The Fair Periodic Assignment Problem

Rolf Nelson van Lieshout
Eindhoven University of Technology, The Netherlands

Bartholomeüs Theodorus Cornelis van Rossum
Eindhoven University of Technology, The Netherlands

Abstract

We study the periodic assignment problem, in which a set of periodically repeating tasks must be assigned to workers within a repeating schedule. The classical efficiency objective is to minimize the number of workers required to operate the schedule. We propose a $\mathcal{O}(n \log n)$ algorithm to solve this problem. Next, we formalize a notion of fairness among workers, and impose that each worker performs the same work over time. We analyze the resulting trade-off between efficiency and fairness, showing that the price of fairness is at most one extra worker, and that such a fair solution can always be found using the Nearest Neighbor heuristic. We characterize all instances that admit a solution that is both fair and efficient, and use this result to develop a $\mathcal{O}(n \log n)$ exact algorithm for the fair periodic assignment problem. Finally, we show that allowing aperiodic schedules never reduces the price of fairness.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms; Mathematics of computing → Graph algorithms; Applied computing → Transportation

Keywords and phrases Cyclic scheduling, Fairness, Traveling Salesman Problem

Digital Object Identifier 10.4230/OASIcs.ATMOS.2025.1

1 Introduction

Public transport schedules exhibit a high degree of periodicity across multiple time scales. On short time scales, timetables often repeat every hour or even more frequently; on longer time scales, crew rosters typically follow multi-week cycles. This recurring structure motivates the study of the *Periodic Assignment Problem* (PAP), where resources – such as vehicles, platforms, or crew members – must be assigned to tasks within a repeating schedule [1, 2, 10].

When assigning vehicles or platforms, the primary objective is operational efficiency and adherence to capacity constraints. However, when assigning crew members, fairness becomes a central concern: it is desirable that all employees perform the same work over time, rather than being locked into fixed subsets of tasks. While fair periodic rosters are widely used in both rail [2] and bus systems [11], the fundamental trade-off between fairness (in workload distribution) and efficiency (in the number of required workers) has not been formally quantified.

In this paper, we make this trade-off explicit. We formalize a natural fairness criterion – requiring that all workers cyclically perform the same set of tasks – and study its impact on scheduling efficiency. We characterize the structure of fair periodic assignments and present efficient algorithms to compute fair schedules that require a minimal number of workers.

1.1 Problem Description

Consider a set of n timed tasks $\mathcal{I}$ to be performed periodically by a pool of homogeneous workers. The schedule has a fixed period T (typically one week in rostering contexts), and each task $i \in \mathcal{I}$ is defined by a T-periodic open interval (a_i, b_i) with $a_i, b_i \in [0, T)$. This interval may wrap around the end of the period, i.e., it is possible that $a_i > b_i$. The duration of task i is

$$c(i) := [b_i - a_i]_T,$$

where $[\cdot]_T$ denotes the modulo-T operator mapping values to $[0, T)$.

The r-th occurrence of task i is performed in the interval $(a_i + rT,\ b_i + rT)$. Workers are immediately available for a new task after completing one (any required rest time can be absorbed into task durations). A worker who completes the r-th occurrence of task i can begin the r-th occurrence of task j if $b_i \leq a_j$, or the $(r+1)$-th occurrence of task j otherwise. In general, the *transition time* between task i and task j is independent of r and defined as

$$c_{ij} := [a_j - b_i]_T.$$

These transitions define a complete directed transition graph $\mathcal{G} = (\mathcal{I}, \mathcal{A})$, where $\mathcal{A} := \mathcal{I} \times \mathcal{I}$ includes all possible task-to-task transitions.

We now formalize the optimization problem of finding an efficient periodic assignment.

▶ **Definition 1.** *Given a period $T \in \mathbb{N}$, a set of T-periodic tasks $\mathcal{I}$, and a transition graph $\mathcal{G} = (\mathcal{I}, \mathcal{A})$, the* Periodic Assignment Problem (PAP) *is to find a subset of arcs $\mathcal{M} \subseteq \mathcal{A}$ such that:*

a) *The total transition time $\sum_{(i,j)\in\mathcal{M}} c_{ij}$ is minimized,*
b) *For every task $i \in \mathcal{I}$, $\mathcal{M}$ includes exactly one arc entering and one arc leaving i.*

The second condition requires every task to have exactly one predecessor and one successor. Consequently, any feasible solution $\mathcal{M}$ defines a collection of disjoint cycles, where each cycle represents the sequence of tasks repeatedly executed by a group of workers. A cycle $\mathcal{C}$ covering tasks $\mathcal{I}_\mathcal{C}$ with transitions $\mathcal{A}_\mathcal{C}$ requires $w(\mathcal{C})$ workers, where

$$w(\mathcal{C}) = \frac{1}{T}\left(\sum_{i\in\mathcal{I}_\mathcal{C}} c(i) + \sum_{(i,j)\in\mathcal{A}_\mathcal{C}} c_{ij}\right).$$

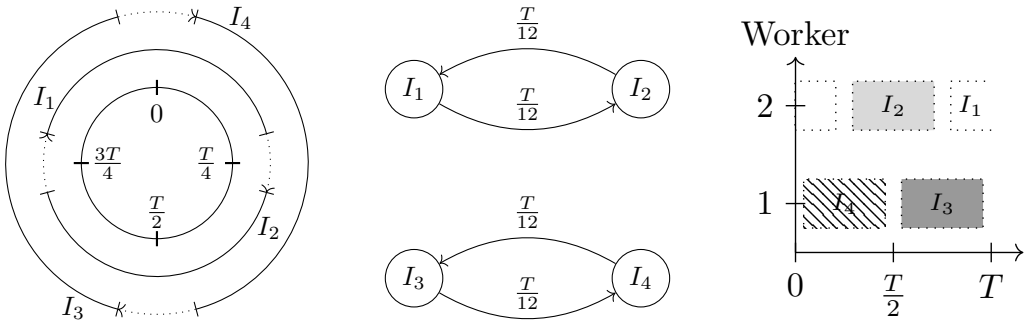
Since the durations of tasks are fixed, minimizing total transition time is equivalent to minimizing the number of workers required to operate the schedule.

Figure 1 shows a PAP instance, together with three possible representations of an efficient solution that requires the minimum of two workers. In Figure 1a, the tasks to be performed periodically are labeled $I_1, \ldots, I_4$ and displayed as circular arcs. Dotted arcs indicate the transition arcs in the efficient solution, requiring two workers to operate two disjoint schedules of two tasks each. Figure 1b represents the same solution as a set of disjoint directed cycles in the transition graph, where each node corresponds to a task and the durations of the transitions are given on the arcs. Finally, Figure 1c visualizes the solution explicitly as a periodic schedule for two workers.

