On the Complexity of a Periodic Scheduling Problem with Precedence Relations

Richard Hladík, Anna Minaeva, and Zdeněk Hanzálek

Czech Institute of Informatics, Robotics, and Cybernetics, Czech Technical University in Prague

Abstract. Periodic scheduling problems (PSP) are frequently found in a wide range of applications. In these problems, we schedule a set of tasks on a set of machines in time, where each task is to be executed repeatedly with a given period. The tasks are assigned to machines, and at any moment, at most one task can be processed by a given machine. Since no existing works address the complexity of PSPs with precedence relations, we consider the most basic PSP with chains and end-to-end latency constraints given in the number of periods. We define a degeneracy of a chain as the number of broken precedence relations within the time window of one period. We address the general problem of finding a schedule with the minimum total degeneracy of all chains. We prove that this PSP is strongly NP-hard even when restricted to unit processing times, a common period, and 16 machines, by a reduction from the job shop scheduling problem. Finally, we propose a local search heuristic to solve the general PSP and present its experimental evaluation.

1 Introduction

Periodic scheduling problems (PSPs) are frequently found in a wide range of applications, including communications [16], maintenance [17], production [1], avionics [7], and automotive [5]. A control loop is a typical example of an application that requires periodic data transmission from sensors over control units and gateways to actuators. The result depends not only on a logically correct computation but also on the end-to-end latency measured from the moment when the sensor acquires a physical value to the moment when the actuator performs its action. Due to the periodic nature of the problem, the end-to-end latency is typically expressed in a number of periods [15].

In a PSP, we are given a set of tasks and a set of machines. Each task has a *processing time* p and is to be executed repeatedly with a given *period* T on a (given) machine. The goal is to schedule the tasks in time so that at any given moment, at most one task is processed by each machine, and the periodical nature of the tasks is satisfied. A PSP can be either preemptive or non-preemptive, when an execution of a task can or cannot be preempted by the execution of another task, respectively. In this work, we deal with the *non-preemptive* version of the PSP.

Many works have addressed the complexity of non-preemptive PSPs. Jeffay et al. in [10] address a PSP on a single machine with arbitrary release dates and deadlines equal to one period from the corresponding release dates, where the release date is the earliest time a task can start in its period and the deadline is the latest time it must complete. In the three-field Graham notation $\alpha|\beta|\gamma$ introduced in [8], where the α field characterizes the resources, the β field reflects properties of tasks, and the γ field contains the criterion, this problem is denoted as $1|T_i, r_i, d_i = r_i + T_i|-$. The authors prove that this problem is strongly NP-hard by a reduction from the 3-Partition problem. However, the proof relies on the different release dates of the tasks. Furthermore,

the case of the harmonic period set, when larger periods are divisible by smaller periods, seems to be an easier problem, since there are efficient heuristics to solve it (e.g., in [4]). However, Cai et al. in [3] strengthened the result of [10] by proving strong NP-hardness of the PSP $1|T_i^{harm}, d_i = T_i|-$ with zero release dates and harmonic periods. Later, Nawrocki et al. in [14] prove that this complexity result holds even if the ratio between the periods is a power of 2, i.e., for $1|T_i^{pow2}, d_i = T_i|-$.

A PSP with a zero jitter requirement (also known as strictly or perfectly periodic, where the position of a task within a period is the same in all periods) is widely assumed ([4], [6]) since a non-zero jitter represents a disturbance to control systems. Korst et al. in [12] show that the PSP with zero jitter requirements on a single machine, $1|T_i, jit_i = 0|-$, is strongly NP-hard. Moreover, the same problem with unit processing times, $1|T_i, jit_i = 0, p_i = 1|-$, is shown to be NP-hard by Bar-Noy et al. [1] by the reduction from the graph coloring problem. Jacobs et al. in [9] strengthen this result by proving that this PSP is strongly NP-hard. As a matter of fact, even deciding whether a single task can be added to the set of already scheduled tasks for this PSP is NP-complete, since it is the problem of computing simultaneous incongruences.

There are no results on the complexity of non-preemptive PSPs with precedence relations. Therefore, in this paper, we focus on a PSP with chains of precedence relations, i.e., a task can only be scheduled after the completion of its predecessor unless it is the first task in the chain. End-to-end latency of a chain is the time from the start time of its first task to the completion time of its last task. We define the degeneracy of a chain as its end-to-end latency divided by its period. Alternatively, it is the number of broken precedence relations within the time window of one period.

We consider a general PSP$_{gen}$, PD$|T_i^{harm}, jit_i = 0, \text{chains}| \sum \delta$, where tasks with harmonic periods and zero jitter requirements are scheduled on multiple dedicated machines (i.e., assignment of tasks to machines is given) so that the total degeneracy of all chains is minimized. Furthermore, we address the complexity of a special case of PSP$_{gen}$ called PSP$_{com}$, PD16$|T_i = T, jit_i = 0, p_i = 1, \text{chains}, \delta_l = 0|-$, where tasks with a *common* period and unit processing times are scheduled on 16 machines, and chains are 0-degenerated (i.e., all precedence relations are satisfied within one period).

The main three contributions of this paper are: 1) We propose a novel formulation of a PSP with chains of precedence constraints called PSP$_{gen}$ based on chains degeneracy. The degeneracy offers a coarser alternative to the widely used end-to-end latency and may be a more suitable metric for some real-world problems. 2) We establish that PSP$_{gen}$ is strongly NP-hard even when restricted to unit execution times, common period, and 16 machines by a reduction from the job shop scheduling problem $J3 \mid p_i = 1 \mid C_{\max}$. This problem is called PSP$_{com}$ and denoted as PD16$|T_i = T, jit_i = 0, p_i = 1, \text{chains}, \delta_s = 0|-$. 3) We provide a local search heuristic algorithm that solves PSP$_{gen}$. Moreover, we experimentally demonstrate the soundness of our algorithm and show that it can solve $92\,\%$ of our instances (with up to 9 000 of tasks) in a few minutes on a desktop computer, and the provably optimal solution is found for more than 75% instances.

2 Problem Description

In this section, we present a general problem PSP$_{gen}$ considered in this work. We first introduce non-collision constraints and then the optimization criterion based on how well the precedence constraints are satisfied within one period. Finally, we constructively prove that the existence of a solution satisfying the non-collision constraints is

equivalent to the existence of the solution satisfying both non-collision and precedence constraints.

2.1 Problem Statement

We are given a set of *tasks* $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ and a set of *machines* $\mathcal{M} = \{\mu_1, \ldots, \mu_{|\mathcal{M}|}\}$. Each task τ_i has a *processing time* $p(\tau_i) \in \mathbb{N}^*$ and executes repeatedly with a given *period* $T(\tau_i) \in \mathbb{N}^*$. Here, $\mathbb{N}^*$ is a set of natural numbers without zero and $\mathbb{N}_0$ is a set of natural numbers with zero. Each task τ_i is also *assigned* to machine $m(\tau_i) \in \mathcal{M}$, on which it must be executed.

