Improved algorithms for tool switching problems with multiple objectives

Martina Furrer  Torsten Mütze

Institute of Theoretical Computer Science, ETH Zurich, Switzerland

martfur@student.ethz.ch  torsten.muetze@inf.ethz.ch

ABSTRACT. We present an algorithmic framework that improves over several known results for a family of optimization problems on flexible manufacturing systems that involve multiple objectives and that have received considerable attention in the literature. We demonstrate the usefulness of our algorithm by theoretical analysis and by experiments with large real-world and random instances.

KEYWORDS: flexible manufacturing system, tool switching, algorithm

1. INTRODUCTION

The main contribution of this paper is an algorithmic framework that improves over several known results for a family of optimization problems on flexible manufacturing systems that involve multiple objectives and that have received considerable attention, mainly due to their relevance in numerous industrial applications [Bar88, TD88a, TD88b, CKOS94, Cra97, CvdK99, PF95, PF00, Ste83, GSS93, LZ93, HLMS98, KIL01, SH02, Müt14, ABZ15]. Before discussing these applications and references in detail, we state the problem in terms of flexible manufacturing systems (we shall see later that our model also captures a wide range of other assignment problems with a different background). The problem formulation is illustrated in Figure 1.

1.1. Problem statement. A flexible manufacturing system has $m$ slots, each of which can hold one of $n$ available tools at a time. We are given a sequence of $\ell$ jobs $J_1, \ldots, J_\ell$, where each job $J_i \subseteq [n] := \{1, \ldots, n\}$ describes the set of tools that need to be mounted to the slots to process the job. The jobs are processed in the given order, i.e., job $J_i$ is processed in the $i$-th step. Denoting the tools of job $J_i$ by $\alpha_1, \ldots, \alpha_k$ (the number of tools in each job may be different), this is achieved by assigning $\alpha_1, \ldots, \alpha_k$ to $k$ of the slots, and the remaining $m - k$ slots remain unused in this step. Such an assignment of tools to slots for each of the jobs is a plan, described by a $\ell \times m$ matrix where rows correspond to jobs and columns correspond to slots. To evaluate the quality of a plan, we consider specific configurations in the plan, so-called tool switches and stops (see Figure 1). A tool switch occurs whenever a tool $\alpha$ is assigned to a slot in some step $i_\alpha$, and a different tool $\beta$ is assigned to the same slot in a later step $i_\beta$ (and the slot is unused in between). Tool switches are a purely combinatorial property of the plan (e.g., the plan $\sigma$ in Figure 1 has 5 tool switches, whereas the plan $\sigma'$ has only 2 tool switches). To further evaluate the relevance of tool switches, we take into account setup times of the tools and processing times of the jobs. Specifically, each tool $\alpha \in [n]$ has an integer setup time $s(\alpha)$ representing the number of consecutive time units it takes to mount tool $\alpha$ to a slot before it can be used, and each job $J_i$ has an integer processing time $p_i$ representing the number of consecutive time units it takes to process the job. We say that a switch from tool $\alpha$ to $\beta$ as above is critical, if the sum of all processing times $p_i$ with $i_\alpha < i < i_\beta$ is strictly smaller than the setup time $s(\beta)$ (e.g., the switch from tool 4 to 3 on slot 1 in the plan $\sigma$ is critical).

1 This work was performed as a research project in cooperation with Supercomputing Systems AG, Zurich, Switzerland. See http://www.scs.ch/optimizer for a game-like demonstration of the problem and the industrial applications discussed in this paper.
Problem instance (left) with $\ell = 6$ jobs, $m = 3$ slots and $n = 4$ tools, and three different plans (middle and right). Tool switches in a plan are indicated by vertical lines with square endpoints, where critical tool switches are drawn in black and non-critical tool switches in grey. Stops are indicated by horizontal dashed lines with a circular endpoint. Resolving the critical tool switches in $\sigma$ requires two stops, the first one either after $J_1$ or $J_2$, and the second one either after $J_4$ or $J_5$. The plans $\sigma'$ and $\sigma''$ have the minimal number of stops, $\sigma'$ also minimizes tool switches ($\sigma$ minimizes neither stops nor tool switches).
In contrast to that, mailroom insert planning as discussed in [ABZ15] is an application of our model where finite setup times are the norm. Here, a line of feeders (=slots) inserts various sets (=jobs) of different advertising brochures (=tools) into a newspaper. Each feeder can be accessed individually to exchange a batch of brochures for another one (=a tool switch) without stops if the feeder is unused sufficiently long (=a non-critical tool switch). The instance in Figure 1 is of this type: E.g., the switch from tool 2 to 1 in slot 3 of the plan $\sigma'$ does not require a stop (it can be performed during operation), which is typical for such an application (but untypical for the single magazine setting of CNC machines). Let us explicitly mention three more applications of this type mentioned in [Müt14] and [LZ93] that emphasize the versatility of our model. In all these examples the goal is to minimize costs incurred by tool switches and by stops that interrupt the production: The first example is in chemical manufacturing, where various chemicals (=jobs), each consisting of a specific set of constituents (=tools), are produced by feeding the constituents to the mixing and reaction stage through a number of supply pipes (=slots), and costs are incurred from retooling and cleaning a pipe whenever a constituent on the pipe changes. The second example is a pharmaceutical packaging facility which bundles together different types of pills (=tools) into patient or region specific boxes (=jobs), where the pills are stored in containers that can be hooked to the feeding holes (=slots) of the packaging unit, and costs are incurred by every container exchange. The third example is the automated assembly of printed circuit boards, where a line of feeders (=slots) mounts various sets (=jobs) of different types of electronic components (=tools) to the circuit board, and costs are incurred whenever the type of component assigned to a feeder changes.

In several of the previously mentioned scenarios, the ordering of jobs is not fixed (as in our model), but also part of the decision process (the goal then is to determine an optimal ordering of the jobs and an optimal assignment of tools to slots). Unfortunately, all reasonable objectives, in particular minimizing tool switches or minimizing stops, are NP-hard in this more general setting already for $m = 2$ slots [CKOS94, ABZ15]. In view of this, finding an optimal assignment of tools to slots becomes the more important once a (heuristically computed) job ordering has been determined in the first phase (put differently, solving the assignment problem optimally allows us to consider the ordering problem separately). In fact, researchers have proposed numerous heuristics/approximation algorithms to tackle the ordering problem [Bar88, TD88a, TD88b, CKOS94, CvdK99, PF95, HLMS98].

Observe that similar planning problems involving cleanup times rather than setup times are captured by our model by reversing the time orientation. Note also that if setup operations can only be performed during stops (as in the single magazine setting of CNC machines), or if the time required for stopping and restarting the manufacturing system is larger than the setup times, minimizing stops is equivalent to minimizing the production makespan (see also the remarks at the end of this paper).

1.3. Our results. The main contribution of this work is an algorithm that computes an optimal plan for the lex-minimization of stops and tool switches. In the following we discuss the performance of our algorithm and how it improves upon the previously mentioned results.

(i) For general instances (with arbitrary setup and processing times), the running time of our algorithm is bounded by $\tilde{O}(\ell^2 m^3(\min\{m, \lfloor n/2 \rfloor\}))$ (the $\tilde{O}()$ hides logarithmic factors as customary; $\binom{n}{k}$ denotes the binomial coefficient). For a constant number of slots $m = c$ (as in [Bar88, TD88a, CKOS94, PF95]) this gives the runtime bound $\tilde{O}(\ell^2 n^c)$. In this case we obtain a positive answer to the problem raised in [ABZ15] to design an efficient algorithm for the lex-minimization of stops and tool switches. Specifically, we get rid of the exponential dependence of the running time on $\ell$ that is incurred when exhaustively enumerating all possible stop placements and minimizing tool switches for each of them (as suggested in [ABZ15]).
(ii) In several interesting cases, characterized by certain frequency patterns of tools in the instance (details can be found in Section 4.3.3), the runtime bound can be improved to $\tilde{O}(\ell^2 m^{c'})$ for some constant $c' \geq 2$ (this bound is polynomial in all three variables $\ell$, $m$ and $n$).

To demonstrate that our algorithm and the above-mentioned runtime bounds are practically useful, we conducted experiments with real-world and random instances where $\ell$, $m$ and $n$ are in the order of magnitude of hundreds to thousands. In all cases, our algorithm computed optimal solutions in few seconds on a standard desktop computer (see Section 5).

Our problem model with arbitrary setup and processing times is quite versatile and subsumes several special cases that are interesting in their own right. Our algorithm provides a unified approach to solve all of these special problems and compares favorably to specialized algorithms from the literature:

(iii) If we set $s(\alpha) := \infty$ for every tool $\alpha \in [n]$, then every tool switch is critical, capturing the single magazine setting of CNC machines. We thus provide the first (provable and efficient) algorithm to solve the lex-minimization of stops and tool switches for this scenario, complementing earlier experimental work [KIL01]. The runtime bounds stated under (i) and (ii) hold here as well, and instances with $s(\alpha) = \infty$ were also speedily solved in our experiments.

(iv) For unit setup and processing times, i.e., $s(\alpha) = 1$ for every tool $\alpha \in [n]$ and $p_i = 1$ for every job $i \in [\ell]$, the runtime bound of our algorithm can be improved to $O(\ell m)$, matching the results presented in [ABZ15, Corollary 8] and [Müt14, Theorem 6] for this special case (this is the main algorithmic result from [Müt14], and only the case of unit setup and processing times is considered in that paper).

(v) If our goal is to minimize tool switches as the only objective (and neglect stops entirely), then we may set $s(\alpha) := 0$ for every tool $\alpha \in [n]$ (then there are no critical tool switches and no stops), and obtain an $O(\ell m)$ algorithm for minimizing tool switches. This matches the running time of the algorithms proposed in [TD88a, CKOS94] and [Müt14]. (The algorithm from [PF95] runs in time $O(\ell^2 m^3)$, but can also handle more general tool switching costs.)

(vi) If our goal is to minimize stops as the only objective (and neglect tool switches entirely), then we may run only the first phase from our general algorithm, and obtain an $O(\ell m)$ procedure for minimizing the number of stops, a polynomial improvement over the $O(\ell^2 m)$ solution proposed in [ABZ15, Algorithm 1]. We may further specialize by setting $s(\alpha) := \infty$ or $s(\alpha) := 1$ for every tool $\alpha \in [n]$ (see (iii) and (iv) above).

1.4. Sketch of our algorithm. In the instance from Figure 1 the one unavoidable stop can be placed at two possible positions (shown on the right hand side of the figure). In general, the number of ways stops can be placed grows exponentially with the number of jobs, and a priori we do not know which stop placement minimizes the number of tool switches. Our algorithm that solves the lex-minimization of stops and tool switches therefore proceeds in two phases: In the first phase it computes a compact (i.e., polynomial size) representation of all possible stop placements. In the second phase it employs a branch-and-bound strategy to enumerate a small subset of all possible stop placements, computing the minimal number of tool switches for each of them, then taking the global minimum. A more specific description of the main ideas of our algorithm will be given in Section 4.1 below, once all formal notions are available.

1.5. Outline of this paper. After formalizing the problem statement in Section 2 we explain the two phases of our algorithm in Section 3 and Section 4. We report on our computational experiments in Section 5 and we indicate some directions for further research in Section 6.
In this section we introduce several crucial definitions and formalize the problem statement. The example at the end of this section illustrates the most important definitions. Table 1 summarizes notations that are used throughout this paper (some defined only later). Here and throughout the rest of the paper, important definitions are highlighted using a bold italic font.

**Problem instances.** For any sequence \( x = (x_1, \ldots, x_N) \) we define first(\( x \)) := \( x_1 \) and last(\( x \)) := \( x_N \) and we denote by \( |x| \) the number of entries of \( x \), \( |x| := N \). For integers \( a \) and \( b \) with \( a \leq b \) we write \([a, b] := \{ a, a+1, \ldots, b \} \), and as before \([a] := [1, a] \). We denote a problem instance as a 5-tuple \( \mathcal{P} = (n, J, m, s, p) \) with the number of tools, the job sequence \( J = (J_1, \ldots, J_\ell) \) where \( J_i \subseteq [n] \) for all \( i \in [\ell] \), the number of slots \( m \), setup times \( s(\alpha) \in \{0, 1, 2, \ldots\} \) for all \( \alpha \in [n] \), and processing times \( p_i \in \{0, 1, 2, \ldots\} \) for all \( i \in [\ell] \).\(^2\) \ We assume w.l.o.g. that each tool appears in at least one job, i.e., \( \bigcup_{i \in [\ell]} J_i = [n] \). For the number of slots \( m \) we assume w.l.o.g. that \( \max_{i \in [\ell]} |J_i| \leq m < n \). The lower bound ensures that there are enough slots to process the largest job (otherwise no plan exists). The upper bound excludes the case where an optimal plan can be trivially computed simply by dedicating each slot to one particular tool, and assigning all occurrences of this tool to this slot. For any integer \( k \in [\ell] \) we write \( \mathcal{P}[k] := (n, (J_1, \ldots, J_k), m, s, (p_1, \ldots, p_k)) \).

