schedule cyclically active. Observe that the order in
which operations are shifted makes a difference in the resulting schedule. Thus, given a schedule resulting from a non-cyclic construction heuristic, we apply `CyclicActiveSchedule()` several times, each time examining jobs in a different order. Within each job, we apply cyclic global left shifts to operations in a topological order. Each time `CyclicActiveSchedule()` terminates, the resulting schedule is submitted to `WrapAndSmooth()` and evaluated. The best schedule found is kept for further improvement.

![Diagram](image)

(a) Schedule before cyclic global left shift of operation 4 with \( C = 8 \)

\[ T_3 + p_3 = 8, \quad T_4 = 9 \]

![Diagram](image)

(b) Schedule after cyclic global left shift of operation 4 with \( C = 8 \)

\[ T_4 = (T_3 + p_3) \mod 8 = 0 \]

Figure 3: Cyclic Global Left Shift Example
4.3 Cyclic Priority Dispatch Rules

The apparent disadvantage of cyclic schedule construction heuristics based on non-cyclic approaches is that WrapAndSmooth() can destroy much of the short flow time structure of the non-cyclic schedule. Calling CyclicActiveSchedule() first helps, but construction heuristics that are explicitly cyclic in nature may enjoy an advantage. In this section, we extend the concept of priority dispatch rules to develop a technique that can generate a feasible cyclic schedule. The basic idea is to simulate forward from time 0 with all machines idle initially, release a new set of jobs every $C$ units of time, and use a priority dispatch rule to schedule the resulting queues of operations at each machine. The algorithm terminates when a feasible cyclic schedule has been identified. We impose a number of restrictions on the dispatch rule to ensure that the algorithm will find such a schedule within a finite number of steps.

In subsection 4.3.1, we develop the notation and terminology needed to describe the algorithm. In subsection 4.3.2, we describe the properties of a dispatch rule that are sufficient for finite convergence of the algorithm. In subsection 4.3.3, we present the algorithm and termination conditions. Finally, in subsection 4.3.4, we prove that the dispatch rule properties are sufficient to ensure that the algorithm will terminate in a finite number of steps. Throughout this section, we assume that all delays are nonnegative: $d_{ii'} \geq 0$ for all $(i,i') \in A$.

4.3.1 Cyclic PDR Framework

Within our cyclic dispatch rule framework, time $nC$ denotes the start of cycle $n$, where $n \in Z_0^+$. At the start of each cycle, a new set of jobs is released. The copy of operation $i$ that is released at the beginning of cycle $n$ is referred to as operation execution $(i,n)$. Let $\Lambda \subseteq O \times Z_0^+$ denote the set of operation executions that have been released but not scheduled, and let $\Lambda_i \equiv \{(i',n) \in \Lambda : i' = i\}$ be the queue of unscheduled operation executions for operation $i, i \in O$. Let $\Phi \subseteq O \times Z_0^+$ denote the set of operation executions, $(i,n)$, that have been scheduled, and let $\tau \equiv \{\tau(i,n) \geq 0 : (i,n) \in \Phi\}$ denote the set of start times of scheduled operation executions. In scheduling operations, we will ensure that the start times of operations are machine sequence feasible. That is, for all $(i,n), (i',n') \in \Phi$
such that \( m_i = m_{i'} \),

\[
\text{either } \begin{cases} 
\tau(i', n') \geq \tau(i, n) + p_i; \text{ or} \\
\tau(i, n) \geq \tau(i', n') + p_{i'}.
\end{cases}
\] (4.6)

An operation execution is said to be eligible if all of its immediate predecessor operation executions have been scheduled to start. Let \( \Theta \equiv \{(i, n) \in \Lambda : P_i \times \{n\} \subset \Phi\} \) be the queue of eligible operation executions, and let \( \Theta_i \equiv \{(i', n) \in \Theta : i' = i\} \) be the queue of eligible operation executions for operation \( i, i \in O \).

An operation execution is said to be schedulable if all of its immediate predecessor operation executions have completed and all requisite inter-operation delays have been satisfied. That is, operation execution \( (i, n) \) becomes schedulable at time \( t \) if

\[
t = \max_{i' \in P_i} (\tau(i', n) + p_{i'} + d_{i'i}).
\]

Clearly, in any feasible schedule, an operation cannot start until it becomes schedulable.

Our simulation approach will be to release a new set of jobs every \( C \) units of time. Let \( r \in \mathbb{Z}_0^+ \) denote the current release cycle and let \( t \in \{rC, rC + 1, \ldots, (r + 1)C - 1\} \) denote the current simulation time within the current release cycle. For clarity, the dependence of \( \Omega, \Phi, \Theta, \) and \( \tau \) upon \( t \) will be suppressed from the notation except where needed.

For an eligible operation execution \( (i, n) \in \Theta \), define the eligible start time of \( (i, n) \) with respect to time \( t \) to be:

\[
\tilde{\tau}(i, n, t) \equiv \max \left\{ t, \max_{i' \in P_i} (\tau(i', n) + p_{i'} + d_{i'i}), \max_{m_{i'} = m_i} (\tau(i', n') + p_{i'}) \right\}.
\] (4.7)

Note that the definition of eligible start time guarantees that an operation execution will be schedulable at this time. If operation execution \( (i, n) \) is chosen at simulation time \( t \) to be scheduled, it will be scheduled to start at time \( \tilde{\tau}(i, n, t) \). Assuming the integrality of processing times and delays, \( \tau \) computed in this way will always be integral, job sequence feasible, and machine sequence feasible.

Throughout the simulation, we will keep track of the operations that have been scheduled recently; that is, within the last \( C \) units of time. Let \( \Phi(t) \equiv \bigcup_{m \in M} \Phi_m(t) \), where

\[
\Phi_m(t) \equiv \{(i, n) \in \Phi : t > \tau(i, n) \geq t - C\}
\]
denotes the operation executions scheduled to start on machine \( m \) during the interval \([t - C, t)\). Let \( \Pi_m(t) \) denote the projection of \( \Phi_m(t) \) onto \( O^M_m \); that is,

\[
\Pi_m(t) \equiv \left\{ i \in O^M_m : \exists n \in \mathbb{Z}_0^+ \text{ s.t. } (i, n) \in \Phi_m(t) \right\}.
\]
Thus, $\Pi_m(t)$ is a list of the operations scheduled to start on machine $m$ within the last $C$ units of time (relative to the current simulation time, $t$). Let $\Pi(t) \equiv \bigcup_{m \in M} \Pi_m(t)$. Intuitively, we want to construct a scheduling algorithm so that eligible operation executions for operations that are not in $\Pi(t)$ will receive higher priority for scheduling at time $t$ than for operations that are in $\Pi(t)$. This will ensure that no queue, $\Theta_i$, will be allowed to grow indefinitely.