Our goal is to find a *schedule*, which is a function $s : \mathcal{T} \to \mathbb{N}_0$ that assigns a *start time* $s(\tau_i)$ to each task τ_i. The task τ_i is then executed every $T(\tau_i)$ units of time, i.e., with zero jitter; its k-th execution (for $k \in \mathbb{N}_0$) spans the interval $[s(\tau_i) + k \cdot T(\tau_i), s(\tau_i) + p(\tau_i) + k \cdot T(\tau_i))$. Let $\mathcal{R}(\tau_i)$ denote the union of all such intervals for task τ_i.

A schedule s *has no collisions* if there is at most one task executed on each machine at any given moment, that is:

$$\mathcal{R}(\tau_i) \cap \mathcal{R}(\tau_j) = \emptyset, \;\; \forall\, i \neq j : m(\tau_i) = m(\tau_j) \tag{1}$$

Korst et al. [11] have shown that for zero-jitter case, Equation (1) is equivalent to

$$p(\tau_i) \leq (s(\tau_j) - s(\tau_i)) \bmod g_{i,j} \leq g_{i,j} - p(\tau_j), \tag{2}$$

where $g_{i,j} = \gcd(T(\tau_i), T(\tau_j))$.

There are precedence constrains in the form of task chains. Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be a partition of $\mathcal{T}$ into pairwise disjoint ordered sets $C_1, \ldots, C_k$ such that in each set, all tasks have the same period.[⟨1⟩] Each of these sets is called a *(precedence) chain*. The r-th task of the c-th chain is denoted by C_c^r (for $r \in \{1, \ldots, |C_c|\}$). We call C_c^{r-1} and C_c^{r+1} (if they exist) *predecessor* and *successor* of C_c^r. Tasks without a predecessor are called *root tasks*. Note that formally, $C_c^r \in \mathcal{T}$.

A schedule s *satisfies precedence relations*, if

$$s(C_c^r) \geq s(C_c^{r-1}) + p(C_c^{r-1}), \; \forall C_c \in \mathcal{C}, \; r = 2, \ldots, |C_c| \tag{3}$$

that is, each task starts only after its predecessor finishes execution. Given that all tasks in a chain have the same period, all further executions of the chain are also ordered correctly.

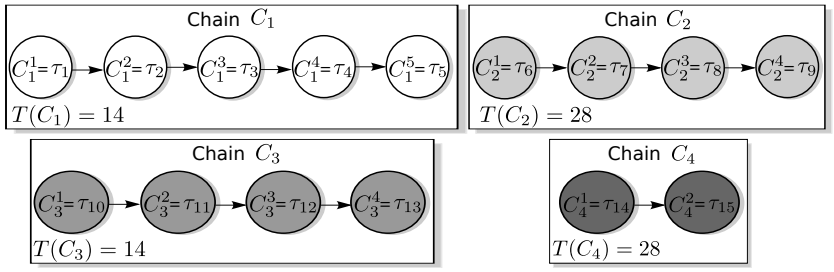
The end-to-end latency $L(C_c)$ of a chain C_c is the distance from the start time of the first task to the completion time of the last task in the chain as given by Equation (4). Then, the *degeneracy* $\delta_s(C_c)$ with respect to schedule s is defined in Equation (5).

$$L(C_c) = s(C_c^{|C_c|}) + p(C_c^{|C_c|}) - s(C_c^1), \tag{4}$$

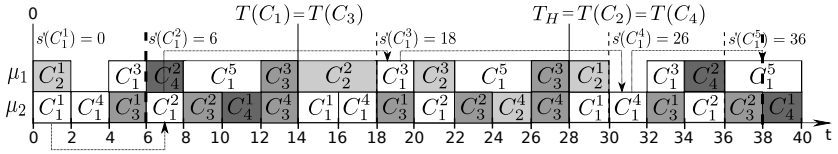
$$\delta_s(C_c) = \left\lceil \frac{L(C_c)}{T(C_c^1)} \right\rceil - 1. \tag{5}$$

In other words, a chain degeneracy is the number of crossed relative period boundaries, with the first period starting at the start time of the first task in the chain. For the

[⟨1⟩] On the other hand, two tasks with equal period are not necessarily in the same C_c



(a) Chains of precedence relations C_1, C_2, C_3, and C_4.



(b) An example schedule with degeneracies $\delta_s(C_1)=2$, $\delta_s(C_2)=0$, $\delta_s(C_3)=1$, and $\delta_s(C_4)=0$. Solid vertical lines mark absolute period boundaries, whereas bold dashed lines depict relative period boundaries for chain C_4.

Fig. 1: A periodic scheduling problem with an example solution. There are four chains: $C_1 = (\tau_1, \tau_2, \tau_3, \tau_4, \tau_5), C_2 = (\tau_6, \tau_7, \tau_8, \tau_9), C_3 = (\tau_{10}, \tau_{11}, \tau_{12}, \tau_{13})$, and $C_4 = (\tau_{14}, \tau_{15})$ with periods 14, 28, 14, and 28 time units, respectively. The task assignments are $m(\tau_3) = m(\tau_5) = m(\tau_6) = m(\tau_7) = m(\tau_8) = m(\tau_{12}) = m(\tau_{15}) = \mu_1$ and $m(\tau_1) = m(\tau_2) = m(\tau_4) = m(\tau_9) = m(\tau_{10}) = m(\tau_{11}) = m(\tau_{13}) = m(\tau_{14}) = \mu_2$, the processing times are 2 except for $p(\tau_5) = p(\tau_7) = 4$.

example in Figure 1, the degeneracy of chain C_1 is 2, since its first task $C_1^1 = \tau_1$ starts at 0 and crosses its relative period boundary (in this case coinciding with its absolute period boundary) 2 times. On the other hand, although C_4 crosses its absolute period boundary at time 28, its degeneracy equals 0, since its relative period boundary is at time $10 + 28 = 38$.

The *degeneracy* of a schedule $\delta(s)$ is then defined as:

$$\delta(s) = \begin{cases} \sum_{C_c \in \mathcal{C}} \delta_s(C_c) & \text{if } s \text{ is feasible,} \\ +\infty & \text{otherwise.} \end{cases}$$

We also say a schedule with degeneracy k is *k-degenerated*. Note that by this definition, the minimum possible degeneracy is zero. In this case, its end-to-end latency does not exceed the length of chain's period.

With the definitions provided, PSP$_{gen}$ is: given a description of tasks and precedence chains, find a schedule with minimal degeneracy. Formally, find

$$\underset{s\,:\,\mathcal{T} \to \mathbb{N}_0}{\arg\min}\ \delta(s), \tag{6}$$

such that non-collision (1) and precedence (3) constraints hold. In the three-field notation, PSP$_{gen}$ is denoted as PD$|T_i^{harm}, jit_i = 0, \text{chains}|\sum \delta$.

