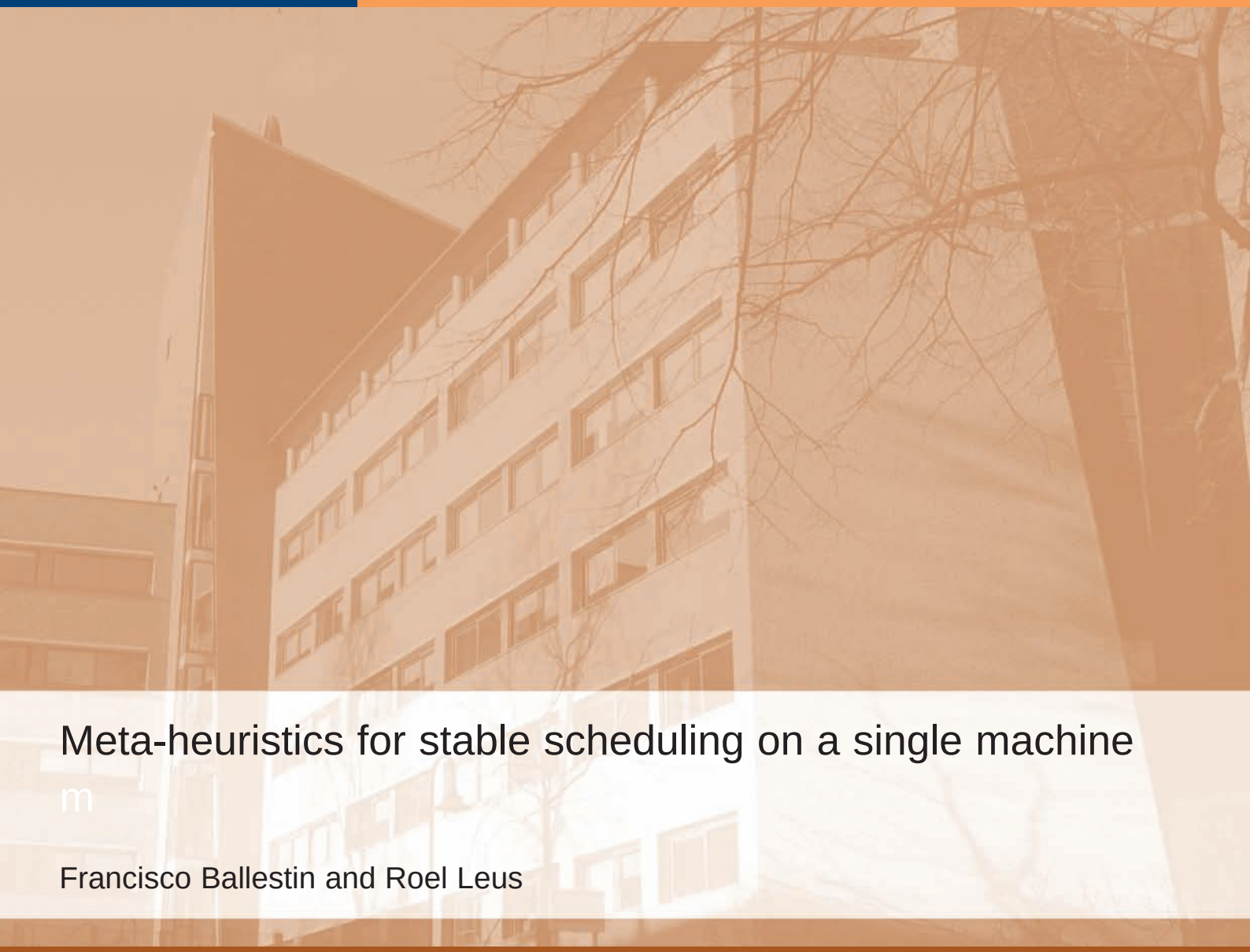
KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculty of Economics and Applied Economics



Meta-heuristics for stable scheduling on a single machine

Francisco Ballestin and Roel Leus

DEPARTMENT OF DECISION SCIENCES AND INFORMATION MANAGEMENT (KBI)

KBI 0607

Meta-heuristics for stable scheduling on a single machine

Francisco Ballestín[1] • Roel Leus[2,*]

[1] Department of Statistics and Operations Research
Universidad Pública de Navarra, Spain
Francisco.Ballestin@unavarra.es

[2] Department of Decision Sciences and Information Management
Katholieke Universiteit Leuven, Belgium
Roel.Leus@econ.kuleuven.be

This paper presents a model for single-machine scheduling with stability objective and a common deadline. Job durations are uncertain, and our goal is to ensure that there is little deviation between planned and actual job starting times. We propose two meta-heuristics for solving an approximate formulation of the model that assumes that exactly one job is disrupted during schedule execution, and we also present a meta-heuristic for the global problem with independent job durations.

Keywords: single-machine scheduling; uncertainty; robustness; meta-heuristics.

1. Introduction

This paper is concerned with the development of a single-machine schedule in an environment with uncertain job durations. Our goal is to ensure that there is little deviation between planned and actual job starting times. The set of jobs to be performed is known at the start of the scheduling horizon and is entirely included into the schedule. This schedule is set up before any job processing takes place, which positions this article within the discipline of *static scheduling*, as opposed to the research in *dynamic scheduling*, where jobs are gradually selected for processing as reality unfolds and information on job-duration realizations becomes available. Slightly different but comparable definitions appear in [10] and [26].

Obviously, in order to produce a schedule, the scheduler needs deterministic job durations, well knowing that deviations from these values will inevitably occur when the schedule is actually implemented. We call the resulting plan a *baseline* schedule or *predictive* schedule. The benefits of adopting such a deterministic predictive schedule in spite of the uncertainty inherent to the environment are twofold. On the one hand, in many production shops, auxiliary resources such as tooling or staffing need to be reserved ahead of time, and the baseline

*Corresponding author

provides time windows for these 'bookings'. On the other hand, and in much the same way, the baseline schedule is also the starting point for communication and coordination with external entities in the company's inbound and outbound supply chain: it constitutes the basis for agreements with suppliers and subcontractors (e.g. for planning external activities such as material procurement and preventive maintenance), as well as for commitments to customers (delivery dates). The usefulness of a predictive schedule is further discussed in [2, 17, 20, 29].

When disruptions take place during schedule execution (a disruption being the receipt of information that a particular job duration was estimated wrongly in the baseline), the baseline schedule needs to be repaired (these computations are sometimes also called *reactive scheduling* or *rescheduling* in the literature). If the scheduler wishes to safeguard the advantages of the baseline schedule as cited above then the actual start of each job should take place as closely as possible to its baseline starting time. This property is referred to as *stability*. A number of sources in the literature have examined how to repair a schedule with a stability objective, e.g. [1, 5, 7, 23, 24, 29].

The current article is concerned with the incorporation into the baseline schedule of advance protection against disruptions. In this way, it classifies under the header of *robust scheduling*, which is a field of scheduling theory that is receiving increasing attention, see e.g. [6, 16]. More particularly, our goal is to introduce stability into the baseline. Previous examples with similar objectives are [20] (in a job-shop environment), [22] (on a single machine) and [28] (for resource-constrained project scheduling). This article is a continuation of the research of Leus and Herroelen ([17]), who analyze the complexity status of a particular single-machine problem (see Section 2 for details) and propose an exact algorithm that is able to produce optimal solutions for small problem instances. The purpose of this paper is to develop sub-optimal algorithms that yield high-quality schedules for large instances. Our contributions are the following: we propose two meta-heuristics for solving an approximate formulation of the model that assumes that exactly one job is disrupted during schedule execution, and we also present a meta-heuristic for the global problem with independent job durations. We show that the easier approximate problem is useful, and can even outperform the correct formulation when both are solved heuristically and variability is low.

In the following section we provide the necessary notation and a formal statement of the two problems that are studied. All of the proposed algorithms follow the same global structure, which is described in Section 3. Section 4 presents two meta-heuristics for the

approximate formulation (named SWOD). Section 5 deals with the generation of good solutions for the problem with independent job durations. Extensive computational results are included within each section. Finally, a number of conclusions are formulated in Section 6.

