mathematics

MDPI

Article

One-Machine Scheduling with Time-Dependent Capacity via Efficient Memetic Algorithms

Raúl Mencía * and Carlos Mencía

Department of Computer Science, University of Oviedo, 33204 Gijón, Spain; menciacarlos@uniovi.es
* Correspondence: menciaraul@uniovi.es

Abstract: This paper addresses the problem of scheduling a set of jobs on a machine with time-varying capacity, with the goal of minimizing the total tardiness objective function. This problem arose in the context scheduling the charging times of a fleet of electric vehicles and it is NP-hard. Recent work proposed an efficient memetic algorithm for solving the problem, combining a genetic algorithm and a local search method. The local search procedure is based on swapping consecutive jobs on a C-path, defined as a sequence of consecutive jobs in a schedule. Building on it, this paper develops new memetic algorithms that stem from new local search procedures also proposed in this paper. The local search methods integrate several mechanisms to make them more effective, including a new condition for swapping pairs of jobs, a hill climbing approach, a procedure that operates on several C-paths and a method that interchanges jobs between different C-paths. As a result, the new local search methods enable the memetic algorithms to reach higher-quality solutions. Experimental results show significant improvements over existing approaches.

Keywords: one-machine scheduling; time-varying capacity; memetic algorithms; local search

Citation: Mencía, R.; Mencía, C. One-Machine Scheduling with Time-Dependent Capacity via Efficient Memetic Algorithms. *Mathematics* **2021**, *9*, 3030. https://doi.org/10.3390/math9233030

Academic Editors: Ana M. Madureira, Joao Ferreira and André Santos

Received: 12 October 2021
Accepted: 22 November 2021
Published: 26 November 2021

1. Introduction

Over the last few decades, scheduling problems have become ubiquitous in a growing number of domains, including manufacturing, transportation or cloud computing, among others [1,2]. These problems often exhibit a high computational complexity [3–6], what makes them an interesting subject of study to several scientific disciplines, as artificial intelligence, operations research or applied mathematics. As a consequence, numerous solving methods, both exact and approximate, have been proposed in the literature, capable of solving increasingly challenging problems.

Exact methods include branch and bound algorithms [7,8], constraint programming [9] or mathematical programming approaches [10], among others.

On the other hand, efficient metaheuristic algorithms have been proposed with the aim of computing high-quality solutions in short time. In this respect, genetic algorithms (GAs) stand out as very effective population-based metaheuristics. These algorithms evolve a population of solutions by means of selection, recombination and replacement genetic operators. GAs have been used to solve numerous scheduling problems, including one-machine [11], parallel machines [12], job shop [13] or resource constrained project scheduling problems [14]. In addition, local search approaches have been widely used in this domain (e.g., to solve one-machine [15], job shop [16] or Earth observation satellite scheduling problems [17], to name a few). In contrast to population-based metaheuristics, these methods work on a single solution, iteratively introducing changes on it to improve its quality. Local search methods have been successfully combined with other metaheuristics as genetic algorithms, resulting in so-called memetic algorithms (MAs). These algorithms have been shown to achieve a proper balance between the exploration and exploitation of the search space, what makes them more effective at solving different scheduling problems, as one-machine [18] or flow shop scheduling problems [19].

Other successful metaheuristics include, among others, differential evolution (DE) [20], ant colony optimization (ACO) [21] or particle swarm optimization (PSO) [22]. In addition, recent work explored hybrid methods combining metaheuristics and machine learning in different domains [23,24].

One-machine scheduling problems have played an important role in scheduling. In general, these problems require scheduling a set of jobs on a unique resource, satisfying diverse constraints, with the goal of optimizing a given objective function. In addition to their many practical applications (e.g., supply chain [25], packet-switched networks [26], or manufacturing [27]), they stand out for acting as building blocks of other more complex problems, usually providing useful approximations or lower bounds [7,28].

This paper studies a problem of this kind that arose in the context of scheduling the charging times of a fleet of electric vehicles [29]. In its formal definition, a set of jobs has to be scheduled on a single machine whose capacity varies over time, with the aim of minimizing the total tardiness objective function. This problem is denoted $(1, Cap(t)||\sum T_i)$ in the conventional $(\alpha|\beta|\gamma)$ notation [30] and it is NP-hard [31,32].

The $(1, Cap(t)||\sum T_i)$ problem has been considered both in online (with real-time requirements) and offline settings. In [29], it was solved by means of the Apparent Tardiness Cost (ATC) priority rule [33], which is of common use in scheduling problems with tardiness objectives. Later, a genetic algorithm [31] was shown to compute much better schedules than classical priority rules, including ATC, at the expense of longer running times. More recently, this genetic algorithm was combined with an efficient local search procedure, resulting in a memetic algorithm [32]. The local search method is based on swapping pairs of consecutive jobs in a so-called C-path, defined as a sequence of consecutive jobs in a feasible schedule. The memetic algorithm was shown to outperform the genetic algorithm by a wide margin and, to our best knowledge, it is the current best-performing offline approach for solving the $(1, Cap(t)||\sum T_i)$ problem. The problem has also been solved in the recent past by means of priority rules evolved by genetic programming [34], as well as ensembles (or sets) of rules [35]. These approaches often produce better schedules than classical rules, as ATC, and are well-suited for solving the problem online, given their very short running times. However, the quality of the schedules they compute was shown to be still significantly lower than that of the schedules calculated by offline methods, as the aforementioned memetic algorithm.