**Stop programs, lex-minimization.** A particular placement of stops is a stop program, formalized as an increasing sequence \( x = (x_1, \ldots, x_N) \) with \( x_a \in [\ell - 1] \) for all \( a \in [N] \) and the interpretation that the \( a \)-th entry \( x_a \) is a stop between jobs \( J_{x_a} \) and \( J_{x_a+1} \). The last stop of \( x \) is last(\( x \)) := \( x_N \). If the stop program has no stops, \( x = () \), then we define last(\( x \)) := \( 0 \). We say that \( x \) is feasible for a given problem instance \( \mathcal{P} \), if there exists a plan \( \sigma \) such that \( x \) resolves all critical tool switches in \( \sigma \). We use \( \mathcal{X} \) to denote the set of all feasible stop programs for \( \mathcal{P} \). We define the minimal number of stops as \( S := \min_{x \in \mathcal{X}} |x| \), and use \( \mathcal{X}^{S} \subseteq \mathcal{X} \) to denote the set of all feasible stop programs with exactly \( S \) stops. Furthermore, we denote the number of tool switches in a plan \( \sigma \) by \( C(\sigma) \).

---

| \(|n| = \{1, 2, \ldots, n\} \) | set of tools | \((l_a, d_a)_{a \in [S], b \in [n_a], (u_a)_{a \in [S]}} \) | interval representation of \( \mathcal{X}^S \) |
| \(J = (J_1, \ldots, J_\ell) \) | sequence of jobs | \( T, T^+ \) | recursion trees |
| \(m\) | number of slots | \( V_i \) | nodes of \( T^+ \) on level \( i \) |
| \(s(\alpha), \alpha \in [n]\) | setup times of tools | \( w_i(\sigma) \) | pre-plan |
| \(p_i, i \in [\ell] \) | processing times of jobs | \( w(\sigma) = \min_{i \in [\ell]} w_i(\sigma) \) | width of pre-plan in row \( i \) |
| \(\mathcal{P} = (n, J, m, s, p) \) | problem instance | \( \lambda \) | width of pre-plan |
| \(x = (x_1, \ldots, x_N) \) | stop program | \( R_i, i \in [\ell] \) | block |
| \(\text{first}(x)/\text{last}(x) \) | first/last entry of \( x \) | \( G_i(\sigma), i \in [\ell] \) | reservations |
| \(|x|\) | length of \( x \) | \( r(\alpha), \alpha \in [n] \) | gaps |
| \(\mathcal{X} \) | set of feasible stop programs | \( \text{prev}(\alpha, i) \) | remaining setup times |
| \(S = \min_{x \in \mathcal{X}} |x| \) | minimal number of stops | \( \text{next}(\alpha, i) \) | previous occurrence of tool \( \alpha \) |
| \(\mathcal{X}^S \) | set of feasible stop programs | \( I(\alpha), \alpha \in [n] \) | next occurrence of tool \( \alpha \) |
| \(\sigma\) | plan (an \( \ell \times m \) matrix) | \( z(\alpha), \alpha \in [n] \) | all occurrences of tool \( \alpha \) |
| \(C(\sigma)\) | number of tool switches in \( \sigma \) | \( Z = \sum_{\alpha \in [n]} |z(\alpha)| \) | split sequence for tool \( \alpha \) |
| \(C(x)\) | min. number of tool switches for stop program \( x \) | | total number of splittings |
| \(C = \min_{x \in \mathcal{X}^S} C(x)\) | min. number of tool switches | | |

---

2With \( s(\alpha) := \infty \) we mean setting \( s(\alpha) \) to a large finite number, e.g. \( s(\alpha) := \sum_{i \in [\ell]} p_i \).
a fixed stop program \(x \in \mathcal{X}\) we define \(C(x)\) as the minimum of \(C(\sigma)\) over all plans \(\sigma\) for which \(x\) resolves all critical tool switches in \(\sigma\). We also define \(C := \min_{x \in \mathcal{X}^S} C(x)\).

The lex-minimization of stops and tool switches consists of computing a plan \(\sigma\) and a stop program \(x\) with the minimal number of stops \(|x| = S\), such that \(x\) resolves all critical tool switches in \(\sigma\) and \(\sigma\) has the minimal number of tool switches \(C(\sigma) = C\).

For an example consider again Figure 1. The figure shows a plan \(\sigma\) and two stop programs \(\tilde{x} = (1, 4)\) and \(\tilde{x} = (2, 4)\) that resolve all critical tool switches in \(\sigma\). The stop programs \(x' = (4)\) and \(x'' = (5)\) resolve all critical tool switches in \(\sigma'\) and \(\sigma''\), respectively. It follows that \(\tilde{x}, \tilde{x}'\) and \(\tilde{x}''\) are all feasible, while e.g. \((1, 3)\) is not feasible. In this example, \(S = 1\) and \(\mathcal{X}^S = \{x', x''\}\). Furthermore, the number of tool switches is \(C(\sigma) = 5\), \(C(\sigma') = C(x') = 2\) and \(C(\sigma'') = C(x'') = 3\). It follows that \(\sigma'\) is a lex-optimal plan and \(x'\) the corresponding stop program.

3. Computing a compact representation of \(\mathcal{X}^S\)

In this section we describe the first phase of our algorithm to solve the lex-minimization of stops and tool switches. As indicated before, in this phase we compute a compact (i.e., polynomial size) representation of the set \(\mathcal{X}^S\) of all feasible stop programs with the minimal number of stops. The algorithm to compute this representation consists of three nested functions, which we will explain from bottom to top in Sections 3.1–3.3 below. Only the top-level function presented in Section 3.3 will be called in the second phase of our algorithm (presented in Section 4). For each algorithm described in this section we first present the pseudocode along with some informal explanations, and then argue about its correctness and running time.

We begin by identifying a crucial obstacle that defeats a straightforward exhaustive enumeration approach to our optimization problem.

Remark 1. A fundamental problem we are facing when solving the lex-minimization of stops and tool switches, is that the set \(\mathcal{X}^S\) in general grows exponentially with the number \(\ell\) of jobs. It follows that an exhaustive enumeration of all stop programs from \(\mathcal{X}^S\) (and minimizing tool switches for each of them, then taking the global minimum) is prohibitively expensive, recall the remarks under (0) from the introduction. To underpin this problem with an example, consider again the instance from Figure 1 with jobs \(J = (J_1, \ldots, J_6)\), \(S = 1\) and \(|\mathcal{X}^S| = 2\). Concatenating \(k\) copies of \(J\) creates an instance with \(\ell = 6k\) jobs, \(S = k\) and \(|\mathcal{X}^S| = 2^k = 2^\ell/6\). In Section 5 we will encounter real-world instances where \(|\mathcal{X}^S|\) is in the order of magnitude of \(10^{20}\) and more.

In the remainder of this section we show how to compute a representation of all stop programs from \(\mathcal{X}^S\) as a data structure of size \(\mathcal{O}(\ell)\) (even though we may have \(|\mathcal{X}^S| = 2^{\mathcal{O}(\ell)}\)).

3.1. Computing the earliest last stop. The algorithm ELS() below forms an elementary building block of the algorithms described in later sections. Given a problem instance with jobs \(J_1, \ldots, J_k\), it computes the earliest last stop \(\min_{x \in \mathcal{X}^S} \text{last}(x) =: i\) among all feasible stop programs \(x\). It follows that if \(i > 0\), then any feasible stop program must contain a stop in the interval \([i, k - 1]\), and \(J_{i+1}, \ldots, J_k\) can be planned without stops. If \(i = 0\), then \(J_1, \ldots, J_k\) can be planned without any stops.

We proceed by explaining the algorithm ELS() and argue about its running time. For the reader’s convenience, Figure 2 shows two detailed examples how the algorithm operates.

3.1.1. Correctness of ELS(). The algorithm ELS() iterates through the instance backward in time, and considers the jobs \(J_i\) for \(i = k, k - 1, \ldots, 0\) (line A2). In this process the algorithm tracks the minimal number of entries that any plan for \(J_1, \ldots, J_k\) without stops must have (the algorithm never actually computes a plan). Specifically, any plan for \(J_1, \ldots, J_k\) without stops must contain enough entries to contain all tools in those jobs plus entries that are reserved for setups (entries that must be
empty to avoid a critical tool switch which would cause a stop). The algorithm tracks the required reserved entries via sets $R_i \subseteq [n]$, referred to as reservations. Each $R_i$ is disjoint from $J_i$ (see lines A1 and A6), and if $\alpha \in R_i$ then there is a job $J_i, i > i$, with $\alpha \in J_i$ and $s(\alpha) > \sum_{i=i+1}^{i-1} p_i$ (in words, setting up the tool $\alpha \in J_i$ requires reserving entries at least up to row $i$). In the algorithm, the inequality involving the setup and processing times is evaluated by defining $r(\alpha) := s(\alpha) - \sum_{i=i+1}^{i-1} p_i$ and by checking whether $r(\alpha) > 0$ (line A6). The time $r(\alpha)$ is the fraction of the setup time of $\alpha \in J_i$ that requires reserving entries in row $i$ of the plan and before. In our algorithm, $r(\alpha)$ is reset in line A5 and updated stepwise in line A4. Suppose $\alpha \in J_i$ is added to $R_{i-1}$ in some iteration $i = \tilde{i}$ in line A6. Then the tool $\alpha$ may not be added to $R_{\tilde{i}-1}$ in a later iteration $i = \tilde{i} < \tilde{i}$ in line A6 either because $r(\alpha) \leq 0$ or because $\alpha \in J_{\tilde{i}-1}$. The reason in the latter case is that the two occurrences of $\alpha$ in $J_{\tilde{i}-1}$ and $J_i$ can be assigned to the same slot, and then no entries in rows $\tilde{i} - 1, \tilde{i} - 2, \ldots$ need to be reserved for the setup of $\alpha \in J_i$ (the entries required for setting up $\alpha \in J_i$ are ‘cut off’ by the repeated occurrence of $\alpha$ in $J_{\tilde{i}-1}$).

Observe that the algorithm correctly returns $i$ in line A3. If the number of entries required for the tools in $J_i$ and the reservations in $R_i$ in any plan exceeds the number of slots $m$, a stop at $i$ is clearly unavoidable. On the other hand, if $|J_i| + |R_i| \leq m$, then it is not hard to see that a plan for the jobs $J_i, \ldots, J_k$ with no stops indeed exists (assign tools to the slots greedily backward in time, ‘cut off’ setups through repeated occurrences of the same tool whenever possible, and avoid stops by leaving entries of the plan that are required for setups empty).

3.1.2. Running time of ELS(). Denoting by $i$ the return value of the algorithm ELS() for an input instance with $k$ jobs and $m$ slots (as specified), the running time is bounded by $O((k - i)m)$. This follows from the observation that $|J_i| \leq m$ and $|R_i| \leq m$ for all $i \in [i, k]$ (recall the assumption max$_{i \in [k]} |J_i| \leq m$ for the input of the algorithm, the termination condition in line A3 and the definition in line A6), so each iteration of the main loop can be performed in time $O(m)$.

3.2. Computing the minimal number of stops. The algorithm MinSP() below forms the next intermediate step in computing a compact representation of the set $\mathcal{X}^S$ (recall that $\mathcal{X}^S$ denotes the set of all feasible stop programs with $S$ stops). Given a problem instance, the algorithm computes a feasible stop program $x$ with the minimal number of stops $|x| = S$, i.e., we have $x \in \mathcal{X}^S$. The computed stop program $x$ has the additional property that each stop is at the earliest possible position among all stop programs from $\mathcal{X}^S$.

We proceed by explaining the algorithm MinSP() and argue about its running time. Figure 2 shows how the algorithm operates on an example.