In general periodic assignments, workers corresponding to different cycles are consistently performing disjoint subsets of tasks (see Figure 1). To enforce *fairness* – that all workers perform exactly the same sequence of tasks – we require that $\mathcal{M}$ forms a single directed cycle covering all tasks, i.e., a Hamiltonian cycle. This gives rise to the fair variant of the PAP:

▶ **Definition 2.** *Given a period $T \in \mathbb{N}$, a set of T-periodic tasks $\mathcal{I}$, and a transition graph $\mathcal{G} = (\mathcal{I}, \mathcal{A})$, the* Fair Periodic Assignment Problem (FPAP) *is to find a subset of arcs $\mathcal{M} \subseteq \mathcal{A}$ such that:*

a) *The total transition time $\sum_{(i,j)\in\mathcal{M}} c_{ij}$ is minimized,*
b) *$\mathcal{M}$ defines a Hamiltonian cycle in $\mathcal{G}$.*

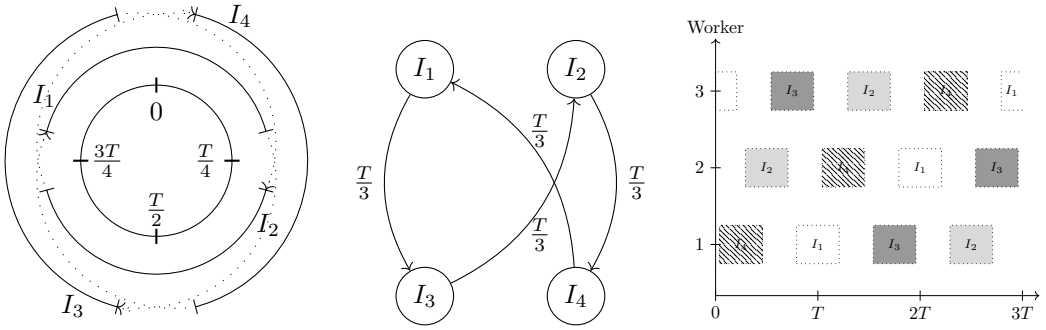


(a) Arc representation. **(b)** Graph representation. **(c)** Schedule representation.

Figure 1 Different representations of an efficient solution.

Clearly, the FPAP is a special case of the classical Traveling Salesman Problem (TSP), where tasks act as cities and transition times define pairwise distances.

Figure 2 displays three representations of a fair solution to a FPAP instance featuring the same set of tasks as Figure 1. Since a fair solution featuring two workers is not possible, the fair solution requires the use of long transition arcs, as shown explicitly in Figures 2a and 2b. The schedule representation in Figure 2c directly shows that the fair assignment requires three workers. Note that the period of the schedule is longer than T, the period of the instance.



(a) Arc representation. **(b)** Graph representation. **(c)** Schedule representation.

Figure 2 Different representations of a fair solution.

Finally, we introduce a lower bound on the number of workers required in any periodic assignment. Let $\mathcal{I}(t)$ denote the number of tasks that intersect time $t \in [0, T)$, and define the system's *load* as

$$L := \max_{t \in [0,T)} \mathcal{I}(t).$$

The instance of Figure 1 has a load of $L = 2$, matching the number of workers in the efficient solution.

1.2 Background and Related Literature

In the special case where there is some t' such that $\mathcal{I}(t') = 0$, [7] show that PAP reduces to coloring an interval graph, yielding a periodic assignment with L workers, matching the lower bound. [6] provide an $\mathcal{O}(n \log n)$ algorithm for computing such a coloring. Also the FPAP is easily solved in this case: at time t' all workers are idling, so L workers can cycle through the L colors to cover all tasks, again matching the lower bound.

In the general case, L workers still suffice for PAP, as follows from Dilworth's theorem [9]. An optimal assignment can be found via linear programming, the Hungarian algorithm, or the $\mathcal{O}(n^2 \log n)$ algorithm of [10].

To the best of our knowledge, we are the first to study the FPAP. However, a similar problem is considered in [4], which was presented at ATMOS 2024 and served as a direct inspiration for this work. Rather than requiring strict fairness, the authors consider *balanced* assignments that are asymptotically fair, i.e., where workers perform the same work in the long-run average. They reduce the problem to a construction involving pebbles on arc-colored Eulerian graphs, showing that if a balanced assignment for w workers exists, it can be found in polynomial time. Their initial construction yields a period bounded by $w^2 \cdot w!$, but in a subsequent extension [5], the authors improve this to a linear period.

Other special cases of the TSP are surveyed in [3] and other cyclic scheduling problems are discussed in [8].

1.3 Main Contributions

The main contributions of this paper are fourfold. First, we present an $\mathcal{O}(n \log n)$ exact algorithm for the PAP, improving upon the previous best $\mathcal{O}(n^2 \log n)$ runtime. Second, we prove that the classical Nearest Neighbor heuristic for TSP yields a fair periodic assignment requiring at most $L+1$ workers, implying that the *price of fairness* is at most one additional worker. This bound is tight. Third, we develop an $\mathcal{O}(n \log n)$ exact algorithm for the FPAP based on subtour patching. Fourth, we show that allowing longer assignment periods and requiring balancedness rather than fairness does not reduce the number of required workers.

2 Periodic Assignment

In this section, we characterize optimal solutions to PAP, which we use later on to solve the FPAP. Moreover, we present a new $\mathcal{O}(n \log n)$ algorithm for solving PAP.

2.1 Theory

We begin by defining notation relevant to the PAP. Transition arc $(i,j) \in \mathcal{A}$ corresponds to the T-periodic interval $[b_i, a_j]$. Analogous to $\mathcal{I}(t)$, for a periodic assignment $\mathcal{M} \subseteq \mathcal{A}$, let $\mathcal{M}(t)$ denote the number of transition arcs in $\mathcal{M}$ whose associated interval intersects time $t \in [0,T)$, and let

$$c(\mathcal{I}) := \sum_{i \in \mathcal{I}} c(i) = \int_0^T \mathcal{I}(t)dt \quad \text{and} \quad c(\mathcal{M}) := \sum_{(i,j) \in \mathcal{M}} c_{ij} = \int_0^T \mathcal{M}(t)dt.$$

Since workers are either performing tasks or transitioning between tasks, it holds that the sum $\mathcal{I}(t) + \mathcal{M}(t)$ equals the number of workers required to operate the schedule for all $t \in [0,T)$, which also equals $(c(\mathcal{I}) + c(\mathcal{M}))/T$. Combining insights from [9] and [10], we now characterize the optimal periodic assignment, which uses exactly L workers.