Building on [32], this paper makes several contributions towards solving the $(1, Cap(t)||\sum T_i)$ problem:

- First, new efficient local search procedures for the problem are proposed and their relevant properties, as correctness and worst-case complexity, are studied. As the previous local search approach, the new methods rely on the notion of C-paths in a feasible schedule. However, they incorporate mechanisms to make them more effective. These include a new condition for swapping pairs of consecutive jobs, the integration of a hill climbing approach, a procedure that operates on several C-paths at the same time and a new way of improving the quality of schedules by interchanging jobs between different C-paths.
- Then, the local search procedures are exploited in combination with a genetic algorithm, giving rise to new memetic algorithms. These algorithms have been designed with the aim of achieving a proper balance between the exploration of the search space and the intensification in its most promising areas.
- An extensive experimental study demonstrates that the memetic algorithms proposed in this work achieve conclusive improvements in practice. The results reveal that the new local search procedures enable the memetic algorithms to reach far better solutions than other methods, including the memetic algorithm proposed in [32] and a constraint programming approach.

The remainder of the paper is structured as follows: Section 2 formally defines the $(1, Cap(t)||\sum T_i)$ problem. Section 3 summarizes the main components of the memetic algorithm proposed in [32], providing the necessary background. The new local search

procedures and the new memetic algorithms are described in Sections 4 and 5, respectively. Section 6 reports the results from the experimental study. Finally, the paper concludes in Section 7.

2. Definition of the Problem

In the $(1, Cap(t)||\sum T_i)$ problem n jobs $\mathcal{J} = \{1, \ldots, n\}$ have to be scheduled on a single machine. Each job $i \in \mathcal{J}$ is available at time $t = 0$ and has a given duration p_i and a due date d_i. Processing a job results in the consumption of one unit of the machine's capacity while it is being processed. The capacity of the machine varies over time: for a time instant $t \geq 0$, $Cap(t)$ denotes its capacity in the interval $[t, t+1)$. It is assumed that $Cap(t) > 0$ for all $t \geq 0$.

A feasible schedule S is an assignment of a starting time s_i to each job $i \in \mathcal{J}$ satisfying the following constraints:

- The capacity of the machine cannot be exceeded at any time, i.e., $X(t) \leq Cap(t)$ for all $t \geq 0$, where $X(t)$ denotes the total consumption of the machine in the interval $[t, t+1)$ due to the jobs scheduled. This corresponds to the number of jobs that are processed in parallel in that interval.
- The processing of a job cannot be preempted, i.e., $C_i = s_i + p_i$ for all $i \in \mathcal{J}$, where C_i denotes the completion time of job i.

In a feasible schedule S, each job $i \in \mathcal{J}$ incurs in a tardiness $T_i = \max\{0, C_i - d_i\}$, which measures its delay when the job is completed after its due date. The total tardiness of S, denoted $T(S)$, is defined as the sum of the tardiness values of all the jobs, that is:

$$T(S) = \sum_{i \in \mathcal{J}} T_i \tag{1}$$

The goal is to find a feasible schedule with the minimum total tardiness possible.

Example 1. *Consider a problem instance with a set of jobs $\mathcal{J} = \{1, \ldots, 12\}$, whose durations and due dates are given in the following table:*

i	1	2	3	4	5	6	7	8	9	10	11	12
p_i	4	4	2	3	4	3	2	3	2	3	3	5
d_i	4	9	13	4	7	8	10	3	13	5	9	7

Figure 1 shows a feasible schedule. For each job, its processing time and its due date is represented in parentheses. The capacity of the machine over time, $Cap(t)$, is shown in the Gantt chart as well. The total consumption $X(t)$ is not explicitly represented, but it can be easily seen that it always holds that $X(t) \leq Cap(t)$. In this schedule, the jobs that incur in a positive tardiness are 1 ($T_1 = 5$), 2 ($T_2 = 5$), 4 ($T_4 = 6$), 5 ($T_5 = 9$), 6 ($T_6 = 4$) and 8 ($T_8 = 8$). So, its total tardiness is 37. Figure 2 shows another feasible schedule. As can be seen, the consumption never exceeds the capacity of the machine, even though job 2 is represented above the capacity line in the Gantt chart. In this case, the jobs that incur in a positive tardiness are 1 ($T_1 = 5$), 2 ($T_2 = 6$), 4 ($T_4 = 6$), 5 ($T_5 = 10$), 6 ($T_6 = 5$) and 8 ($T_8 = 8$). So, its total tardiness is 40.

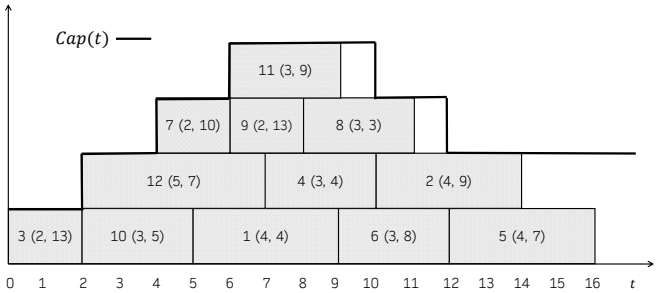


Figure 1. Feasible schedule for the instance in Example 1.

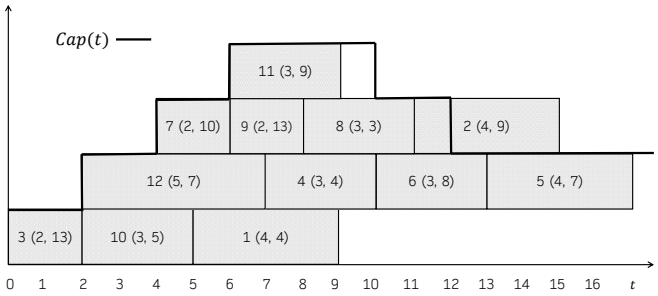


Figure 2. Another feasible schedule for the instance in Example 1.

This problem arose in the context of scheduling the charging times of a fleet of electric vehicles in a community park [29]. In this scenario, it appears as a subproblem of the Electric Vehicles Charging Scheduling Problem (EVCSP), which considers a station with three charging lines and power and balance constraints on their load. The $(1, Cap(t)||\sum T_i)$ problem focuses on scheduling the charging times of the vehicles in one line at a given point in time, subject to maximum load constraints, which result in the definition of $Cap(t)$ for a given problem instance. In [29], $Cap(t)$ was expected to be a unimodal step function, first growing until reaching a peak and then decreasing until getting stabilized at a value greater than 0. Nevertheless, the formal definition of the problem does not impose $Cap(t)$ to be of any given form.