3.2.1. Correctness of MinSP(). The algorithm MinSP() iterates through the instance backward in time, and repeatedly calls the function ELS() on an initial subsection of the instance that consists
possible stop positions

\[
\begin{array}{c|c|c|c|c}
\text{job set} & 0 & 1 & 2 & \cdots \\
J_1 & \{1\} & \ast & \ast & \cdots \\
J_2 & \{2\} & \ast & \ast & \cdots \\
J_3 & \{3, 4, 5\} & \ast & \ast & \cdots \\
J_4 & \{4\} & \ast & \ast & \cdots \\
J_5 & \{5, 6\} & \ast & \ast & \cdots \\
J_6 & \{6, 7\} & \ast & \ast & \cdots \\
J_7 & \{3\} & \ast & \ast & \cdots \\
J_8 & \{2\} & \ast & \ast & \cdots \\
J_9 & \{1\} & \ast & \ast & \cdots \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\text{ELS} & 3 & 0 & 1 & \cdots \\
\text{ELS} & 3 & 0 & 1 & \cdots \\
\text{ELS} & 3 & 0 & 1 & \cdots \\
\text{ELS} & 3 & 0 & 1 & \cdots \\
\text{ELS} & 3 & 0 & 1 & \cdots \\
\text{ELS} & 3 & 0 & 1 & \cdots \\
\text{ELS} & 3 & 0 & 1 & \cdots \\
\text{ELS} & 3 & 0 & 1 & \cdots \\
\text{ELS} & 3 & 0 & 1 & \cdots \\
\end{array}
\]

Figure 2. Example for the algorithms ELS() and MinSP() for the instance \( P = (n, J, m, s, p) \), \( J = (J_1, \ldots, J_\ell) \), shown on the left with \( \ell = 9 \) jobs, \( m = 4 \) slots, \( n = 7 \) tools, setup times \( s(\alpha) = 3 \) for every tool \( \alpha \in [n] \) and unit processing times \( p_i = 1 \) for every job \( i \in [\ell] \).

Algorithm 2: MinSP(\( P \))

Input: an instance \( P = (n, J, m, s, p) \), \( J = (J_1, \ldots, J_\ell) \), with \( \max_{i \in [\ell]} |J_i| \leq m < n \)

Output: a stop program \( x \in X^S \) for the instance \( P \); moreover, any \( x' \in X^S \) has exactly one stop in the interval \( I_a := [x_a, x_{a+1} - 1] \) for any \( a \in [S] \)

B1 \( x := () \) /* initially no stops */
B2 \( k := \ell \) /* start from the end */
B3 while \( i := \text{ELS}(P[k]) \geq 0 \) do /* is another stop necessary? */
B4 \( x := (i, x) \) /* add unavoidable stop after \( J_i \) */
B5 \( k := i \) /* done with jobs \( J_{i+1}, \ldots, J_\ell \) */
B6 return \( x \)

only of the first \( k \) from all \( \ell \) jobs. The variable \( k \) is initialized with \( \ell \) (only in the first call to ELS(), all jobs are considered), and in each iteration reduced to the previous return value of ELS() (lines B2, B3 and B5 see also Figure 2). The final stop program \( x \) consists of all return values computed by ELS() except the last one (lines B1, B3 and B4 the last return value of ELS() is 0).

Let \( x = (x_1, \ldots, x_N) \) be the stop program computed by MinSP(\( P \)). We first argue that \( x \) is feasible for the instance \( P \). As \( x_1, \ldots, x_N \) are the return values of repeated calls to ELS(), there is a plan with no stops for each of the following subsections of the instance: jobs \( J_1, \ldots, J_{x_1} \), jobs \( J_{x_1+1}, \ldots, J_{x_2+1} \) for all \( a \in [N-1] \), and jobs \( J_{x_N+1}, \ldots, J_\ell \). Those plans can be combined, showing that \( x \) is indeed feasible for the entire instance. It remains to show that \( x \) has the minimal number of stops \( |x| = N = S \) (*), and that any \( x' \in X^S \) has exactly one stop in the interval \( I_a := [x_a, x_{a+1} - 1] \) for any \( a \in [S] \) (**). To avoid case distinctions, we introduce the artificial value \( x_{N+1} := \ell \) (without changing \( |x| \)). We fix an arbitrary feasible stop program \( x' \) for the instance \( P \). Observe that for any \( a \in [N] \), the stop programs \( (x_1, \ldots, x_a) \) and \( (x'_1)_{i \geq 1} : x'_{i} < x_{a+1} \) are both feasible for \( P_{x_{a+1}} \). Moreover, by the correctness of ELS() we have \( x_a = \min \text{last}(\bar{x}) \) where the minimum is over all feasible stop programs \( \bar{x} \) for \( P_{x_{a+1}} \). It follows that \( x' \) has at least one stop in the interval \( I_a \) for any \( a \in [N] \). We conclude that \( |x| = |x'| \), proving (*). Observe further that if \( |x'| = S \), i.e., if \( x' \in X^S \), then \( x' \) has exactly one stop in the interval \( I_a \) for any \( a \in [S] \), proving (**).
3.2.2. Running time of \textsc{Minsp}(). Let \( x = (x_1, \ldots, x_S) \) be the stop program computed by \textsc{Minsp}(). The running time of \textsc{Minsp}() is determined by the running times of the calls to \textsc{Els}(), which sum to \( O\left( (\ell - x_S) + (x_S - x_{S-1}) + \cdots + (x_2 - x_1) + (x_1 - 0) \right) \cdot m) = O(\ell m) \).

3.3. Computing the interval representation of \( \mathcal{X}^S \). In this section we combine the two algorithms presented in the previous sections to compute a compact representation of \( \mathcal{X}^S \). We begin by defining a data structure of size \( O(\ell) \) that represents the set \( \mathcal{X}^S \) (recall that \( \ell \) denotes the number of jobs). This data structure is referred to as the interval representation of \( \mathcal{X}^S \). We then establish various important monotonicity properties of this data structure (Lemma 2 and Lemma 3 below), and finally present an algorithm that computes it efficiently. Figure 3 illustrates the concepts introduced in this section.

For the following arguments, we consider a fixed instance \( \mathcal{P} = (n, J, m, s, p) \), \( J = (J_1, \ldots, J_\ell) \). From the previous section (see the output of the algorithm \textsc{Minsp}()) we know that there are disjoint intervals \( I_a, a \in [S] \), such that for any \( x \in \mathcal{X}^S \) and any \( a \in [S] \) we have \( x_a \in I_a \) (the \( a \)-th stop of \( x \) is contained in the interval \( I_a \); see Figure 2). We now describe the stop programs from \( \mathcal{X}^S \) more precisely by specifying tight lower and upper bounds for the possible stop positions. Specifically, we recursively define sequences of intervals \( l_a, l_{a,b}, u_a, u_{a,b} \) with \( a \in [S], b \in [\nu_a] \) and \( l_{a,b}, u_{a,b} \in I_a \) as follows: We define \( l_S := 1, l_{S,1} := \min x_S \) and \( u_{S,1} := \max x_S \) where the minimum and maximum is over all \( x \in \mathcal{X}^S \). For \( a = S - 1, S - 2, \ldots, 1 \) we define \( \nu_a := v_{a+1} = l_{a+1,1} + 1 \) and for any \( b \in [\nu_a] \) we define \( l_{a,b} := \min x_a \) and \( u_{a,b} := \max x_a \) where the minimum and maximum is over all \( x \in \mathcal{X}^S \) with \( x_{a+1} = l_{a+1,1} + b - 1 \) (i.e., the \((a+1)\)-th stop of \( x \) is at the \( b \)-th position in the interval \([l_{a+1,1}, u_{a+1,1}]\)).

The definition of \( l_{a,b} \) and \( u_{a,b} \) for the boundary indices \( b = 1 \) and \( b = \nu_a \) clearly yields finite values (the corresponding subsets of \( \mathcal{X}^S \) over which minimum and maximum are taken are non-empty).

The next lemma shows that this is true also for all intermediate indices \( b \in [\nu_a] \).

Lemma 2. Let \( a \in [S] \) and \( x, x' \in \mathcal{X}^S \) stop programs with \( x_a < x'_a \). For any integer \( i \) with \( x_a < i < x'_a \) there is a stop program \( x'' \in \mathcal{X}^S \) with \( x''_a = i \).

Proof. We fix an integer \( i \) with \( x_a < i < x'_a \). As the intervals \( I_{a-1}, I_a \) and \( I_{a+1} \) are disjoint we have \( x'_{a-1} < x_a \) and \( x'_a < x_{a+1} \). It follows that \( x'' := (x'_1, \ldots, x'_{a-1}, i, x_{a+1}, \ldots, x_S) \) is a stop program with the minimal number of stops \( |x''| = S \) and \( x''_a = i \). Further observe that \( (x'_1, \ldots, x'_{a-1}) \) is feasible for \( \mathcal{P}[i] \) and \( (x_{a+1}, \ldots, x_S) \) is feasible for \( \mathcal{P}[i+1, \ell] \), where \( \mathcal{P}[i+1, \ell] := (n, (\emptyset, \ldots, \emptyset, J_{i+1}, \ldots, J_\ell), m, s, p) \) (here we use \( i < x'_a \) and \( x_a < i \), respectively). This implies that \( x'' \) is feasible for \( \mathcal{P} \).

The definition of \( l_{a,b} \) and \( u_{a,b} \) implies that \( l_{a,b} \leq u_{a,b} \) for any \( a \in [S], b \in [\nu_a] \). The next lemma asserts further important monotonicity properties (see Figure 3).

Lemma 3. For any \( a \in [S] \) and any \( b \in [\nu_a - 1] \) we have \( l_{a,b} \leq l_{a,b+1} \) and \( u_{a,b} = u_{a,b+1} \).

Proof. For any \( a \in [S] \) and any \( k \in [\ell] \) we denote by \( \mathcal{X}^a_{[k]} \) the set of all stop programs (increasing sequences with entries from \([\ell - 1]\)) with exactly \( a \) stops that are feasible for \( \mathcal{P}[k] \). For any \( a \in [S] \) and any \( k \in [\ell - 1] \) we clearly have \( \mathcal{X}^a_{[k]} \supseteq \mathcal{X}^a_{[k+1]} \). For any \( a \in [S] \) and any \( b \in [\nu_a - 1] \), the relation \( \mathcal{X}^a_{[k]} \supseteq \mathcal{X}^a_{[k+1]} \) with \( k := l_{a+1,1} + b - 1 \) shows that \( l_{a,b} \leq l_{a,b+1} \) and \( u_{a,b} \geq u_{a,b+1} \). To see that \( u_{a,b} = u_{a,b+1} \), note that if there are stop programs \( (x_1, \ldots, x_a) \in \mathcal{X}^a_{[k]} \) and \( (x'_1, \ldots, x'_a) \in \mathcal{X}^a_{[k+1]} \) with \( x_a > x'_a \), then \( (x_1, \ldots, x_a) \) is also contained in \( \mathcal{X}^a_{[k+1]} \) (this does not show that \( \mathcal{X}^a_{[k]} = \mathcal{X}^a_{[k+1]} \)).

By Lemma 3 we may simply write \( u_a := u_{a,1} = \cdots = u_{a,\nu_a} \) for any \( a \in [S] \).

The previous arguments show that \( \mathcal{X}^S \) is given by the union over all stop programs \( (x_1, \ldots, x_S) \) that satisfy the following recursive condition: \( x_S \) is contained in the interval \([l_{S,1}, u_S]\), and for any
possible stop positions

\[ J_1 = \{1\} \quad J_2 = \{2\} \quad J_3 = \{3, 4, 5\} \quad J_4 = \{4\} \quad J_5 = \{5, 6\} \quad J_6 = \{6, 7\} \quad J_7 = \{3\} \quad J_8 = \{2\} \quad J_9 = \{1\} \]

\[ l_{1,1} = 1 \quad l_{1,2} = 1 \quad l_{1,3} = 2 \quad l_{2,1} = 3 \quad l_{2,2} = 4 \quad l_{2,3} = 5 \quad u_1 = 2 \quad u_2 = 5 \quad u_3 = 8 \]

The recursion tree \( T \). The interval representation of \( X^S \) is used to recursively enumerate all stop programs from \( X^{S'} \). Simply place stops one after the other backward in time within the intervals \([l_a, u_a] \) according to the previously mentioned condition. This enumeration gives rise to the recursion tree \( T \) shown in Figure 3 (middle), in which every path from the root to a leaf corresponds to one stop program from \( X^S \) (in the figure, the root of \( T \) is at the bottom, leaves at the top). As the number of leaves of \( T \) equals \( |X^S| \), this tree can be astronomically large (recall Remark 1). In Section 4 below we will use the interval representation of \( X^S \) to enumerate (some) stop programs from \( X^S \). Clearly, due to the size of \( T \) our algorithm must never perform an exhaustive enumeration, but focus only on a small fraction of \( T \).

Consider the algorithm \( \text{IntRep()} \) below. Given a problem instance, it computes the interval representation of \( X^S \). It thus forms the top-level function of the first phase of our algorithm to solve the lex-minimization of stops and tool switches.