▶ **Theorem 3.** *Let $\mathcal{M}^* \subseteq \mathcal{A}$ denote a feasible periodic assignment. The following statements are equivalent:*

- **(i)** *$\mathcal{M}^*$ is an optimal solution to PAP,*
- **(ii)** *$c(\mathcal{I}) + c(\mathcal{M}^*) = LT$,*
- **(iii)** *$\mathcal{I}(t) + \mathcal{M}^*(t) = L$ for all $t \in [0,T)$,*
- **(iv)** *$\mathcal{M}^*(t') = 0$ for some $t' \in [0,T)$.*

Proof. $(iv) \Rightarrow (iii)$: Since all workers are either busy performing tasks or transitioning, the total number of workers at any time t is $\mathcal{I}(t) + \mathcal{M}^*(t)$. By assumption, there exists a $t' \in [0, T)$ such that $\mathcal{M}^*(t') = 0$, implying $\mathcal{I}(t') + \mathcal{M}^*(t') \leq L$. Conversely, since $\mathcal{I}(t)$ attains its maximum at some t'', we have $\mathcal{I}(t'') = L$, so $\mathcal{I}(t'')+\mathcal{M}^*(t'') \geq L$. Because the total number of workers is constant over time, we must have that $\mathcal{I}(t) + \mathcal{M}^*(t) = L$ for all $t \in [0, T)$.

$(iii) \Rightarrow (ii)$: Integrating over the interval $[0, T)$ yields:

$$c(\mathcal{I}) + c(\mathcal{M}^*) = \int_0^T \mathcal{I}(t) + \mathcal{M}^*(t)\, dt = \int_0^T L\, dt = LT.$$

$(ii) \Rightarrow (i)$: If $c(\mathcal{I}) + c(\mathcal{M}^*) = LT$, then the schedule uses exactly L workers over time, matching the theoretical lower bound. Hence, $\mathcal{M}^*$ must be optimal.

$(i) \Rightarrow (iv)$: Suppose, for contradiction, that $\mathcal{M}^*(t) \geq 1$ for all $t \in [0, T)$. Then, there exists a subset of transition arcs whose intervals fully cover $[0, T)$. Without loss of generality, let these intervals be $\mathcal{N}^* = \{[b_1, a_1], \ldots, [b_m, a_m]\}$, where

$$b_1 < a_m < b_2 < a_1 < \ldots < b_m < a_{m-1}.$$

We can now construct a shorter matching $\mathcal{N}' = \{[b_1, a_m], [b_2, a_1], \ldots, [b_m, a_{m-1}]\}$, which reduces the total transition time. This contradicts the optimality of $\mathcal{M}^*$. Therefore, $\mathcal{M}^*(t) = 0$ must hold for some $t \in [0, T)$. ◀

2.2 Algorithms

Every transition arc $(i, j) \in \mathcal{A}$ in the PAP corresponds to a periodic interval $[b_i, a_j]$, and its cost depends only on the length of this interval. Therefore, solving PAP is equivalent to matching each end time b_i to a start time a_j such that the resulting intervals minimize the total transition time.

Crucially, the identity of individual tasks does not affect the optimality of the assignment – only the multiset of start and end times matters. More specifically, tasks with identical start and end times can be freely swapped without affecting the number of required workers. This insight enables an efficient algorithm, which we call Shift-Sort-and-Match. It improves upon the $\mathcal{O}(n^2 \log n)$ method of [10] by reducing the complexity to $\mathcal{O}(n \log n)$, primarily by exploiting this structural invariance.

Algorithm 1 outlines the procedure. The algorithm first shifts all start and end times such that $\mathcal{I}(0) = L$, the maximum number of simultaneously active tasks. It then sorts all start and end times into a single array, and greedily matches each end time to the earliest available start time. Note that the algorithm computes an optimal assignment in terms of a matching between end and start times (still denoted by $\mathcal{M}$); the task-to-task assignment can be recovered by tracing these matched time points back to their associated tasks.

The shifting step of Algorithm 1 requires both the maximum L and a maximizer t^* of $\mathcal{I}(t)$. After sorting all events, these can be computed by sweeping through the timeline and maintaining a counter that increments at every start time and decrements at every end time, while keeping track of the interval (a_i, b_j) where the counter reaches its highest value. The maximum load L is then given by the largest value observed by the counter. Finally, any point within the interval (a_i, b_j) where this maximum occurs can be chosen as the maximizer t^*. This procedure takes $\mathcal{O}(n \log n)$ due to the initial sorting step.

▶ **Theorem 4.** *The Shift-Sort-and-Match algorithm computes an optimal periodic assignment $\mathcal{M}^*$ for the Periodic Assignment Problem in $\mathcal{O}(n \log n)$ time.*

Algorithm 1 SHIFT-SORT-AND-MATCH.

```
1: Shift all start and end times such that 𝓘(0) = L
2: Construct array R with all starts and ends, each tagged with its type
3: Sort R in increasing order; break ties by placing ends before starts
4: Initialize empty stack Q for unmatched ends
5: Initialize matching 𝓜 ← ∅
6: Set i ← 1
7: while i ≤ |R| do
8:    if R[i] is a start then
9:       Pop top element e from stack Q
10:      Add arc (e, R[i]) to 𝓜
11:      i ← i + 1
12:   else
13:      Push R[i] onto stack Q
14:      i ← i + 1
15:   end if
16: end while
17: return 𝓜
```

Proof. We begin by analyzing the time complexity. Recall that computing L and the maximizer t^* takes $\mathcal{O}(n \log n)$. Sorting $2n$ time points also takes $\mathcal{O}(n \log n)$, and the main loop of the algorithm executes $\mathcal{O}(n)$ operations, each taking $\mathcal{O}(1)$ time. Hence, the total runtime is $\mathcal{O}(n \log n)$.

To prove correctness, we show that the algorithm constructs a feasible assignment satisfying condition (iv) from Theorem 3, which implies optimality.

We first show that the stack size when processing an event at time t equals $L - \mathcal{I}(t)$. Initially, the stack is empty and $\mathcal{I}(t) = L$. Each step of the algorithm maintains this invariant through the following cases:

- **Case 1 (Line 8):** If $R[i]$ is a start, then $\mathcal{I}(t)$ increases by 1, and the stack size decreases by 1.
- **Case 2 (Line 12):** Otherwise, $R[i]$ is an end; $\mathcal{I}(t)$ decreases by 1, and the stack size increases by 1.

This invariant yields the following consequences:

1. The stack size remains nonnegative at all times.
2. When processing a start, the stack must be nonempty, ensuring a valid match.
3. At termination, $\mathcal{I}(t) = L$ again, so the stack is empty.

Together, these observations confirm that all starts are matched to ends – by popping from the stack in Line 9 – ensuring feasibility of the assignment.