The $(1, Cap(t)||\sum T_i)$ problem was proven NP-hard [32] by reducing the $(1||\sum T_i)$ and the $(P||\sum T_i)$ problems to it. These problems are known to be NP-hard [36]. In the $(1||\sum T_i)$ problem the machine has a constant capacity of one unit, whereas in the $(P||\sum T_i)$ problem there are m identical parallel machines. Any instance of these problems can be reduced to the $(1, Cap(t)||\sum T_i)$ problem by simply defining the capacity of the machine as $Cap(t) = 1$ or $Cap(t) = m$ for all $t \geq 0$, respectively.

3. Preliminaries

This section summarizes the main components of the memetic algorithm proposed in [32], namely, the schedule builder used to define the search space, the genetic algorithm, the local search procedure and their combination.

3.1. Schedule Builder

The definition of a suitable search space is an essential step in the development of effective scheduling algorithms. To this aim, *schedule builders*, or *schedule generation schemes*, have been commonly used (e.g., [37–42]). Schedule builders are non-deterministic constructive methods that allow the computation and enumeration of a subset of the feasible schedules, thus implicitly defining a search space.

Algorithm 1 shows the pseudocode of the schedule builder proposed in [31,32] for the $(1, Cap(t)||\sum T_i)$ problem. It maintains a set US containing the jobs to be scheduled, which is initialized to the set of all jobs $\mathcal{J}$. The algorithm proceeds iteratively: at each iteration a job $u \in US$ is selected (non-deterministically) and it is scheduled at the earliest possible time s_u such that the capacity of the machine is not exceeded at any time. After scheduling the job u, the consumption of the machine $X(t)$ is updated accordingly and u is removed from US. The algorithm terminates when all the jobs have been scheduled, returning a feasible schedule.

Notice that the job to be scheduled at each iteration is selected non-deterministically. This way, the schedule computed depends on the sequence of choices made. For example, considering the problem instance in Example 1, the sequence of choices $\pi_1 = (3, 12, 10, 7, 1, 9, 11, 4, 8, 6, 2, 5)$ would result in the schedule shown in Figure 1. The sequence $\pi_2 = (3, 10, 12, 7, 1, 11, 9, 4, 8, 6, 2, 5)$ would lead the schedule builder to compute the same schedule, so this mapping is many-to-one.

Regardless of these choices, the schedule builder always returns a so-called *left-shifted* schedule, in which no job can be scheduled earlier without delaying the starting time of another job [43]. An example of such a schedule is the one shown in Figure 1. However, the schedule shown in Figure 2 is not left-shifted since, for instance, jobs 2, 5 or 6 could be moved to start earlier without delaying any other job.

Algorithm 1 Schedule Builder ([31,32])

Data: A $(1, Cap(t)||\sum T_i)$ problem instance $\mathcal{P}$.
Result: A feasible schedule S for $\mathcal{P}$.
$US \leftarrow \{1, 2, ..., n\}$;
$X(t) \leftarrow 0, \forall t \geq 0$;
while $US \neq \emptyset$ **do**
  Non-deterministically pick job $u \in US$;
  Assign $s_u = min\{t' | \forall t \text{ with } t' \leq t < t' + p_u : X(t) < Cap(t)\}$;
  Update $X(t) \leftarrow X(t) + 1, \forall t$ with $s_u \leq t < s_u + p_u$;
  $US \leftarrow US - \{u\}$;
end
return *Feasible schedule* $S = (s_1, s_2, ..., s_n)$;

In addition, the non-deterministic selection of jobs in Algorithm 1 enables the definition of a search space containing all left-shifted schedules, by considering all possible sequences of choices (i.e., permutations of the set of jobs). This search space is guaranteed to contain at least one optimal solution to any $(1, Cap(t)||\sum T_i)$ problem instance. For further details, the interested reader is referred to [32].

Among different possibilities, the schedule builder can be used in combination with a priority rule or as the decoder of a genetic algorithm, as described below.

3.2. Genetic Algorithm

Genetic algorithms (GAs) are population-based metaheuristics inspired by the theory of evolution [44]. GAs have been successful at solving combinatorial optimization problems, including scheduling problems (e.g., [11,14,45,46]).

Algorithm 2 depicts the main structure of the genetic algorithm proposed in [31,32] for solving the $(1, Cap(t)||\sum T_i)$ problem. The GA has four parameters: crossover and mutation probabilities (P_c and P_m), number of generations (#*Gen*) and population size (*PopSize*). Initially, the first population is generated at random and evaluated. Then, at each generation, the population is evolved by the application of selection, recombination, evaluation and replacement operators. In the selection phase chromosomes are organized into pairs at random. Each of these pairs undergoes crossover and mutation operators with probabilities P_c and P_m, respectively, what results in two offspring. Then, the new individuals are evaluated, obtaining the actual solutions they represent. Finally, the new population is built in the replacement phase, by a process in which the parents and their offpring compete in a tournament. The GA terminates when #*Gen* generations have been completed, returning the best schedule found. However, other termination criteria could be used instead, as establishing a time limit.

Chromosomes in the GA are permutations of the set of job indices, defining total orderings among the jobs. The GA uses the well-known *Order Crossover* (OX) operator [47], by which an offspring inherits the positions of a (random) subset of the jobs from the first parent and the relative order of the remaining jobs from the second parent. As mutation operator, the GA uses a simple procedure that swaps two random elements in the chromosome. The evaluation of a chromosome is done by means of the schedule builder shown in Algorithm 1, scheduling the jobs in the order they appear in the chromosome. Specifically, given a chromosome $c = (c_1, \ldots, c_n)$, at the i-th iteration the schedule builder selects and schedules the job c_i. This results in a feasible left-shifted schedule. For example, considering the problem instance in Example 1, the chromosome (3, 12, 10, 7, 1, 9, 11, 4, 8, 6, 2, 5) would lead to the schedule shown in Figure 1.