3.3.1. Correctness of \( \text{IntRep()} \). The algorithm \( \text{IntRep()} \) first computes the earliest stop positions \( l_{1,1}, \ldots, l_{S,1} \) by a call to \( \text{MinSp()} \) (line C1; recall Lemma 3 and see Figure 3). The main loop iterates over all stops \( a = 1, \ldots, S \) (line C2). In the \( a \)-th iteration the algorithm computes the possible positions for the \( a \)-th stop and, if \( a \geq 2 \), for the \( (a-1) \)-th stop. Specifically, it computes \( u_a \) (line C13; this is the latest position for the \( a \)-th stop) and, if \( a \geq 2 \), the values \( l_{a-1,b} \) for all \( b \in [2, \nu_{a-1}] \) (lines C4-C10). These are the earliest positions for the \( (a-1) \)-th stop if the \( a \)-th stop

a \in [S - 1], if \( x_{a+1} \) is at the \( b \)-th position in the interval \([l_{a+1,1}, u_{a+1}]\) (i.e., \( b = x_{a+1} - l_{a+1,1} + 1 \)), then \( x_a \) is contained in \([l_a, u_a] \). Consequently, we refer to the sequences \((l_a, u_a)_{a \in [S]}, b \in [\nu_a] \) and \((u_a)_{a \in [S]} \) as the interval representation of \( X^S \). As \( l_a, u_a \in I_a \) and the intervals \( I_a, a \in [S] \), are all disjoint we have \( \sum_{a \in [S]} \nu_a = (u_2 - l_{2,1} + 1) + \cdots + (u_S - l_{S,1} + 1) + 1 \leq \ell \), i.e., the interval representation of \( X^S \) has indeed size \( O(\ell) \).

Figure 3. Interval representation of \( X^S \) for the instance from Figure 2 computed by \( \text{IntRep()} \) (left) and corresponding tree representations of \( X^S \) (middle and right). Each interval \([l_a, u_a] \) of the interval representation is indicated by a bold vertical line with the bullets showing possible stop positions. The black bullets correspond to the stop program computed by \( \text{MinSp()} \) (cf. Figure 2). The upward arrows indicate the dependencies between stop positions for a stop program \( x \in X^S \): We have \( |x| = S = 3 \) and \( x_3 \in [6, 8] \). Furthermore, if \( x_3 = 6 \) then we have \( x_2 \in [3, 5] \), if \( x_3 = 7 \) then \( x_2 \in [4, 5] \), if \( x_3 = 8 \) then \( x_2 = 5 \); if \( x_2 = 3 \) or \( x_2 = 4 \) then \( x_1 \in [1, 2] \), if \( x_2 = 5 \) then \( x_1 = 2 \). The tree representations are explained in detail in Section 4.
stop is at the $b$-th position in the interval $[l_{a,1}, u_a]$. The position of the $a$-th stop in the interval $[l_{a,1}, u_a]$ is controlled via the variable $b$ that is increased step by step (line C5). If the $a$-th stop, $a \geq 2$, is at the $b$-th position in $[l_{a,1}, u_a]$, then the earliest position for the $(a-1)$-th stop can clearly be computed by $\text{ELS}(\mathcal{P}_{l_{a,1}+b-1})$ (lines C6 and C10). Once $b$ has been increased such that the return value $\text{ELS}(\mathcal{P}_{l_{a,1}+b-1})$ is outside (after) the interval $[l_{a-1,1}, u_{a-1}]$ (this is detected by the condition in line C8), then $b$ has moved outside (after) the interval $[l_{a,1}, u_a]$ and this yields the value $u_a$ (line C13). In the case $a = 1$ a similar condition detects the endpoint of the interval $[l_{1,1}, u_1]$ (lines C12 and C13). In particular, the loop in line C5 is iterated only finitely many times, namely $u_a - l_{a,1} + 1$ times.

3.3.2. Running time of $\text{IntRep}()$. The call to $\text{MinSP}()$ in the first step takes time $O(\ell m)$. The running time of the main loop is determined by the running time of the calls to $\text{ELS}()$. In the $a$-th iteration, $a \in [S]$, the function $\text{ELS}()$ is called once, and the call takes time $O(\ell m)$. As $\sum_{a \in [S]} (u_a - l_{a,1} + 1) \leq \ell$ we obtain the overall bound $O(\ell^2 m)$.

In the case $l_{a,1} = u_a$ for all $a \in [S]$, this bound can be improved by a more careful analysis. In this case, in the $a$-th iteration, $a \in [S]$, the function $\text{ELS}()$ is called once, and the call takes time $O((u_1 + 1)m)$ for $a = 1$ and $O((u_a - l_{a-1,1} + 1)m)$ for $a \geq 2$. As $(u_1 + 1) + (u_2 - l_{1,1} + 1) + \cdots + (u_S - l_{S-1,1} + 1) = (u_1 - l_{1,1}) + (u_2 - l_{2,1}) + \cdots + (u_{S-1} - l_{S-1,1}) + u_S + S \leq 3\ell$, we obtain the bound $O(\ell m)$ for the running time of $\text{MinSP}()$ in this case. In the case $S = 0$ the running time of $\text{MinSP}()$ is also $O(\ell m)$, as the main loop is skipped entirely.

4. Branch-and-bound algorithm

In this section we describe the second phase of our algorithm to solve the lex-minimization of stops and tool switches. We first give an informal outline of the branch-and-bound strategy followed by the algorithm in Section 4.1 below. As the next intermediate step, in Section 4.2 we present an algorithm to minimize tool switches for a fixed stop program. Finally, in Section 4.3 we assemble the building blocks from all previous sections to the complete algorithm. For each algorithm described
in this section we first give an informal overview of the main ideas, then present the pseudocode, followed by the correctness proof and the runtime analysis.

4.1. **Branch-and-bound strategy.** The main idea of our algorithm to solve the lex-minimization of stops and tool switches is the following: We use the interval representation of \( \mathcal{X}^S \) to recursively enumerate stop programs from \( \mathcal{X}^S \), and compute the minimal number of tool switches for each of them, then taking the global minimum. A naïve recursive enumeration corresponds to an exhaustive depth-first-search (DFS) through the tree \( T \) introduced in the previous section (see Figure 3), where each path from the root to a leaf corresponds to one stop program \( x \in \mathcal{X}^S \). A branching step occurs whenever there are several choices where to place the next stop \( x_a, a = S, S - 1, \ldots, 1 \). However, due to the size of \( T \) we cannot afford exploring the entire tree (recall Remark 1), but only a small fraction of it. We achieve this by exploring \( T \) via a breadth-first search (BFS) from the root to the leaves, level by level backward in time. By comparing nodes on the same level we can prune certain branches of \( T \) — this is the bounding step — which drastically reduces the problem size (observe that such node comparisons are impossible when \( T \) is explored via DFS). The pruning criterion determines how many nodes survive on each level, and this number is the dominating factor in the running time of our algorithm.

4.2. **Minimizing tool switches for a fixed stop program.** In this section we present an algorithm to minimize tool switches for a fixed stop program \( x \in \mathcal{X}^S \). Given a problem instance and a stop program \( x \in \mathcal{X}^S \), the algorithm \( \text{SplitBlocks()} \) computes the number \( C(x) \) defined in Section 2 and a plan \( \sigma \) with \( C(x) \) many tool switches such that \( x \) resolves all critical tool switches in \( \sigma \). We can think of the computation of \( \text{SplitBlocks()} \) as proceeding along one particular root-leaf path in \( T \), the path corresponding to the stop program \( x \). Conceptually, the branch-and-bound algorithm \( \text{LexMin()} \) presented in Section 4.3 below consists of multiple parallel executions of \( \text{SplitBlocks()} \), one for each root-leaf path in \( T \), in the discussed BFS manner (in the bounding step, most of these executions are aborted prematurely).

4.2.1. **Idea of the algorithm \( \text{SplitBlocks()} \).** The algorithm \( \text{SplitBlocks()} \) is essentially a combinatorial interpretation of the linear programming algorithm for minimizing tool switches presented in [CKOS94], suitably extended to account for entries of the plan that have to be reserved (i.e., left empty) for setups under the given stop program. Informally, the idea is the following (see the right hand side of Figure 4 below): For every tool \( \alpha \in [n] \), we group together several occurrences of \( \alpha \) in the instance to a so-called block, with the goal of assigning these occurrences to the same column of the plan (to the same slot), and keeping all entries in between empty (the corresponding slot is unused in those time slots). Tool switches can thus only occur between different blocks (and not between the tools inside one block), so the number of tool switches is essentially given by the number of blocks. Therefore, we minimize tool switches by minimizing the number of blocks. To minimize the number of blocks we start with maximal blocks — for every tool \( \alpha \in [n] \) exactly one block that contains all occurrences of \( \alpha \) — and split these blocks into as few smaller blocks as possible (without splitting, the blocks will not fit into the plan). I.e., we minimize the number of blocks by minimizing the number of splittings of the maximal blocks. As it turns out, the number of splittings can be minimized by a simple greedy algorithm.

To make these ideas precise, we introduce some definitions. For the rest of this section, we consider a fixed instance \( \mathcal{P} = (n, J, m, s, p) \), \( J = (J_1, \ldots, J_J) \), and a fixed stop program \( x \in \mathcal{X}^S \).

**Blocks, pre-plan, width.** For any tool \( \alpha \in [n] \) we denote by \( I(\alpha) \) the increasing sequence of all \( i \in [\ell] \) with \( \alpha \in J_i \) (so \( |I(\alpha)| \) is the total number of occurrences of \( \alpha \) in the instance). For any tool \( \alpha \in [n] \) and any \( i \in [\ell] \) we define \( \text{prev}(\alpha, i) := \max\{i < i \mid \alpha \in J_i\} \) and \( \text{next}(\alpha, i) := \min\{i > i \mid \alpha \in J_i\} \). For any tool \( \alpha \in [n] \) and any \( i \in [\ell] \) with \( \alpha \in J_i \) let \( \text{res}_{\alpha}(i) \) denote the largest of the four numbers \( \max\{i \mid i \leq \sum_{i=1}^{\ell-1} p_i \geq s(\alpha)\}, \text{prev}(\alpha, i) + 1, \max\{x_a+1 \mid x_a < i\} \) and 1. The interpretation is that
for setting up a tool \( \alpha \in J_i \) we have to reserve an entry in the plan in all rows \([\text{res}_x(\alpha, i), i - 1]\): The first number in the definition of \( \text{res}_x(\alpha, i) \) corresponds to the reservation requirements arising from the setup and processing times, the second and third number reflect that the previous occurrence of \( \alpha \) in the instance or a stop ‘cut off’ those reservation requirements, and the fourth number covers the exceptional case that the sets in the first three maximizations are all empty (then the three corresponding numbers are \(-\infty \) and \( \text{res}_x(\alpha, i) = 1 \)).

Given any consecutive non-empty subsequence \((i_1, \ldots, i_k)\) of \( I(\alpha) \), we refer to the sequence \((\text{res}_x(\alpha, i_1), i_1, \ldots, i_k) := \lambda \) as an \( x \)-induced block of \( \alpha \)'s, and we define \( I(\lambda) := (i_1, \ldots, i_k) \). An \( x \)-induced pre-plan \( \sigma \) is a mapping that assigns to every tool \( \alpha \in [n] \) a sequence of \( x \)-induced blocks \( \lambda_1, \ldots, \lambda_k \) of \( \alpha \)'s such that \( I(\lambda_1), \ldots, I(\lambda_k) \) is a partition of \( I(\alpha) \) into consecutive subsequences. We sometimes simply write ‘block’ or ‘pre-plan’ to refer to this assignment procedure does not get stuck. Furthermore, as \( \sigma \) is a partition that assigns to every tool \( \alpha \in [n] \), any tool \( \alpha \) originating from two different blocks never appear consecutively in the same column of \( \sigma \). In particular, we have to reserve an entry in the plan in all rows \( \text{res}_x(\alpha, i), i - 1 \) for setting up a tool \( \alpha \in J_i \).

\begin{align*}
\text{res}_x(\alpha, i) := \text{max}_{\lambda \in [\text{first}(\alpha), \text{last}(\lambda)]} \{ \text{res}_x(\alpha, i)_\lambda \}.
\end{align*}

To complete the construction, we can derive from it a plan \( \sigma \) with \( C(\sigma) = |\sigma| - m \) (recall that \( C(\sigma) \) denotes the number of tool switches in \( \sigma \)) such that \( x \) resolves all critical tool switches in \( \sigma \). We construct \( \sigma \) by considering the blocks of \( \sigma \) in increasing order of their first entries [3] and by greedily assigning them one after the other to the other to the \( m \) available columns (=slots), where we assign each of the first \( m \) blocks to a different column (here we need \( m < n \), i.e., \( \sigma \) contains enough blocks to fill all columns). Specifically, assigning an \( x \)-induced block \((\text{res}_x(\alpha, i_1), i_1, \ldots, i_k)\) of \( \alpha \)'s to a column \( j \in [m] \) of \( \sigma \) means setting the entries in rows \( i_1, \ldots, i_k \) and column \( j \) to \( \alpha \), and keeping all entries in rows \([\text{res}_x(\alpha, i_1), i_1, \ldots, i_k] \) \( \setminus \{i_1, \ldots, i_k\} \) and column \( j \) empty. The definition of \( \text{res}_x(\alpha, i_1) \) thus ensures that \( x \) indeed resolves all critical tool switches in \( \sigma \) (as \( \sigma \) has the minimal number of blocks, we have \( \alpha \notin J_i \) for \( i = \text{res}_x(\alpha, i_1) - 1 \)). The block processing order and the condition \( w(\sigma) \leq m \) ensure that this assignment procedure does not get stuck. Furthermore, as \( \sigma \) has the minimal number of blocks, two entries \( \alpha \) originating from two different blocks never appear consecutively in the same column of \( \sigma \) (otherwise the two blocks could be merged without violating the condition \( w(\sigma) \leq m \)). It follows that each except the first \( m \) blocks of \( \sigma \) causes exactly one tool switch, i.e., \( C(\sigma) = |\sigma| - m \).