For optimality, note that all arcs match from a lower index to a higher index. In particular, no transition arcs "loop" around time $t = 0$. Hence $\mathcal{M}^*(0) = 0$, which implies the solution is optimal by property (iv) of Theorem 3. ◀

3 Fair Periodic Assignment

Imposing fairness in the periodic assignment problem amounts to requiring that the assignment ($\mathcal{M}$) defines a Hamiltonian cycle in the transition graph. Unlike in the standard PAP, this condition makes the specific identities of transition arcs essential, as these determine whether there are any subtours or not.

This additional constraint makes the problem strictly harder: it is no longer guaranteed that a solution exists using only L workers. As illustrated in Figure 2, certain FPAP instances necessarily involve long transition arcs, forcing any fair solution to use $L+1=3$ workers instead of $L=2$. Remarkably, we show that this is the worst-case scenario: every FPAP instance can be solved using either L or $L+1$ workers.

This section proceeds as follows. First, we prove that the Nearest Neighbor heuristic always yields a fair solution with at most $L+1$ workers. Then, we characterize the class of FPAP instances that admit a fair solution using exactly L workers. Building on this result, we then present an exact algorithm for solving FPAP.

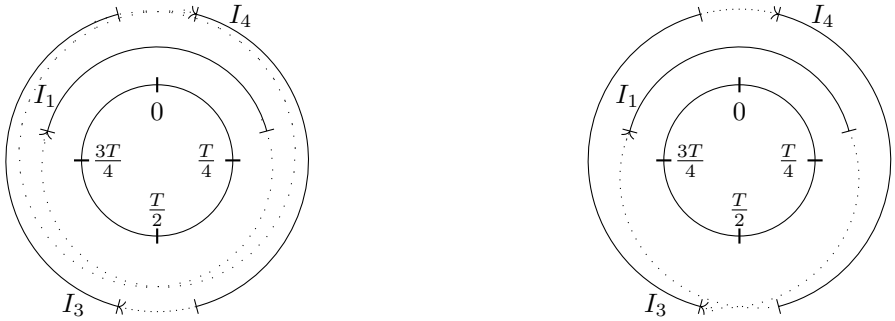
3.1 Nearest Neighbor

The Nearest Neighbor heuristic is a classical heuristic for the TSP, and repeatedly matches the current task with the closest unvisited task until all tasks are visited. This is formalized in Algorithm 2.

Algorithm 2 Nearest Neighbor.

```
Set current task i to arbitrary task u
Store unvisited tasks 𝓘 \ {i} in 𝓤
Sort 𝓤 based on increasing start time
Initialize matching 𝓜 ← ∅
while |𝓤| ≥ 1 do
    Let v be task in 𝓤 closest to i
    Add arc (i, v) to 𝓜
    Remove v from 𝓤
    i ← v
end while
Add arc (i, u) to 𝓜
return 𝓜
```

To demonstrate the heuristic, we consider the same instance as in Figure 1 but now without task I_2. Figure 3 illustrates that Nearest Neighbor, starting from task I_4, returns a solution with three workers, as indicated by the use of long transition arcs. In contrast, the optimal solution only uses two. As the following theorem shows, this corresponds to the worst-case performance of this heuristic.



(a) Nearest Neighbor. **(b)** Optimal.

Figure 3 Nearest Neighbor solution versus optimal solution.

▶ **Theorem 5.** *The* Nearest Neighbor *algorithm returns a fair periodic assignment requiring at most* $L + 1$ *workers in* $\mathcal{O}(n \log n)$ *time.*

Proof. We first analyze the runtime. A crucial observation is that the distance of the current task to the next depends solely on the end time of the current task and start time of the next task. If unvisited tasks are stored in increasing order of start time, one can efficiently find the closest unvisited task. To this end, we store the unvisited tasks $\mathcal{U}$ in a self-balancing binary search tree, allowing us to maintain a fixed task order even as tasks are visited and removed throughout the course of the algorithm. Constructing this tree takes $\mathcal{O}(n \log n)$. Finding the next task v and removing it from $\mathcal{U}$ both take $\mathcal{O}(\log n)$. Since $\mathcal{O}(n)$ operations are required until all tasks are visited, the total time complexity is $\mathcal{O}(n \log n)$.

Nearest Neighbor always returns a fair periodic assignment. It remains to show that this assignment requires at most $L+1$ workers. Suppose, to the contrary, that the assignment requires more than $L + 1$ workers. Let $t = 0$ correspond to the start of the first task that is visited. If the assignment requires more than $L+1$ workers, there is some task $i = (a, b)$ that is started after L periods, i.e., after L complete revolutions around the circle. This means that in the first L periods, a task is performed at time a. Together with task i, this implies the existence of some small $\varepsilon > 0$ such that $\mathcal{I}(a + \varepsilon) = L + 1$, a contradiction. ◀

The performance guarantee of Nearest Neighbor allows us to upper-bound the number of additional workers required to operate a fair assignment, i.e., the price of fairness.

▶ **Corollary 6.** *The price of fairness is* $1/L$.

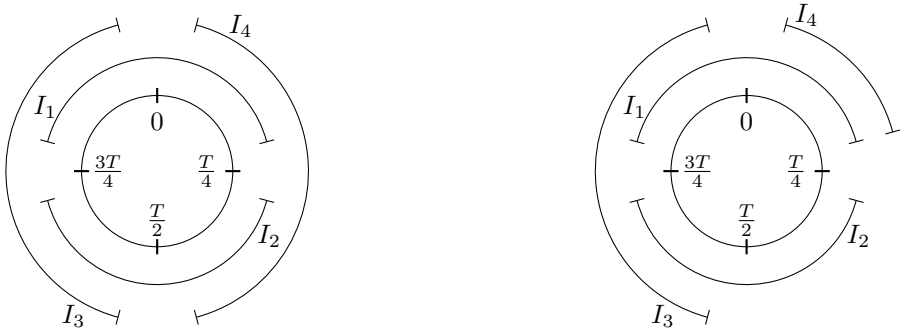
Proof. An efficient assignment always requires exactly L workers. Nearest Neighbor returns a fair solution with at most $L + 1$ workers. This upper bound is tight, see, e.g., Figure 2. It follows that the price of fairness equals $\frac{(L+1)-L}{L} = \frac{1}{L}$. ◀

3.2 Theory

We now aim to characterize the instances that admit a solution that is both fair and *efficient*, i.e., require no more than L workers. Remark that, in any efficient solution, workers are only idle during periods in which the load is not maximal, i.e., for which $\mathcal{I}(t) < L$. We refer to such intervals as *idle intervals*. Since a worker is idle right before and after performing a task, tasks act as connections between idle intervals. Moreover, the transition arcs in any efficient solution are fully contained in idle intervals. As transition arcs determine which tasks are performed consecutively, they implicitly define which idle intervals are visited along each (sub)tour.