Algorithm 2 Genetic Algorithm ([31,32])

Data: A $(1, Cap(t)||\sum T_i)$ problem instance $\mathcal{P}$ and a set of parameters: crossover probability P_c, mutation probability P_m, number of generations $\#Gen$ and population size $PopSize$.
Result: A feasible schedule for $\mathcal{P}$.
Generate and evaluate the initial population $P(0)$;
for *t=1 to #Gen-1* **do**
  Selection: organize the chromosomes in $P(t-1)$ into pairs at random;
  Recombination: mate each pair of chromosomes and mutate the two offspring in accordance with P_c and P_m;
  Evaluation: evaluate the resulting chromosomes;
  Replacement: make a tournament selection among every two parents and their offspring to complete $P(t)$;
end
return *The best schedule built so far*;

3.3. Local Search Procedure

Local search algorithms have been widely used for solving a variety of hard scheduling problems (e.g., [15–17]). These methods aim at iteratively improving the quality of a given solution by performing changes on it, moving to neighbouring solutions.

The local search procedure our contributions build on is based on swapping pairs of *consecutive* jobs in a feasible left-shifted schedule. Two jobs i and j are consecutive in a schedule S if $s_i = s_j + p_j$ or $s_j = s_i + p_i$, i.e., if one of the jobs starts its processing just after the other one is completed.

As proven in [32], swapping a pair of consecutive jobs (i, j) in a schedule S results in a new feasible schedule S' where all the other jobs keep their starting time (and so their tardiness). As a consequence, if the total tardiness of S is known in advance, the total tardiness of S' can be computed in constant time as $T(S') = T(S) - (T_i + T_j) + (T'_i + T'_j)$, where $T'_i = \max\{0, (s_i + p_i + p_j) - d_i\}$ and $T'_j = \max\{0, (s_i + p_j) - d_j\}$. This allows for establishing an efficient improvement condition of S' over S from swapping the pair of consecutive jobs (i, j): S' improves S if and only if $(T'_i + T'_j) < (T_i + T_j)$.

The results above serve to define a neighbourhood structure, consisting of all the pairs of consecutive jobs in a given schedule. This structure could be exploited by any standard local search approach (e.g., simulated annealing [48], tabu search [49], etc.). However, as pointed out in [32], jobs with earlier starting times in a schedule could be expected to contribute less to the total tardiness than those that start their processing later. This observation led to the definition of an efficient local search procedure, aiming at delaying jobs with low tardiness values in favor of jobs with higher values.

The procedure exploits the notion of C-path, defined as a maximal sequence of pairwise consecutive jobs in a feasible schedule. As an example, in the schedule shown in Figure 1, the sequence of jobs $(3, 10, 1, 6, 5)$ constitutes a C-path. Other examples are $(3, 12, 4, 2)$, $(7, 9, 8)$ and $(7, 11)$. This concept is similar to the notion of *critical block* commonly used in the context of shop scheduling problems [16,50]. Throughout, for a C-path P, $T(P)$ will denote its tardiness, i.e., the sum of the tardiness values of all the jobs contained in it. In addition, $P[k]$ will denote the k-th job in P, with k an integer index in the interval $[1, \ldots, |P|]$.

Algorithm 3 shows the local search approach (the pseudocode of the local search procedure proposed in [32] has been split in Algorithms 3 and 4 to improve the presentation of the new local search algorithms described in Section 4), referred to as *Single C-path local search* (SCP) herein. It is based on the observation that rearranging the jobs in a C-path P does not alter the tardiness of any job outside P. As can be observed, given a feasible left-shifted schedule S, SCP consists of two phases: it first computes a random C-path P in S, which is then *processed* in order to improve its tardiness. As a result a (potentially) improved feasible left-shifted schedule S' is returned, i.e., $T(S') \leq T(S)$.

Computing an optimal order of the jobs in a C-path can be seen as an instance of the $(1||\sum Ti)$ problem. This problem is known to be NP-hard [51], so solving it to optimality may be too time-consuming. As an alternative, SCP uses the efficient procedure *ProcessPath*, shown in Algorithm 4. This procedure operates on a feasible schedule S' and on a C-path P', initialized as copies of the input schedule S and input C-path P. The algorithm traverses P' from left to right. At each iteration, it considers the job $P'[i]$, and swaps it with the next job in the C-path while the aforementioned improvement condition is fulfilled. Upon termination, the algorithm returns S' and P'.

Notice that, although *ProcessPath* is deterministic, it operates on a random C-path, what introduces a source of randomness to the SCP procedure.

The procedure *ProcessPath* performs at most $\mathcal{O}(|P^2|)$ swap operations, what gives an upper bound on its runtime complexity, since both testing the improvement condition and swapping jobs can be done in constant time. As a result, the SCP local search procedure runs in $\mathcal{O}(n^2)$, with n the number of jobs, as in the worst case $|P| = n$.

Algorithm 3 Single C-path local search (SCP)

Data: A feasible schedule S.
Result: A feasible schedule S' with $T(S') \leq T(S)$.
$P \leftarrow ComputeCPath(S)$;
$\langle S', P' \rangle \leftarrow ProcessPath(S, P)$;
return S';

Algorithm 4 *ProcessPath*

Data: A feasible schedule S, a C-path P in S.
Result: A feasible schedule S' with $T(S') \leq T(S)$, the improved C-path P'.
$\langle S', P', i \rangle \leftarrow \langle S, P, 1 \rangle$;
while $i < |P'|$ **do**
  $j \leftarrow i$;
  while $j < |P'| \wedge Improves(P'[j], P'[j+1], S')$ **do**
    $Swap(P'[j], P'[j+1], S')$;
    $j \leftarrow j + 1$;
  end
  if $i = j$ **then** $i \leftarrow i + 1$;
end
return $\langle S', P' \rangle$;

3.4. Memetic Algorithm

Memetic algorithms (MAs) are hybrid metaheuristics that result from combining genetic algorithms and local search methods [52]. These algorithms are often able to keep a proper balance between the exploration and exploitation of the search space. The GA searches for solutions globally, guiding the process towards promising areas, whereas local search intensifies the search locally, what leads to finding better solutions. As a result, MAs have been very effective at solving scheduling problems (e.g., [18,19,53–55]).