To complete the proof of the lemma, it suffices to show that the above construction can be reversed: Given a plan \( \sigma \) such that \( x \) resolves all critical tool switches, we can derive from it an \( x \)-induced pre-plan \( \sigma \) with \( w(\sigma) \leq m \) and with \( C(\sigma) \geq |\sigma| - m \): The construction consists of two steps. In the first step we construct a plan \( \sigma' \) from \( \sigma \) with \( C(\sigma) \geq C(\sigma') \) such that \( x \) resolves all critical tool switches in \( \sigma' \) by exhaustively applying the following exchange operation: For any tool \( \alpha \in [n] \) and any consecutive subsequence \((i_1, \ldots, i_k)\) of \( I(\alpha) \), if \( \alpha \) appears consecutively in rows \( i_1 \) and \( i_k \) in some column \( j \in [m] \) (so the entries in rows \([i_1 + 1, i_k - 1]\) and column \( j \) are empty), then we move the entries \( \alpha \) in rows \( i_2, \ldots, i_{k-1} \) to column \( j \). In the second step, for any column \( j \in [m] \) of \( \sigma' \), any tool \( \alpha \in [n] \) and any maximal sequence of consecutive entries \( \alpha \) in rows \( i_1, \ldots, i_k \) and column \( j \), we add the block \((\text{res}_x(\alpha, i_1), i_1, \ldots, i_k)\) to \( \sigma \) (note that \((i_1, \ldots, i_k)\) is a consecutive subsequence of \( I(\alpha) \) as a result of the first step, and the entries in rows \([\text{res}_x(\alpha, i_1), i_1 - 1]\) and column \( j \) of \( \sigma' \) must be empty, as \( x \) resolves all critical tool switches in \( \sigma' \)). By construction, we have \( w(\sigma) \leq m \). Furthermore,

\[ w(\sigma) := \max_{i \in [\text{first}(\lambda), \text{last}(\lambda)]} w(\sigma, i) \].

[3] This yields an algorithm that processes the blocks forward in time. Alternatively, by considering the blocks in decreasing order of their last entries, we obtain an algorithm that runs backward in time.
denoting by \( k \) the number of non-empty columns of \( \sigma' \), we have \( C(\sigma') = |\sigma| - k \geq |\sigma| - m \), as desired.

4.2.2. A greedy block-splitting algorithm. Motivated by Lemma \([4]\) we now describe an algorithm that computes an \( x \)-induced pre-plan with \( w(\sigma) \leq m \) and the minimal number of blocks. For this we introduce a few more definitions.

**Reservations, gaps, splitting blocks.** For any \( i \in [\ell] \) we define \( R_i \) as the union of all tools \( \alpha \in [n] \setminus J_i \) with \( i \geq \text{res}_x(\alpha, i) \) where \( i = \text{next}(\alpha, i) \leq \ell \). In words, \( \alpha \in R_i \) if and only if the setup of tool \( \alpha \in J_i \) requires an entry in row \( i \) of the plan. For any \( x \)-induced pre-plan \( \sigma \) and any \( i \in [\ell] \), we define \( G_i(\sigma) \) as the union of all tools \( \alpha \in [n] \setminus (J_i \cup R_i) \) for which \( \sigma \) contains a block \( \lambda \) of \( \alpha \)'s with \( i \in [\text{first}(\lambda), \text{last}(\lambda)] \). In words, \( \alpha \in G_i(\sigma) \) if and only if assigning the block \( \lambda \) to the plan occupies an entry in the plan in row \( i \) that must remain empty if a tool switch between the two entries \( \alpha \) in rows \( \text{prev}(\alpha, i) \) and \( \text{next}(\alpha, i) \) should be avoided (and this entry is not required for setting up \( \alpha \in J_{\text{next}(\alpha, i)} \)). We refer to the sets \( R_i \) and \( G_i(\sigma) \) as reservations and gaps, respectively. Note that the sets \( J_i, R_i, G_i(\sigma) \) are all disjoint and that

\[
w_i(\sigma) = |J_i| + |R_i| + |G_i(\sigma)|.
\]

For any tool \( \alpha \in G_i(\sigma) \), splitting a block \( \lambda \) of \( \alpha \)'s in row \( i \) means replacing \( \lambda \) by the two blocks \( \lambda' \) and \( \lambda'' \) with last(\( \lambda' \)) = \( \text{prev}(\alpha, i) \) and first(\( \lambda'' \)) = res\( _x(\alpha, \text{next}(\alpha, i)) \). This splitting operation reduces the width of the pre-plan \( \sigma \) by one in all rows with \( \text{last}(\lambda') + 1, \text{first}(\lambda'') - 1 \) (this includes row \( i \)). The greedy algorithm for computing an \( x \)-induced pre-plan \( \bar{\sigma} \) with \( w(\bar{\sigma}) \leq m \) and the minimal number of blocks can be described as follows (see the right hand side of Figure \( 3 \)).

**Greedy block-splitting algorithm:** Start with the pre-plan \( \sigma \) that contains only maximal blocks, i.e., exactly one block \( \hat{\lambda}(\alpha) \) for every tool \( \alpha \in [n] \) with \( I(\hat{\lambda}(\alpha)) = I(\alpha) \) (this block contains all occurrences of \( \alpha \)). For \( i = \ell, \ell - 1, \ldots, 1 \) do the following: If \( w_i(\sigma) > m \), then split exactly \( d_i := w_i(\sigma) - m \) many blocks of \( \alpha \)'s with \( \alpha \in G_i(\sigma) \) in row \( i \) (this reduces \( w_i(\sigma) \) to exactly \( m \)). Specifically, split the \( d_i \) blocks of \( \alpha \)'s, \( \alpha \in G_i(\sigma) \), with minimal values \( \text{prev}(\alpha, i) \) (ties can be broken arbitrarily) — this is the greedy splitting rule.

This algorithm clearly produces an \( x \)-induced pre-plan \( \sigma \) with \( w(\sigma) \leq m \). The fact that \( \sigma \) has the minimal number of blocks can be seen as follows: Clearly, for any \( i = \ell, \ell - 1, \ldots, 1 \) with \( w_i(\sigma) > m \) at least \( d_i = w_i(\sigma) - m \) blocks of \( \alpha \)'s, \( \alpha \in G_i(\sigma) \), have to be split. By splitting exactly \( d_i \) blocks we do not lose any optimal solution, as every additional block can still either be split or not in a later step. Furthermore, splitting the \( d_i \) blocks with minimal values \( \text{prev}(\alpha, i) \) reduces the width \( w_i(\sigma) \) in all rows \( i \in [i - 1] \) by the maximum possible amount. By a straightforward monotonicity argument this generates the minimal number of splits and hence the minimal number of blocks. We refer to \([2, 3]\) and \([4]\) for a reformulation of this algorithm and correctness proof in the language of linear programming (the problem is essentially an interval covering problem, and its dual an interval packing problem; see also \([5, 6] \ p. 562-573\)).

The pseudocode of \text{SplitBlocks}() implements the greedy algorithm outlined above. We proceed by explaining the steps of \text{SplitBlocks}() in detail and argue about its running time. Figure \( 4 \) shows how the algorithm operates on an example (the stop program \( x \) in this example corresponds to the leftmost root-leaf path in the recursion tree \( T \) in Figure \( 3 \)).

4.2.3. Correctness of \text{SplitBlocks}(). The algorithm consists of two stages: In the first stage, it computes an \( x \)-induced pre-plan \( \sigma \) with \( w(\sigma) \leq m \) and the minimal number of blocks using the greedy strategy outlined in the previous section (lines \( \text{D1-D19} \)). In the second stage, it computes a plan \( \sigma \) with \( C(\sigma) = |\sigma| - m \) from the pre-plan \( \sigma \) (this is done in the function \text{PLAN}() in line \( \text{D20} \) that will be explained shortly).
Figure 4: Example for the algorithm SplitBlocks() for the instance from Figure 2 and the stop program $x = (1, 3, 6)$. The resulting $x$-induced pre-plan $\sigma$ and plan $\sigma$ are shown on the right. Initially, there are seven maximal blocks, three of them are split (indicated by the crosses), so eventually there are $|\sigma| = 7 + 3 = 10$ blocks (indicated by the solid rectangles). The grey boxes represent elements from the sets $R_i$. The plan $\sigma$ has $C(\sigma) = C(x) = |\sigma| - m = 10 - 4 = 6$ tool switches.

Algorithm 4: SplitBlocks($\mathcal{P}, x$)

Input: an instance $\mathcal{P} = (n, J, m, s, p)$, $J = (J_1, \ldots, J_\ell)$, with $\max_{i \in [\ell]} |J_i| \leq m < n$, a stop program $x \in \mathcal{X}^S$

Output: the number $C(x)$ defined in Section 2 and a plan $\sigma$ with $C(\sigma) = C(x)$ such that $x$ resolves all critical tool switches in $\sigma$

D1 $\forall \alpha \in [n]: z(\alpha) := ()$ /* initially empty split sequences */
D2 $a := |x|$ /* count stops backwards */
D3 $R_\ell := \emptyset; G_\ell := \emptyset$ /* initially no reservations and gaps */
D4 for $i := \ell$ downto 1 do /* backward loop through the instance */
D5 \[
    d_i := (|J_i| + |R_i| + |G_i|) - m
\] /* width in row $i$ too large? */
D6 if $d_i > 0$ then /* need to split $d_i$ many blocks */
D7 \[
    D_i := \text{an arbitrary subset of $d_i$ elements } \alpha \in G_i \text{ with minimal values } \text{prev}(\alpha, i)
\]
D8 $\forall \alpha \in D_i: z(\alpha) := (i, z(\alpha))$ /* greedy splitting rule */
D9 $G_i := G_i \setminus D_i$ /* blocks have been split */
D10 $\forall \alpha \in R_i: r(\alpha) := r(\alpha) - p_i$ /* update remaining setup time */
D11 $\forall \alpha \in J_i: r(\alpha) := s(\alpha)$ /* reset remaining setup time */
D12 if $(a = 0) \text{ or } (x_a < i - 1)$ then /* no stop at $i - 1$ */
D13 \[
    R_{i-1} := \{ \alpha \in (J_i \cup R_i) \setminus J_{i-1} | r(\alpha) > 0 \}
\] /* update reservations and... */
D14 $G_{i-1} := \{ \alpha \in (J_i \cup R_i) \setminus J_{i-1} | r(\alpha) \leq 0 \wedge \text{prev}(\alpha, i - 1) \geq 1 \} \cup G_i \setminus J_{i-1}$ /* ... gaps */
D15 else /* stop at $x_a = i - 1$ */
D16 $R_{i-1} := \emptyset$ /* stop 'cuts off' reservations */
D17 $G_{i-1} := \{ \alpha \in (J_i \cup R_i) \setminus J_{i-1} | \text{prev}(\alpha, i - 1) \geq 1 \} \cup G_i \setminus J_{i-1}$
D18 \[
    a := a - 1
\] /* passed $a$-th stop of $x$ */
D19 $\sigma := \{ \alpha \mapsto \text{split}(\hat{\lambda}(\alpha), z(\alpha)) | \alpha \in [n] \}$ /* split maximal blocks */
D20 $\sigma := \text{PLAN}(\sigma, \ell, m)$ /* compute plan from pre-plan */
D21 return $(|\sigma| - m, \sigma)$
The first stage of the algorithm is a backward loop over all \( i = \ell, \ell - 1, \ldots, 1 \) (line D4) that computes for every tool \( \alpha \in [n] \) a sequence \( z(\alpha) \) of rows where to split the maximal blocks \( \lambda(\alpha) \), \( \alpha \in [n] \). The split sequences \( z(\alpha) \), \( \alpha \in [n] \), are initially empty (line D1) this corresponds to not splitting the maximal blocks at all, and augmented step by step (line D8) each additional splitting increases the number of blocks by +1. The final pre-plan \( \pi \) is derived by splitting each block \( \lambda(\alpha) \) in every row recorded in \( z(\alpha) \), this splitting operation is denoted by \( \text{split}(\lambda(\alpha), z(\alpha)) \) (line D19).