In case a solution consists of multiple subtours, we distinguish two cases. If two disjoint cycles share transition arcs in the same idle interval, exchanging the end tasks between two transition arcs in this interval may *patch* the cycles together, reducing the number of subtours by one without increasing the transition costs and bringing us closer to a fair solution. If they are not simultaneously idle, however, such a procedure is impossible and a fair solution with L workers is out of reach. In this case, the disjoint cycles can only be patched together by transition arcs that are active outside of idle intervals, implying the use of at least one additional worker compared to an efficient solution. The connectivity of idle intervals thus contains crucial information regarding the existence of a fair solution with L workers.

In the remainder of this section, we show that the way in which tasks connect idle intervals provides a necessary and sufficient condition for the existence of solutions that are both efficient and fair. In particular, we show that such solutions can always be constructed from efficient solutions through patching.



(a) Instance A. **(b)** Instance B.

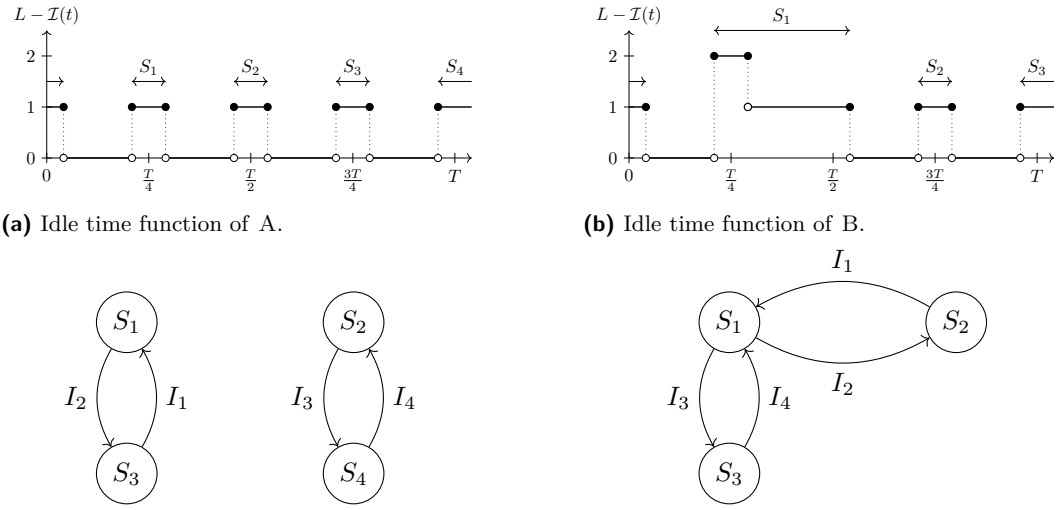
Figure 4 Two instances of the periodic assignment problem. Instance A does not admit a fair and efficient solution, while instance B does.

Throughout, we illustrate our analysis on the instances in Figure 4. The instance in Figure 4a equals that of Figure 2 and does not admit a fair solution with $L = 2$ workers. In contrast, the slightly modified instance in Figure 4b does admit a fair and efficient solution.

We proceed by formalizing the notion of idle interval.

▶ **Definition 7.** *An idle interval k is a maximal T-periodic closed interval $[s_k, e_k]$ satisfying $L - \mathcal{I}(t) > 0$ for all $t \in [s_k, e_k]$.*

An interval is maximal when it is not contained in another interval. It is easy to see that each instance admits at most n idle intervals. Moreover, the start and end time of each interval correspond to the end and start time of some tasks, respectively. The idle intervals of instances A and B are shown in Figures 5a and 5b, respectively. Observe that the number of idle workers $L - \mathcal{I}(t)$ need not be constant within an idle interval.



(a) Idle time function of A. **(b)** Idle time function of B.

(c) Idle interval graph of A. **(d)** Idle interval graph of B.

Figure 5 Idle time function and idle interval graph of instance A and B.

By optimality, efficient assignments do not make use of transition arcs outside idle intervals.

▶ **Lemma 8.** *In any optimal periodic assignment, every transition arc is contained in an idle interval.*

Proof. Denote the optimal assignment by $\mathcal{M}^*$ and suppose, to the contrary, that transition arc $(i,j) \in \mathcal{M}^*$ is not contained in any idle interval. Choose any t in the T-periodic interval $[b_i, a_j]$. Since (i,j) is not contained in any idle interval, it follows from Definition 7 that $\mathcal{I}(t) = L$. As arc (i,j) is also active at time t, it holds that $\mathcal{I}(t) + \mathcal{M}^*(t) \geq L + 1$. By statement (iii) of Theorem 3, this contradicts optimality of $\mathcal{M}^*$. ◀

Our goal is to study how the connectivity of idle intervals determines the number of disjoint cycles in a periodic assignment. To this end, we introduce the idle interval graph, representing how the various tasks connect different idle intervals.

▶ **Definition 9.** *Let $\mathcal{S}$ denote the set of idle intervals. The idle interval graph is the directed multigraph $\mathcal{H} = (\mathcal{S}, \mathcal{B})$ that contains an arc (k,l) for every task $i \in \mathcal{I}$ satisfying $a_i \in [s_k, e_k]$ and $b_i \in [s_l, e_l]$.*

To see that this is well-defined, note that $\mathcal{I}(a_i), \mathcal{I}(b_i) < L$ for all $i \in \mathcal{I}$. In other words, for each task there are unique idle intervals at its start and end time, respectively. By construction, the number of arcs $|\mathcal{B}|$ is always equal to exactly n. Moreover, any feasible periodic assignment covers all tasks once, and thereby implicitly includes all arcs of $\mathcal{H}$.

Figures 5c and 5d illustrate the idle interval graphs of instances A and B, respectively. The number of idle intervals, and hence the number of nodes in the graph, differs across the two instances. Moreover, we find that the idle interval graph of instance B is connected, while that of instance A is not.

It turns out that efficient solutions on instances with a connected idle interval graph always satisfy the following condition: if they are not fair, they contain overlapping transition arcs belonging to disjoint cycles.