We will also track which operations have been scheduled within the current release cycle. Let $\Gamma^r_m$ denote the set of operations that have been scheduled to start on machine $m$ in the current release cycle $r$:

$$\Gamma^r_m \equiv \left\{ i \in O^M_m : \exists n \in Z^*_0 \text{ s.t. } rC \leq \tau(i, n) < (r + 1)C \right\}.$$  

The dependence of $\Gamma^r_m$ upon $\tau$ and hence upon $t$ has been suppressed from the notation. Note that at the end of the current release cycle (i.e., when $t = (r + 1)C$), $\Gamma^r_m$ will equal $\Pi_m((r + 1)C)$, but that during the current release cycle (i.e., $rC \leq t < (r + 1)C$), $\Gamma^r_m$ will be a subset of $\Pi_m(t)$. Let $\Gamma^r \equiv \bigcup_{m \in M} \Gamma^r_m$ denote the set of all operations that have been scheduled within the current release cycle.

We will need to determine if the operations scheduled in the current release cycle result in a feasible cyclic schedule. Let $T^r$ denote the set of (latest) operation execution start times for each operation type scheduled within release cycle $r$:

$$T^r \equiv \left\{ T^r_i = \max_{n(i, n) \in \Phi} \tau(i, n) : i \in \Gamma^r \right\}.$$  

Considering only the most recently scheduled executions of each operation in the current release cycle, let $t^r_m$ denote the earliest operation start time on machine $m$. That is,

$$t^r_m \equiv \begin{cases} (r + 1)C, & \text{if } \Gamma^r_m = \emptyset; \\ \min_{i \in \Gamma^r_m} T^r_i, & \text{otherwise.} \end{cases}$$  

We refer to $t^r_m$ as the release cycle start time for machine $m$ in cycle $r$. As the next lemma states, if all operations are scheduled to start in the current release cycle and if all of these operations will finish within one cycle length after the release cycle start time for each machine, then $T^r$ gives rise to a set of feasible cyclic start times.

**Lemma 1** If $\Gamma^r = O$, and $T^r_i + p_i \leq t^r_m + C$, $\forall i \in O$, then $T^r \mod C \in T$.

**Proof.** Since the start times scheduled by the simulation will be integral and will satisfy (4.6), the additional condition ensures that $T^r \mod C$ satisfies (3.2) and (3.3). ■
Finally, an operation which is scheduled to start in the previous cycle but does not complete until this cycle is said to be cycle-spanning in this cycle. Let $S^r$ denote the set of all cycle-spanning operations in cycle $r$:

$$S^r \equiv \{ i \in O : \exists (i,n) \in \Phi \text{ s.t. } \tau(i,n) < rC < \tau(i,n) + p_i \}.$$  

For simulation time $t$, let $S(t) = S^r$ where $r = \left\lfloor \frac{t}{C} \right\rfloor$. Let $s^r_m$ denote the earliest feasible start time on machine $m$ in cycle $r$:

$$s^r_m = \max \left\{ rC, \max_{(i,n) \in \Phi_m \cap \{rC\}} (\tau(i,n) + p_i) \right\}.$$  

Observe that $s^r_m > rC$ if and only if an operation on machine $m$ is cycle-spanning in cycle $r$. Also, by definition, release cycle start times are at least as large as earliest feasible start times:

$$t^r_m \geq s^r_m.$$  

The notation developed for the implementation of the cyclic dispatch rule algorithm is summarized in Table 2.

### 4.3.2 Cyclic Dispatch Rule Properties

In this subsection, we describe three properties of a dispatch rule that will ensure that a feasible cyclic schedule will be found in a finite simulation run. We refer to these properties as cycle-fitting, cycle-counting, and anti-aging.

- **Cycle-fitting**: No operation will be scheduled to begin on machine $m$ in the current release cycle, $r$, unless it can complete before time $t^r_m + C$.

Observe that if the dispatch rule is cycle-fitting, then by Lemma 1, a sufficient condition for the start times of the current cycle to give rise to a set of feasible cyclic start times is simply $\Gamma^r = O$. That is, a feasible cyclic schedule can be identified as soon as a cycle is simulated in which each operation type has been scheduled at least once.

- **Cycle-counting**: At most one execution of any operation type $i$ may be scheduled to begin in a release cycle.
<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Lambda$</td>
<td>The set of released operation executions that have not yet been scheduled.</td>
</tr>
<tr>
<td>$\Phi$</td>
<td>The set of scheduled operation executions.</td>
</tr>
<tr>
<td>$\tau$</td>
<td>The set of start times of scheduled operation executions.</td>
</tr>
<tr>
<td>$\Theta$</td>
<td>The set of eligible operation executions.</td>
</tr>
<tr>
<td>$\tilde{\tau}(i, n, t)$</td>
<td>The eligible start time of operation execution $(i, n)$ with respect to simulation time $t$.</td>
</tr>
<tr>
<td>$\Phi_m(t)$</td>
<td>The set of operation executions scheduled to start on machine $m$ during the interval $[t - C, t)$.</td>
</tr>
<tr>
<td>$\Pi_m(t)$</td>
<td>The set of operations scheduled to start on machine $m$ during the interval $[t - C, t)$. That is, the projection of $\Phi_m(t)$ onto $C_m^M$.</td>
</tr>
<tr>
<td>$\Gamma_m^r$</td>
<td>The set of operations scheduled to start on machine $m$ during the current release cycle $r$.</td>
</tr>
<tr>
<td>$T^r$</td>
<td>The set of latest operation execution start times for each operation type scheduled to start during the current release cycle $r$.</td>
</tr>
<tr>
<td>$t_{r_m}^*$</td>
<td>The earliest start time of any operation execution scheduled on machine $m$ during the current release cycle $r$.</td>
</tr>
<tr>
<td>$\psi$</td>
<td>The selection criterion for the dispatch rule.</td>
</tr>
<tr>
<td>$\Theta(t)$</td>
<td>The set of cyclically-eligible operation executions at simulation time $t$.</td>
</tr>
<tr>
<td>$S^r$</td>
<td>The set of cycle-spanning operations in cycle $r$.</td>
</tr>
<tr>
<td>$s_{r_m}^*$</td>
<td>The earliest feasible start time on machine $m$ in cycle $r$.</td>
</tr>
</tbody>
</table>

Table 2: Cyclic PDR Notation
This restriction ensures that $\tau^r = \tau^f$. Subsequent operations to be scheduled on machine $m$ in the current release cycle must be chosen from $O^m_\tau \setminus \Gamma^r_m$.