In the following we show that the variables \( R_i \) and \( G_i \), \( i \in [\ell] \), in the algorithm correspond exactly to the sets \( R_i \) and \( G_i(\pi_i) \) defined in the previous section for the pre-plan \( \pi_i := \{ p \mapsto \text{split}(\lambda(\alpha), z(\alpha)) \mid \alpha \in [n] \} \) where \( z(\alpha) \), \( \alpha \in [n] \), are the values of the split sequences at the beginning of the for-loop where the loop variable has the value \( i \) (so \( \pi_i \) is the initial pre-plan consisting only of maximal blocks, and \( \pi_0 \) is the pre-plan obtained after termination of the main loop). Given this correspondence, lines \( D5 \)–\( D8 \) are a straightforward implementation of the greedy splitting rule described in the previous section, where \( w_i(\pi_i) \) is evaluated via \( I \). The sets \( R_i \), \( i \in [\ell] \), are computed analogously to the algorithm ELS() using auxiliary variables \( r(\alpha), \alpha \in [n] \) (compare lines \( D3 \)–\( D10 \), \( D11 \), and \( D13 \)) with lines \( A1 \)–\( A4 \), \( A5 \), and \( A6 \) respectively, and note that the condition \( r(\alpha) > 0 \) that decides whether \( \alpha \in (J_i \cup R_i) \setminus J_{i-1} \) is added to \( R_{i-1} \) is equivalent to \( i - 1 \geq \text{res}_{\alpha}(\alpha, i) \) for \( i = \text{next}(\alpha, i - 1) \). The only adaptation is that the reservations are reset whenever a stop is encountered: Line D16 is executed if \( x_a = i - 1 \), where \( a \) is the stop counter (see lines D2, D12, and D18). To understand how the sets \( G_i(\pi_i) \) are computed, consider first lines \( D13 \)–\( D14 \), \( D16 \), and \( D17 \). Note that the condition \( \text{prev}(\alpha, i - 1) \geq 1 \) in lines D14 and D17 simply checks if the tool \( \alpha \) occurs again in some \( J_i, i \leq i - 1 \) (\( i = \text{prev}(\alpha, i - 1) \)). Observe that any tool \( \alpha \in (J_i \cup R_i) \setminus J_{i-1} \) for which this is true and that is not added to \( R_{i-1} \) is added to \( G_{i-1} \) (this is the first contribution to the set unions in lines \( D13 \) and \( D17 \)). Furthermore, \( \alpha \) is passed on to \( G_{i-2}, G_{i-3}, \ldots, G_{i+1} \) (this is the second contribution to the set unions, see also line D3), unless the corresponding block of \( \alpha \)'s is split in one of these rows (line D9). (A small subtlety concerning line D9 is that when splitting a block of \( \alpha \)'s in row \( i \) by removing \( \alpha \) from \( G_i \), we omit removing \( \alpha \) also from the sets \( G_{i+1}, \ldots, G_{i-1} \), \( i = \text{res}_{\alpha}(\alpha, \text{next}(\alpha, i)) \), as these are not needed anymore later in the algorithm.) In the second stage of the algorithm, the function \( \text{Plan}() \) (line D20) constructs an \( \ell \times m \) plan \( \sigma \) from the pre-plan \( \pi \) with \( C(\sigma) = |\pi| - m \) such that \( x \) resolves all critical tool switches in \( \sigma \). The function is defined by translating the construction from the first part of the proof of Lemma 4 into an algorithm. A detailed pseudocode description can be found in [Müt14, Function 3].

Summarizing, we showed that \( \text{SplitBlocks}() \) implements the greedy algorithm outlined in the previous section, which, as argued before, yields an \( x \)-induced pre-plan \( \pi \) with \( w(\pi) \leq m \) and the minimal number of blocks. We also showed that the plan \( \pi \) computed by \( \text{SplitBlocks}() \) satisfies \( C(\pi) = |\pi| - m \) and that \( x \) resolves all critical tool switches in \( \sigma \). The correctness of the return values of \( \text{SplitBlocks}() \) (line D21) hence follows from Lemma 4.

4.2.4. Running time of \( \text{SplitBlocks}() \). We argue that the running time of \( \text{SplitBlocks}() \) is bounded by \( \mathcal{O}(\ell m) \). We clearly have \( |J_i| \leq m \) for all \( i \in [\ell] \) by the assumption \( \max_{i \in [\ell]} |J_i| \leq m \) for the input of the algorithm. Furthermore, a straightforward induction over the number of iterations of the main loop shows that \( |R_i| \leq m \) and \( |G_i| \leq m \) for all \( i \in [\ell] \) (the main ingredient here is that \( |J_i| + |R_i| + |G_i| \leq m \) after the instructions in lines \( D5 \)–\( D9 \)). It follows that all operations involving the sets \( J_i, R_i \) and \( G_i \) in lines \( D5 \)–\( D18 \) can be implemented to run in time \( \mathcal{O}(m) \): To achieve this we maintain for every tool \( \alpha \in [n] \) an index \( \nu(\alpha) \) into the sequence \( I(\alpha) \) such that \( I(\alpha)_{\nu(\alpha)} = \text{prev}(\alpha, i) \) (the sequences \( I(\alpha) \) can be precomputed in time \( \mathcal{O}(\sum_{i \in [\ell]} |J_i|) \leq \mathcal{O}(\ell m) \)). This allows us to query \( \text{prev}(\alpha, i) \) in constant time, which is needed in lines \( D7, D14 \), and \( D17 \). Furthermore, to select \( d_i \) elements from \( G_i \) with the minimal values \( \text{prev}(\alpha, i) \) in line \( D7 \) we employ a linear-time selection algorithm. It follows that the first stage of the algorithm runs in time \( \mathcal{O}(\ell m) \). With the help of
the sets \( R_i, i \in [\ell] \), computed earlier, the splitting operation in line \([D19]\) can also be performed in time \( O(\ell m) \) (for any \( \alpha \in [n] \) and \( i \in [\ell] \) with \( \alpha \in J_i \), we do not need to recompute \( \text{res}_\alpha(x, i) \)). The same runtime bound also applies for the function \( \text{PLAN}(\) (see [Müt14] Section 2.3)). Adding these contributions yields the desired overall runtime bound.

4.3. Lex-minimization of stops and tool switches. We are now ready to assemble the various building blocks from all previous sections to the algorithm \( \text{LEXMIN}(\) that solves the lex-minimization of stops and tool switches for any given instance \( \mathcal{P} = (n, J, m, s, p), J = (J_1, \ldots, J_\ell) \).

4.3.1. Idea of the algorithm \( \text{LEXMIN}(\). As outlined in Section 4.1, the basic idea is to explore the tree \( T \) defined in Section 3.3 (see Figure 3) via a BFS, and to prune branches whenever possible. For technical reasons our algorithm operates on a slightly modified tree \( T^+ \) that we define in the following.

**The tree \( T^+, stop/auxiliary nodes, levels.** As the inner nodes of \( T \) describe the possible stop positions for stop programs from \( \mathcal{X}^S \), we refer to them as stop nodes (they are indicated by grey bullets in Figure 3). We refer to the stop position associated with a stop node as the level of this node (possible levels for stop nodes are \( 1, \ldots, \ell - 1 \)). This notion can be extended to the root of \( T \), which is assigned the level \( \ell \), and to the leaves, which are assigned the level \( 0 \). The tree \( T^+ \) is derived from \( T \) in two steps (see the right hand side of Figure 3): Ignore the shaded regions and crossed out nodes for the moment: We first subdivide \( T \) by adding nodes, referred to as auxiliary nodes (indicated by white bullets in the figure), such that every root-leaf path has exactly \( \ell + 1 \) nodes, one for each level (exactly \( S \) of them are stop nodes, and the remaining ones, including the root and the leaf, are auxiliary nodes). We then merge several auxiliary descendants of a node to a single auxiliary node (so every node either has only a single descendant or exactly one stop descendant and one auxiliary descendant).

The benefit of the tree \( T^+ \) is that a BFS starting from the root visits all nodes level by level (we neglect the effect of pruning branches for the moment). As indicated earlier, during this BFS we perform multiple parallel executions of the algorithm \( \text{SPLITBLOCKS}(\) one for each path from the root to a node on the current level. As the BFS jumps back and forth between different nodes/different executions of \( \text{SPLITBLOCKS}(\), we assign a state information to each node of \( T^+ \). This enables us to continue the computation from this node in a later step. During the BFS we compute the state information of every node on level \( i - 1 \) from the state information of the corresponding parent node on level \( i \) by performing the instructions in lines \([D5,D18]\) of \( \text{SPLITBLOCKS}(\) ). Specifically, to a node \( v \) on level \( i \) of \( T^+ \) we assign the state information \((x^v, R^v, r^v, G^v, Z^v)\) where the entries of this 5-tuple are defined as follows: \( x^v \) denotes the stop program corresponding to the path from the root of \( T^+ \) to the node \( v \) (stops in \( x^v \) given by the stop nodes are in increasing order as usual, auxiliary nodes are ignored). The next three entries correspond to the values of the variables \( R_i, (r(\alpha))_{\alpha \in [n]} \), and \( G_i \) in the algorithm \( \text{SPLITBLOCKS}(\) ) with input \( \mathcal{P} \) and \( x^v \) at the beginning of the for-loop where the loop variable has the value \( i \). The last entry \( Z^v \) is an integer tracking the total number of splittings

\[
Z^v := \sum_{\alpha \in [n]} |z^v(\alpha)| ,
\]

where \( z^v(\alpha), \alpha \in [n], \) are the split sequences (the variables \( z(\alpha), \alpha \in [n] \)) computed by the algorithm \( \text{SPLITBLOCKS}(\) ) with input as before. We emphasize here that the split sequences are not actually computed during the BFS, but \( Z^v \) can be maintained by substituting line \([D8]\) in \( \text{SPLITBLOCKS}(\) ) by \( Z^v := Z^v + d_i \). When the BFS reaches the leaves of \( T^+ \) (i.e., the nodes on level \( 0 \)), a stop program that minimizes tool switches is given by a leaf node \( v \) with minimal value \( Z^v \). This is because the number of blocks of the corresponding pre-plan \( \bar{\sigma}^v \) is given by \( |\bar{\sigma}^v| = n + \sum_{\alpha \in [n]} z^v(\alpha) = n + Z^v \) (recall from line \([D19]\) that \( \bar{\sigma}^v \) is obtained by splitting \( n \) maximal blocks, one for every tool \( \alpha \in [n], \) and each splitting increases the number of blocks by one) and \( C(x^v) = |\bar{\sigma}^v| - m = n + Z^v - m \).
(recall line [D2] and Lemma [4] note also that $n$ and $m$ are the same for each leaf, so the leaf $v$ with minimal value $Z^v$ minimizes $C(x^v)$). So during the BFS we neither need to compute the split sequences $z^v(\alpha), \alpha \in [n]$, nor the pre-plan $\sigma^v$ — all information necessary to decide which leaf of $T^+$ is optimal is encoded in the split counters $Z^v$.

Consider now the algorithm $\text{LEXMIN}()$ that implements the outlined branch-and-bound algorithm on the tree $T^+$ and thereby solves the lex-minimization of stops and tool switches. We proceed to explain the algorithm in detail (in particular, we specify the pruning criterion, which is crucial for the algorithm’s performance and has not been discussed so far) and argue about its running time.

4.3.2. Correctness of $\text{LEXMIN}()$. The algorithm conveniently identifies each node $v$ of $T^+$ with the corresponding state information $(x^v, R^v, r^v, G^v, Z^v)$. It tracks all visited nodes on level $i$ of $T^+$ — i.e., their state information — via a sequence $V_i$ (without pruning, $V_i$ would contain all nodes on level $i$, but with pruning, much fewer). The main loop implements the level by level BFS (line [E4] and [E5]), starting from the root (line [E3]). Once the BFS is finished, a stop program $x^v \in \mathcal{X}^S$ that minimizes tool switches is computed based on the split counters $Z^v$, by minimizing over all nodes $v$ in $V_0$, as discussed before (line [E7]). For this optimal stop program an actual plan is computed by calling the function $\text{SPLITBLOCKS}()$ (line [E8]).