▶ **Lemma 10.** *If the idle interval graph $\mathcal{H}$ is weakly connected, and an optimal periodic assignment $\mathcal{M}^*$ consists of disjoint cycles, then there exists a pair of overlapping transition arcs belonging to disjoint cycles.*

Proof. We first show that at least two disjoint cycles must contain transition arcs traversing the same idle interval. To the contrary, suppose that no two cycles share arcs in an idle interval. Take any cycle $\mathcal{C}_1 \in \mathcal{M}^*$, and denote by $\mathcal{S}_1 \subseteq \mathcal{S}$ all the idle intervals traversed along this cycle. By assumption, all the cycles in $\mathcal{M}' = \mathcal{M}^* \setminus \mathcal{C}_1$ only traverse idle intervals in $\mathcal{S}' = \mathcal{S} \setminus \mathcal{S}_1$. Clearly, $\mathcal{S}'$ is nonempty. Let $\mathcal{B}' \subseteq \mathcal{B}$ be the arcs connecting $\mathcal{S}'$ and $\mathcal{S}_1$. By connectivity of $\mathcal{H}$, this set is nonempty. Consider any arc $(k,l) \in \mathcal{B}'$. Recall that this arc is induced by some task $i \in \mathcal{I}$. Without loss of generality, assume that $k \in \mathcal{S}_1$ and $l \in \mathcal{S}'$. If task i is covered by $\mathcal{C}_1$, this cycle would contain a transition arc in l, a contradiction. Similarly, if task i is covered by some cycle in $\mathcal{M}'$, it would contain a transition arc in k, another contradiction. Since the task must be covered on exactly one cycle, we conclude that there must exist two disjoint cycles containing transition arcs in the same idle interval.

Now assume that two disjoint cycles contain transition arcs in the same idle interval $[s,e] \in \mathcal{S}$. We show that their transition arcs must overlap at some time in $[s,e]$.

Let $\mathcal{C}_1$ be the cycle whose earliest transition arc starts at time s. By definition of the idle interval and optimality of $\mathcal{M}^*$, such a cycle exists. We distinguish two cases. First, suppose that the transition arcs in $\mathcal{C}_1$ are continuously active from time s until time e. Let $\mathcal{C}_2$ be any other disjoint cycle active in the same idle interval. Clearly, any transition arc from $\mathcal{C}_2$ will overlap with some arc of $\mathcal{C}_1$. Alternatively, suppose that the transition arcs in $\mathcal{C}_1$ are active

from time s until some time $t_1 < e$, and potentially later in the idle interval too. Let $\mathcal{C}_2$ be the disjoint cycle passing through the same idle interval whose earliest transition arc has the second-earliest start time. Denote this start time by t_2. If $t_2 \leq t_1$, the corresponding transition arc must intersect with some arc in $\mathcal{C}_1$. If $t_2 > t_1$, no transition arcs are active in the interval (t_1, t_2). This contradicts the definition of the idle interval and optimality of $\mathcal{M}^*$, stating that $\mathcal{M}^*(t) \geq 1$ for all $t \in [s, e]$. Hence, at least two transition arcs from disjoint cycles must overlap. ◀

As outlined before, these overlapping arcs provide opportunities for patching. Indeed, we show that patching can always be used to obtain a fair and efficient solution on instances satisfying the conditions of Lemma 10.

▶ **Theorem 11.** *An instance admits a fair solution with L workers if and only if the idle interval graph $\mathcal{H}$ is weakly connected.*

Proof. First, let $\mathcal{M}^*$ be any fair solution with L workers. This single cycle visits all nodes (idle intervals) and arcs (tasks) of the idle interval graph. Clearly, the idle interval graph must be connected.

Now suppose that the idle interval graph is connected, and let $\mathcal{M}^*$ be any solution with L workers. If it consists of a single cycle, we are done. Assume that it consists of at least two disjoint cycles. By Lemma 10, there exists a pair of overlapping transition arcs $(i_1, j_1), (i_2, j_2)$ belonging to disjoint cycles. We can reduce the number of disjoint cycles by one through a patching operation. In particular, we replace the original, overlapping transition arcs by (i_1, j_2) and (i_2, j_1). The total transition time of the assignment, and hence the number of required workers, is unaffected by this operation. The number of disjoint cycles, however, decreases by one. As the original number of disjoint cycles is at most n, repeating this procedure at most $n - 1$ times is guaranteed to return a fair solution with L workers. ◀

To illustrate the patching idea, we return to the idle interval graphs of instances A and B in Figures 5c and 5d, respectively. In line with Theorem 11, we find that the graph of instance A, for which no fair solution with $L = 2$ workers exists, is not connected. In contrast, the graph of instance B is connected, showing that this instance admits a fair solution with two workers. It does not imply, however, that every efficient solution to this instance is fair. For example, the solution consisting of two disjoint cycles $(I_1 \to I_2 \to I_1)$ and $(I_3 \to I_4 \to I_3)$ is efficient but not fair. However, these two disjoint cycles can be patched to form a single fair solution with two workers. For example, transition arcs (I_1, I_2) and (I_4, I_3) overlap at time $t = \frac{T}{4}$. Patching these two arcs yields the fair and efficient solution $(I_1 \to I_3 \to I_4 \to I_2 \to I_1)$.

We conclude our theoretical analysis by pointing out that Theorem 11 provides an elegant, alternative way of arriving at the price of fairness in Corollary 6. In particular, we can view the price of fairness as the minimum increase in the load required to make the idle interval graph connected. Consider an instance for which the idle interval graph is unconnected. It can easily be made connected by adding a series of artificial tasks that span the period exactly once and start and end at the start and end times of consecutive intervals, respectively. The resulting instance has load $L + 1$, and always admits a fair assignment using $L + 1$ workers.

3.3 Patching

The proof of Theorem 11 shows that, on instances admitting a fair and efficient solution, any efficient solution containing disjoint cycles can be made fair by a finite number of patching operations. We now show how to efficiently implement such a patching procedure and obtain a $\mathcal{O}(n \log n)$ exact algorithm for FPAP, which we call PATCHING.

Algorithm 3 PATCHING.

Compute efficient assignment $\mathcal{M} \leftarrow$ SHIFT-SORT-AND-MATCH
Compute the number of disjoint cycles C in $\mathcal{M}$
Initialize array U, where entry $U[i]$ stores the original cycle index of task i
Initialize array V of size C, where $V[k]$ stores the index of the cycle to which the cycle originally indexed by k has been patched. Initially, $V[k] = k$ for all $k \in [C]$
Construct array of transition arcs R, sorted by increasing start time
Initialize patching arc $(i, j) \leftarrow R[1]$
for $m = 2, \ldots, |R|$ **do**
  Retrieve current arc $(k, l) \leftarrow R[m]$
  if $b_k > a_j$ **then**
    Update patching arc $(i, j) \leftarrow (k, l)$
  else if $V[U[k]] \neq V[U[i]]$ **then**
    Perform patching by replacing arcs $(i, j), (k, l)$ in $\mathcal{M}$ with $(i, l), (k, j)$
    Update cycle indices $V[U[k]], V[U[l]] \leftarrow V[U[i]]$
    $C \leftarrow C - 1$
    if $a_j > a_l$ **then**
      Update patching arc $(i, j) \leftarrow (k, j)$
    else
      Update patching arc $(i, j) \leftarrow (i, l)$
    end if
  else if $a_l > a_j$ **then**
    Update patching arc $(i, j) \leftarrow (k, l)$
  end if
end for
if $C = 1$ **then**
  return $\mathcal{M}$
end if
return NEAREST NEIGHBOR

The algorithm is described in Algorithm 3. It effectively performs the patching procedure outlined in the proof of Theorem 11. Starting from an efficient periodic assignment, it processes all transition arcs in chronological order, patching overlapping arcs that belong to disjoint cycles. We store the cycle indices in a dedicated two-array data structure, to ensure that the cycle indices can be updated in constant time after each patching operation. Once all arcs have been processed, the assignment is guaranteed to be free from overlapping transition arcs belonging to disjoint cycles. By Theorem 11, this returns a fair and efficient solution on all instances whose idle interval graph is connected. If the assignment still consists of disjoint cycles, we conclude that the idle interval graph must be disconnected. In this case, at least $L + 1$ workers are required in a fair solution, and NEAREST NEIGHBOR is called to find such a solution.