The cycle-fitting and cycle-counting restrictions could result in the build-up of queues of unscheduled operation executions. A further restriction is required to ensure that none of these queues grows indefinitely. Let $\Psi$ denote the set of possible selection criteria for a dispatch rule. Let the real-valued function $\text{Priority}(i,n,\Pi(t),S(t),\psi)$ return the priority value of operation execution $(i,n) \in \Theta$ using criterion $\psi \in \Psi$, given that operations in $\Pi(t)$ have been scheduled recently, and the set of cycle-spanning operations for the current cycle is given by $S(t)$.

- **Anti-aging:**

  1. If $n < n'$ then

      \[ \text{Priority}(i,n,\Pi(t),S(t),\psi) > \text{Priority}(i,n',\Pi(t),S(t),\psi); \quad (4.8) \]

  2. If $m_i = m_{i'}, i \not\in \Pi(t)$, and $i' \in \Pi(t)$, then

      \[ \text{Priority}(i,n,\Pi(t),S(t),\psi) > \text{Priority}(i',n',\Pi(t),S(t),\psi), \quad (4.9) \]

      regardless of $n$ and $n'$; and

  3. If $m_i = m_{i'}, i \not\in S(t)$, and $i' \in S(t)$, then

      \[ \text{Priority}(i,n,\Pi(t),S(t),\psi) > \text{Priority}(i',n',\Pi(t),S(t),\psi), \quad (4.10) \]

      for $t \in \{rC, rC + 1, \ldots, T_{r-1}^r + C\}$.

Requirement (4.8) ensures that older executions of the same operation type receive higher priority than more recently released executions. Requirement (4.9) ensures that operation types that have not been scheduled to start recently (i.e., within the past $C$ units of time) receive higher priority than other operation types on the same machine. Requirement (4.10) ensures that cycle-spanning operations receive lower priority than other operation types on the same machine. Observe that $i \in S(t) \Rightarrow i \in \Pi(t)$ for $t \in \{rC, rC + 1, \ldots, T_{r-1}^r + C\}$, so that (4.10) can be viewed as a refinement of (4.9).
Example 1 Suppose that operations $i \in O$ are numbered from 1 to $|O|$. The following priority function $\psi = \text{"NUM"}$, which gives higher priority to higher-numbered operations, is anti-aging:

$$\text{Priority}(i, n, \Pi(t), S(t), \text{"NUM"}) = \left( \frac{i}{|O|} \right) \left( \frac{\max_{n' \in \Theta} \{n' : (i, n') \in \Theta\} - n}{\max_{n' \in \Theta} \{n' : (i, n') \in \Theta\}} \right) + 1_{\{i \in \Pi(t)\}} + 1_{\{i \in S(t)\}}.$$

Example 2 The following priority function $\psi = \text{"SPT"}$, which uses the shortest processing time criterion, is anti-aging:

$$\text{Priority}(i, n, \Pi(t), S(t), \text{"SPT"}) = \left( \frac{\max_{t \in O} P_t - p_i}{\max_{t \in O} P_t} \right) \left( \frac{\max_{n' \in \Theta} \{n' : (i, n') \in \Theta\} - n}{\max_{n' \in \Theta} \{n' : (i, n') \in \Theta\}} \right) + 1_{\{i \in \Pi(t)\}} + 1_{\{i \in S(t)\}}.$$

Note that in this manner, any of the traditional dispatch rule selection criteria ("most work remaining", "least work remaining", "shortest processing time", etc.) may be used to construct a cyclic priority dispatch rule.

4.3.3 The Cyclic Priority Dispatch Rule Algorithm

The simulation algorithm described in this subsection maintains the cycle-counting, cycle-fitting, and anti-aging properties outlined in the previous section. In doing so, it finds a feasible cyclic schedule in a finite number of iterations.

An eligible operation execution $(i, n)$ is said to be cyclically-eligible at time $t$ in the current release cycle $r$ if, in addition to being eligible, it maintains the cycle-counting and cycle-fitting properties of the dispatch rule (i.e., it belongs to $O_{m_i}^t \setminus \Gamma_{m_i}^r$ and it can complete before time $t_{m_i}^r + C$). Let $\tilde{\Theta}(t)$ denote the set of cyclically-eligible operations at time $t$, where $t \in \{rC, rC + 1, \ldots, (r + 1)C - 1\}$. That is:

$$\tilde{\Theta}(t) \equiv \{(i, n) \in \Theta : i \in O_{m_i}^t \setminus \Gamma_{m_i}^r, \hat{r}(i, n, t) \leq \min \{(r + 1)C - 1, t_{m_i}^r + C - p_i\} \}.$$

The function $\text{SimulatePDR()}$, described in Figure 4, implements the basic simulation step of a priority dispatch algorithm. The algorithm identifies the earliest possible start time among the operation executions in $\tilde{\Theta}(t)$. If there are multiple operation executions in $\tilde{\Theta}(t)$ that can start at this earliest time, then the priority rule is used to select an operation execution to schedule. The remaining steps update the various inputs, including $\Gamma^r$ and $t^r$.
**Input:** Cycle length $C \in \mathbb{Z}^+$; dispatch rule $\psi \in \Psi$; operation executions released but unscheduled $\Lambda \subseteq O \times \mathbb{Z}_0^+$; queue of eligible operation executions $\Theta \subseteq \Lambda$; most recent scheduled operations $\Phi \subseteq O \times \mathbb{Z}_0^+$; start times of most recent scheduled operations $\tau \subseteq \times \mathbb{Z}_0^+$; current release cycle $\tau \in \mathbb{Z}_0^+$; current simulation time $t \in \{rC, rC + 1, rC + 2, \ldots, (r + 1)C - 1\}$; cyclically-eligible operation executions $\tilde{\Theta} \subseteq \Theta$; operations scheduled to start in the current release cycle $\{\Gamma_r^m \subseteq O_m^M : m \in M\}$; and release cycle start times by machine $\{t_m^r : m \in M\}$.

**Output:** Updated values of: operation executions released but unscheduled $\Lambda'$; queue of eligible operation executions $\Theta'$; most recent scheduled operations $\Phi'$; start times of most recent scheduled operations $\tau'$; current simulation time $t'$; operations scheduled to start in the current release cycle $\{\Gamma_m' : m \in M\}$; and release cycle start times $\{t_m' : m \in M\}$.