2. Notation and problem statement

2.1 Definitions and objective function

A set of jobs $N = \{1, 2, \ldots, n\}$ with deterministic baseline durations d_i ($i \in N$) is to be scheduled on a single machine; all jobs are available for processing at the beginning of the planning period. A baseline schedule is an n-vector $\mathbf{s}$, which specifies a starting time s_i for each job i. There is a common deadline ω for all the jobs (e.g. one day's production-shift length): $s_i + d_i \leq \omega, \forall i \in N$. The actual duration of i is a stochastic variable D_i, which need not always equal d_i. The actual starting time $S_i(\mathbf{s})$ of job i is a random variable that is dependent on $\mathbf{s}$ (see below). A non-negative integer cost c_i is incurred per unit-time deviation in the start time of job i, as a penalty for the resulting system nervousness, shop-coordination difficulties and the delivery delay to the customer. The expected weighted deviation between *actual* and *planned* job starting times is the stability measure for schedule $\mathbf{s}$: we minimize objective function $\sum_{i \in N} c_i |E[S_i(\mathbf{s})] - s_i|$, where $E[\cdot]$ is the expectation operator. In the remainder of the article, we omit the argument $\mathbf{s}$ when there is no danger of confusion.

Stochastic job duration D_i is modeled by means of discrete scenarios: let random variable L_i denote the increase in d_i if i is 'disrupted', which takes place with probability π_i; D_i equals the baseline duration d_i with probability $(1-\pi_i)$. L_i is discrete with probability-mass function $g_i(\cdot)$, which associates non-zero probability with positive values $l_{ik} \in \Psi_i$, where Ψ_i denotes the set of disruption scenarios for the duration of job i. $\sum_{k \in \Psi_i} g_i(l_{ik}) = 1$ and g_{ik} is used as shorthand for $g_i(l_{ik})$; the disruption lengths l_{ik} are indexed from small to large for given i. Values l_{ik} are assumed to be integer and the D_i for different jobs i are independent.

Sequencing decisions are represented by a bijection $L : \{1, \ldots, n\} \rightarrow N$, where $L(p)$ is the index of the job in position p in the sequence (L represents a job list). Each such bijection is in one-to-one correspondence with a total order on set N. Λ denotes the set of all job lists. A schedule $\mathbf{s}$ is unequivocally determined by a job list, representing the sequencing decisions, together with the following set of decision variables:

F_p = amount of idle time inserted immediately after $L(p)$ ($p = 1, \ldots, n$).

Inserted idle time can be envisaged as *buffer* time used to cushion the propagation of a disruption towards the (machine) successors of the disrupted job. Quantities F_p are collected in n-vector $\mathbf{f}$. The values in $\mathbf{f}$ are valid if

$$\sum_{p=1}^{n} F_p = \omega - \sum_{i=1}^{n} d_i.$$

Φ is the set of all valid buffer-size vectors; we restrict our search to integral buffer sizes. A pairwise-interchange argument shows that for any two consecutive jobs $L(p) = i$ and $L(p+1) = j$ $(p = 1, \ldots, n-1)$ in an optimal solution either $\pi_i E_i[L_i] c_j \leq \pi_j E_j[L_j] c_i$ or $F_p > 0$, otherwise the solution is dominated, with $E_i[\cdot]$ the expectation operator with respect to L_i. A combination of sequencing decisions L and buffer sizes $\mathbf{f}$ completely determines a baseline schedule $\mathbf{s}(L, \mathbf{f})$ in the following way:

$$s_i(L, \mathbf{f}) = \sum_{p=1}^{L^{-1}(i)-1} \left(d_{L(p)} + F_p \right) \qquad i = 1, \ldots, n. \tag{1}$$

Note that implicitly $s_{L(1)} = 0$.

We assume that jobs are never started earlier than their baseline starting time: $s_i \leq S_i$, $\forall i \in N$, which guarantees that actual production will strictly copy the baseline if no disruptions occur. In effect, the baseline starting times become 'release dates' for schedule execution. A motivation for this approach is given in [17]. The realization of D_i becomes known when job i is executed; the exact timing of this information is not important since we reschedule by right-shifting the remaining jobs without re-sequencing:

$$\begin{cases} S_{L(1)} = s_{L(1)} \\ S_{L(p)} = \max\{s_{L(p)}; S_{L(p-1)} + D_{L(p-1)}\}, \qquad p = 2, \ldots, n. \end{cases}$$

The resulting problem with independent job durations and objective function

$$\min_{L \in \Lambda, \mathbf{f} \in \Phi} \qquad g(\mathbf{s}(L, \mathbf{f})) = \sum_{i=1}^{n} c_i (E[S_i] - s_i) \tag{2}$$

is called STABILITY.

2.2 One-disruption model

To evaluate the objective function for STABILITY for a feasible solution $\mathbf{s}$, little less is possible than to evaluate all $\prod_{i \in N} (|\Psi_i| + 1)$ possible combinations of duration disruptions. A pseudo-polynomial time algorithm can be used but remains computationally unattractive.

Efficiently producing optimal scheduling solutions to STABILITY therefore seems illusory. This was the motivation in [17] to develop a model that focuses only on the main effects of the separate disruption of each of the n jobs rather than on all possible disruption interactions.

Define I_i to be the indicator variable that is 1 if job i is disrupted, 0 otherwise, so $K := \sum_{i \in N} I_i$ is the number of disrupted jobs. The objective function (2) is altered as follows, yielding problem STABILITY WITH ONE DISRUPTION (SWOD):

$$\min_{L \in \Lambda, \mathbf{f} \in \Phi} \qquad h(\mathbf{s}(L, \mathbf{f})) = \sum_{i=1}^{n} c_i (E[S_i | K = 1] - s_i). \tag{3}$$

The model assumes that exactly one job suffers a disruption from its baseline duration. The resulting restricted model is useful when disruptions are sparse and spread over time so that the number of interactions is limited. Computational results show that the model is quite robust to variations in the expected number of disrupted jobs $E[K] = \sum_{i \in N} \pi_i$ and performs best for low $E[K]$. Stand-alone evaluation of $h()$ requires $O(n^2 \Psi_{\max})$ time.

We let $p_i = Pr[I_i = 1 | K = 1]$ represent the probability that job i is the unique disrupted job, conditional on exactly one job being disrupted. Objective function (3) can then be rewritten as

$$\min \qquad \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{|\Psi_i|} \alpha_{ijk} \Delta_{ijk}. \tag{4}$$

In this expression, $\alpha_{ijk} = p_i g_{ik} c_j$ and

$$\Delta_{ijk} = \max \left\{ 0 \quad ; \quad l_{ik} - \left(\sum_{p=L^{-1}(i)}^{L^{-1}(j)-1} F_p \right) \right\}, \qquad i, j = 1, \ldots, n; k = 1, \ldots, |\Psi_i|,$$

the delay in the start time of job j due to a disruption according to scenario k of job i when $K = 1$. Δ_{ijk} is equal to zero or to the disruption length of i minus the buffer size in place between the positions of jobs i and j, whichever is larger. In (4), the expected value of the starting-time delay of job j is computed by summing the values Δ_{ijk} weighted with probability $p_i g_{ik}$ and cost c_j.

The scheduling problem SWOD as set out above has been shown to be $\mathcal{NP}$-hard in the ordinary sense in [19], even if all $|\Psi_i| = 1$. A similar proof can be set up to show strong $\mathcal{NP}$-hardness ([17]). However, determination of optimal idle times $\mathbf{f}$ for a *given* sequence L can be performed in polynomial time using network-flow techniques (see [15, 17]). In [17] the intractability of STABILITY is also discussed and it is shown that, without loss of generality, we can set all job durations equal to zero if we accordingly subtract $\sum_{i=1}^{n} d_i$ from ω. This

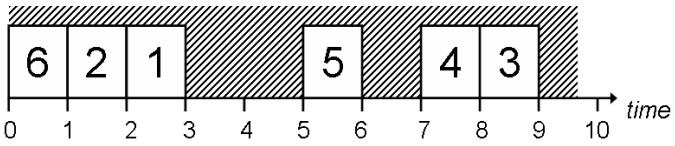


Figure 1: Schedule for the example problem when $\omega = 9$.

operation is assumed to have been applied to all input instances in our computations, except for the computational illustration in Section 2.3, where we consider unit durations for ease of exposition. When the available float is zero, i.e. for the case $\omega = \sum_{i=1}^{n} d_i$, ordering the jobs in non-decreasing expected weighted disruption length $p_i E_i[L_i]/c_i$ leads to an optimal schedule for SWOD, which can be shown by an adjacent-interchange argument. The same holds for STABILITY for quantity $\pi_i E_i[L_i]/c_i$. We refer to this rule as the *EWDL*-rule (for *expected weighted disruption length*); the rule always leads to the same sequence(s) for the two problems.