The memetic algorithm proposed in [32] for the $(1, Cap(t)||\sum T_i)$ problem has the same structure as the GA shown in Algorithm 2. However, after a feasible schedule is computed by the schedule builder (Algorithm 1) in the evaluation phase, the MA uses the SCP local search procedure (Algorithm 3) in an attempt to improve it.

In order to incorporate the characteristics of the improved schedules into the population, the MA implements a Lamarackian evolution model by which the improved schedule is coded back into the chromosome it came from. This is done by replacing the chromosome by a new one where the jobs in the C-path P processed by the SCP procedure appear in same order as in the resulting C-path P', and all the other jobs keep the same positions. This way, the new chromosome would lead the schedule builder to build the improved schedule P'.

Throughout, this memetic algorithm will be referred to as MA_{SCP}.

4. New Local Search Procedures

This section describes the proposed new local search procedures for the $(1, Cap(t)||\sum T_i)$ problem. The new methods build on the SCP local search algorithm described in Section 3.3. These are aimed at making the local search more effective and, at the same time, keeping their complexity low.

4.1. Enhancements to Single C-Path Local Search

First, two enhancements to the SCP procedure are proposed, both operating on a single C-path. The first one extends the condition for swapping two consecutive jobs, whereas the second one integrates a hill climbing approach for improving a C-path to a greater extent.

4.1.1. Slack-Aware Improvement Condition

The SCP procedure swaps two consecutive jobs (i, j) in a C-path if the improvement condition $(T'_i + T'_j) < (T_i + T_j)$ holds. This way, performing a swap operation always results in a feasible schedule with less total tardiness. However, there may be situations where the condition is not fulfilled, but swapping the jobs may still be beneficial.

Consider the case that $(T'_i + T'_j) = (T_i + T_j)$, i.e., swapping the jobs would not have any (immediate) effect in the total tardiness of the resulting schedule. Since the C-path is processed from left to right, further improvements would be more likely to occur in subsequent iterations if the job that is left in the second place had a greater *slack*, defined as the difference between its due date and its completion time. Notice that a positive slack means that the job would be completed before its deadline, whereas a negative slack indicates that the job would incur in some (positive) tardiness.

This way, if the jobs i and j are not swapped, the job left in the second position would be j, and its slack would be $d_j - C_j$. On the other hand, if the jobs are swapped, the second job would be i, with a slack of $d_i - C'_i$, where $C'_i = s_i + p_i + p_j$. Since $C'_i = C_j$ it suffices to compare the due dates of the jobs to determine whether $d_i - C'_i > d_j - C_j$. This results in the extended improvement condition shown in the following equation:

$$[(T'_i + T'_j) < (T_i + T_j)] \vee [(T'_i + T'_j) = (T_i + T_j) \wedge d_j < d_i] \tag{2}$$

Example 2. *Consider the schedule depicted in Figure 1, and the C-path (3, 10, 1, 6, 5). If the procedure ProcessPath is run with the original improvement condition, no swap operation would be performed. However, the use of the slack-aware condition results in several swaps, leading to reductions in the total tardiness along the process. The resulting C-path is shown in Figure 3, with a total tardiness of 13, given by the jobs 10 ($T_{10} = 2$), 6 ($T_6 = 2$) and 5 ($T_5 = 9$). The resulting schedule has a total tardiness of 32, five units less than the original one.*

Figure 3. Resulting C-path in Example 2.

By using the slack-aware improvement condition, there is the guarantee that the final schedule S' will be such that $T(S') \leq T(S)$, since no swap operation that increases the total tardiness is ever performed. In addition, the new condition does not affect the worst-case complexity of the method, as it can be checked in constant time and the maximum possible number of swap operations remains the same.

Throughout, the predicate $Improves(i, j)$ will indicate whether the improvement condition holds for a pair of consecutive jobs (i, j). Besides, the SCP procedure using the new improvement condition will be referred to as iSCP.

4.1.2. Hill Climbing on a Single C-Path

An analysis of the procedure *ProcessPath* (Algorithm 4) reveals that the resulting C-path may be further improved by swapping some pair of consecutive jobs.

Suppose that in a given iteration there are three consecutive jobs (i, j, k) in the C-path P, i.e., $P = (P[1], P[2], \ldots, i, j, k, \ldots, P[|P|])$, and that the improvement condition $Improves(i, j)$ does not hold. This way, the jobs i and j are not swapped and, by construction of the algorithm, i will not be swapped with any other job that appears after it. If in the next iteration the algorithm swaps j and k, the final C-path will be of the form $P' = (P'[1], \ldots, i, \ldots, k, \ldots, j, \ldots, P'[|P|])$, since j and k could be swapped with other jobs in subsequent iterations. In this situation, swapping i with the next job in the C-path might result in an improvement.

Example 3. *Consider the schedule depicted in Figure 1, and the C-path (7, 9, 8). In the first iteration, the procedure ProcessPath (using the slack-aware improvement condition) does not swap the jobs 7 and 9. In the second iteration, it swaps the jobs 9 and 8, what results in the C-path (7, 8, 9). Now, if this procedure is issued again, the jobs 7 and 8 would be swapped and the final C-path would be (8, 7, 9), shown in Figure 4. The new C-path has a total tardiness of 4, given by job 8 ($T_8 = 4$), four units less than the original one.*

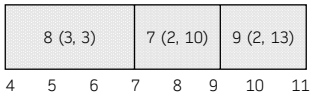


Figure 4. Resulting C-path in Example 3.