1. Initialize: $(\Lambda'; \Theta'; \Phi'; \tau'; t'; \{\Gamma_m'\}; \{t_m'\}) \leftarrow (\Lambda; \Theta; \Phi; \tau; t; \{\Gamma_m\}; \{t_m\})$

2. If $\tilde{\Theta} \neq \emptyset$ (i.e., if there exists at least one cyclically-eligible operation execution) then:
   (a) Find earliest time to schedule a new operation: $t' \leftarrow \min_{(i,n) \in \tilde{\Theta}} \tilde{\tau}(i,n,t)$;
   (b) Identify operations scheduled in past $C$ time units: $\Pi \leftarrow \bigcup_{m \in M} \Pi_m(t')$;
   (c) Identify operations that are cycle-spanning: $S \leftarrow S(t)$;
   (d) Select highest priority cyclically-eligible operation execution to schedule next:
      $$(i^*, n^*) \leftarrow \arg\max_{(i,n) \in \tilde{\Theta}} \text{Priority}(i,n,\Pi,S,\psi)$$
      $\tilde{\tau}(i^*,n^*,t')$;
   (e) Schedule selected operation execution: $\tau \leftarrow \tau \cup \{\tau(i^*,n^*) = t'\}$;
   (f) If this operation is first on its machine in the current cycle then record the release cycle start time for the affected machine: If $\Gamma_{m,i}^r = \emptyset$ then $t_m' \leftarrow t'$;
   (g) Update sets of scheduled and unscheduled operation executions: $\Phi' \leftarrow \Phi \cup \{(i^*,n^*)\}$; $\Lambda' \leftarrow \Lambda \backslash \{(i^*,n^*)\}$; $\Theta' \leftarrow \Theta \backslash \{(i^*,n^*)\}$;
   (h) Update set of operations scheduled this cycle: $\Gamma_{m,i}^r \leftarrow \Gamma_{m,i}^r \cup \{i^*\}$;
   (i) Expand set of eligible operation executions based on precedence constraints: $\Theta' \leftarrow \Theta' \cup \{(i',n) \in \Lambda : i' \in S_{i^*}; P_r \times \{n\} \subseteq \Phi\}$.

3. Else advance simulation time to end of current release cycle: $t' \leftarrow (r + 1)C$;

4. Return $\Lambda'; \Theta'; \Phi'; \tau'; t'; \{\Gamma_m' : m \in M\}; \{t_m' : m \in M\}$.

Figure 4: Function SimulatePDR
If there are no cyclically-eligible operation executions (i.e., if $\hat{\Theta} = \emptyset$), then the simulation time is advanced to the end of the current release cycle.

At the end of a release cycle, $r$, the simulation terminates if $\Gamma^r = \emptyset$. In that event, by Lemma 1, the set of operation start times, modulo $C$, will be cyclically feasible: $T^r \mod C \in T$. Function \texttt{CyclicPDR()}, described in Figure 5, manages the outer loop of a cycle-counting, cycle-fitting, and anti-aging priority dispatch rule.

### 4.3.4 Finite Termination of the Cyclic Priority Dispatch Rule Algorithm

By Lemma 1, if function \texttt{CyclicPDR()} terminates, it yields a feasible cyclic schedule. In this subsection, we assume that the conditions of cycle-fitting, cycle-counting, and anti-aging hold and show that they are sufficient to ensure termination of the algorithm within a finite number of cycles.

**Lemma 2** Consider the time $s^r_m$, the earliest feasible start time on machine $m$ in the release cycle, $r$. At that time, if there exists $(i, n) \in \Theta_i$ such that $m_i = m$ and $\varphi(i, n, s^r_m) = s^r_m$ then \texttt{CyclicPDR()} will schedule some execution of operation $i$ to start in cycle $r$.

**Proof.** Consider the time $t_i = \min \{(r + 1)C - 1, s^r_m + C - p_i\}$, the latest possible time to schedule operation $i$ within the current cycle under the cycle-fitting restriction. Suppose that no execution of operation $i$ was scheduled to start at any time $t \in \{s^r_m, s^r_m + 1, \ldots, t_i\}$. Then, $(i, n) \in \hat{\Theta}(t)$ for every $t \in \{s^r_m, s^r_m + 1, \ldots, t_i\}$, since at each of these times it maintained the cycle-counting and cycle-fitting properties of the dispatch rule. Thus, since operation $i$ was not scheduled in $\{s^r_m, s^r_m + 1, \ldots, t_i\}$, and since $(i, n)$ was cyclically-eligible at each of these times, the machine $m$ must have been busy from $s^r_m$ until at least $t_i + 1$ with other operations, and $t^r_m = s^r_m$. Let $\Gamma$ denote the set of operations scheduled to start on machine $m$ during $[t^r_m, t_i]$. By the cycle-counting restriction, at most one operation of each type can be scheduled to start during the period $[t^r_m, (r + 1)C - 1]$. Hence $\Gamma \subseteq O^M_m \setminus \{i\}$. It must be the case that $s^r_m + \sum_{\nu \in \Gamma} p_{\nu} > t_i$, else operation $i$ could have been scheduled at time $t_i$. Let $k \in S^r$ denote the cycle-spanning operation on machine $m$ in cycle $r$, if it exists. By the anti-aging requirement (4.10), $k \notin \Gamma$, since operation $k$ cannot be scheduled until after operation $i$ is scheduled. If a cycle-spanning operation $k$ exists and $i \neq k$, then:

$$t_i < s^r_m + \sum_{\nu \in \Gamma} p_{\nu} < rC + p_k + \sum_{\nu \in \Gamma} p_{\nu} \leq rC + \sum_{\nu \in O^M_m \setminus \{i\}} p_{\nu} \leq rC + C - p_i = (r + 1)C - p_i \leq t_i,$$
Input: Cycle length $C \in \mathbb{Z}^+$; and dispatch rule $\psi \in \Psi$.

Output: Feasible cyclic start times $T \in T$.