Algorithm 5: $\text{LEXMIN}(\mathcal{P})$

\begin{itemize}
  \item [Input:] an instance $\mathcal{P} = (n, J, m, s, p)$, $J = (J_1, \ldots, J_\ell)$, with $\max_{i \in [\ell]} |J_i| \leq m < n$
  \item [Output:] the numbers $S$ and $C$ defined in Section 2, a plan $\sigma$ with $C(\sigma) = C$ and a stop program $x \in \mathcal{X}^S$ that resolves all critical tool switches in $\sigma$
  \item [E1] $(l_{a,b})_{a \in [S], b \in [\nu_a]}, (u_{a})_{a \in [S]} : = \text{IntRep}(\mathcal{P})$ /* interval representation of $\mathcal{X}^S$ */
  \item [E2] $a := S$ /* count stop intervals backwards */
  \item [E3] $V_i := ((((), \emptyset, \emptyset, \emptyset, 0))$ /* state information at root node of $T^+$ */
  \item [E4] for $i := \ell$ downto 1 do /* BFS through $T^+$ starting at the root */
    \item [E5] $V_{i-1} := \text{UPDATE}_{\mathcal{P}, l, a}(V_i, i, a)$ /* compute nodes on next level */
    \item [E6] if $a \geq 1$ and $i - 1 = l_{a, 1}$ then $a := a - 1$ /* passed $a$-th stop interval */
    \item [E7] $v^*$ := an arbitrary node $v$ in $V_0$ with minimal value $Z^v$ /* pick optimal leaf */
    \item [E8] $(C, \sigma) := \text{SPLITBLOCKS}(\mathcal{P}, x^v)$ /* compute actual plan */
    \item [E9] return $(S, C, \sigma, x^v)$
\end{itemize}

The function $\text{UPDATE}_{\mathcal{P}, l, a}(V_i, i, a)$ is the core of our branch-and-bound algorithm, and consists of the steps shown below. The function performs the \textit{bounding step} (step (b) below) by removing certain nodes from $V_i$ based on comparing their state information (these nodes on level $i$ are pruned from the tree $T^+$, and the corresponding executions of $\text{SPLITBLOCKS}()$ are terminated prematurely). Using the precomputed interval representation of $\mathcal{X}^S$ (these are the sequences $l$ and $u$ computed in line [E1]), and the backward stop counter $a$ (lines [E2] and [E6]), the function performs the \textit{branching step} (step (c) below) by computing the descendants of the nodes that survived the bounding step (the descendants are on level $i - 1$ of $T^+$). It also computes the state information of these descendants from the state information of the corresponding parent nodes (steps (a) and (d)). Together these steps yield the nodes $V_{i-1}$ for the next iteration:

(a) For every node $v$ in the sequence $V_i$ perform the instructions in lines [D5]–[D11] of $\text{SPLITBLOCKS}()$, where line [DS] is substituted by $Z^v := Z^v + d_i$.

(b) For any two different nodes $v, w$ in the sequence $V_i$, remove $w$ from $V_i$, if the following three conditions hold: first$(x^v) = \text{first}(x^w)$ (**), the sorted sequence $(\text{prev}(\alpha, i))_{\alpha \in G^v_i}$ equals the sorted sequence $(\text{prev}(\alpha, i))_{\alpha \in G^w_i}$ (**), and $Z^v \leq Z^w$ (***)

The function $\text{UPDATE}_{\mathcal{P}, l, a}(V_i, i, a)$ is the core of our branch-and-bound algorithm, and consists of the steps shown below. The function performs the \textit{bounding step} (step (b) below) by removing certain nodes from $V_i$ based on comparing their state information (these nodes on level $i$ are pruned from the tree $T^+$, and the corresponding executions of $\text{SPLITBLOCKS}()$ are terminated prematurely). Using the precomputed interval representation of $\mathcal{X}^S$ (these are the sequences $l$ and $u$ computed in line [E1]), and the backward stop counter $a$ (lines [E2] and [E6]), the function performs the \textit{branching step} (step (c) below) by computing the descendants of the nodes that survived the bounding step (the descendants are on level $i - 1$ of $T^+$). It also computes the state information of these descendants from the state information of the corresponding parent nodes (steps (a) and (d)). Together these steps yield the nodes $V_{i-1}$ for the next iteration:

(a) For every node $v$ in the sequence $V_i$ perform the instructions in lines [D5]–[D11] of $\text{SPLITBLOCKS}()$, where line [DS] is substituted by $Z^v := Z^v + d_i$.

(b) For any two different nodes $v, w$ in the sequence $V_i$, remove $w$ from $V_i$, if the following three conditions hold: first$(x^v) = \text{first}(x^w)$ (**), the sorted sequence $(\text{prev}(\alpha, i))_{\alpha \in G^v_i}$ equals the sorted sequence $(\text{prev}(\alpha, i))_{\alpha \in G^w_i}$ (**), and $Z^v \leq Z^w$ (***)
(c) Compute the descendants $V_{i-1}^{\text{aux}}$ and $V_{i-1}^{\text{stop}}$ of the nodes $V_i$ in $T^+$ as follows (these sequences are the auxiliary nodes and stop nodes, computed separately): For every node $v \in V_i$, if $a \geq 1$ and $(|x^v| = 0) \lor (u_a < \text{first}(x^v))$ and $i - 1 \leq u_a$, then add $v$ to the sequence $V_{i-1}^{\text{stop}}$. Define $b := 1$ if $a = S$ and $b := \text{first}(x^v) - l_{a+1,1} + 1$ otherwise. If $a = 0$ or $(|x| \geq 1) \land (u_a \geq \text{first}(x))$ or $i - 1 > l_{a,b}$, then add $v$ to the sequence $V_{i-1}^{\text{aux}}$.

(d) For every node $v$ in the sequence $V_{i-1}^{\text{aux}}$ perform the instructions in lines $\text{D13}$-$\text{D14}$ of $\text{SplitBlocks}()$.

For every node $v$ in the sequence $V_{i-1}^{\text{stop}}$ set $x^v := (i - 1, x^v)$ and perform the instructions in lines $\text{D16}$-$\text{D17}$. Define $V_{i-1} := (V_{i-1}^{\text{aux}}, V_{i-1}^{\text{stop}})$.

When discussing the algorithm $\text{SplitBlocks}()$ in Section 4.2.3 we have already shown that steps $\text{(a)}$ and $\text{(d)}$ correctly compute the state information of the nodes on level $i-1$ from the state information of the nodes on level $i$ (recall also the remarks after (2)). To verify step $\text{(c)}$ note that these somewhat technical conditions simply check whether a node $v$ in $V_i$ has an auxiliary node and/or a stop node as descendants on level $i-1$ of $T^+$ (this can be derived from the interval representation of $X^i$ and from the first entry of $x^v$, which determines the shape of the subtree of $T^+$ rooted at $v$). It remains to argue that the bounding step $\text{(b)}$ removes only nodes that are irrelevant for an optimal solution. Specifically, we show that whenever a node $w$ is removed in step $\text{(b)}$, then any leaf $\bar{w}$ of the subtree of $T^+$ rooted at $w$ satisfies $Z^\bar{w} \geq Z^v$ for $v^*$ as computed in line $\text{E7}$.

Consider two different nodes $v$, $w$ in the sequence $V_i$ that satisfy the conditions $\text{(*)}$, $\text{(**)}$ and $\text{(***)}$ in step $\text{(b)}$ (as a consequence, node $v$ is kept, and node $w$ is removed in this step). First observe that by $\text{(*)}$, the subtrees of $T^+$ rooted at $v$ and $w$ are isomorphic (for any stop program from $X^i$, the possible positions for the $a$-th stop only depend on the position of the $(a+1)$-th stop for all $a \in [S - 1]$). It therefore suffices to show that for any leaf $\bar{v}$ of the subtree rooted at $v$, the corresponding leaf $\bar{w}$ of the subtree rooted at $w$ satisfies $Z^{\bar{w}} \leq Z^{\bar{v}}$. We fix two such corresponding leaves $\bar{v}$ and $\bar{w}$, and compare the computations of the algorithm $\text{SplitBlocks}()$ on the path from $v$ to $\bar{v}$ and on the path from $w$ to $\bar{w}$. A straightforward induction shows that for any level $i = i, i - 1, \ldots, 0$, the two nodes $\bar{v}$ and $\bar{w}$ on this level on the path from $v$ to $\bar{v}$ and on the path from $w$ to $\bar{w}$, respectively, satisfy $R_i^{\bar{v}} = R_i^{\bar{w}}$, $r^\bar{v} = r^\bar{w}$ and $|G^i_\bar{v}| = |G^i_\bar{w}|$ (the first two equalities hold because $x^\bar{v}$ and $x^\bar{w}$ agree in all stops $\leq i$, the last equality follows from $\text{(**)}$). This implies that $Z^\bar{v} - Z^v = Z^{\bar{w}} - Z^w$ (i.e., the split counters increase equally in both computations of $\text{SplitBlocks}()$), in particular $Z^{\bar{v}} - Z^v = Z^{\bar{w}} - Z^w$. Combining this with $\text{(***)}$ yields $Z^\bar{v} \leq Z^{\bar{w}}$, as desired.

We conclude this section by an example how the pruning criterion works for the instance from Figure 2 (see also the right hand side of Figure 3). Running the algorithm $\text{LEXMIN}()$ for this example, it turns out that in each subset of nodes of $T^+$ that are shaded in Figure 3 only one node survives the pruning step (the six pruned nodes are crossed out). E.g., for each of the three highlighted nodes $v$ on level 5 we have $R_5^v = \emptyset$, $G_5^v = \{2, 3\}$ and $Z^v = 1$, and consequently only one of them survives (in this particular case the state information is exactly the same, so the decision is arbitrary). In total, from the 53 nodes of $T^+$, only 31 are visited by our algorithm (counting pruned nodes also as visited), and the optimal stop program is chosen among the two surviving leaves (these leaves are marked with * in the figure). Specifically, for both leaves $v$ in $V_9$ the split counter has the value $Z^v = 2$, i.e., the minimal number of tool switches is $C = \min_{x \in X^S} C(x) = \min_{v \in V_0} C(x^v) = n + Z^v - m = 7 + 2 - 4 = 5$.

4.3.3. Running time of $\text{LEXMIN}()$. Steps $\text{[a]}$ $\text{[c]}$ and $\text{[d]}$ of the function $\text{UPDATE}()$ with input $V_i$ can be implemented to run in time $O(|V_i| \cdot m)$ (the arguments are given in Section 4.2.4). The bounding step $\text{[b]}$ can be implemented as follows:

(b1) Divide the nodes $v$ in $V_i$ into groups according to the first entry of $x^v$ (so all nodes within the same group satisfy $\text{(*)}$).

(b2) For each group, for every node $v$ sort the sequence $(\text{prev}(\alpha, i))_{\alpha \in G^i_v}$. We denote the sorted sequence by $y^v$. 
(b3) For each group, sort all nodes \( v \) lexicographically according to the sequences \( y^v \) (so nodes satisfying (**) appear consecutively in the sorted list).

(b4) For each group, for any pair of nodes \( u, v \) that are consecutive in the sorted list and that satisfy \( y^u = y^v \), if \( Z^u \leq Z^v \) then remove \( v \), otherwise remove \( u \) (this is condition (***)).

To bound the running time of these steps, recall from Section 4.2.4 that any node \( v \) in \( \mathcal{V}_i \) satisfies \(|G^i_v| \leq m\), so the length of the sequence \( y^v \) is bounded by \(|y^v| \leq m\). We conclude that step (b1) runs in time \( \mathcal{O}(|\mathcal{V}_i|) \), step (b2) in time \( \mathcal{O}(m \log m) \) per node and \( \mathcal{O}(|\mathcal{V}_i| \cdot m \log m) \) in total, step (b3) in time \( \mathcal{O}(m \cdot |\mathcal{V}_i| \log |\mathcal{V}_i|) \), and step (b4) in time \( \mathcal{O}(|\mathcal{V}_i| \cdot m) \). Summing these contributions yields the bound \( \mathcal{O}(|\mathcal{V}_i| m \log(|\mathcal{V}_i| m)) \) for step (b). This clearly dominates the bounds derived for steps (a), (c) and (d) so the main loop of \textsc{LexMin()} runs in time
\[
\mathcal{O}(|\mathcal{V}_i| \ell m \log(|\mathcal{V}_i| m)) \ .
\]

To bound \( |\mathcal{V}_i| \), we denote the resulting sequence of nodes after step (b4) by \( \mathcal{V}^-_i \) (\( \mathcal{V}^-_i \) is obtained from \( \mathcal{V}_i \) by removing nodes, so \(|\mathcal{V}^-_i| \leq |\mathcal{V}_i|\)). Furthermore, we have
\[
|\mathcal{V}_i| \leq 2 |\mathcal{V}^-_{i+1}| \ ,
\]
where the factor 2 comes from the branching step (c). We now show that
\[
|\mathcal{V}^-_i| \leq \ell(m + 1) \left(\min_{m_{\ell,n/2}}^n \right) \ , \hspace{1em} i \in [\ell] \ .
\]
To see this first observe that the number of groups resulting from step (b1) is at most \( \ell \) (there are at most \( \ell \) different stop positions). Furthermore, for every node \( v \) in \( \mathcal{V}_i \), there are at most \( n \) possible values for \( \text{prev}(\alpha, i) \), \( \alpha \in G^i_v \subseteq [n] \), and \( |G^i_v| \leq m \). Consequently, the number of different sequences \( y^v \) arising in step (b2) is at most \( \sum_{i=0}^{m} \binom{n}{\text{last}(I(\alpha))} \leq (m + 1) (\min_{m_{\ell,n/2}}^n) \). As in each group only nodes with different sequences survive step (b4) we obtain (5).