2.2 Equivalence Proof

We show that if there is a schedule satisfying non-collision constraints for PSP$_{gen}$, it is possible to modify it to also satisfy precedence constraints. We actually formulate this in a stronger form, which we use later in Section 3:

Lemma 1. *Let s be a schedule satisfying (1), and let $\tau^\star$ be a fixed root task. Then there exists a schedule s' satisfying both (1) and (3) such that $s'(\tau^\star) = 0$ and $s'(\tau) \leq T(\tau)$ for all other root tasks τ. Moreover, if s already satisfies (3), then $\delta(s') \leq \delta(s)$.*

Proof. We define s'' as s moved in time so that the start time of the chosen root task $\tau^\star$ is zero:

$$s''(\tau_i) = s(\tau_i) - s(\tau^\star).$$

Schedule s'' satisfies non-collision constraint (2) since the move does not change relative positions of start times of the tasks. However, some $s''(\tau_i)$ may be negative since $\tau^\star$ may not be a task with the minimum start time in s. We may fix that by moving each task by a suitable multiple of its period to the right in time. Nevertheless, precedence constraint (3) may still be violated, and therefore we fix both of these issues simultaneously as described in the following paragraph.

For $\tau_i \in \mathcal{T}$, $t_0 \in \mathbb{N}_0$, let $\mathrm{shift}(\tau_i, t_0)$ be a minimum value $t = s''(\tau_i) + k \cdot T(\tau_i)$ (where $k \in \mathbb{Z}$) such that $t \geq t_0$. We construct s' chain by chain, traversing each chain in the order of precedences. Given a chain C_c, we set $s'(C_c^1) = \mathrm{shift}(C_c^1, 0)$. For each subsequent task C_c^r, we set $s'(C_c^r) = \mathrm{shift}(C_c^r, s'(C_c^{r-1}) + p(C_c^{r-1}))$. That is, the shift operation guarantees that each task starts at the earliest time after its predecessor finishes and do not collide with other tasks. This automatically ensures that the precedence constraint holds for s'.

With this construction, $s'(\tau_i) \bmod T(\tau_i) = s''(\tau_i) \bmod T(\tau_i)$ for each τ_i, which means s' satisfies non-collision constraint (1) due to Constraint (2).

Finally, since $\tau^\star$ is a root task, $s'(\tau^\star) = \mathrm{shift}(\tau^\star, 0) = 0$. For all other root tasks τ, $s'(\tau) < T(\tau)$ by the definition of shift. Assume s satisfies precedence constraints (3) and observe that $\delta(s'') = \delta(s)$. For each C_c, $\delta_{s'}(C_c) \leq \delta_{s''}(C_c)$, since the tasks are scheduled as close as possible. Thus, $\delta(s') \leq \delta(s'') = \delta(s)$. □

An example of the result of this constructive proof can be seen in Figure 1(b), where for chain C_1, an original schedule s can be $s(\tau_1) = 0, s(\tau_2) = 6, s(\tau_3) = 4, s(\tau_4) = 2, s(\tau_5) = 8$, and a constructed schedule s' is $s'(\tau_1) = 0, s'(\tau_2) = 6, s'(\tau_3) = 18, s'(\tau_4) = 30$, and $s'(\tau_5) = 36$.

3 Problem Complexity

In this section, we prove that even a less general version of PSP$_{gen}$ is strongly NP-hard by a polynomial transformation from a special version of the job shop scheduling problem.

3.1 PSP$_{com}$

The proof of NP-hardness will be carried out on a restricted variant of PSP$_{gen}$ called PSP$_{com}$. This is a decision problem based on PSP$_{gen}$ with the following modifications: the number of machines is at most 16, all tasks have unit processing time and a common

period T_H, and the problem is to decide whether there exists a 0-degenerated schedule. Thus, it is denoted as PD16$|T_i = T, jit_i = 0, p_i = 1, \text{chains}, \delta_l = 0|-$ in the three-field notation. Note that (strong) NP-hardness of PSP$_{com}$ implies (strong) NP-hardness of PSP$_{gen}$ since the latter is more general.

Definition 1. *PSP$_{com}$ is defined by a 4-tuple $(\mathcal{T}, \mathcal{M}, \mathcal{C}, T_H)$ consisting of the task set, $\mathcal{T}$, machine set, $\mathcal{M}$, chain set, $\mathcal{C}$, and the common period T_H. The problem is to decide if there exists a feasible schedule s such that precedence constraint (3), non-collision constraint* (7)*, and 0-degeneracy constraint* (8) *hold.*

$$s(\tau_i) \bmod T_H \neq s(\tau_j) \bmod T_H, \ \forall\, i \neq j : m(\tau_i) = m(\tau_j) \tag{7}$$

$$s(C_c^k) - s(C_c^1) < T_H, \qquad \forall C_c \in \mathcal{C}. \tag{8}$$

Due to unit processing times, non-collision constraint (1) resulted in Constraint (7) and 0-degeneracy constraint (8) is simply a constraint on chains' end-to-end latency (4).

Finally, a schedule of PSP$_{com}$ *is feasible* if *it satisfies precedence constraint (3), non-collision constraint (7), and 0-degeneracy constraint (8).*

3.2 Job Shop Scheduling Problem JS3

We prove NP-hardness of PSP$_{com}$ by reduction from a specific variant of the *job shop scheduling problem* JS3 denoted in the three-field notation as $J3 \mid p_i = 1 \mid C_{\max}$. Thus, the tasks with unit processing times are scheduled on three machines so that the makespan (i.e., the completion time of the latest task) is minimal. We formulate it briefly in the following paragraphs.

Definition 2. *JS3 is defined by a 4-tuple $(\hat{\mathcal{T}}, \hat{\mathcal{M}}, \hat{\mathcal{C}}, L)$ consisting of the task set, $\hat{\mathcal{T}} = \{\hat{\tau}_1, \ldots, \hat{\tau}_{\hat{n}}\}$, machine set, $\hat{\mathcal{M}} = \{\hat{\mu}_1, \hat{\mu}_2, \hat{\mu}_3\}$, chain set, $\hat{\mathcal{C}} = \{\hat{C}_1, \ldots, \hat{C}_{\hat{k}}\}$, and the maximum makespan L. The problem is to decide if there exist schedule s such that non-collision constraint (1) and precedence constraint (3) (with s substituted for $\hat{s}$, τ_i for $\hat{\tau}_i$, and similarly for other variables) such that makespan constraint* (9) *holds.*

$$\max_{\hat{\tau}_i \in \hat{\mathcal{T}}} \{\hat{s}(\hat{\tau}_i) + \hat{p}(\hat{\tau}_i)\} - \min_{\hat{\tau}_i \in \hat{\mathcal{T}}} \{\hat{s}(\hat{\tau}_i)\} \leq L, \tag{9}$$

The definitions of the task set, machine set, and chain set are the same as those of PSP$_{com}$. However, the tasks are not periodic. Therefore, 0-degeneracy constraint states that the time elapsed between the first task starting and the last task finishing among all tasks must be at most L.

Lenstra et al. have shown [13] that this problem is strongly NP-hard.

3.3 Naive Incomplete Reduction

We show that PSP$_{com}$ (and, therefore, PSP$_{gen}$) is strongly NP-hard by constructing a polynomial reduction from JS3 to PSP$_{com}$.