1. Initialize: $r \leftarrow 0$; $\Gamma^r \leftarrow \emptyset$; $t \leftarrow 0$; $\Phi \leftarrow \emptyset$; $\Lambda \leftarrow \emptyset$; $\Theta \leftarrow \emptyset$;

2. While $\Gamma^r \neq O$ (i.e., while schedule for previous cycle failed to include all operations):
   begin
   (a) Release operation executions for next cycle: $\Lambda \leftarrow \Lambda \cup (O \times \{r\})$; $\Theta \leftarrow \Theta \cup (\bigcup_{j \in J} I_j \times \{r\})$;
   (b) Initialize cycle-counting and cycle-fitting: For $m \in M : \Gamma^r_m \leftarrow \emptyset$; $t^r_m \leftarrow rC$;
   (c) While $t < (r + 1)C$ (i.e., until simulation time advances beyond end of current cycle):
       begin
       i. Restrict attention to cyclically-eligible operation executions (i.e., to cycle-fitting operations that do not already have one execution scheduled in this cycle):
          $$\bar{\Theta}(t) \leftarrow \{(i, n) \in \Theta : i \in O^M_{m_i} \setminus \Gamma^r_m; \overline{\tau}(i, n, t) \leq \min \{(r + 1)C - 1, t^r_m + C - p_i\}\};$$
       ii. Schedule the next eligible operation execution and advance the simulation time:
          $$(\Lambda; \Theta; \Phi; \tau; t; \{\Gamma^r_m\}; \{t^r_m\}) \leftarrow \text{SimulatePDR} \left(\Lambda; \Theta; \Phi; \tau; t; \bar{\Theta}(t); \{\Gamma^r_m\}; \{t^r_m\}\right)$$
       end
   (d) Extract start times of the most recent cycle: $T \leftarrow \{\tau(i, n) : i \in O; \tau(i, n) \geq rC\}$;
   (e) Advance the current cycle indicator: $r \leftarrow r + 1$;
   end

3. Express cyclic start times modulo $C$: For all $i \in O : T_i \leftarrow T_i \mod C$;

4. Return $T$.

Figure 5: Function CyclicPDR
a contradiction. If a cycle-spanning operation $k$ exists and $i = k$, then

$$s_m^r + \sum_{i' \in \Gamma} p_{i'} \leq s_m^r + \sum_{i' \in O_m \setminus \{k\}} p_{i'} \leq s_m^r + C - p_i < rC + p_i + C - p_i = (r + 1)C,$$

and hence:

$$t_i < s_m^r + \sum_{i' \in \Gamma} p_{i'} \leq \min \{(r + 1)C - 1, s_m^r + C - p_i\} = t_i,$$

a contradiction. If there is no cycle-spanning operation, then $t_m^r = s_m^r = rC$, and

$$t_i < s_m^r + \sum_{i' \in \Gamma} p_{i'} \leq rC + \sum_{i' \in O_m \setminus \{i\}} p_{i'} \leq rC + C - p_i = (r + 1)C - p_i \leq t_i,$$

a contradiction. We conclude that some execution of operation $i$ must be scheduled to start at some time $t \in \{s_m^r, s_m^r + 1, \ldots, t_i\}$. ■

The chief implication of Lemma 2 is that all queues of unscheduled operations are bounded and hence the makespan of any job released in the CyclicPDR() simulation is bounded. To show this, we will bound the amount of time a released operation execution $(i, n)$ remains unscheduled using the depth of operation $i$. The depth of an operation $i \in O_j$ is defined to be the length, in number of arcs, of the longest directed path from any initial node in $I_j$ to node $i$ in the graph $(O_j', A_j)$.

**Lemma 3** If $i \in I \equiv \bigcup_{j \in J} I_j$ is an initial operation for some job $j$ (i.e., an operation with depth $d = 0$), then for all cycles $n \geq 0$, CyclicPDR() will schedule operation executions so that $t_m^0 = nC$, and $nC \leq t(i, n) \leq t(i, n) + p_i \leq (n + 1)C$. That is, for initial operation types $i$, $(i, n)$ will be scheduled to start and finish in cycle $n$.

**Proof.** By induction on $n$. At the beginning of cycle $n = 0$, machine $m_i$ is idle, so $s_m^0 = 0$, and $(i, 0) \in \Theta_i$ with $\bar{t}(i, 0, 0) = 0$. Hence, some job will be scheduled on machine $m_i$ at time 0, so $t_m^0 = 0$. Since $(i, 0)$ is the only eligible execution of operation $i$ (and will remain the only one until time $C$), by Lemma 2, it will be scheduled to start by $C - p_i$, and hence will finish by $C$.

Suppose the induction hypothesis is true for all $n \leq n'$, and consider cycle $n' + 1$. Since $t_m^0 = n'C$, by the cycle-fitting property we have that no operation will be scheduled to start on machine $m_i$ in cycle $n'$ unless it can finish by $n'C + C = (n' + 1)C$. Hence, $s_m^{n' + 1} = (n' + 1)C$. Since a new set of jobs is released every cycle, $(i, n' + 1) \in \bar{\Theta}((n' + 1)C)$ with $\bar{t}(i, n' + 1, (n' + 1)C) = (n' + 1)C$, and some operation will be scheduled on machine
$m_i$ at time $(n' + 1)C$. This implies that $t_{m_i}^{(n'+1)} = (n' + 1)C$. By the induction hypothesis, operation execution $(i, n')$ was scheduled to start and finish in cycle $n'$, so at time $(n' + 1)C$, $(i, n' + 1)$ is the only eligible execution of operation $i$, and will remain the only one until time $(n' + 2)C$. By Lemma 2, $(i, n' + 1)$ will be scheduled to start by $(n' + 2)C - p_i$, and hence will finish by $(n' + 2)C$.

Lemma 4 For all operation types $i \in O$ and all release cycles $n \geq 0$, **CyclicPDR()** schedules operation executions such that $(i, n)$ becomes eligible and is scheduled before $(i, n+1)$.

**Proof.** By induction on $d$, the depth of operation $i$. For depth $d = 0$, the lemma is immediately true from Lemma 3. Suppose that the lemma is true for all operations having depth $d$ or less, and consider an operation $i$ of depth $d + 1$. For any $n \geq 0$, $(i, n)$ becomes eligible at time $t^* = \max_{i' \in P_i} \tau(i', n)$. By definition, all predecessors $i' \in P_i$ have depth at most $d$. Thus, by the induction hypothesis, we have that $\tau(i', n) < \tau(i', n + 1)$ for all $i' \in P_i$, and hence $\max_{i' \in P_i} \tau(i', n) < \max_{i' \in P_i} \tau(i', n + 1)$. This establishes that $(i, n)$ becomes eligible before $(i, n + 1)$. Moreover, for all times $t \geq t^*$ at which both $(i, n)$ and $(i, n + 1)$ are eligible and unscheduled, $(i, n + 1) \in \tilde{\Omega}(t) \Rightarrow (i, n) \in \tilde{\Omega}(t)$, and $\tilde{\tau}(i, n, t) \leq \tilde{\tau}(i, n + 1, t)$. Hence, by the anti-aging property (4.8), we will have $\tau(i, n) < \tau(i, n + 1)$.