The observation above serves to develop an improved version of *ProcessPath*, by performing a *hill-climbing* approach. The new version of the method, termed *ProcessPath+*, is shown in Algorithm 5. As can be observed, the method repeatedly invokes *ProcessPath* until there are no reductions in the total tardiness objective function. Upon termination, the method reaches a fixpoint where swapping any pair of consecutive jobs in the C-path P' does not result in an improvement. As can be observed, the structure of *ProcessPath+* is similar to the Bubble Sort algorithm: each invocation to the procedure *ProcessPath* could be related to one pass of the sorting algorithm, and it terminates when no swaps are made in a given iteration.

Noticeably, given a schedule S and a C-path P in S as input, the procedure *ProcessPath+* performs at most $|P|$ iterations.

To show this, it is first proven that after the i-th iteration, the last i jobs in the C-path become fixed, i.e., they will not be rearranged in any future iterations of the method. This follows from the next result:

Lemma 1. *Let P be the initial C-path, $L = |P|$ and $P^{(i)}$, with $i \geq 1$, the C-path computed after the i-th iteration of ProcessPath+. Then, $P^{(i)}[k] = P^{(i+1)}[k]$ for all $k \in [L - i + 1, L]$.*

Proof. The lemma is proven by induction on the number of iterations.

(Base case: $i = 1$) $P^{(1)} \neq P^{(2)}$ only if at some step in the computation of $P^{(1)}$ there are two consecutive jobs (j_1, j_2) such that the condition $Improves(j_1, j_2)$ does not hold and $P^{(1)} = (P^{(1)}[1], \ldots, j_1, k, \ldots, j_2, \ldots, P^{(1)}[L])$ with $k \neq j_2$. Suppose, w.l.o.g., that there is only one such pair of consecutive jobs. Since $Improves(j_1, j_2)$ does not hold, j_1 will not be swapped with j_2 in the computation of $P^{(2)}$. So, $P^{(1)}[L] = P^{(2)}[L]$.

(Inductive step: $i > 1$) Assume as inductive hypothesis that $P^{(i-1)}[k] = P^{(i)}[k]$ for all $k \in [L - (i - 1) + 1, L]$. $P^{(i)} \neq P^{(i+1)}$ only if at some step in the computation of $P^{(i)}$ there are two consecutive jobs (j_1, j_2) such that the condition $Improves(j_1, j_2)$ does not hold and $P^{(i)} = (P^{(i)}[1], \ldots, j_1, k, \ldots, j_2, \ldots, P^{(i)}[L])$, with $k \neq j_2$. Suppose, w.l.o.g., that there is only one such pair of consecutive jobs. As the inductive hypothesis holds, it must be that $Improves(P^{(i-1)}[k], P^{(i-1)}[k + 1])$ does not hold for any $k \in [L - (i - 1) + 1, L - 1]$. So, $P^{(i)} = (P^{(i)}[1], \ldots, j_1, k, \ldots, j_2, \ldots, P^{(i)}[L - i + 1], P^{(i-1)}[L - (i - 1) + 1], \ldots, P^{(i-1)}[L])$.

Since $Improves(j_1, j_2)$ does not hold, j_1 will not be swapped with j_2 in the computation of $P^{(i+1)}$. Hence, $P^{(i)}[L-i+1] = P^{(i+1)}[L-i+1]$ and hence $P^{(i)}[k] = P^{(i+1)}[k]$ for all $k \in [L-i+1, L]$. □

Now, the bound on the maximum number of iterations performed by *ProcessPath+* can be easily shown.

Proposition 1. *ProcessPath+ performs at most $|P|$ iterations.*

Proof. By Lemma 1, after the $(|P|-1)$-th iteration, it holds that $P^{(|P|-1)}[k] = P^{(|P|)}[k]$ for all $k \in [2, |P|]$. Since there is only one job left (the first one), it must also hold that $P^{(|P|-1)}[1] = P^{(|P|)}[1]$. So, in the $|P|$-th iteration no swaps are made, what results in the termination of *ProcessPath+*. □

Interestingly, the procedure *ProcessPath+* exhibits the same worst-case complexity than *ProcessPath*, as shown next:

Proposition 2. *ProcessPath+ runs in $\mathcal{O}(|P|^2)$.*

Proof. Note that if two consecutive jobs i and j are swapped, they will not be swapped in subsequent iterations, since the improvement condition would not hold in the opposite direction. In P there are $\binom{|P|}{2} = \frac{|P|^2-|P|}{2}$ possible unordered pairs of jobs, what gives an upper bound of the total number of swap operations that the algorithm can possibly perform along its execution. Now, let sw_i denote the number of swap operations performed at the i-th iteration. Since the whole C-path is traversed, this results in $(|P| + sw_i - 1)$ checks of the improvement condition. By Proposition 1, there can be at most $|P|$ iterations. Hence, the total number of checks will be $(|P| + sw_1 - 1) + \cdots + (|P| + sw_{|P|} - 1) = |P|^2 + \sum_{i=1}^{|P|} sw_i - |P| \leq |P|^2 + \binom{|P|}{2} - |P| \in \mathcal{O}(|P|^2)$. Since both performing a swap operation and testing the improvement condition are done in constant time, the complexity of *ProcessPath+* is $\mathcal{O}(|P|^2)$. □

The use of *ProcessPath+* gives rise to a new local search method, termed SCP+, that is shown in Algorithm 6. As SCP, the new method computes a random C-path P and tries to improve it. However, in the second phase it invokes *ProcessPath+* instead of the simpler *ProcessPath*. In the worst case $|P| = n$, with $n = |\mathcal{J}|$, so SCP+ runs in $\mathcal{O}(n^2)$.