An obvious, but incorrect attempt at the reduction is: given an arbitrary JS3 instance of $\hat{\mathcal{I}} = (\hat{\mathcal{C}}, \hat{\mathcal{M}}, \hat{\mathcal{C}}, L)$, we create one PSP$_{com}$ task for each JS3 task: $\mathcal{T} = \{\, \tau_i \mid \hat{\tau}_i \in \hat{\mathcal{T}} \,\}$, and similarly $\mathcal{M} = \{\, \mu_i \mid \hat{\mu}_i \in \hat{\mathcal{M}} \,\}$. We also define $\mathcal{C} = \{\, C_1, \ldots, C_k \,\}$, where $k = \hat{k}$ and $C_i = \hat{C}_i$. At last, for each $\tau_i \in \mathcal{T}$ we define $p(\tau_i) = 1$ and $T(\tau_i) = T_H = L$.

Unlike PSP$_{com}$, which imposes the time limit T_H on time elapsed between the first task starting and the last task finishing *for each chain separately*, JS3 requires L to be a global limit *for tasks among all chains*. Thus, some solutions feasible for PSP$_{com}$ are infeasible for JS3. For the example in Figure 1(b), the schedule for chains C_2 and C_4 with period $T_H = 28$ is feasible for PSP$_{com}$, but not feasible for JS3 due to makespan constraint (9). Although both chains individually span over less than 28 time units (C_2 from 0 to 26 and C_4 from 10 to 36), together, they run from time 0 to time 36, which is more than the corresponding makespan value of $L = 28$. Thus, *the two conditions are equivalent only if we ensure that all chains start at the same time*. We focus on that in the following subsections.

3.4 Anchoring Chains

We allocate new machines, tasks, and chains to enforce a particular configuration of the schedule. By introducing several "dense" chains (i.e., chains with the number of tasks equal to the hyper-period), we make all chains start at the same time. To simplify the analysis, we limit ourselves to the following subclass of schedules.

Definition 3. *Let τ be a root task. A schedule s is* τ-initial*, if $s(\tau) = 0$, and $s(\tau') < T_H$ for all other root tasks τ'. A schedule s is* initial *if it is τ-initial for some task τ.*

Without loss of generality, we may consider only τ-initial schedules: Lemma 1 guarantees that if the instance has a feasible schedule, it also has a τ-initial feasible schedule. The converse is true since τ-initial feasible schedule is a feasible schedule by definition.

To make the reduction in Section 3.3 complete, we formulate and prove Lemma 2. It states that for any PSP$_{com}$ instance, except for special cases, we can create another instance that is feasible if and only if there exists a schedule for the initial instance satisfying the makespan constraint in job shop scheduling problem.

Lemma 2. *Given a PSP$_{com}$ instance $\mathcal{I} = (\mathcal{T}, \mathcal{M}, \mathcal{C}, T_H)$ with $\mathcal{M} = \{\mu_1, \mu_2, \mu_3\}$, $|\mathcal{C}| > 1$ and $T_H > 2$, it is possible to create a PSP$_{com}$ instance $\mathcal{I}'$ such that $\mathcal{I}'$ is feasible if and only if*

$$\begin{gathered}\exists s, \text{ a feasible schedule of } \mathcal{I}, \text{ such that} \\ \forall \tau_i \in \mathcal{T} : s(\tau_i) + p(\tau_i) \leq T_H.\end{gathered} \tag{10}$$

To prove Lemma 2, we formulate Lemma 3. It states that the space of solutions (schedules) for a PSP$_{com}$ problem instance with 16 machines and no additional restrictions is equivalent to the space of schedules for a PSP$_{com}$ instance with 4 machines and the enforced configuration shown in Figure 2. In this configuration, tasks mapped to one machine may be executed in a time interval $[0, x]$, whereas all other tasks may be executed in a time interval $[x, {T_H}'']$ for a fixed (of our choosing) $x \in \{2, \ldots, T_H - 3\}$. Therefore, this lemma allows for working in this constrained space of schedules.

Lemma 3. *Given a PSP$_{com}$ instance $\mathcal{I}'' = (\mathcal{T}'', \mathcal{M}'', \mathcal{C}'', {T_H}'')$ and a parameter $x > 1$, where $\mathcal{M}'' = \{\mu_1, \mu_2, \mu_3, \mu_\star\}$, and ${T_H}'' > x + 2$, it is possible to create an instance $\mathcal{I}'$ such that $\mathcal{I}'$ is feasible if and only if*

$$\begin{gathered}\exists s, \text{ a feasible schedule of } \mathcal{I}'', \text{ such that:} \\ \forall \tau_i \in \mathcal{T}'' : m(\tau_i) = \mu_\star \implies s(\tau_i) \bmod {T_H}'' \in [0, x), \\ \forall \tau_i \in \mathcal{T}'' : m(\tau_i) \neq \mu_\star \implies s(\tau_i) \bmod {T_H}'' \in [x, {T_H}'').\end{gathered} \tag{11}$$

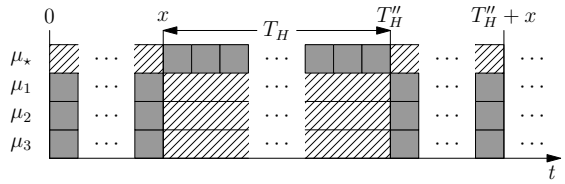


Fig. 2: Illustration of the complexity proof. Gray solid area stand for the tasks added in Lemma 3. In the lower right hatched area, JS3 tasks are located, whereas the upper left hatched area contains the anchoring tasks added in Lemma 2.

Equation (11) states that the tasks from $\mathcal{T}''$ are forbidden to execute in the gray solid areas of Figure 2 (and in any of their congruent copies).

We first prove Lemma 2 using the result of Lemma 3 and then we prove Lemma 3.

Proof (Lemma 2). The main idea is to use Lemma 3 to work with schedules with the configuration displayed in Figure 2. We "anchor" each chain by prepending to it a special task assigned to $\mu_\star$. This guarantees that each chain starts before time x. Since tasks of $\mathcal{I}$ (i.e., JS3 tasks) must be executed in the time interval $[0, {T_H}'' + x)$ due to the end-to-end latency constraint, and at the same time cannot be executed in time intervals $[0, x)$ and $[{T_H}'', {T_H}'' + x)$, they must be executed in interval $[x, {T_H}'']$. Therefore, the makespan of the resulting JS3 chains is not more than $T_H = {T_H}'' - x$. The details follow.

For each chain, we create a new anchor task τ_{a_c} and prepend it to the chain: $C_c'' = (\tau_{a_c}, C_c^1, \ldots, C_c^{|C_c|})$. We create an instance $\mathcal{I}'' = (\mathcal{T}'', \mathcal{M}'', \mathcal{C}'', {T_H}'')$ with $\mathcal{T}'' = \mathcal{T} \cup \{\tau_{a_1}, \ldots, \tau_{a_k}\}$, $\mathcal{M}'' = \mathcal{M} \cup \{\mu_\star\}$, $\mathcal{C}'' = \{C_1'', \ldots, C_k''\}$, and ${T_H}'' = T_H + k$. We assign the anchor tasks to the auxiliary machine: $m(\tau_{a_c}) = \mu_\star$ for all c.