We are now ready to bound the amount of time any released operation remains unscheduled. Let $\bar{q}_i = \max_{i' \in P_i} [(p_{i'} + d_{i'}) / C]$, and note that this is an upper bound on the number of cycles required for operation execution $(i, n)$ to become schedulable once its predecessors $(i', n), i' \in P_i$, have been scheduled to start. Recall that the delays $d_{i'}$ may be arbitrarily long. Let $\bar{Q} = \max_{i \in O} \bar{q}_i$ be the upper bound over all operation types.

Lemma 5 For all operations $i \in O$ and all release cycles $n \geq 0$, **CyclicPDR()** will schedule operation execution $(i, n)$ to start before or during cycle $n + d(\bar{Q} + 2)$, where $d$ denotes the depth of operation $i$.

**Proof.** The proof is by induction on $d$. For $d = 0$, the truth of the lemma is immediate from Lemma 3.

Suppose that the lemma is true for all operations having depth $d$ or less, and consider an operation $i$ of depth $d + 1$. We will show that the lemma is true for all $(i, n), n \geq 0$, using a nested induction on $n$.  

29
First, we consider operation execution \((i, 0)\). Since all predecessor operations \(i' \in P_i\) have depth \(d\) or less, the induction hypothesis implies that operation executions \((i', 0), i' \in P_i\), will all be scheduled to start before or during cycle \(d(Q + 2)\). Hence, \((i, 0)\) will be schedulable before or during cycle \(d(Q + 2) + Q\), and (if not already scheduled) \(\tau(i, 0, s_{mi}^{d(Q+2)+Q+1}) = s_{mi}^{d(Q+2)+Q+1}\). By Lemma 2, an execution of operation \(i\) will be scheduled to start in cycle \(d(Q + 2) + Q + 1 < (d + 1)(Q + 2)\), and by Lemma 4, this will be execution \((i, 0)\).

Next, we make a second inductive hypothesis that the lemma holds for all executions of operation \(i\) (having depth \(d + 1\)) that are released on or before cycle \(n\). It remains to show that the lemma holds for operation execution \((i, n + 1)\). (This will simultaneously confirm the induction on both \(d\) and \(n\).) By the first induction hypothesis, we have that for all \(i' \in P_i\), \((i', n + 1)\) will be scheduled to start before or during cycle \((n + 1) + d(Q + 2)\). This implies that \((i, n + 1)\) will be schedulable by cycle \(r = (n + 1) + d(Q + 2)\). This, and (if not already scheduled) \(\tau(i, n + 1, s_{mi}^{r+1}) = s_{mi}^{r+1}\). By the second induction hypothesis, we have that \((i, n)\) will be scheduled to start before or during cycle \(n + (d + 1)(Q + 2) = r + 1\). Thus, at time \(s_{mi}^{r+2}\) (if not already scheduled), \((i, n + 1)\) will be the oldest unscheduled execution of operation \(i\), and \(\tau(i, n + 1, s_{mi}^{r+2}) = s_{mi}^{r+2}\). By Lemmas 2 and 4, execution \((i, n + 1)\) will be scheduled to start during cycle \(r + 2 = (n + 1) + (d + 1)(Q + 2)\), concluding the proof. 

**Proposition 6** The function \(\text{CyclicPDR()}\) terminates after a finite number of cycles and returns a set of feasible cyclic start times: \(T \in T\).

**Proof.** By Lemma 1, cycle-counting, cycle-fitting and the termination criterion ensures \(T \in T\). Let \(\overline{d}\) denote the maximum depth of any node in the graph \((O, A)\). By Lemma 5, the maximum makespan of any job in the \(\text{CyclicPDR()}\) simulation is \((\overline{d}(Q + 2) + 2)C\). Let \(\overline{r} = \overline{d}(Q + 2) + 2\). The amount of work released each cycle is \(\overline{p} = \sum_{i \in O} p_i\). Since this work will be completed within \(\overline{r}\) cycles, a lower bound for the cumulative amount of work scheduled (and completed) by the end of cycle \(r\) is given by \(L(r)\), where

\[
L(r) = \begin{cases} 
0 & \text{if } r \leq \overline{r}; \\
(r - \overline{r}) \overline{p} & \text{otherwise}.
\end{cases}
\]

The cycle-counting property of the dispatch rule implies that the maximum work that can be scheduled to start in each cycle is \(\overline{p}\). For the algorithm to terminate, we must ultimately have a cycle in which the work scheduled to start equals \(\overline{p}\) (i.e., \(\Gamma = O\)). In any cycle, if the
amount of work scheduled to start is less than \( \bar{p} \), then it is at most \( \tilde{p} = \bar{p} - \min_{i \in O} p_i \). Hence, if the algorithm has not terminated at the end of cycle \( r \), then the cumulative amount of work scheduled is \textit{at most} \( r\tilde{p} = r\bar{p} - r(\min_{i \in O} p_i) \) and \textit{at least} \( L(r) = r\bar{p} - r\bar{p} \). This can only happen for \( r \leq \frac{r\bar{p}}{\min_{i \in O} p_i} = \frac{r\bar{p}}{\bar{p} - \tilde{p}} \). Hence, on or before cycle \( r\bar{p}/(\bar{p} - \tilde{p}) \), there must be a cycle in which \( \Gamma = O \) and CyclicPDR() terminates. 

5 Evaluation

Function \texttt{Evaluate()}, described in [7], begins with an arbitrary schedule of start times, \( T \), makes it cyclically feasible using the function \texttt{WrapAndSmooth()}, finds the best corresponding values for the cycle offset variables, \( D \), and then finds local timing improvements without altering the machine sequences. Function \texttt{Evaluate()} returns a feasible cyclic schedule \((T^*, D^*)\) and the corresponding value of the weighted flow time objective, \( WFT^* \). The relative performance of different construction heuristics can be compared after this first call to \texttt{Evaluate()}.

Two heuristics to improve the weighted flow time of a feasible cyclic schedule are described in the companion paper [7]. Both are based on the concept of \textit{compression}. The basic step in a compression heuristic is to identify a precedence constraint, \((u, v) \in A\), with positive slack time and to move the successor operation, \( v \), to start at the earliest possible time relative to the predecessor operation, \( u \). The intention of the \textit{coarse compression} heuristic is to visit many schedules in the neighborhood of the starting schedule quickly and identify a new schedule whose neighborhood is promising for more refined improvement. The focus is on speed and roughly guided moves rather than carefully restricted ones. By contrast, the \textit{fine compression} heuristic is more conservative and is intended to find a local optimum within a promising neighborhood using carefully controlled compression steps.