2.3 Illustration

We illustrate the problem setting by means of a brief example. Consider a problem instance with $n = 6$ jobs where all jobs have equal duration $d_i = 1$, and a time horizon of $\omega = 9$ time units is allotted to the set of jobs. Consequently, we have three spare units of time that can serve as a buffer. Tasks indexed 5 and 6 are considered to be of high importance, the cost of delay in their starting times is $c_5 = c_6 = 4$; the other jobs $i \neq 5, 6$ have $c_i = 1$. Further data are provided in Table 1. Job 1, for instance, has a probability of three out ten of suffering a duration disruption, and if this occurs, it will be an increase of either one or two time units, both equally likely.

job i	1	2	3	4	5	6
π_i	0.3	0.05	0.3	0.1	0.25	0.1
$\lvert\Psi_i\rvert$	2	2	1	2	2	1
$l_{i1}(g_{i1})$	1 (0.5)	1 (0.7)	2 (1)	2 (0.5)	1 (0.5)	2 (1)
$l_{i2}(g_{i2})$	2 (0.5)	2 (0.3)	-	4 (0.5)	2 (0.5)	-
p_i	0.292474	0.035918	0.292474	0.075827	0.22748	0.075827
$\pi_i E_i[L_i]/c_i$	0.45	0.065	0.6	0.3	0.09375	0.05
$p_i E_i[L_i]/c_i$	0.438712	0.046693	0.584949	0.22748	0.085305	0.037913

Table 1: Disruption data for the example problem.

An optimal solution to the corresponding instance of SWOD is depicted in Figure 1, its objective-function value is 0.805. In this schedule the starting time of job 5 is protected from a disruption of up to two time units in the duration of jobs 1, 2 or 6. The available idle time is put to good use: if we reduce ω to 6 (no idle time anymore), the optimal solution attains an associated cost of 3.4574 for job sequence 6-2-5-4-1-3. Sequence 6-2-1-5-4-3 (optimal for $\omega = 9$) corresponds with a cost of 5.0823 when $\omega = 6$, whereas 6-2-5-4-1-3 leads to a cost of at least 1.2509 when the scheduling horizon is nine time units.

Multiple sets of values π_i exist corresponding with the same values p_i: the dependent durations have one less 'degree of freedom' (for details we again refer to [17]). For increasing $E[K]$, corresponding with more variability in the system, Table 2 computes the appropriate π_i-values for the dependent durations in the example (where $E[K] = 1.1$) and also gives the objective-function values for $\mathbf{s}_{SWOD}$, the optimal SWOD-schedule (which is the same for each line of the table), as well as for $\mathbf{s}_{STAB}$, the schedule that is produced by the heuristic procedure for solving STABILITY that we present in Section 5. The objective function is evaluated using simulation. Notice that $p_i \neq \pi_i$, $i = 1, \ldots, 6$, even for $E[K] = 1$.

$\mathbf{s}_{STAB}$ is consistently better than $\mathbf{s}_{SWOD}$, which is expectable since the latter is a heuristic solution for STABILITY. We also observe that the difference increases with $E[K]$, which is in line with our remark that SWOD is most useful for a low number of disrupted jobs. However, the remainder of this paper will show that, for heuristic purposes and with the same time limit, an algorithm for SWOD can search many more solutions than an algorithm for STABILITY, which will enable it to produce better solutions in some cases.

$E[K]$	π_1	π_2	π_3	π_4	π_5	π_6	$g(\mathbf{s}_{SWOD})$	$g(\mathbf{s}_{STAB})$	difference
0.5	0.14	0.0196	0.14	0.0405	0.1124	0.0405	0.477	0.461	3.52%
1	0.2726	0.044	0.2726	0.0885	0.2257	0.0885	1.149	1.063	8.07%
1.5	0.3963	0.0746	0.3963	0.1455	0.338	0.1455	2.068	1.824	13.40%
2	0.508	0.1125	0.508	0.2112	0.4454	0.2112	3.271	2.747	19.07%
2.5	0.6081	0.1601	0.6081	0.2869	0.5469	0.2869	4.802	3.851	24.72%
3	0.6962	0.2196	0.6962	0.3727	0.6406	0.3727	6.711	5.023	33.60%
4	0.8361	0.3851	0.8361	0.5694	0.7986	0.5694	11.712	7.285	60.77%
5	0.9346	0.6372	0.9346	0.7876	0.9175	0.7876	18.272	8.994	103.16%
6	1	1	1	1	1	1	26.107	10.904	139.43%

Table 2: Comparison of STABILITY and SWOD for different values of $E[K]$.
'difference' = $[g(\mathbf{s}_{SWOD}) - g(\mathbf{s}_{STAB})]/g(\mathbf{s}_{STAB})$.

2.4 Implementation and data generation

The performance of the algorithms presented in the next sections is examined by means of computational experiments using randomly generated datasets. The coding was performed in C using the Microsoft Visual C++ 6.0 programming environment, and the experiments were run on a Samsung X15 Plus portable computer with Pentium M processor with 1,400 MHz clock speed and 512 MB RAM, equipped with Windows XP.

The experimental design adopted for this study consists of datasets of 25 problem instances per value of n; we consider $n = 10, 20$ and 50 jobs. For each job i in each instance of each dataset, we directly generate p_i-values rather than π_i. All job durations are uncertain: for each job i, a value q_i is selected from the discrete domain $[1; 10]$ and these values are normalized to probabilities p_i. Cost coefficients c_i are integer values randomly selected from $[1; 5]$. The disruption length L_i is a discrete random variable for which g_i is a discretization of the linear function $G_i(x) = 2(1/C_i - x/C_i^2)$, for which the intercept C_i with the abscissa is a realization of a discrete uniform random variable with support $[2; 25]$. Scenarios $k \in \Psi_i$ are determined as follows: l_{i1} is randomly selected from the discrete values in $[1; \min\{4, C_i - 1\}]$ and additional scenarios $l_{ik} = l_{i,k-1} + 5$ are added while $l_{ik} \leq C_i - 1$. These choices are largely the same as those made in [17]. As for ω, we consider three settings, which are dependent on the number of jobs: $\omega = \sqrt{n}$ (*low* sum of buffer sizes), $\omega = 2\sqrt{n}$ (*medium* protection), and $\omega = 4\sqrt{n}$ (*high* cumulative buffers), each time rounded to the nearest integer.

Jobs with zero cost coefficient can be sequenced last: this is always a dominant decision. Similarly, jobs i with zero $p_i E_i[L_i]$ can be scheduled first on the machine without loss of better solutions. Such instances are actually never encountered in our experiments because of the way in which the data are generated. In the following we also work with $\mathbf{f}$ as an $(n-1)$-vector (instead of dimension n): without loss of better solutions, we assume $F_n = 0$.

3. Global algorithmic structure

The three algorithms that will be presented in this article follow the same overall logic structure represented in pseudocode as Algorithm 1. Deviation from this framework might allow for even more efficient or effective solution approaches, but we have decided to adhere to the same structure so that the intrinsic underlying solution 'philosophy', which differs between the three algorithms, is put into perspective: we examine one algorithm (SW1) for the approximate formulation SWOD, which adopts a hierarchic treatment of sequencing and

timing decisions; one algorithm (SW2) for SWOD with more monolithic treatment of the same two decisions; and one solution approach for the exact problem formulation (STAB).