We use Lemma 3 on $\mathcal{I}''$ with $x = k$ to obtain $\mathcal{I}' = (\mathcal{T}', \mathcal{M}', \mathcal{C}', {T_H}')$. We want to prove that $\mathcal{I}'$ simulates the JS3 makespan constraint (10) in $\mathcal{I}$. By Lemma 3, $\mathcal{I}'$ enforces the configuration depicted in Figure 2 in $\mathcal{I}''$. We only have to prove that $\mathcal{I}''$ with the configuration constraints (11) simulates the JS3 makespan constraint (10) in $\mathcal{I}$. Formally, proving the lemma is now equivalent to proving that $\mathcal{I}''$ satisfies configuration Constraints (11) if and only if $\mathcal{I}$ satisfies Constraint (10). Next, we prove both implications.

"$\mathcal{I}$ satisfies Constraint (10) $\Rightarrow$ $\mathcal{I}''$ satisfies Constraint (11)": Let s be the feasible schedule of $\mathcal{I}$ satisfying Constraint (10). We create s'' as follows:

$$s''(\tau_i) = \begin{cases} c - 1 & \text{if } \tau_i = \tau_{a_c} \text{ for some } \tau_{a_c}, \\ s(\tau_i) + k & \text{otherwise.} \end{cases}$$

The definition is valid, since if $\tau_i \neq \tau_{a_c}$, then $\tau_i \in \mathcal{T}$, and $s(\tau_i)$ is defined. Both the satisfaction of Constraints (11) of s'' and its 0-degeneracy with ${T_H}'' = T_H + k$ are guaranteed by the construction.

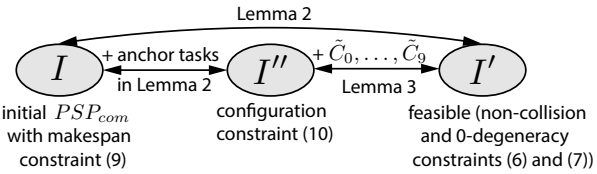


Fig. 3: Illustration of the connection of instances $\mathcal{I}$, $\mathcal{I}'$, and $\mathcal{I}''$ in Lemmas 2 and 3

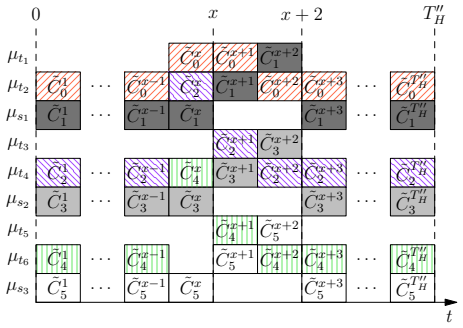


Fig. 4: Assignment of tasks in $\tilde{C}_0, \ldots, \tilde{C}_5$ to machines $\mu_{t_1}, \ldots, \mu_{t_6}, \mu_{s_1}, \ldots, \mu_{s_3}$ and the only possible schedule of these tasks. Tasks in the same chain have the same background. The purpose of this configuration is to make free only intervals $[x, x+2)$ on machines μ_{s_1}, μ_{s_2} and μ_{s_3}.

"$\mathcal{I}''$ satisfies Constraint (11) $\Rightarrow$ $\mathcal{I}$ satisfies Constraint (10)": Let s'' be the feasible schedule of $\mathcal{I}''$ satisfying Constraints (11). Since there are k anchor tasks assigned to machine $\mu_\star$ and k time moments where the tasks may be scheduled, there must exist τ_{a_c} such that $s''(\tau_{a_c}) \bmod {T_H}'' = 0$. We may assume s'' is τ_{a_c}-initial, otherwise we invoke Lemma 1 with τ_{a_c} and make it such. Observe that the shifted schedule still satisfies Constraints (11). Since s'' is initial, $s''(\tau_{a_c}) \leq {T_H}''$ and therefore, due to configuration constraints (11), $s''(\tau_{a_c}) \leq k$ for all τ_{a_c}. Then, $s(\tau)'' < {T_H}'' + k$ for all $\tau \in \mathcal{T}$ because of 0-degeneracy, and, finally, $s''(\tau) < {T_H}'' = T_H + k$ since Constraints (11) hold for s''.

We may thus set $s(\tau_i) = s''(\tau_i) - k$ for all $\tau_i \in \mathcal{T}$ and observe the resulting schedule is feasible and satisfies Constraint (10).

□

Now we proceed by proving Lemma 3.

Proof (Lemma 3). We create 12 new machines and $10 \times T_H$ new tasks. We create a new instance $\mathcal{I}' = (\mathcal{T}', \mathcal{M}', \mathcal{C}', {T_H}')$ with $\mathcal{M}' = \mathcal{M}'' \cup \{\mu_{t_1}, \ldots, \mu_{t_9}, \mu_{s_1}, \mu_{s_2}, \mu_{s_3}\}$, ${T_H}' = {T_H}''$, $\mathcal{C}' = \mathcal{C}'' \cup \{\tilde{C}_0, \ldots, \tilde{C}_9\}$, and $\mathcal{T}' = \mathcal{T}'' \cup \tilde{C}_0 \cup \cdots \cup \tilde{C}_9$. Each chain consists of ${T_H}''$ new tasks with unit processing times and we shall prove that the start times of all auxiliary tasks are predetermined across all initial feasible schedules.

We prove the lemma in two steps: first, we describe the assignment of the tasks in $\tilde{C}_0, \ldots, \tilde{C}_5$, and show that they enforce a special configuration on machines μ_{s_1}, μ_{s_2}, and μ_{s_3}. Then, we describe the assignment of the remaining chains and conclude the proof.

The assignment of tasks in $\tilde{C}_0, \ldots, \tilde{C}_5$ to machines is shown in Figure 4. All tasks in each chain except for two tasks are assigned to the same machine, whereas the remaining two tasks are assigned to two other mutually distinct machines. This assignment ensures that the shown schedule s is the only possible for any $\tilde{C}_0^1$-initial feasible schedule. Since each added chain contains exactly T_H tasks, fixing the start time of one task in a chain results in uniquely determined start times of all other tasks in this chain due to non-collision and 0-degeneracy constraints (7) and (8), respectively. Thus, the start times of the tasks in $\tilde{C}_0$ are uniquely determined because of $\tilde{C}_0^1$. Then, $\tilde{C}_1^{x+1}$ can only start at x since for any other choice, $\tilde{C}_0$ and $\tilde{C}_1$ would collide. Therefore, $s(\tilde{C}_1^r) = r - 1$ for all r. We proceed with the same reasoning, and conclude that $s(\tilde{C}_c^r) = r - 1$ for all $c \in \{0, \ldots, 5\}$, $r \in \{1, \ldots, T_H\}$.

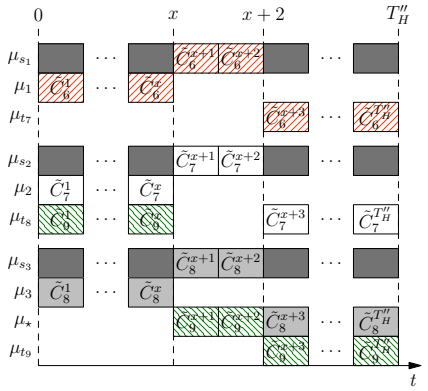


Fig. 5: Assignment of tasks in $\tilde{C}_6, \ldots, \tilde{C}_9$ to machines $\mu_{s_1}, \mu_{s_2}, \mu_{s_3}, \mu_{t_7}, \mu_{t_8}, \mu_{t_9}, \mu_1, \mu_2, \mu_3$, $\mu_\star$ and the only possible schedule of these tasks. Dark gray rectangles are the tasks fixed in Figure 4. The purpose of this configuration is to occupy interval $[0, x]$ on machines μ_1, μ_2, and μ_3 and $[x, {T_H}'']$ on $\mu_\star$.