Hybrid algorithms to solve \textit{WFTCSP} consist of one of the construction heuristics (to generate a starting schedule), followed by one application of \texttt{CoarseCompression()}, then one application of \texttt{FineCompression()}. In this paper we compare the relative performance of different construction heuristics at two points: after the first call to \texttt{Evaluate()} and after \texttt{CoarseCompression()}. It is possible that a construction heuristic might perform poorly when initially evaluated but might work better than others in combination with the coarse compression algorithm. The two evaluation points will be referred to as \textit{before compression}.
and after coarse compression, respectively. The companion paper focuses on the comparison and evaluation of coarse compression and fine compression.

6 Empirical Results

For testing purposes, we adopted the following weighting scheme for the weighted average flow time objective. Let $R_j = \{(i, f) \in I_j \times F_j : \exists \text{ a routing path from } i \text{ to } f\}$ describe the set of maximal routing paths for job type $j$ and let $w = LCM (\{|R_j| : j \in J\})$, the least common multiple of the number of routing paths in each job. Then let

$$w_{if} = \begin{cases} \frac{w}{|R_j|}, & \text{if } (i, f) \in R_j; \\ 0, & \text{otherwise}. \end{cases}$$

From (3.4), it can be seen that the resulting objective weights each job equally (i.e., $\sum_{i \in I_j} \sum_{f \in F_j} w_{if} = w$ for every job $j$), and within each job $j \in J$, the objective weights each pair in $R_j$ equally. By construction, the weights are integers.

6.1 Test Data

As a testbed, we selected a variety of different (non-cyclic) job shop instances. Included are well-known problems from Adams, Balas, & Zawack (abz5-abz9), Fisher & Thompson (ft06, ft10, ft20), Lawrence (la01-la40), Applegate & Cook (orb01-orb10), Storer, Wu, & Vaccari (swv01-swv20), and Yamada & Nakano (yn1-yn4). The instances are summarized in Table 3. There are a total of 82 problem instances in this selected set.

We considered four types of routing structures based on these selected problem instances: serial routings, assembly routings, partial order routings, and re-entrant flow routings. The problem instances were generated as follows:

- **Serial Routing Instances**: The selected base problems are defined in terms of serial routings. No modifications were required.

- **Assembly Routing Instances**: We used the selected base problems and generated assembly routings as follows: for each job $j$ originally containing $|O_j|$ operations in a serial line routing (WLOG, assume the operations are numbered $1, 2, \ldots, |O_j|$), we select the unique job successor of $i$, $s_i$, uniformly at random from $\{i+1, i+2, \ldots, |O_j|\}$. 

32
Table 3: Job Shop Problem Instances

- **Partial Order Routing Instances:** We used the selected base problems and generated partial order routings as follows: for each job \( j \) originally containing \( |O_j| \) operations in a serial line routing (WLOG, assume the operations are numbered \( 1, 2, \ldots, |O_j| \)), we select one or two job successors of \( i \) uniformly at random from \( \{i+1, i+2, \ldots, |O_j|\} \). This method guarantees that the resulting individual job routing graphs will be connected and acyclic.

- **Re-Entrant Flow Routing Instances:** We used the selected base problems and generated re-entrant flows as follows: for each operation in each job select at random a machine to which to assign the operation. We limit the number of machines to five, thereby forcing every job to visit at least one machine more than once, provided the original data contained more than five operations per job. (Sixteen of the 82 base problems contain exactly five operations per job; all others contain more than five operations per job.)

The problem instances were converted into cyclic problems simply by specifying a cycle length. For each problem instance, we considered three cycle lengths, corresponding to the 75%, 85%, and 95% utilization levels on the bottleneck machine. That is, if a problem instance has the maximum machine load \( L_{max} = K \), then we considered the instance with \( C = [K/0.75], C = [K/0.85], \) and \( C = [K/0.95] \).
<table>
<thead>
<tr>
<th>Abbrev.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>Initial Sequence</td>
</tr>
<tr>
<td>SBJ Desc.</td>
<td>Sequence by Job in descending order of $WFT_j$</td>
</tr>
<tr>
<td>SBJ Asc.</td>
<td>Sequence by Job in ascending order of $WFT_j$</td>
</tr>
<tr>
<td>Shifting BN</td>
<td>Shifting Bottleneck</td>
</tr>
<tr>
<td>MWR-A</td>
<td>Most Work Remaining (Active)</td>
</tr>
<tr>
<td>MWR-SA</td>
<td>Most Work Remaining (Semi-active)</td>
</tr>
<tr>
<td>LWR-A</td>
<td>Least Work Remaining (Active)</td>
</tr>
<tr>
<td>LWR-SA</td>
<td>Least Work Remaining (Semi-active)</td>
</tr>
<tr>
<td>SPT-A</td>
<td>Shortest Processing Time (Active)</td>
</tr>
<tr>
<td>SPT-SA</td>
<td>Shortest Processing Time (Semi-active)</td>
</tr>
<tr>
<td>Cyclic NUM</td>
<td>Cyclic PDR with Highest Operation Number</td>
</tr>
</tbody>
</table>

Table 4: Construction Heuristics

With 82 selected base problems, four routing structures, and three utilization levels, we considered a total of 984 problem instances. For each, we ran a total of eleven construction heuristics. Table 4 lists the different heuristic methods applied. The heuristic labelled “Initial” is a naive mechanism which arbitrarily sequences machine operations before evaluation. It serves as a control mechanism for our construction heuristics. That is, in addition to comparing the construction heuristics with one another, we wish to establish that using some form of intelligent schedule construction before applying improvement is a worthwhile undertaking. Also, note that in testing the cyclic priority dispatch rule, we used the naive selection criterion “NUM”. Our reason for doing this was to ascertain whether the construction method itself, apart from the selection criterion adopted, performed favorably compared to the noncyclic heuristic methods. Thus, we chose a simple, naive selection rule to embed into CyclicPDR.

For each construction method, we are interested in measuring the average ratio of the weighted flow times (WFT) found to the WFT lower bound for different types of problem instances. Clearly we expect heuristic performance to degrade as utilization level increases, as problem size increases, and as routing structures become more complex. However, we wish to ascertain whether a particular heuristic, or class of heuristics, tends to perform
better under certain conditions. Note that for each utilization level there are 328 problem instances (82 base problems x 4 routing structures), and for each routing structure there are 246 problem instances (82 base problems x 3 utilization levels). The number of instances of a particular problem size is given by the number of base problems of that size (listed in Table 3) times 12 (3 utilization levels x 4 routing structures).