Algorithm 5 *ProcessPath+*

Data: A feasible schedule S, a C-path P in S.
Result: A feasible schedule S' with $T(S') \leq T(S)$, the improved C-path P'.
$\langle S', P' \rangle \leftarrow ProcessPath(S, P)$;
while $T(P') < T(P)$ **do**
  $P \leftarrow P'$;
  $\langle S', P' \rangle \leftarrow ProcessPath(S', P')$;
end
return $\langle S', P' \rangle$;

Algorithm 6 SCP with Hill Climbing (SCP+)

Data: A feasible schedule S.
Result: A feasible schedule S' with $T(S') \leq T(S)$.
$P \leftarrow ComputeCPath(S)$;
$\langle S', P' \rangle \leftarrow ProcessPath+(P, S)$;
return S';

4.2. Cover-Based Local Search

The previous methods aim at improving a single C-path P in a schedule, leaving all the jobs outside P unaltered. Clearly, processing these jobs could bring additional improvements.

This section proposes a new procedure based on computing and processing a *cover* of C-paths, that is, a set of C-paths such that the union of their elements hits all the jobs in $\mathcal{J}$. In order to be able to efficiently process the C-paths independently from each other, the cover is restricted to be a partition of the set $\mathcal{J}$. This way, given a schedule S a cover $\mathcal{C} = \{P_1, \ldots, P_k\}$ is a set of maximal sequences of pair-wise consecutive jobs in S, such that each job in $\mathcal{J}$ belongs to exactly one P_i. Notice that such sequences may not be maximal w.r.t. all the jobs in S. Anyway, each P_i in the cover will be maximal w.r.t. the jobs that do not belong to any other P_j, with $i \neq j$. So, slightly abusing notation, it is referred to as a C-path.

Once a cover is computed, the new cover-based local search procedure, termed CB, processes each of the C-paths, aiming at reducing their total tardiness.

Example 4. *Consider the schedule shown in Figure 1. The first step taken by CB is computing a cover of C-paths, for instance* $\mathcal{C} = \{P_1, \ldots, P_4\}$, *with* $P_1 = (3, 10, 1, 6, 5)$, $P_2 = (12, 4, 2)$, $P_3 = (9, 7, 8)$ *and* $P_4 = (11)$. *Then, the procedure ProcessPath+ is invoked on each C-path in the cover, what results in the improved C-paths* $P'_1 = (1, 10, 6, 3, 5)$, $P'_2 = (4, 2, 12)$, $P'_3 = (8, 7, 9)$ *and* $P'_4 = (11)$. *Note that since there is only one job in* P_4 *no improvements are possible. After all the C-paths have bee processed, the resulting schedule, shown in Figure 5 has a total tardiness of 25, twelve units less than the original one.*

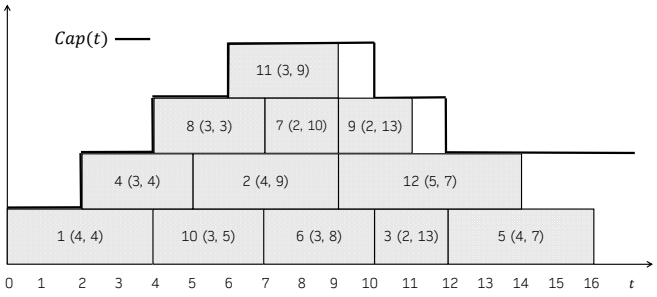


Figure 5. Resulting schedule in Example 4.

The computation of a cover of C-paths is depicted in Algorithm 7. The algorithm operates on a list R of jobs and maintains a set $\mathcal{C}$ of sequences of pair-wise consecutive jobs. $\mathcal{C}$ is initialized as the empty set and will eventually contain the cover of C-paths returned by the method. The list R contains the jobs sorted non-decreasingly by their starting times in the schedule S given as input. Then, R is traversed from left to right. At the i-th iteration, the method looks for a sequence P_c to append the job $R[i]$, i.e., such that its last job is completed exactly at the starting time of $R[i]$. To this aim, it uses the function *LookForPath*, that returns the index $c > 0$ of P_c in $\mathcal{C}$ if such sequence exists and the value 0 otherwise. If a sequence P_c is found, the job $R[i]$ is appended at its end. Otherwise a new sequence $(R[i])$ is created and added to $\mathcal{C}$.

It is easy to see that computing a cover of C-paths is done in $\mathcal{O}(n^2)$. First, the complexity of sorting the jobs is $\mathcal{O}(n \times log(n))$, e.g., by using the *Heapsort* algorithm [56]. Second, the loop iterates over all the n jobs, and for each job it may have to traverse the whole set $\mathcal{C}$, what gives the complexity $\mathcal{O}(n^2)$.

Algorithm 7 *ComputeCoverCPaths*

Data: A feasible schedule S.
Result: A cover of C-paths $\mathcal{C} = \{P_1, \ldots, P_k\}$ of S.
$R \leftarrow Sort(\mathcal{J}, S)$;
$\mathcal{C} \leftarrow \emptyset$;
for $i \in (1, \ldots, |R|)$ **do**
  $c \leftarrow LookForPath(\mathcal{C}, R[i])$;
  if $c > 0$ **then**
    $P_c \leftarrow AppendJob(P_c, R[i])$;
  else
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{(R[i])\}$;
  end
end
return $\mathcal{C}$;

Algorithm 8 shows the pseudocode of the cover-based local search procedure. As can be observed, it first computes a cover of C-paths $\{P_1, P_2, \ldots, P_k\}$ by using Algorithm 7. Then, the method attempts to improve each C-path P_i in the cover by means of the procedure *ProcessPath+*. Notice that processing P_i does not interfere with any other C-paths, since in every job belongs to only one C-path. Finally, the improved schedule S' is returned.

Algorithm 8 Cover-based Local Search Procedure (CB)

Data: A feasible schedule S.
Result: A feasible schedule S' with $T(S') \leq T(S)$.
$\{P_1, P_2, \ldots, P_k\} \leftarrow ComputeCoverCPaths(S)$;
$S' \leftarrow S$;
for $P_i \in \{P_1, \ldots, P_k\}$ **do**
  $\langle S', P_i' \rangle \leftarrow ProcessPath+(S', P_i)$;
end
return S';

As invoking *ProcessPath+* never increases the total tardiness, it follows that $T(S') \leq T(S)$. Besides, as the previous methods, the new local search procedure has a worst-case quadratic complexity, as shown next:

Proposition 3. *CB runs in* $\mathcal{O}(n^2)$.