We proceed by proving the various runtime bounds stated under (i), (vi) in the introduction.

(i) Combining (3), (4) and (5) proves that the main loop of \textsc{LexMin()} runs in time \( \overline{\mathcal{O}}(\ell^2 m \log(\min_{m_{\ell,n/2}}^n)) \).

As this bound dominates the bounds \( \mathcal{O}(\ell^2 m) \) for \textsc{IntRep()} and \( \mathcal{O}(\ell m) \) for \textsc{SplitBlocks()} (see Section 3.3.2 and Section 4.2.4 respectively), it also holds for the running time of the entire algorithm \textsc{LexMin()}, proving the general runtime bound stated under (i) in the introduction.

(ii) Even though the binomial coefficient in the bound (5) cannot be eliminated in general, the bound is nevertheless overly pessimistic in most situations (we will support this by experimental data in Section 5 below). In the following we derive improved bounds for two such cases by considering for every tool \( \alpha \in [n] \) the sequence \( I(\alpha) \) of all occurrences of \( \alpha \) in the instance (recall the definition of \( I(\alpha) \) from Section 4.2.2):

- Every tool appears only in a bounded range of jobs: Formally, there is a constant \( c_1 \) such that the cardinality of \( \{ \alpha \in [n] \mid i \in [\text{first}(I(\alpha)), \text{last}(I(\alpha))] \} \) is bounded by \( c_1 \) for every \( i \in [\ell] \).
- In this case, using that \( \alpha \in G^i_v \) implies \( \text{first}(I(\alpha)) \leq i \leq \text{last}(I(\alpha)) \), an analogous argument as given after (5) shows that \( \binom{n}{\text{last}(I(\alpha))} \) can be replaced by \( \binom{c_1}{\min_{m_{\ell,n/2}}^n} = \mathcal{O}(1) \) in (5). Continuing the argument as before yields the improved runtime bound \( \mathcal{O}(\ell^2 m^2) \) for the algorithm \textsc{LexMin()}.

- Every tool appears frequently: Formally, there is a constant \( c_2 \geq 1 \) such that for any tool \( \alpha \in [n] \), no two consecutive entries of \( I(\alpha) \) differ by more than \( c_2 \). In this case the number of different sequences \( y^v \) arising in step (b2) is at most \( \sum_{i=0}^{m} (i + 1)^{c_2 - 1} \leq (m + 1)^{c_2} \) (the term \( (i + 1)^{c_2 - 1} \) bounds the number of partitions of the integer \( i \) into at most \( c_2 \) summands), yielding the bound \( \overline{\mathcal{O}}(\ell^2 m^{c_2 + 1}) \) for the running time of \textsc{LexMin()}.

This proves the runtime bound stated under (ii) in the introduction.

(iv), (v) For unit setup and processing times \( (s(\alpha) = 1 \text{ for every tool } \alpha \in [n] \text{ and } p_i = 1 \text{ for every job } i \in [\ell]) \) the interval representation computed in line E1 satisfies \( l_{a,1} = u_a \) for all \( a \in [S] \), i.e.,
we have \( |\mathcal{X}^S| = 1 \) (there is no choice where to place stops, and therefore only a single minimal stop program). For zero setup times and arbitrary processing times \( s(\alpha) = 0 \) for every tool \( \alpha \in [n] \), we have \( S = 0 \) (no stops at all) and consequently \( \mathcal{X}^S = \{()\} \) (only a single minimal stop program with no stops). Consequently, in both cases the tree \( T^+ \) consists only of a single path with exactly one node on each level, i.e., we have \( |V_i| = 1 \) for all \( i \in [\ell] \). It follows that the bounding step \[ \mathbf{b} \] can be skipped, so that the main loop of LEXMIN() runs in time \( \mathcal{O}(\ell m) \). The runtime bound for INTREP() improves to \( \mathcal{O}(\ell m) \) in both cases (see Section 3.3.2), the same bound also holds for SPLITBLOCKS(). Consequently, the algorithm LEXMIN() runs in time \( \mathcal{O}(\ell m) \) in these cases, as claimed under \[ \mathbf{iv} \] and \[ \mathbf{v} \] in the introduction.

\[ \mathbf{vi} \] If our goal is to minimize stops as the only objective (and neglect tool switches entirely), then we only need to run the algorithm MINSPP() (this is the first step of the algorithm INTREP()), which constitutes the first step of LEXMIN()), which runs in time \( \mathcal{O}(\ell m) \) (see Section 3.2.2).

5. Computational results

We implemented the algorithm LEXMIN() in C++ and tested it on several real-world instances from an industrial application that is similar to mailroom insert planning as described in the introduction (see [ABZ15]). These instances with up to \( \ell = 2650 \) jobs and up to \( m = 25 \) slots were solved optimally in less than 1 s on a standard desktop computer (3.4 GHz Intel i7 processor). We also generated random instances with up to \( \ell = 1000 \) jobs and \( m = 1000 \) slots, and obtained only slightly larger running times (see the rightmost column in Table 2). We emphasize here that in many of these cases the cardinality of \( \mathcal{X}^S \), i.e., the number of ways stops can be placed, is astronomically large (see the second to last column in the table and recall Remark [1]). Moreover, several of these results were obtained for setup times \( s(\alpha) = \infty \) for every tool \( \alpha \in [n] \), which in general are hardest to solve (recall \[ \mathbf{iii} \] from the introduction, and note that large setup times create many dependencies between different jobs.

| Name | \( n \) | \( \ell \) | \( s(\alpha) \) | \( p_i \) avg, min, max | \( m \) | \( d \) | \( S \) (stops) | \( C \) (tool switches) | \( |\mathcal{X}^S| \) | runtime [s] |
|------|-------|-------|----------------|------------------------|-------|-------|-------------|-----------------|-----------------|----------|
| real-1 | 123   | 330   | (0, 900, \( \infty \)) | 507, 17, 3146 | 24 | 0.58 | (0.102, 113) | \{813, 913, 929\} | \{1, 13 \cdot 10^4, 17 \cdot 10^9\} | 0.03, 0.03, 0.03 |
| real-2 | 149   | 610   | (0, 900, \( \infty \)) | 784, 5, 5350 | 25 | 0.57 | (0, 10, 43) | \{270, 302, 288\} | \{1, 16 \cdot 10^4\} | 0.03, 0.05, 0.06 |
| real-3 | 152   | 540   | (0, 900, \( \infty \)) | 941, 5, 6790 | 21 | 0.59 | (0, 19, 48) | \{265, 286, 285\} | \{1, 432, 32 \cdot 10^9\} | 0.03, 0.03, 0.05 |
| real-4 | 179   | 530   | (0, 900, \( \infty \)) | 916, 5, 9435 | 21 | 0.62 | (0, 75, 124) | \{545, 598, 594\} | \{1, 15 \cdot 10^4, 24 \cdot 10^2\} | 0.03, 0.05, 0.02 |
| real-5 | 113   | 2650  | (0, 900, \( \infty \)) | 230, 5, 7599 | 19 | 0.47 | (0, 47, 116) | \{446, 498, 488\} | \{1, 26 \cdot 10^3, 11 \cdot 10^2\} | 0.05, 0.08, 0.17 |
| rand-1 | 1500  | 1000  | (0, 1, \( \infty \)) | 1 | 1000 | 0.52 | (0, 0, 182) | \{27399, 28617, 35122\} | \{1, 1, 11 \cdot 10^9\} | 0.34, 0.34, 1.34 |
| rand-2 | 2500  | 1000  | (0, 1, \( \infty \)) | 1 | 1000 | 0.47 | (0, 1, 277) | \{75278, 84483, 90244\} | \{1, 1, 30 \cdot 10^9\} | 0.42, 0.44, 1.23 |

Table 2. Computational results for five real-world and two random instances. Columns 2–6 and 8–9 show the input and output parameters as in the pseudocode description of the algorithm LEXMIN(). Column 7 shows the plan density \( d := \sum_{i \in [\ell]} |J_i|/(\ell m) \). The values for \( |\mathcal{X}^S| \) in column 10 were computed exactly from the interval representation of \( \mathcal{X}^S \), and rounded to two significant digits. The last column shows the running time of our algorithm. For the instances real-x, the processing times \( p_i \) are all different, and the table shows the average, minimum and maximum over all jobs. The setup time is \( s(\alpha) = 900 \) for every tool \( \alpha \in [n] \), and we also considered the modified problems where either \( s(\alpha) = 0 \) or \( s(\alpha) = \infty \) for all \( \alpha \in [n] \) (recall the remarks under \[ \mathbf{iii} \]–\[ \mathbf{v} \] from the introduction). The three result values in columns 8–11 correspond to these three setup time settings. For the instances rand-x, we chose unit processing times \( p_i = 1 \) and three different setup time settings: either \( s(\alpha) = 0 \), \( s(\alpha) = 1 \) or \( s(\alpha) = \infty \) for every tool \( \alpha \in [n] \).
To further explore the practical limits of the algorithm \textsc{LexMin}(), we varied the setup times and the number of slots for our largest real-world instance \textbf{real-5}. The left hand side of Figure 5 shows the results of increasing the setup times (for a fixed number of slots). The right hand side of the figure shows the results of increasing the number of slots from \( m = 17 \) to \( m = 113 \) (for fixed setup times \( s(\alpha) = \infty \) for every tool \( \alpha \in [n] \); note that \( m = 113 \) is the largest interesting value, as this instance has \( n = 113 \) tools). Observe that while most of the resulting problems were solved similarly fast as before, the hardest one took 35 s (this happened to be for \( m = 66 \) slots). As can be seen from this figure, the running time of \textsc{LexMin}() is essentially determined by the number of nodes of \( T^+ \) that are visited by the algorithm, and the hard instances are the ones where our pruning criterion is least effective.

Overall, our experiments provide strong evidence that the general runtime bound for the algorithm \textsc{LexMin}() stated under (i) in the introduction is overly pessimistic in most cases, and that instead the improved bounds stated under (ii) apply (recall the corresponding points (i) and (ii) from Section 4.3.3).

6. Concluding remarks

We conclude this paper by indicating some directions for further research.

Even though the algorithms presented in this paper solve the lex-minimization of stops and tool switches efficiently for many interesting cases (including probably most instances that are relevant in practice), it remains an interesting theoretical problem to precisely pin down the complexity status of the general problem (polynomial or NP-hard?) when the number of slots \( m \) is not constant. The runtime bound for our algorithm stated under [i] in the introduction is clearly polynomial in \( \ell \) and \( n \) (the number of jobs and tools, respectively), but not in \( m \).
In [PF95] an algorithm based on flow networks was presented that allows tool switch minimization for more general switching costs that are e.g. dependent on the two tools being exchanged (this is a vast generalization of our model where every tool switch costs +1). It would be of great practical value to devise an algorithm that allows specifying such general switching costs, and that simultaneously considers also other objectives, as e.g. minimizing the number of stops.

In our model, setup requirements never extend past a stop (each stop is ‘long enough’ to complete all setups that can be performed during the stop). This captures scenarios where setup operations can only be performed during stops (as e.g. in the single magazine setting of CNC machines mentioned in the introduction), or where the time required for stopping and restarting the manufacturing system is larger than the setup times. As mentioned before, in those cases minimizing stops is equivalent to minimizing the production makespan. However, in applications where the time consumed by a stop depends on the setup times of the tools (so setup requirements can extend beyond a stop, depending on its duration), one might aim for minimizing the production makespan, rather than the number of stops (then the two objectives are not equivalent anymore in general). It is not hard to see that in this setting minimizing the makespan (as a single objective) can be achieved by a straightforward adaption of the greedy backward-in-time algorithm presented in Section 3.2. It would be interesting to investigate whether the makespan objective can be combined e.g. with the minimization of tool switches (as a secondary objective), possibly using a similar branch-and-bound approach as described in this paper.

References