The assignment of the tasks in chains $\tilde{C}_6, \ldots, \tilde{C}_9$, is shown in Figure 5. The first x tasks in each chain are assigned to the same machine μ_1, μ_2, μ_3 and μ_{t_8}, the next two tasks in each chain are assigned to the same machine $\mu_{s_1}, \mu_{s_2}, \mu_{s_3}$ and $\mu_\star$, and the rest of the tasks in each chain are assigned to the same machine $\mu_{t_7}, \mu_{t_8}, \mu_\star$, and μ_{t_9}, respectively. Note that machines μ_1, μ_2, μ_3, $\mu_\star$ are free exactly at times indicated by Constraint (11) (compare with Figure 2).

Using the same reasoning as in the previous part of the proof, we can prove that the configuration shown in Figure 5 is the only possible for any $\tilde{C}_0^1$-initial feasible schedule. We shall now verify that $\mathcal{I}'$ satisfies the requirements of the lemma, i. e., that it simulates configuration constraints (11) in $\mathcal{I}''$. We prove both implications:

"$\mathcal{I}''$ satisfies configuration constraints (11) $\implies$ $\mathcal{I}'$ is feasible": Let s'' be the schedule of $\mathcal{I}''$ satisfying Constraints (11). We define $s'(\tau_i) = s''(\tau_i)$ for $\tau_i \in \mathcal{T}$ and use the idea from Figures 4 and 5 for the remaining auxiliary tasks. By the construction, s' is 0-degenerated and has no collisions.

"$\mathcal{I}'$ is feasible $\implies$ $\mathcal{I}''$ satisfies configuration constraints (11)": Let s' be a feasible schedule of $\mathcal{I}'$. We assume without loss of generality that s' is $\tilde{C}_0^1$-initial, otherwise we make it such using Lemma 1. We define $s''(\tau_i) = s'(\tau)$ for all $\tau \in \mathcal{T}$. Due to s' being $\tilde{C}_0^1$-initial, we know that it must schedule tasks from the auxiliary chain as displayed in Figure 5. Then Constraints (11) become just the no-collision constraints on s' between tasks from $\mathcal{T}$ and tasks from $\tilde{C}_6$, $\tilde{C}_7$, $\tilde{C}_8$ and $\tilde{C}_9$. Since these hold by construction, we are done. □

Observation. *If Lemma 2 is used on $\mathcal{I} = (\mathcal{T}, \mathcal{M}, \mathcal{C}, T_H)$, the resulting instance has $\mathcal{O}(|\mathcal{T}| + T_H)$ tasks, and the time complexity of constructing it is also $\mathcal{O}(|\mathcal{T}| + T_H)$.*

Theorem 1. *There exists a polynomial reduction from JS3 to PSP$_{com}$.*

Proof. Let $\mathcal{J} = (\hat{\mathcal{T}}, \hat{\mathcal{M}}, \hat{\mathcal{C}}, L)$ be a JS3 instance. Let $\mathcal{I} = (\mathcal{T}, \mathcal{M}, \mathcal{C}, T_H)$ be the instance obtained from $\mathcal{J}$ by the naive reduction described in Section 3.3. We consider two cases:

- The conditions of Lemma 2 do not hold. Then either $|\mathcal{C}| = 1$, or $T_H = L \leq 2$. In both cases, we may decide the feasibility of the instance in polynomial time.

- If $T_H > |\mathcal{T}|$, the instance is always feasible.
- Otherwise, we use Lemma 2 to obtain in time $\mathcal{O}(|\mathcal{T}| + T_H) = \mathcal{O}(|\mathcal{T}|)$ an instance $\mathcal{I}'$ that is feasible if and only if there is a feasible schedule s of $\mathcal{I}$ satisfying Constraint (10), which is equivalent to $\mathcal{J}$ having a schedule satisfying Constraint (9).

Corollary 1. *PSP$_{com}$ and PSP$_{gen}$ are strongly NP-hard.*

4 Heuristic Approach

We propose a local search heuristic to solve PSP$_{gen}$ described in Section 2. The algorithm first generates a (possibly infeasible) schedule and then, for a fixed number of iterations, creates a new schedule based on the current one. The old schedule is swapped with the new one if the quality of the latter is not worse in terms of degeneracy.

Schedule representation In PSPs, schedules are usually represented directly by task start times. However, inspired by the disjunctive graphs in the job shop scheduling problem [2], we represent a schedule as a *queue* of tasks. We reconstruct the schedule whenever we want to compute its degeneracy. A queue Q is a totally ordered list of tasks: $Q = (\tau_{\pi(1)}, \ldots, \tau_{\pi(n)})$, where π is a permutation of $\{1, \ldots, n\}$. Let $Q(i) = \tau_{\pi(i)}$.

Reconstruction A *reconstruction function* $f : Q \to s$ is a function that takes Q as an argument and returns a schedule (i.e., task start times). An important property of reconstruction functions is that a small change in a queue results in a small change in the reconstructed schedule. This property is the main motivation for the design decisions of this function. Moreover, two schedules represented by start times might look very different, but in fact they might be nearly identical from the "search space" viewpoint.

The reconstruction starts with an empty schedule (i.e., $s(\tau_i) = \emptyset$ for all τ_i) and it schedules task $Q(\ell)$ in the ℓ-th iteration such that non-collision (1) and precedence (3) (if the predecessor of $Q(\ell)$ is already scheduled) constraints are respected. Once a task is scheduled, it remains fixed until the end of the reconstruction. Since tasks may not be in their precedence order in a queue Q, missing precedence constraints are handled at the end of the reconstruction by the shifting procedure from Lemma 1 in Section 2.2. Allowing broken precedence constraints in Q gives the heuristic the freedom to decide that a particular precedence relation should be broken. Note that the partial schedule may not be extendable to a feasible schedule. Then, we return $\emptyset$ instead of the schedule.

We next describe the strategy to assign the start time for task $Q(\ell) = \tau_i$ in details. For each machine μ_q, we maintain the time $\mathrm{head}(\mu_q) = \max\{s(\tau_i) + p(\tau_i) \mid s(\tau_i) \neq \emptyset, m(\tau_i) = \mu_q\}$ at which the last task assigned to this machine (scheduled so far) finishes executing. We schedule task τ_i at the earliest time t such that: 1. it is not sooner than the last task on the corresponding machine, i.e., $t \geq \mathrm{head}(m(\tau_i))$; 2. non-collision constraints are satisfied, i.e., no tasks are already scheduled in intervals $[t + k \cdot T(\tau_i), t + k \cdot T(\tau_i) + p(\tau_i))$; and 3. $t \geq s(\tau_j) + p(\tau_j)$, where τ_j is the predecessor of τ_i (if $\tau_j = \emptyset$ or $s(\tau_j) = \emptyset$, we set $s(\tau_j) = -\infty$). If no such t exists, we return $\emptyset$. Due to the efficient schedule representation, finding the smallest t is done quickly. However, in this work we do not elaborate on it due to the space limit.