The algorithms manipulate a collection of solutions (called 'population') of constant size *popsize* rather than a single solution at each stage, and fit within the general framework of 'adaptive memory programming' ([27]). The discussion in the following sections will highlight the implementation of each of the main elements of this code separately. Simple common components are the following:

- The termination_criterion for the procedures is the reaching of a bound on the running time.
- The better a job list, the more likely it will be chosen by select as father or mother. This procedure uses regret-based biased random sampling (cfr. [8]).
- 'Bernoulli(p_{bin})' stands for a realization of a Bernoulli-distributed random variable with parameter (success probability) equal to p_{bin}. Obviously, evaluate and fast_descent are applied to the daughter only if a daughter is actually constructed, so if the Bernoulli variable is 1.
- At the end of each iteration, insert checks for son and daughter separately whether it is better than the worst individual in the current population, and if so, replaces the latter by the new solution.

Algorithm 1 Global algorithmic structure

```
construct initial population
while termination_criterion not met do
  select two solutions father and mother from population
  if Bernoulli(p_bin) = 1 then
    obtain son, daughter from binary_operator(father, mother)
    son = mutate(son); daughter = mutate(daughter)
  else
    son = unary_operator(father)
  end if
  evaluate(son); evaluate(daughter)
  son = fast_descent(son); daughter = fast_descent(daughter)
  evaluate(son); evaluate(daughter)
  insert son and/or daughter into population if they are better than current worst solution
end while
```

4. Stability with one disruption (SWOD)

In this section we propose two meta-heuristics for solving problem SWOD. The two algorithms differ considerably from each other, in that the buffer sizes (values $\mathbf{f}$) are determined optimally by a subroutine that functions as a black box in the first algorithm (which we have named algorithm 'SW1'), while in the second algorithm (which is referred to as 'SW2'), buffer sizes *and* sequencing decisions, i.e. the two elements that jointly completely determine a schedule, are both decided on by the meta-heuristic search. The two algorithms are presented in Sections 4.1 and 4.2, respectively.

4.1 Algorithm SW1: separation of sequencing and scheduling

The solution representation of our first algorithm SW1 is a job list L. This gives rise to a close resemblance between SW1 and meta-heuristics for scheduling problems for which the search space can be restricted to active schedules, since job lists are often a preferred solution representation for the latter type of problems (see, for instance, [14]). The meta-heuristic attempts to find a job list $L^* = \arg\min_{L\in\Lambda}\{\min_{\mathbf{f}\in\Phi} h(\mathbf{s}(\mathbf{f}, L))\}$. Below we elaborate on the implementation in SW1 of some of the functions appearing in the global algorithmic framework that was described in Section 3.

4.1.1 evaluate

The objective-function value $h(\mathbf{s}(\mathbf{f}^*(L), L))$ for a given value of L is established by means of (polynomial-time) network-flow techniques (referenced in Section 2.2), which implicitly determine associated optimal buffer sizes $\mathbf{f}^*(L)$. More specifically, for solving the encountered minimum-cost flow problems, we use CS2[1], version 3.9, a practical implementation of a scaling push-relabel method ([11]), which is called as a subroutine. The code was slightly adapted to run under Windows and to guarantee convenient memory management, the algorithm itself was kept unchanged. CS2 take all integer inputs; since quantities p_i and g_{ik} may be fractional, values α_{ijk} are multiplied by factor 100,000 and rounded to the lower integer.

[1]Copyright © 1995-1999 IG Systems, Inc. Commercial use of CS2 requires a license; cfr. website http://www.igsystems.com/cs2/

4.1.2 construct

In order to compose an initial job list we can use one or more priority rules. We order the jobs of N in decreasing value of c_i, $1/p_i$, $1/E_i[L_i]$ and $c_i/(p_iE_i[L_i])$ (the denominator of these fractions is non-zero, cfr. Section 2.4). If we follow this approach deterministically (ties broken by job number), we obtain one job list for each rule. Computational experiments show that the best rule is $c_i/(p_iE_i[L_i])$ for all values of ω, with the largest differences corresponding with low ω – cfr. Section 4.1.8, where a random order is also included. In order to introduce randomness into the ordering process, we generate multiple different solutions starting from the same priority rule. More specifically, *nr_changes* jobs in each list are selected and inserted elsewhere in the list. The initial population is constructed in this way, using each of the foregoing five priority rules for 20% of the population members (all lists resulting from the random rule are independently generated, and the job insertion is not applied).

4.1.3 binary_operator

This part of SW1 has been borrowed from the literature on genetic algorithms (GA) (see [12] for general information on GAs). We use a two-point crossover to combine father and mother into son and daughter. This crossover has proved to work well in a range of different scheduling environments, including scheduling under uncertainty (see e.g. [3, 4, 9, 13]). Therefore, we consider case $p_{bin} = 1$ as a 'touchstone' version of SW1. Let job lists L_F and L_M represent father and mother, respectively. For a two-point crossover we draw two random integers q_1 and q_2 with $1 \leq q_1 < q_2 \leq n$. The daughter's job list L_D is determined as follows:

$$
\begin{aligned}
&L_D(p) = L_M(p), && p = 1, \ldots, q_1;\\
&L_D(p) = \arg\min_{k \in N}\{L_F^{-1}(k) | k \notin \{L_D(1), \ldots, L_D(p-1)\}\}, && p = q_1 + 1, \ldots, q_2;\\
&L_D(p) = \arg\min_{k \in N}\{L_M^{-1}(k) | k \notin \{L_D(1), \ldots, L_D(p-1)\}\}, && p = q_2 + 1, \ldots, n.
\end{aligned}
$$

The first part of the daughter is determined by the mother, the next set of positions is derived from the father but already selected jobs may not be included again. The last positions are again supplied by the mother. The son is constructed similarly by exchanging the role of father and mother.

4.1.4 unary_operator

We employ a steepest-descent algorithm as unary operator for SW1; an overview is provided as Algorithm 2. Since calls to the network solver are highly costly as far as time consumption is concerned, this part of SW1 leaves the implicit buffer sizes, which were optimal for the input job list, unchanged. With each outer iteration of Algorithm 2, the best found neighborhood solution is retained, even if this is not an improving move; this makes the algorithm of the type steepest-descent/shallowest-ascent, which adds to the diversification of the overall search procedure. Only after the final iteration (in the subsequent call to evaluate) is the output job list passed to the network solver to obtain new optimal buffer sizes. It is important to note that this evaluation may actually lead to a better objective-function value than the combination of job list and buffers we started with.

The last line of Algorithm 2 is a call to function EWDL_reordering. This function rearranges its input list L so that each subset of jobs that are adjacent in L and have zero intermediate buffers in input vector $\mathbf{f}_0$ are reordered in *EWDL*-order, which is optimal for fixed buffer sizes.

Algorithm 2 unary_operator for SW1

Input: job list L, buffer sizes $\mathbf{f}_0$ (with normally $\mathbf{f}_0 = \mathbf{f}^*(L)$)
$g_0 = g(\mathbf{s}(L, \mathbf{f}_0))$; *betterfound* = FALSE
for $i = 1$ to *nr_iter* **do**
  for $j = 1$ to *nr_pairs* **do**
    randomly select $\{a_1, a_2\} \subset N$ (all pairs for the same i are different)
    $L' = L$; exchange the positions of a_1 and a_2 in L'
    if $g(\mathbf{s}(L', \mathbf{f}_0)) < g_0$ **then**
      betterfound = TRUE; $L_{better} = L'$; $g_0 = g(\mathbf{s}(L', \mathbf{f}_0))$
    end if
    remember $g(\mathbf{s}(L', \mathbf{f}_0))$
  end for
  set L equal to the list L' with best value g encountered for the current value of i
  L=EWDL_reordering(L,$\mathbf{f}_0$)
end for
if *betterfound* **then**
  return L_{better}
else
  return L
end if

4.1.5 mutate

Mutation is applied to the son and daughter in order to increase the diversity of the population; this function is not guided by the impact on the objective function. We have considered three mutation procedures for SW1, applied to an input job list. Each of these three procedures takes a value $p_{mut} \in [0; 1]$ as parameter.