6.2 Computational Results

The code for these experiments, as well as Andrew Goldberg’s minimum cost flow code (see [7] for details of its use), were written in C. Both were compiled as DOS executable files using the Microsoft Visual C++ compiler (version 1.52), and the experiments were run on an IBM Thinkpad with a Pentium III processor.

Figure 6 shows the average ratio (over all 984 problem instances) of total weighted flow time to the WFT lower bound achieved by each construction method before and after coarse compression. It is immediately clear that Cyclic NUM outperforms all other construction methods on one or more problem dimensions, both before and after coarse compression. Further investigation revealed that this dominance is largely due to the performance of Cyclic NUM on problem instances with complex routing structures (i.e., assembly and partial order routings).

Figure 7 gives a detailed breakdown of the results by routing structure. The first chart lists the average and standard deviation of the performance ratios achieved by each method by routing structure, both before and after coarse compression. The second chart illustrates the averages in a bar chart format. The third chart illustrates the percentage of problems for which each construction heuristic method yielded one of the best three solutions for that problem (of the eleven solutions). Observe that for assembly and partial order routing structures, the average flow times achieved by Cyclic NUM after compression were less than half of the average flow times achieved by all other methods after compression except for LWR-Active (and the Cyclic NUM averages were only slightly more than half of the LWR-Active averages). The gaps before compression were even larger. Moreover, the Cyclic NUM solution was among the top three solutions for nearly all assembly and partial order instances. Thus, for these instances, the solutions found by the Cyclic NUM method are dominant in magnitude as well as in frequency.
Figure 6: Average Total Weighted Flow Time to Lower Bound Ratio Before and After Coarse Compression

Because of the domination exhibited by Cyclic NUM for assembly and partial order routing structures (which together constitute one half of all problem instances tested), we decided to disregard these instances for purposes of analyzing the remaining problem dimensions, lest they unfairly skew the results. Hence, for the remainder of this section (unless otherwise stated), we consider only the re-entrant flow instances (called "reflow" for short) and the serial routing instances (492 instances total).

For reflow and serial instances, Figure 8 gives a detailed breakdown of the average performance of each construction method by utilization level, and Figure 9 gives a similar breakdown by problem size. While Cyclic NUM outperforms all other methods (on average) at the 85% and 95% utilization levels, there is no clear winner at the 75% utilization level. (Had we included the assembly and partial order instances in these averages, we would have seen overwhelming dominance at all three utilization levels.) Similarly, Cyclic NUM consistently gives the best average performance for problem instances having 150 or more operations, but several other methods outperform Cyclic NUM for smaller problems. In fact, it is curious to note that Cyclic NUM is among the worst performers for problems having 36, 50, or 75 operations. Considering the magnitude of the differences, it is unlikely that this "poor" performance by Cyclic NUM on small problem instances holds any significance. It is more likely the case that for "easy" problem instances (i.e., low utilization and/or few operations), most intelligent construction heuristics will work reasonably well, particularly
Figure 7: Average Performance of Construction Methods by Routing Structure
when combined with a local improvement scheme like coarse compression. (Even the Initial method outperforms Cyclic NUM for small problems after compression, but not before.) As is usually the case, it is the harder problem instances that determine which methods are robust and likely to scale well. From this standpoint, Cyclic NUM is still the method of choice.

There are several other conclusions that can be drawn from Figures 8 and 9. Note that for serial and reflow routing structures, the LWR-SA heuristic will tend to sequence operations in a manner that closely resembles the sequence of the SBJ Ascending heuristic. Thus, it is not surprising that the average performance of LWR-SA is remarkably similar to SBJ-Asc for all utilization levels and all problem sizes. In addition, the two SBJ heuristics (Asc and Desc) were very close in terms of average performance across the board, indicating that the initial sequence of job placement may not be as important as the initial relative placement of operations within a job (which is the same in both methods).

At every utilization level and for all larger problem sizes, the semi-active versions of the non-cyclic priority dispatch rules (MWR-SA, LWR-SA, and SPT-SA) outperformed their active counterparts (MWR-A, LWR-A, and SPT-A) on average. It is possible that we are seeing a positive impact from cyclic global left shifts, since more such shifts are possible in semi-active (non-cyclic) schedules than in active (non-cyclic) schedules.

At every utilization level and for all larger problem sizes, the performance of the Initial method was by far the worst, lending some credence to our claim that intelligent construction does matter, and myopic local improvement schemes, by themselves, are unlikely to result in competitive schedules.

While coarse compression substantially reduces the average flow times initially achieved by almost all of the construction heuristics, it is clearly limited in its efficacy. For assembly and partial order routing instances, the average flow times achieved by the non-cyclic heuristics after coarse compression are still more than double those achieved by Cyclic NUM, implying that the former are nowhere close to optimal. Moreover, for assembly and partial order instances, coarse compression is virtually ineffective when applied after Cyclic NUM. (This phenomenon is explored in greater detail in the companion paper.) As might be expected, coarse compression has the greatest impact on those construction methods for which the average quality of the starting solution is poor, and the least impact on Cyclic NUM.
Figure 8: Average Performance of Construction Methods for Reflow and Serial Instances by Utilization Level
Figure 9: Average Performance of Construction Methods for Reflow and Serial Instances by Problem Size
Finally, we note that the Cyclic NUM heuristic yields a WFT that is, on average, 240% of the WFT lower bound after coarse compression. This percentage is high, but we strongly suspect that the WFT lower bound we have used for comparison is in general of poor quality, particularly for larger and more complex problem instances. Further research is needed to establish tighter lower bounds. Moreover, the application of fine compression, discussed in [7], reduces this average ratio to 225%.

7 Conclusions

The cyclic scheduling problem as formulated by Graves et al. [26] is NP-hard even for the case of identical jobs with serial routings. In this paper, we considered non-identical jobs with general acyclic precedence constraints and investigated two classes of heuristic construction methods. Non-cyclic methods are derived from traditional job-shop scheduling methods, and a procedure was developed to convert the resulting schedules into cyclically feasible ones. Cyclic methods apply priority dispatch rules in a manner that guarantees cyclic feasibility without further conversion. We reported empirical evidence demonstrating that the choice of schedule construction method has a strong impact on the flow time performance of the resulting schedule, even after applying the coarse compression improvement method. For problem instances having larger size and complexity, the naive Cyclic Priority Dispatch Rule described in this paper outperformed all non-cyclic rules derived from the literature (sequence by job, shifting bottleneck, and priority dispatch) in terms of average performance, standard deviation of performance, and the percentage of cases in which it yielded one of the best three solutions. We recommend cyclic priority dispatch rules as a benchmark to evaluate all future heuristic construction techniques for cyclic schedules.

References