Neighbor function A neighbor function takes the current Q and modifies it. The resulting queue is a neighbor. Multiple functions have been tested, out of which the most notable are presented in Table 1. Note that in this work, all random choices are assumed to be in a form of the uniform distribution. The idea used by the most successful functions,

as shown later in the experiments, is to rearrange the tasks of a randomly chosen chain such that they go in the order of precedence relations (`chain_sort` function). For example, queue $Q = (\cdots, C_c^2, \cdots, C_c^3, \cdots, C_c^1)$ becomes $Q' = (\cdots, C_c^1, \cdots, C_c^2, \cdots, C_c^3)$ for a chosen chain C_c.

Table 1: List of neighbor functions

Name	Description
`swap_uniform`	Swap two randomly chosen tasks
`chain_uniform`	Perform `swap_uniform` for two tasks of a random chain
`chain_adjacent`	Swap two adjacent tasks in a random chain
`swap_ran`	Perform one of the previous three randomly
`chain_sort`	Rearrange tasks of a random chain to satisfy precedence relations
`chain_sort_patient`	Perform `chain_sort` on a chain with broken precedences
`chain_sort_loc`	Perform `chain_sort` until the neighbor degeneracy decreases
`combined_local`	Perform `chain_sort_loc`; if all chains are sorted, do `swap_ran`
`combined_random`	Perform `swap_ran` or `chain_sort_patient` randomly
`switch`	In the initial iterations, perform `chain_sort_loc`, later perform `combined_random`

In `chain_sort_loc`, the change is discarded if the degeneracy of a queue with the rearranged tasks is worse, and the change is kept if the degeneracy has not changed. In one iteration, we apply this function until the neighbor degeneracy becomes less than the initial degeneracy or until the queue is sorted. The intuition behind is that although sorting the current chain may not change the degeneracy, it may nevertheless improve the solution. Finally, `switch` is the only function that change its strategy with iterations. Here, `chain_sort_loc` is used until all chains are sorted. From the first iteration when there is no unsorted chain onward, we always perform `combined_random`. The idea is that in the initial burn-in phase, we want to sort as many chains as possible. Once we achieve that, we want to avoid getting stuck in a local minimum. Thus, we switch to `combined_random`, allowing more unpredictable changes.

Initialization To find an initial Q, we sort all tasks in the ascending order using a custom comparison $\prec$ defined as follows ("$<$" denoting the lexicographic comparison on ordered pairs): $\tau_i \prec \tau_j \iff (T(\tau_i), -p(\tau_i)) < (T(\tau_j), -p(\tau_j))$. In other words, we place tasks with smaller periods first, and in case of a tie, we place longer-executing tasks first.

4.1 Algorithm Overview

The proposed local search heuristic is presented in Algorithm 1. It is parameterized by the choice of two neighbor functions, $\mathcal{N}$ and $\mathcal{N}_{+\infty}$, and the running time $time\_limit$. We use function $\mathcal{N}$ when the current schedule is feasible (i.e., the reconstruction managed to find a non-collision schedule for a queue in the corresponding iteration), whereas we use $\mathcal{N}_{+\infty}$ when the schedule is not feasible. The parameter $time\_limit$ can be chosen based on the time that is acceptable by the system designer. The algorithm starts by generating an initial queue on Line 1. Then, the heuristic repeats the following procedure for a fixed number of iterations (finishing early if $\delta(Q) = 0$ (Line 8)). First, the schedule is reconstructed, and its degeneracy is computed (Line 3). If the reconstructed schedule is infeasible or the reconstruction fails, we set $\delta(Q) = +\infty$. In this case, as mentioned earlier, a neighbor is generated by calling $\mathcal{N}_{+\infty}(Q)$. Otherwise, it is done using $\mathcal{N}(Q)$. If the new schedule is not worse than the old one (Line 6), the old schedule

Input: $\mathcal{N}, \mathcal{N}_{+\infty}, time\_limit$
1 Q.initialize();
2 **while** *elapsed time* $< time\_limit$ **do**
3 $s = Q$.reconstruct(), $\delta_1 = \delta(s)$;
4 **if** $\delta_1 = +\infty$ **then** $Q' = \mathcal{N}_{+\infty}(Q)$ **else** $Q' = \mathcal{N}(Q)$;
5 $s' = Q'$.reconstruct(), $\delta_2 = \delta(s')$;
6 **if** $\delta_2 \leq \delta_1$ **then**
7 $Q = Q'$;
8 **if** $\delta(Q)' = 0$ **then**
Output: s
9 **end**
10 **end**
11 **end**
Output: s

Algorithm 1: Local search heuristic

is swapped with the new one (Line 7), and the next iteration continues working with this updated schedule. The idea behind using non-strict inequality on Line 6 is that we want to explore as much solution space as possible and not get stuck in a local optimum.

4.2 Experimental Results

Experimental setup We randomly generated 7 sets of problem instances differing in the following parameters: minimum and maximum number of tasks in each chain, $l(C)_{\min}$ and $l(C)_{\max}$, respectively, and the upper limit on the utilization of each machine, $\sigma_{\max} = \max_{\mu \in \mathcal{M}} \sigma_\mu$. The utilization is defined as $\sigma_\mu = \sum_{\tau:\, m(\tau)=\mu} \frac{p(\tau)}{T(\tau)}$, which is the fraction of time this machine is occupied. In Sets 1-6, the utilization of 95% of the machines lies in the interval $[\sigma_{\max} - 0.2, \sigma_{\max}]$. In Set 7, the utilization of more than 90% of the machines equals 1, and the utilization of the rest of the machines lies in the interval $[0.96, 1)$. The number of tasks in all sets varies from 100 to 9 000, with more than half of the instances having more than 1 000 tasks in each set. Each of these seven test sets consists of 4 groups of 200 problem instances, with the following parameters ($\mathcal{P}$ is the set of periods, $|\mathcal{M}|$ is the number of machines): 1. $\mathcal{P} = \{2^0, 2^1, \ldots, 2^{10}\}$, $|\mathcal{M}| = 5$, 2. $\mathcal{P} = \{2^0, 2^1, \ldots, 2^{10}\}$, $|\mathcal{M}| = 10$, 3. $\mathcal{P} = \{2, 10, 20, 100, 200, 1000, 2000, 4000\}$, $|\mathcal{M}| = 5$, and 4. $\mathcal{P} = \{8, 16, 64, 256, 1024, 2048\}$, $|\mathcal{M}| = 5$. The generation procedure ensures that each of the 5 400 generated problem instances has a 0-degenerated schedule.

We ran the heuristic 6 times on each instance with $\mathcal{N}_{+\infty} =$ `chain_uniform` with a different random seed. For each run of the heuristic, we set $time\_limit$ to 3 minutes. We choose the best of these runs as a result. Finally, we performed the experiments on a machine equipped with four Intel(R) Xeon(R) Silver 4110 CPUs, all clocked at 2.10GHz, each having 8 cores and 16 threads. Similar results were achieved on a middle-end laptop.