1. Interchange of pairs of adjacent jobs. For each job pair $(L(p), L(p+1))$ separately $(p < n)$, p_{mut} represents the probability that the mutation is applied: with probability $(1 - p_{mut})$, nothing happens (with L the input list to mutate).

2. Interchange of pairs of jobs, not necessarily adjacent. We compute $n_{mut} = n \times p_{mut}$ and arbitrarily select n_{mut} different pairs (i, j), $i \neq j$, for interchange.

3. For each position p separately, with probability p_{mut}, we insert job $L(p)$ into a randomly chosen position $q \neq p$ by appropriately 'shifting' the other jobs.

Preliminary experiments have shown that 1. actually performs best, and the mutation in the final version of SW1 is therefore based on this (simplest) setting.

4.1.6 fast_descent

This local-search procedure is based on calculating the difference in objective function if two adjacent jobs in a list L are switched (without changing the intermediate buffers). If an improvement is made we continue the procedure with the new job list. As was the case for unary_operator, new optimal buffer sizes for the resulting job list are produced by means of network-flow computations after the end of the procedure.

Interchange of adjacent jobs is computationally attractive because the impact on the objective function is easily computed. The change in the objective function when we exchange the positions of jobs $i = L(p)$ and $j = L(p+1)$ is equal to $c_i p_j E_j[L_j] - c_j p_i E_i[L_i]$ if the buffer $F_p = 0$. For $F_p > 0$, we still only need to recompute Δ_{ixk} and Δ_{jxk}, $\forall x \in N : p \leq L^{-1}(x)$, and Δ_{xik} and Δ_{xjk}, $\forall x \in N : L^{-1}(x) \leq p$, for all relevant scenarios k. Note that jobs that are swapped in unary_operator are not necessarily adjacent in the input list. Although it is still not unavoidable to recompute the objective function 'from scratch' after each move in the latter case, repeated evaluation nevertheless turns out to be overly costly, which is why we only consider adjacent jobs in fast_descent.

Various approaches are possible to exchanging the positions of adjacent jobs. We have tested three, the main distinction being the order in which the switches are made. Preliminary testing has indicated that setting 2. performs best.

1. For $p = 1$ to $(n-1)$, we investigate the exchange of jobs $L(p)$ and $L(p+1)$.

2. $(n-1)$ trials to switch jobs $L(p)$ and $L(p+1)$, once for each value of $p \in \{1, 2, \ldots, n-1\}$, but values are selected in random order.

3. The third version of fast_descent performs an unpredictable number of iterations according to the code description titled 'Fastest descent version 3', which iteratively looks for local improvements.

Algorithm 3 Fastest descent version 3

```
Eligible = {1, 2, ..., n − 1}
while Eligible not empty do
  select and remove a position p from Eligible
  investigate the switch of positions p and (p + 1)
  if improving then
    implement the switch
    if (p − 1) > 0 and (p − 1) ∉ Eligible then add (p − 1) to Eligible
    if (p + 1) < n and (p + 1) ∉ Eligible then add (p + 1) to Eligible
  end if
end while
```

4.1.7 Different versions of the algorithm

For the dataset with $n = 50$, we compare the computational results of algorithm SW1 for various parameter settings; the time limit is n seconds. The reference setting is $p_{bin} = 0$, $p_{mut} = 0.2$, $popsize = 10$, $nr\_changes = 3$, $nr\_iter = 1$, $nr\_pairs = 20$. For each version of the algorithm we report the increase in objective function on changing from the reference to the setting under examination, expressed as a percentage of the reference (each time averaged over the dataset).

Table 3 contains the results for a number of different choices with respect to the construction of the initial population. The reference setting comes out best overall, although for $\omega = 7$ it is not superior. These figures also show that SW1 is robust: it is very little sensitive to changes in parameter choices, which follows from the fact that the absolute value of each of the deviations is below 1%. Our final choice $popsize = 10$ is rather low compared with the existing literature on population-based search but turns out to lead to the best results.

	popsize		*nr_changes*		
ω	5	25	0	1	5
7	−0.09%	−0.13%	−0.02%	−0.20%	−0.03%
14	0.34%	1.38%	0.55%	0.45%	0.38%
28	0.11%	1.47%	−0.12%	0.14%	−0.15%
avg	0.12%	0.91%	0.14%	0.13%	0.07%

Table 3: Computational results for SW1 for different settings for the initial population.

	p_{bin}				*nr_iter* = 2	*nr_iter* = 2
ω	0.1	0.25	0.5	1.0	*nr_pairs* = 20	*nr_pairs* = 10
7	−0.03%	0.16%	1.09%	4.94%	0.19%	0.10%
14	0.10%	1.12%	1.81%	5.50%	1.49%	0.51%
28	0.55%	0.82%	1.86%	5.29%	1.70%	1.17%
avg	0.21%	0.70%	1.59%	5.24%	1.13%	0.59%

Table 4: Computational results for SW1 for additional parameter values.

Choices for other parameters are examined in Table 4. It turns out that the choice $p_{bin} = 0$ is best, which effectively means that the binary operator is never invoked and the steepest descent algorithm is always selected. A rather large difference can be observed between the final variant of SW1 and the touchstone version using the two-point crossover (for which $p_{bin} = 1$). We also conclude from the final columns in the table (as well as from additional experiments, the computational results of which are not included) that SW1 is relatively insensitive to changes in the values of *nr_iter* and *nr_pairs*, but that, overall, the choices made in the reference setting are preferable.

4.1.8 Objective-function comparisons

For $n = 10$, the deviations from the optimal solutions produced by the branch-and-bound (B&B) algorithm described in [17] are given in Table 5, for SW1 and for the truncated B&B; for $n = 20$ we compare the truncated B&B and SW1. We base our comparisons on the objective function as it is computed by CS2 (cfr. Section 4.1.1). The time limit for SW1 is still n seconds, the B&B is truncated after $10n$ seconds (we allot more time to B&B because the algorithm was never actually designed to be interrupted). We observe that the truncated B&B and SW1 exhibit a comparable heuristic performance for $n = 10$, and only fail to produce all optimal solutions for high ω (=13), where for three out of the 25 instances both heuristics do not find a global optimum. For $n = 20$, SW1 significantly outperforms the truncated variant of the B&B algorithm.

	$n = 10$			$n = 20$
	time		truncated	
ω	full B&B	SW1	B&B	SW1
low	194.01s	0.00%	0.00%	−6.37%
medium	418.99s	0.00%	0.00%	−7.48%
high	786.70s	0.15%	0.15%	−9.10%

Table 5: Objective-function comparison with optimal solutions for $n = 10$ and with the truncated B&B for $n = 20$.

ω	rule 1	rule 2	rule 3	rule 4	random	trunc. B&B
7	16.51%	69.42%	61.08%	38.94%	117.38%	10.07%
14	21.17%	68.96%	60.83%	36.62%	104.51%	13.52%
28	24.97%	64.70%	57.51%	32.44%	87.00%	15.95%
avg	20.88%	67.70%	59.81%	36.00%	102.96%	13.18%

Table 6: Performance of the priority rules and the truncated B&B for SWOD compared with the reference setting for SW1; $n = 50$.

We have examined the performance of the different priority rules that are used in the composition of the initial population as a stand-alone heuristic for SWOD when they are combined with optimal buffer insertion (again carried out by CS2); these results are provided in Table 6 and pertain to $n = 50$. The numbering of the four rules follows the order in which they are listed in Section 4.1.2. Even the best priority rule exhibits over 20% deviation from the SW1 solution *on average*. A comparison of SW1 with truncated B&B is also included; we observe that SW1 outperforms the latter algorithm by 10% to 15% on average.