To ensure that overlapping transition arcs of disjoint cycles can always be patched, PATCHING makes use of a so-called *patching arc*. This patching arc is the transition arc with the maximum end time among all previously processed arcs, including arcs obtained through patching. Since arcs are processed in increasing order of start time, the next processed transition arc will always overlap with the patching arc, provided that it belongs to the same idle interval. This way, any transition arc belonging to a different cycle can be successfully patched. It follows that PATCHING correctly eliminates all overlapping transition arcs belonging to disjoint cycles.

▶ **Theorem 12.** *The* Patching *algorithm returns an optimal fair periodic assignment in* $\mathcal{O}(n \log n)$ *time.*

Proof. We start with a complexity analysis. Computing an efficient assignment with Shift-Sort-and-Match takes $\mathcal{O}(n \log n)$. One can compute all cycles in $\mathcal{O}(n)$ by iterating once over all transition arcs. Constructing the cycle index arrays U and V also takes linear time. Sorting the transition arcs R by increasing start time requires $\mathcal{O}(n \log n)$. All $\mathcal{O}(n)$ operations in the for-loop take $\mathcal{O}(1)$: patching itself takes constant time, and the arrays U and V allow us to update the cycle index of each task in constant time as well. Finally, calling Nearest Neighbor takes $\mathcal{O}(n \log n)$. The overall time complexity becomes $\mathcal{O}(n \log n)$.

It is clear that Patching returns a fair periodic assignment. To prove optimality, we distinguish between two cases. If the instance admits a fair solution with L workers, by the proof of Theorem 11 it suffices to show that Patching successfully patches all overlapping transition arcs belonging to different cycles. If the instance does not admit such a solution, Nearest Neighbor returns an optimal fair solution requiring $L + 1$ workers.

We now show that the algorithm correctly eliminates all overlapping transition arcs belonging to different cycles. In particular, we show that patching arc (i, j) always overlaps with, and hence can be patched with, the next processed transition arc (k, l), whenever (k, l) belongs to the same idle interval as (i, j).

Without loss of generality, assume that the initial patching arc $R[1]$ marks the start of an idle interval. Assume that the current patching arc is (i, j), and we are processing arc (k, l). In case the IF-statement on Line 9 evaluates to true, it must hold that (k, l) belongs to a different idle interval than (i, j). To the contrary, suppose that they belong to the same idle interval but $b_k > a_j$. Since we always update the patching arc to the processed transition arc with the latest end time, this would imply that no transition arcs are active in the interval (a_j, b_k), contradicting the definition of idle interval and efficiency of $\mathcal{M}$. Since the arcs belong to different idle intervals, we do not perform patching but simply update the patching arc.

Now, assume that the two arcs belong to the same idle interval but to disjoint cycles, i.e., Line 11 evaluates to true. We can patch the two arcs whenever they overlap, i.e., whenever the interval $[b_i, a_j] \cap [b_k, a_l] = [\max\{b_i, b_k\}, \min\{a_j, a_l\}]$ is nonempty. Since we process transition arcs in increasing order of their start time, it holds that $b_i \leq b_k$. From Line 9, it follows that $a_j \geq b_k$. Hence, the two arcs overlap, and we perform a patching operation. Moreover, we update the patching arc to the processed transition arc with the latest end time.

Finally, in case the two arcs belong to the same idle interval but the same cycle, we do not perform a patching operation. Instead, on Line 21 we update the patching arc to preserve the required property that it has the maximum end time among all processed transition arcs. ◀

Interestingly, we are not aware of a more efficient algorithm for testing the *existence* of a fair and efficient periodic assignment than Patching, which directly computes the optimal fair assignment. The obvious alternative way of testing the conditions of Theorem 11 is to construct the idle interval graph and perform a depth-first search to determine its connectivity. While the latter step takes linear time as $\mathcal{O}(|\mathcal{S}| + |\mathcal{B}|) = \mathcal{O}(n)$, computing the idle intervals themselves requires a sorting of the tasks, bringing the time complexity to $\mathcal{O}(n \log n)$, i.e., the same as Patching.

4 Fair versus Balanced Assignments

Thus far, we considered assignments that define periodic schedules: if there are q workers, a worker performing some task in period p also performs this task in periods $p + zq$ for any $z \in \mathbb{Z}_{\geq 0}$. In contrast, [4] and [5] consider general *aperiodic* assignments and develop

conditions for when such an assignment is *balanced*. Informally, this means that an assignment is fair in the long term average. In this section, we show that there are no benefits for allowing aperiodic assignments, or periodic assignments that repeat with a longer period than the number of workers: if there is a balanced, potentially aperiodic assignment, there also exists a fair periodic assignment with the same number of workers.

Following [4], an assignment for q workers is a function $f : \mathcal{I} \times \mathbb{Z}_{\geq 0} \to [q]$, where $f(i, r) = m$ means that the r-th occurrence of task i is performed by worker m. An assignment is feasible if no worker is assigned to two overlapping tasks. More formally, this requires that $f(i, r) \neq f(i', r')$ whenever $(a_i + r, b_i + r) \cap (a_{i'} + r', b_{i'} + r') \neq \emptyset$ for all $i \neq i'$ and r, r'. An assignment is balanced if for all tasks $i \in \mathcal{I}$ and all workers m

$$\lim_{p \to \infty} \frac{1}{p} |\{r \in [p] : f(i, r) = m\}| = \frac{1}{q}.$$

In other words, in the long term average, all workers perform all tasks with the same proportion.