Results Table 3 presents the average degeneracy of the heuristic algorithm with the most promising neighbor functions on problem instances of Set 3, which we consider moderately difficult. The percentage of problem instances for which the heuristic found a feasible solution is 98.98. The degeneracy of the heuristic with all neighbor functions is relatively small, however `combined_local` and `switch` show the best results

Table 2: Parameters of the generated sets

Name	$\sigma_{\max}$	$l(C)_{\min}$	$l(C)_{\max}$
Set 1	0.8	5	15
Set 2	0.9	2	5
($\star$) Set 3	0.9	5	15
Set 4	0.9	15	25
Set 5	0.93	5	15
Set 6	0.96	5	15
Set 7	1	5	15

Table 3: Results on Set 3 with various functions $\mathcal{N}$

Neighbor function	average degeneracy
chain_sort	1.49
chain_sort_patient	1.49
combined_local	0.13
combined_random	1.93
switch	0.12

Table 4: Results of the heuristic with $\mathcal{N} =$ `combined_local` on different sets

Set	% of solved instances	Average degeneracy
Set 1, $\sigma_{\max} = 0.8$, medium chains	99.56	0
Set 5, $\sigma_{\max} = 0.93$, medium chains	97.98	0.49
Set 6, $\sigma_{\max} = 0.96$, medium chains	96.65	1.34
Set 7, $\sigma_{\max} = 1$, medium chains	54.50	58.52
Set 2, $\sigma_{\max} = 0.9$, short chains	98.56	0.01
Set 3, $\sigma_{\max} = 0.9$, medium chains	98.98	0.13
Set 4, $\sigma_{\max} = 0.9$, long chains	98.90	0.96

with statistically insignificant difference. Since the former is conceptually easier, we use it in the second experiment.

Table 4 shows the percentage of problem instances for which a feasible solution was found and the average degeneracy for each test set. Note that Sets 1, 3, 5, 6, and 7 have equal parameters except for $\sigma_{\max}$. As expected, increased machine utilization leads to larger degeneracy with the significant difference for Set 7 with $\sigma_{\max} = 1$. On the other hand, Sets 2, 3, and 4 differ in the chain length only. Instances with longer chains have larger average degeneracy, but the results do not suggest such a dramatic increase as in Set 7.

5 Conclusions and Future Work

This paper addresses the periodic scheduling problem PSP_{gen} with chains of precedence relations. In this problem, periodic tasks are scheduled in time on dedicated machines so that at any moment, at most one task is executed on each machine. We define a degeneracy of a chain as the number of broken precedence relations within the time window of one period. The problem is to find a schedule with the minimum degeneracy.

We prove that this problem is strongly NP-hard even when restricted to unit processing times, a single period, and 16 machines (called PSP_{com}), by a reduction from a variant of a job shop scheduling problem. In this reduction, by introducing auxiliary tasks, machines, and chains, we prove that the entire space of solutions of PSP_{com} is equivalent to the space of solutions respecting the job shop constraint on the total length of the schedule (missing in PSP_{com}). Furthermore, we present a local search heuristic algorithm that solves PSP_{gen}. We generated 5 400 problem instances allowing 0-degenerated solutions. On them, we experimentally demonstrate the soundness of our algorithm and show that it can successfully solve 92% of the instances with average resulting degeneracy of 8.78, each in a few minutes.

As future work, it would be interesting to explore the complexity of PSP_{com} on a smaller number of machines. Whereas the PSP_{com} with a common period on one machine is polynomially solvable (by placing one task after another of the first chain in precedence order, then the second chain, etc., in an arbitrary order), scheduling on two and more resources (up to 15) is an open question.

Acknowledgments

Research leading to these results has received funding from the EU ECSEL Joint Undertaking and the Ministry of Education of the Czech Republic under grant agreement 826452 (project Arrowhead Tools).

References

1. Bar-Noy, A., Bhatia, R., Naor, J.S., Schieber, B.: Minimizing service and operation costs of periodic scheduling. Mathematics of Operations Research **27**(3), 518–544 (2002)
2. Blażewicz, J., Pesch, E., Sterna, M.: The disjunctive graph machine representation of the job shop scheduling problem. European Journal of Operational Research (2000)
3. Cai, Y., Kong, M.: Nonpreemptive scheduling of periodic tasks in uni-and multiprocessor systems. Algorithmica (1996)
4. Dvorak, J., Hanzalek, Z.: Multi-variant time constrained FlexRay static segment scheduling. 2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014) pp. 1–8 (2014)
5. Dvořák, J., Hanzálek, Z.: Multi-variant scheduling of critical time-triggered communication in incremental development process: Application to flexray. IEEE Transactions on Vehicular Technology **68**(1), 155–169 (2018)
6. Eisenbrand, F., Hähnle, N., Niemeier, M., Skutella, M., Verschae, J., Wiese, A.: Scheduling periodic tasks in a hard real-time environment. In: International colloquium on automata, languages, and programming. pp. 299–311. Springer (2010)
7. Eisenbrand, F., Kesavan, K., Mattikalli, R.S., Niemeier, M., Nordsieck, A.W., Skutella, M., Verschae, J., Wiese, A.: Solving an avionics real-time scheduling problem by advanced ip-methods. In: European Symposium on Algorithms. pp. 11–22. Springer (2010)
8. Graham, R.L., Lawler, E.L., Lenstra, J.K., Kan, A.R.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals of discrete mathematics **5**, 287–326 (1979)
9. Jacobs, T., Longo, S.: A new perspective on the windows scheduling problem. arXiv preprint arXiv:1410.7237 (2014)
10. Jeffay, K., Stanat, D.F., Martel, C.U.: On non-preemptive scheduling of periodic and sporadic tasks. In: IEEE real-time systems symposium. pp. 129–139. US: IEEE (1991)
11. Korst, J., Aarts, E., Lenstra, J., Wessels, J.: Periodic multiprocessor scheduling. PARLE '91 Parallel Architectures and Languages Europe **505**, 166–178 (Jun 1991)
12. Korst, J., Aarts, E., Lenstra, J.K.: Scheduling periodic tasks. INFORMS journal on Computing **8**(4), 428–435 (1996)
13. Lenstra, J.K., Kan, A.R.: Computational complexity of discrete optimization problems. In: Annals of Discrete Mathematics, vol. 4, pp. 121–140. Elsevier (1979)
14. Nawrocki, J.R., Czajka, A., Complak, W.: Scheduling cyclic tasks with binary periods. Information processing letters **65**(4), 173–178 (1998)
15. Ogata, K.: Discrete-time control systems, vol. 2. Prentice Hall Englewood Cliffs, NJ (1995)
16. Oliver, R.S., Craciunas, S.S., Steiner, W.: IEEE 802.1 Qbv gate control list synthesis using array theory encoding. In: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 13–24. IEEE (2018)
17. Wei, W., Liu, C.: On a periodic maintenance problem. Operations Research Letters **2**(2), 90–93 (1983)