4.2 Algorithm SW2: integration of sequencing and scheduling

Contrary to the approach discussed in Section 4.1, algorithm SW2 takes active control of both job sequencing (under the form of a list of the jobs in N) and of buffer sizes between consecutive pairs of jobs. This integration of the two decisions to be made constitutes the innovative character of SW2: up till now, similar studies ([17, 20, 22, 28]) have always adhered to a hierarchical bi-level approach, where resource allocation (sequencing, in our context) is performed in the first stage and timing decisions (buffer insertion) are made in the second stage. The solution representation for SW2 is a combination of a job list L and an $(n-1)$-vector $\mathbf{f}$ of buffer sizes. Concurrently, we also maintain a set of associated starting times based on Equation (1). In SW2, buffer sizes are computed heuristically rather than optimally but we avoid the time-expensive calls to CS2 (see the discussion in Section 4.2.1).

4.2.1 evaluate

An optimal assignment of buffers to a job list can be established in polynomial time. Nevertheless, this procedure (cfr. Section 4.1.1) still consumes a lot of time when the number of jobs is ten or higher, since it is invoked very frequently. This is the main reason why we have decided to create SW2. In SW1, the determination of optimal buffers was part of the calculation of the objective function. The computational cost of evaluating a solution is drastically decreased for SW2 since starting times are easily derived from the solution codification. As a consequence, we are able to examine more solutions within the same time. The population size is therefore also significantly increased compared to SW1. Still, it should be pointed out that SW1 scans only a dominant set of solutions, namely job lists with optimal associated buffers (most of the time). This is no longer true for SW2, whose search space is larger and contains a whole range of dominated solutions (since we are usually not working with optimal buffers).

4.2.2 construct

We have tested three algorithms for creating initial solutions for SW2 (which are actually stand-alone heuristics for SWOD).

1. Sequential job list and buffer insertion: we create the job lists exactly as in SW1. However, instead of using an exact algorithm for inserting the buffers we use ADFF, a simple heuristic described in [17].

2. Minimization of the cost of inserting a job in a given position. The structure of this function is presented as Algorithm 4. From back to front of the machine, we fill one job position and buffer per iteration. Rather than simply selecting the job leading to (locally) smallest cost (see below), we again introduce randomness. More specifically, F_p is selected as follows: let $\omega_{\max}$ be a random integer in $[\omega/2, \omega]$. F_p is non-zero with probability P; in this case it is a random integer in $[0; \min\{\omega_r, \omega_{\max}\}]$, so that the average buffer size *if* non-zero will be $\min\{\omega_r, \omega_{\max}\}/2$. At the start of the procedure, we wish to spread out the buffers with equal treatment of all positions on the machine, so our desire is for values F_p to follow a multinomial distribution with sum ω and equal probabilities. The expected value of each F_p in this case would be $\omega/(n-1)$. More generally, at an arbitrary stage of the procedure, we wish the expected value of F_p to

equal $\omega_r/(e-1)$, with $e = |\text{Eligible}|$. This is achieved by selecting

$$P = \frac{2\omega_r}{(e-1)\min\{\omega_r, \omega_{\max}\}}.$$

The cost of placing job k in position p is calculated as the cost of the (possible) delay of successor jobs if k is disrupted. This can be calculated exactly since we know all jobs and buffers after $L(p)$.

Algorithm 4 Initial solutions for SW2 version 2

$L(n) = \arg\max_{i \in N} p_i E_i[L_i]/c_i$; Eligible $= N \backslash \{L(n)\}$; $\omega_r = \omega$
for $p = (n-1)$ to 2 **do**
  compute buffer size F_p; $\omega_r = \omega_r - F_p$
  $j = \arg\min_{k \in \text{Eligible}}\{\text{cost of insertion of } k \text{ at position } p\}$
  $L(p) = j$; remove j from Eligible
end for
$F_1 = \omega_r$; $L(1) =$ the single remaining job in Eligible

3. Minimization of the cost of insertion of a job *or* buffer in any position. We construct job lists L_k and buffer sizes $\mathbf{f}_k$ for increasing number of jobs k ($k = 2, \ldots, n$): see Algorithm 5, where 'Uniform(0;1)' stands for a realization of a continuous uniformly distributed random variable on $[0; 1]$, and again $e = |\text{Eligible}|$. Function prior yields the job with highest priority in |Eligible| according to one of the priority rules discussed for SW1.

   In order to compute where to place a unit of buffer in the solution, we analyze every possibility. In this process it is not necessary to re-calculate the entire objective function for each candidate position: the values for adjacent positions are interrelated. With respect to insertion of a job j, we split the partial solution into sublists where the jobs share the same starting time (zero intermediate buffers). We need only consider allocation of j to each of the sublists.

4.2.3 binary_operator

Let $\mathbf{s}_M$, $\mathbf{s}_F$, $\mathbf{s}_D$ and $\mathbf{s}_S$ represent the starting-time vectors for mother, father, daughter and son, respectively. For $i = 1, \ldots, n$, we compute $(\mathbf{s}_D)_i$ as the integer closest to $\alpha(\mathbf{s}_M)_i + (1-\alpha)(\mathbf{s}_F)_i$, with α a random number in $[0; 1]$. L_D orders the jobs in increasing $(\mathbf{s}_D)_i$; in case of ties, the same order as L_M is adopted. The son is created symmetrically. In both cases buffers are implicit from the starting times.

Algorithm 5 Initial solutions for SW2 version 3

Eligible $= N$; $\omega_r = \omega$; $\mathbf{f}_2 = 0$
i=prior; remove i from Eligible; j=prior; remove j from Eligible
if $p_i E_i[L_i]c_j \leq p_j E_j[L_j]c_i$ **then**
  $L_2(1) = i$; $L_2(2) = j$
else
  $L_2(1) = j$; $L_2(2) = i$
end if
while $e > 0$ or $\omega_r > 0$ **do**
  if Uniform(0;1) $< \left(\frac{\omega_r}{\omega_r + e}\right)$ **then**
    increase by one the buffer in $\mathbf{f}_{n-e}$ with best change in objective function; $\omega_r = \omega - 1$
  else
    $j =$ prior
    construct L_{n-e+1} and $\mathbf{f}_{n-e+1}$ by inserting j into L_{n-e} in the best position
    remove j from Eligible
  end if
end while
return L_n and $\mathbf{f}_n$

4.2.4 unary_operator

This procedure is the same as the one described in Section 4.1.4, so without attention to buffers. We underline that this is a deliberate choice: preparatory tests with operators manipulating buffers and job lists at the same time did not lead to improved results.

4.2.5 mutate

The mutation of the job list for SW2 is the same as for SW1. As for the buffers, a parameter $p_{mut} \in [0; 1]$ is chosen ($p_{mut} = 0.1$ in our computations). For each F_p separately, $p = 1, \ldots, n-1$, mutation takes place with probability p_{mut}. If that happens, one of three operations is applied, each equally likely (in all cases, position q is chosen randomly):

1. exchange of F_p with buffer F_q, $p \neq q$, $F_p \neq F_q$.
2. transfer of a random number of time units from buffer $F_q > 0$, $p \neq q$, to F_p.
3. transfer of a random number of time units from F_p to buffer F_q, $p \neq q$, if $F_p > 0$.

4.2.6 fast_descent

We first apply fast_descent for job lists as it is described in Section 4.1.6. Afterwards, local search is applied for the buffer sizes. A number of different moves are investigated. We

differentiate between zero and non-zero buffers and work again with sublists (cfr. Section 4.2.2). The first four moves constitute changes within a sublist; if an improving movement is found, we implement it before going to the next sublist. Each of the following moves is examined for a non-zero buffer F_p:

- FD1fw: Exchange F_p with zero F_q for which $p < q \leq n$ and $\nexists F_r > 0 : p < r < q$.
- FD1bw: Exchange F_p with zero F_q for which $1 \leq q < p$ and $\nexists F_r > 0 : q < r < p$.
- FD2fw: Subtract a random integer quantity $Q \in \{1, 2, \ldots, F_p - 1\}$ from F_p and add Q to zero F_q for which $p < q \leq n$ and $\nexists F_r > 0 : p < r < q$.
- FD2bw: Subtract a random integer quantity $Q \in \{1, 2, \ldots, F_p - 1\}$ from F_p and add Q to zero F_q for which $1 \leq q < p$ and $\nexists F_r > 0 : q < r < p$.