We let $\mathcal{I}^R$ denote the roll-out of the task set, i.e., the set containing the intervals $(a_i + rT, b_i + rT)$ for all $i \in \mathcal{I}$ and $r \in \mathbb{Z}_{\geq 0}$. Let $\mathcal{I}^R(t)$ denote the number of active tasks at time $t \geq 0$. The definition of idle intervals naturally extends to the roll-out:

▶ **Definition 13.** *A rolled-out idle interval k is a maximal closed interval $[s_k, e_k]$ satisfying $L - \mathcal{I}^R(t) > 0$ for all $t \in [s_k, e_k]$.*

Since tasks repeat periodically, it holds that $\mathcal{I}^R(t + rT) = \mathcal{I}(t)$ for all $r \in \mathbb{Z}_{\geq 0}$ and all t. Hence, every regular idle interval (as defined in Section 3) is associated with an infinite number of corresponding rolled-out idle intervals. Analogous to the periodic case, we can define a graph that represents the connections between the idle intervals:

▶ **Definition 14.** *Let $\mathcal{S}^R$ denote the set of idle intervals. The rolled-out idle interval graph is the directed multigraph $\mathcal{H}^R = (\mathcal{S}^R, \mathcal{B}^R)$ that contains an arc (k, l) for every task $i \in \mathcal{I}^R$ satisfying $a_i \in [s_k, e_k]$ and $b_i \in [s_l, e_l]$.*

Since there is an infinite number of rolled-out idle intervals, the rolled-out idle interval graph is an infinite graph. Moreover, while the periodic idle interval graph is cyclic, the rolled-out idle interval graph is acyclic, as all arcs go forward in time. Figure 6 shows the first three periods of the rolled-out idle interval graph corresponding to a roll-out of instance B.

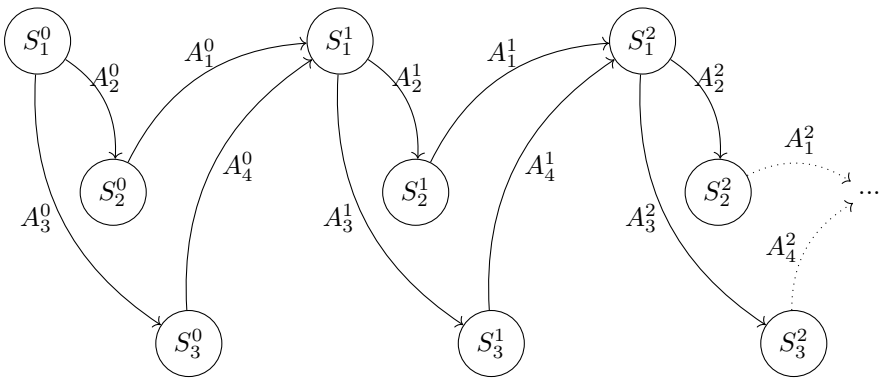


Figure 6 Rolled-out idle interval graph of instance B, showing the first three periods.

There is a many-to-one mapping from nodes and arcs in the rolled-out graph to nodes and arcs in the periodic graph. This results in the following observation:

▶ **Observation 15.** *The rolled-out idle interval graph $\mathcal{H}^R$ is weakly connected if and only the idle interval graph $\mathcal{H}$ is connected.*

We can now prove a counterpart of Theorem 11 for aperiodic instances.

▶ **Theorem 16.** *An instance admits a balanced assignment with L workers if and only if the rolled-out idle interval graph $\mathcal{H}^R$ is weakly connected.*

Proof. If $\mathcal{H}^R$ is weakly connected, $\mathcal{H}$ is connected. It follows from Theorem 11 that there exists a fair periodic assignment with L workers, which also defines a balanced assignment.

To prove the other direction, assume that the rolled-out idle interval graph is not weakly connected. This implies that there are tasks that are not connected through idle intervals. In a balanced schedule, however, workers must perform all (periodic) tasks, necessitating the use of long transition arcs that cross idle intervals. At such instants, there are L active task arcs and at least one active transition arc, so at least $L + 1$ workers are required in a balanced solution. ◀

Recall that there always exists a fair periodic assignment with $L + 1$ workers. It directly follows from Theorem 16 that imposing balancedness instead of the stricter fairness criterion does not provide any benefits in terms of efficiency:

▶ **Corollary 17.** *An instance admits a balanced assignment with q workers if and only if the instance admits a fair periodic assignment with q workers.*

References

1. Enrico Bortoletto, Rolf N van Lieshout, Berenike Masing, and Niels Lindner. Periodic Event Scheduling with Flexible Infrastructure Assignment. In *24th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2024)*, volume 123 of *Open Access Series in Informatics (OASIcs)*, pages 4:1–4:18, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.ATMOS.2024.4.
2. Thomas Breugem, Twan Dollevoet, and Dennis Huisman. Is equality always desirable? Analyzing the trade-off between fairness and attractiveness in crew rostering. *Management Science*, 68(4):2619–2641, 2022. doi:10.1287/mnsc.2021.4005.
3. Rainer E Burkard, Vladimir G Deineko, Rene Van Dal, Jack AA van der Veen, and Gerhard J Woeginger. Well-solvable special cases of the traveling salesman problem: A survey. *SIAM Review*, 40(3):496–546, 1998. doi:10.1137/S0036144596297514.
4. Héloïse Gachet and Frédéric Meunier. Balanced Assignments of Periodic Tasks. In Paul C. Bouman and Spyros C. Kontogiannis, editors, *24th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2024)*, volume 123 of *Open Access Series in Informatics (OASIcs)*, pages 5:1–5:12, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.ATMOS.2024.5.
5. Héloïse Gachet and Frédéric Meunier. Balanced assignments of periodic tasks. *CoRR*, 2025. arXiv:2407.05485.
6. Gupta, Lee, and Leung. An optimal solution for the channel-assignment problem. *IEEE Transactions on Computers*, C-28(11):807–810, 1979. doi:10.1109/TC.1979.1675260.
7. Jan H. M. Korst, Emile H. L. Aarts, Jan Karel Lenstra, and Jaap Wessels. Periodic assignment and graph colouring. *Discrete Applied Mathematics*, 51(3):291–305, 1994. doi:10.1016/0166-218X(92)00036-L.
8. Eugene Levner, Vladimir Kats, David Alcaide López de Pablo, and T.C.E. Cheng. Complexity of cyclic scheduling problems: A state-of-the-art survey. *Computers & Industrial Engineering*, 59(2):352–361, 2010. doi:10.1016/j.cie.2010.03.013.

9 James B Orlin. Minimizing the number of vehicles to meet a fixed periodic schedule: An application of periodic posets. *Operations Research*, 30(4):760–776, 1982. doi:10.1287/OPRE.30.4.760.

10 Rolf N van Lieshout. Integrated periodic timetabling and vehicle circulation scheduling. *Transportation Science*, 55(3):768–790, 2021. doi:10.1287/trsc.2020.1024.

11 Lin Xie and Leena Suhl. Cyclic and non-cyclic crew rostering problems in public bus transit. *OR Spectrum*, 37:99–136, 2015. doi:10.1007/s00291-014-0364-9.