Additionally, we investigate the following moves, which maintain the sublists:

- FD3fw: Subtract a random integer quantity $Q \in \{1, 2, \ldots, F_p - 1\}$ from $F_p > 0$ and add Q to non-zero F_q, with $p < q \leq n$.
- FD3bw: Subtract a random integer quantity $Q \in \{1, 2, \ldots, F_p - 1\}$ from $F_p > 0$ and add Q to non-zero F_q, with $1 \leq q < p$.

Again, there is no need for re-computation of the entire objective function for every move: we extensively exploit previous computations. The six moves are applied one after the other in the order of description.

4.2.7 Computational results

We have selected $p_{bin} = 0.5$ and method 2 for the construction of the initial population as the best setting for SW2. Table 7 compiles some comparisons of this reference setting with different parameter choices; all results pertain to the dataset with $n = 50$ and a time limit of $n/4$ seconds. The table shows that the parameter choices for the reference setting lead to the best results. Contrary to what was found for SW1, binary_operator is useful in SW2: $p_{bin} > 0$ is preferable – although (based on the bad performance of $p_{bin} = 1$) the unary operator still contributes most to the search procedure (we restrict the examined p_{bin} values to 0, 0.5 and 1 since we do not want to 'finetune' the parameters too much). In the table, methods 1, 2

			construct			fast_descent
ω	$p_{bin} = 0$	$p_{bin} = 1$	method 1	method 3	1, 2 and 3	only for job lists
7	0.95%	47.37%	1.23%	0.87%	1.05%	1.08%
14	0.80%	42.10%	1.15%	1.43%	1.83%	0.50%
28	−0.04%	31.53%	0.93%	1.07%	0.81%	1.00%
avg	0.57%	40.34%	1.10%	1.12%	1.23%	0.86%

Table 7: Computational results for different versions of SW2: we report the percentage increase in the objective function compared with the reference setting.

and 3 together for construct refers to the algorithm where the three methods are invoked in equal proportions.

The results in Table 8 demonstrate that, compared with SW1, the allotted computation times can be brought down for SW2 while at the same time higher-quality schedules can be obtained. Specifically, a time limit of $n/4$ seconds is imposed on SW2, which is only one fourth of the time accorded to SW1. A large gain in CPU-time is achieved by not invoking CS2, as explained in Section 4.2.1, and this more than offsets the disadvantage of not working with optimal buffer sizes.

$\omega =$	7	14	28	avg
	1.77%	2.57%	2.34%	2.22%

Table 8: The improvement in the objective function by using SW2 rather than SW1, for $n = 50$.

5. Stability with independent job durations

The meta-heuristic that we propose and which works with independent job durations is called 'STAB' for short. STAB adopts the same algorithmic framework as SW1 and SW2, which was outlined in Algorithm 1. Additionally, STAB also shares with SW2 its solution representation and the functions binary_operator and mutate.

5.1 evaluate

We have mentioned in Section 2.2 that exact evaluation of the STABILITY objective function is overly time-consuming, which is why we approximate the objective-function values by means of simulation. Similar decisions in the context of scheduling under uncertainty have been made by [3, 18, 21, 25], among others. We define a 'replication' as a set of values

$(\lambda_1, \ldots, \lambda_n) \in \times_{i \in N}\{\Psi_i \cup \{0\}\}$ representing realizations of disruption lengths for all jobs (with zero included). Computation of the objective function $g()$ of a given solution is done by averaging the summed weighted deviation for each of the replications. The number of replications *nr_rep* is a parameter of STAB. The same set of replications is maintained for different calls to evaluate, which leads to better results than the case where new replications are sampled at each call, as will be confirmed by the computational results (Table 9).

In order to compare algorithms, we produce a better approximation of the objective-function value of the output schedule of STAB by means of a larger set of one million replications (which is the same for all algorithms included in the comparison); these replications are independent from the initial set of replications used in evaluate. The foregoing approach to handling replications has been borrowed from [3].

Efficiency gains are obtained as follows: since the events $K = 1, K = 2, \ldots$ and $K = n$ are disjoint and jointly exhaustive, we see (from the law of total expectation) that, for a schedule $\mathbf{s}$,

$$g(\mathbf{s}) = \sum_{k=0}^{n} \sum_{i=1}^{n} c_i(E[S_i|K = k] - s_i).$$

Since $S_i = s_i$ for all $i \in N$ when $K = 0$, we can dismiss all replications that are zero vectors and afterwards multiply our estimate with coefficient $(1 - Pr[K = 0])$, where $Pr[K = 0] = \prod_{i \in N}(1 - \pi_i)$. This technique is especially useful when $E[K]$ is low.

5.2 construct

For the construction of the initial population, we create the job list exactly as in SW1. Instead of an exact algorithm for inserting the buffers, however, we use the following heuristic. We distribute ω_1 time units among the buffers that separate job pairs $\{i, j\}$ that are *not* in *EWDL*-order (ω_1 being a parameter to STAB); priority is given to pairs $\{i, j\}$ with high value $|p_j E_j[L_j]/c_j - p_i E_i[L_i]/c_i|$. Only afterwards (since repeated evaluation would computationally be overly expensive), the STABILITY objective g() is computed, leading to an estimate of the cost $c_j(E[S_j] - s_j)$ associated to each job j. Subsequently, the remaining $(\omega - \omega_1)$ units of idle time are randomly spread across the scheduling horizon, with a bias towards positions in front of the jobs with higher cost.

5.3 unary_operator

Based on estimates of the contribution of each job to the objective function (cfr. Section 5.2), *nr_moves* jobs are selected in unary_operator, with a bias towards jobs with higher cost. Each thus selected job is advanced in the list a (random) number of positions. The buffer sizes are kept unaltered.

5.4 fast_descent

If $E[K] \leq 3$, the fast_descent-implementation of SW2 is applied for the SWOD objective function $h()$. The schedule that is obtained replaces the original solution if its score on objective function $g()$ is better than the old solution, otherwise it is discarded. When $E[K] > 3$ we only apply function EWDL_reordering (see Section 4.1.4) and leave the buffer sizes unchanged.

5.5 Computational results

All results in this section were obtained for the dataset with $n = 50$; a time limit of $n/4$ seconds is imposed (the same as for SW2). The parameter values that are retained for the final version of STAB (called 'reference setting') are $p_{bin} = 0.5$ and $nr\_rep = 1000$.

Table 9 provides a comparison of the performance of different versions of STAB; deviations are always expressed as a percentage of the objective-function value for the reference setting. The results are described for different amounts of available idle time (represented by ω) and for different degrees of variability in the system (measured by $E[K]$). The table provides (rudimentary) numerical ground for the parameter choices in the reference setting as well as for the inclusion of fast_descent; it also substantiates our earlier remarks with respect to the implementation of function evaluate (made in Section 5.1). In Table 9, 'no fast_descent' means that function fast_descent is not invoked; comparisons are only useful for $E[K] \leq 3$. 'new replic' refers to the setting where new replications are used for each new call to evaluate. 'include zero' indicates that replications equal to the zero vector are not dismissed from the computations (see Section 5.1); comparisons are only useful for low values of $E[K]$. Contrary to what was found for SW2 (and even more so for SW1), the binary operator is more useful than the unary operator.

Table 10 contains objective-function comparisons between SW2 and STAB (the reference is STAB). We pointed out in Section 2.2 that the SWOD-formulation is most valuable for low

ω	$E[K]$	*nr_rep* = 500	*nr_rep* = 5000	$p_{bin} = 0$	$p_{bin} = 1$	no fast_descent	new replic	include zero
7	1	1.67%	0.35%	1.42%	0.74%	0.75%	5.61%	1.34%
	10	0.15%	0.34%	4.24%	0.19%		1.79%	
	20	−0.01%	0.20%	5.46%	0.05%		1.10%	
14	1	2.06%	0.67%	1.99%	0.49%	1.79%	7.32%	1.14%
	10	0.05%	1.06%	8.17%	0.52%		3.04%	
	20	0.03%	0.59%	10.71%	0.42%		2.12%	
28	1	0.95%	0.19%	1.57%	−0.03%	4.40%	7.25%	0.86%
	10	−0.07%	2.02%	10.78%	0.55%		4.41%	
	20	−0.07%	1.92%	18.53%	1.06%		3.97%	
avg		0.53%	0.82%	6.99%	0.44%	2.31%	4.07%	1.11%

Table 9: Computational results for different versions of STAB: we report the percentage increase in the objective function compared with the reference setting.

	$E[K]$					
ω	0.5	1	1.5	2	3	4
7	−3.17%	−1.78%	0.48%	2.52%	5.83%	9.91%
14	−4.06%	−2.17%	2.10%	4.88%	10.78%	17.84%
28	−5.81%	−4.29%	−1.41%	1.06%	7.03%	11.55%
avg	−4.35%	−2.75%	0.39%	2.82%	7.88%	13.10%

Table 10: Comparison between SW2 and STAB (deviation of SW2 from STAB, as a percentage of the STAB objective function).

values of $E[K]$, which is confirmed here. Additionally, we arrive at the new observation that, when both the formulations STABILITY and SWOD are solved sub-optimally, heuristics that solve the latter (easier) problem can even outperform algorithms that tackle the exact problem statement – but only when there is little variability in the system (as measured by $E[K]$). As $E[K]$ increases, the quality of the solution provided by SW2 becomes increasingly poorer compared to the one obtained by STAB.

6. Summary and conclusions

In this article we have examined a single-machine scheduling problem with variable job durations; our goal was to ensure that little deviation occurs between planned and actual job starting times. Earlier studies have shown that the problem is hard and that exact solutions can only be found for small instances. We have developed three meta-heuristic algorithms that yield high-quality schedules for large instances. The algorithms follow the

same overall logic structure, which has allowed us to bring into focus the intrinsic differences between the solution approaches that underlie each of the algorithms. In particular, we have examined one algorithm (SW1) with hierarchic treatment of sequencing and timing decisions for an approximate formulation called SWOD, in which it is assumed that exactly one job is disrupted during schedule execution; one algorithm (SW2) with more monolithic treatment of the same two decisions for SWOD; and one solution approach (STAB) for the exact problem formulation. The algorithm STAB is also easily adaptable to other modeling choices for the job durations (e.g. when these are continuous and/or dependent random variables) since its objective-function evaluation is based on simulation.

Our conclusions are the following: (1) SW1 is a meta-heuristic that delivers higher-quality schedules than a truncated B&B-algorithm for the problem SWOD; (2) SW2 is an algorithm that attempts to integrate the decisions of resource allocation and timing of the jobs to be performed, and it outperforms SW1, which follows a (more classical) hierarchic approach to these decisions; (3) the approximate formulation SWOD is useful and can even lead to better results than the exact problem formulation in the case where meta-heuristics are used to solve both problems and when there is little variability in the system; and (4) when the chances of individual activity disruption are higher, it is necessary to apply a specialized algorithm for the problem in order to obtain good-quality solutions, e.g. our algorithm STAB.

Acknowledgments

This research was partially supported by the Ministerio de Ciencia y Tecnología (Spain) under contract TIC2002-02510 and the Agencia Valenciana de Ciencia y Tecnología, GRUPOS03/174. Roel Leus is partly funded as Postdoctoral Fellow of the Research Foundation – Flanders (Belgium). This work was done during a stay of the first author at the Katholieke Universiteit Leuven.

References

[1] M.S. Akturk and E. Gorgulu. Match-up scheduling under a machine breakdown. *European Journal of Operational Research*, 112:81–97, 1999.

[2] H. Aytug, M.A. Lawley, K. McKay, S. Mohan, and R. Uzsoy. Executing production

schedules in the face of uncertainties: A review and some future directions. *European Journal of Operational Research*, 161:86–110, 2004.

[3] F. Ballestín. When it is worthwile to work with the stochastic RCPSP? *Journal of Scheduling*. To appear.

[4] F. Ballestín, V. Valls, and S. Quintanilla. Due dates and RCPSP. In J. Jozefowska and J. Weglarz, editors, *Perspectives in Modern Project Scheduling*. Kluwer.

[5] J.C. Bean, J.R. Birge, J. Mittenthal, and C.E. Noon. Match-up scheduling with multiple resources, release dates and disruptions. *Operations Research*, 39:470–483, 1991.

[6] L. Bölöni and D.C. Marinescu. Robust scheduling of metaprograms. *Journal of Scheduling*, 5:395–412, 2002.

[7] K.M. Calhoun, R.F. Deckro, J.T. Moore, J.W. Chrissis, and J.C. Van Hove. Planning and re-planning in project and production planning. *Omega*, 30:155–170, 2002.

[8] A. Drexl. Scheduling of project networks by job assignment. *Management Science*, 37:1590–1602, 1991.

[9] B. Franck, K. Neumann, and C. Schwindt. Truncated branch-and-bound, schedule-construction, and schedule-improvement procedures for resource-constrained project scheduling. *OR Spektrum*, 23:297–324, 2001.

[10] K. Gary, R. Uzsoy, S.P. Smith, and K. Kempf. Measuring the quality of manufacturing schedules. In D.E. Brown and W.T. Scherer, editors, *Intelligent scheduling systems*, pages 211–369. Kluwer, 1994.

[11] A.V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms*, 22:1–29, 1997.

[12] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[13] S. Hartmann. A self-adapting genetic algorithm for project scheduling under resource constraints. *Naval Research Logistics*, 49:433–448, 2002.

[14] S. Hartmann and R. Kolisch. Experimental evaluation of state-of-the-art heuristics for resource-constrained project scheduling problem. *European Journal of Operational Research*, 127:394–407, 2000.

[15] W. Herroelen and R. Leus. The construction of stable project baseline schedules. *European Journal of Operational Research*, 156:550–565, 2004.

[16] P. Kouvelis and G. Yu. *Robust Discrete Optimization and its Applications*. Kluwer Academic Publishers, 1997.

[17] R. Leus and W. Herroelen. Scheduling for stability in single-machine production systems. *Journal of Scheduling*. To appear.

[18] R. Leus and W. Herroelen. Stability and resource allocation in project planning. *IIE Transactions*, 36:667–682, 2004.

[19] R. Leus and W. Herroelen. The complexity of machine scheduling for stability with a single disrupted job. *Operations Research Letters*, 33:151–156, 2005.

[20] S.V. Mehta and R.M. Uzsoy. Predictable scheduling of a job shop subject to breakdowns. *IEEE Transactions on Robotics and Automation*, 14:365–378, 1998.

[21] R.H. Möhring and F.J. Radermacher. The order-theoretic approach to scheduling: the stochastic case. In R. Slowinski and J. Weglarz, editors, *Advances in project scheduling*, chapter III.4. Elsevier, 1989.

[22] R. O'Donovan, R. Uzsoy, and K.N. McKay. Predictable scheduling of a single machine with breakdowns and sensitive jobs. *International Journal of Production Research*, 37:4217–4233, 1999.

[23] A.S. Raheja and V. Subramaniam. Reactive recovery of job shop schedules - a review. *International Journal of Advanced Manufacturing Technology*, 19:756–763, 2002.

[24] R. Rangsaritratsamee, W.G.Jr. Ferrel, and M.B. Kurz. Dynamic rescheduling that simultaneously considers efficiency and stability. *Computers and Industrial Engineering*, 46:1–15, 2004.

[25] F. Stork. *Stochastic resource-constrained project scheduling*. PhD thesis, Technische Universität Berlin, 2001.

[26] V. Suresh and D. Chaudhuri. Dynamic scheduling – a survey of research. *International Journal of Production Economics*, 32:53–63, 1993.

[27] E.D. Taillard, L.M. Gambardella, M. Gendreau, and J.-Y. Potvin. Adaptive memory programming: a unified view of metaheuristics. *European Journal of Operational Research*, 135:1–16, 2001.

[28] S. Van de Vonder, E.L. Demeulemeester, and W.S. Herroelen. An investigation of efficient and effective predictive-reactive project scheduling procedures. *Journal of Scheduling*. To appear.

[29] S.D. Wu, H.S. Storer, and P.-C. Chang. One-machine rescheduling heuristics with efficiency and stability as criteria. *Computers and Operations Research*, 20:1–14, 1993.