Proof. Computing the cover of C-paths in the first phase is done in $\mathcal{O}(n^2)$. Then, in the second phase the method performs k iterations. At the i-th iteration, by Proposition 2, invoking *ProcessPath+* on P_i has a complexity of $\mathcal{O}(|P_i|^2)$. Notice that $|P_1|^2 + \cdots + |P_k|^2 \leq (|P_1| + \cdots + |P_k|)^2$. Since $\sum_{i=1}^{k} |P_i| = n$, the runtime complexity of the loop is bounded by $\mathcal{O}(n^2)$. So, CB runs in $\mathcal{O}(n^2)$. □

4.3. Interchanging Jobs between C-Paths

The CB procedure processes the C-paths in a cover independently from each other, aiming at reducing their total tardiness. As an alternative, the global quality of the schedule could be improved by swapping jobs that belong to different C-paths.

Let S be a feasible schedule, P_1 and P_2 two C-paths in a cover, and consider the jobs $i \in P_1$ and $j \in P_2$ such that $p_i \leq p_j$. The jobs i and j can be interchanged without interfering with any other C-paths if the following capacity condition holds in S: just after the completion of the last job in P_1 there are at least $p_j - p_i$ time instants t where $X(t) < Cap(t)$. In this situation, interchanging the jobs results in a new schedule S' where $T_i' = max\{0, (s_j + p_i) - d_i\}$ and $T_j' = max\{0, (s_i + p_j) - d_j\}$. In addition, all the jobs in P_1 after i are delayed $p_j - p_i$ time units (thus potentially increasing their tardiness), and the jobs in P_2 after j are scheduled $p_j - p_i$ time units earlier (potentially reducing their tardiness), guaranteeing that S' is a left-shifted schedule.

If the new C-paths P'_1 and P'_2 are such that $T(P'_1) + T(P'_2) < T(P_1) + T(P_2)$, S' would have a higher quality than S in terms of total tardiness.

Example 5. *Consider the schedule from Example 4 depicted in Figure 5, and the cover of C-paths* $\mathcal{C} = \{P_1, \ldots, P_4\}$, *with* $P_1 = (1, 10, 6, 3, 5)$, $P_2 = (4, 2, 12)$, $P_3 = (8, 7, 9)$ *and* $P_4 = (11)$. *The jobs 1 (in* P_1*) and 8 (in* P_3*) can be interchanged, as the capacity condition is fulfilled, and this operation results in a feasible schedule with less total tardiness. The same applies to the jobs 5 (in* P_1*) and 11 (in* P_4*). As a consequence, the resulting schedule, depicted in Figure 6, has a total tardiness of 22, three units less than the original one.*

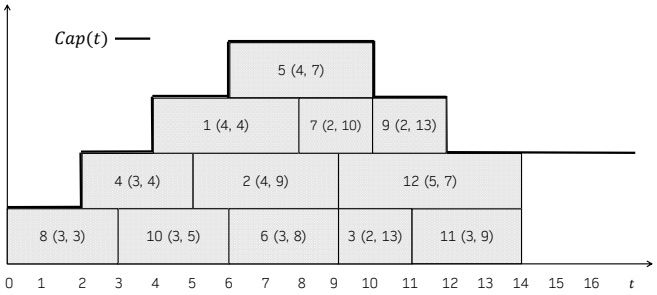


Figure 6. Resulting schedule in Example 5.

Building on the previous idea, Algorithm 9 shows a new local search procedure, termed ICP, that attempts to reduce the total tardiness of a schedule S by means of interchanging jobs between C-paths. First, a cover of C-paths $\mathcal{C} = \{P_1, \ldots, P_k\}$ is computed using Algorithm 7. At this point, the algorithm proceeds iteratively. It operates over a feasible schedule S', initialized as a copy of S. At each iteration, the C-path $P_m \in \mathcal{C}$ with the greatest tardiness is selected. Then, the algorithm traverses all the C-paths P_p different from P_m, and for each job $P_m[i]$ in P_m and each job $P_p[j]$ in P_p, it tests whether interchanging them is possible (i.e., if the aforementioned capacity condition holds). If so, the jobs are interchanged resulting in the new left-shifted schedule S''. This operation is performed by the procedure *Interchange*, which delays and moves earlier the jobs in the C-paths after $P_m[i]$ and $P_p[j]$ if necessary. If $T(S'') < T(S')$, S' is replaced by S''. Finally, P_m is removed from $\mathcal{C}$ and a new iteration is performed. The method terminates when $|\mathcal{C}| = 1$, returning the (possibly) improved schedule S'.

Algorithm 9 Local search by interchanging jobs between C-paths (ICP)

Data: A feasible schedule S.
Result: A feasible schedule S' with $T(S') \leq T(S)$.
$\mathcal{C} \leftarrow ComputeCoverCPaths(S)$;
$S' \leftarrow S$;
while $|\mathcal{C}| > 1$ **do**
  $P_m \leftarrow argmax\{P_i \in \mathcal{C} | T(P_i)\}$;
  for $P_p \in \mathcal{C} \setminus \{P_m\}$ **do**
    for $i \in (1, \ldots, |P_m|)$ **do**
      for $j \in (1, \ldots, |P_p|)$ **do**
        if $CapacityCondition(P_m, P_p, P_m[i], P_p[j], S')$ **then**
          $S'' \leftarrow Interchange(P_m, P_p, P_m[i], P_p[j], S')$;
          if $T(S'') < T(S')$ **then** $S' \leftarrow S''$;
        end
      end
    end
  end
  $\mathcal{C} \leftarrow \mathcal{C} \setminus \{P_m\}$;
end
return S';

By construction, in the worst case, the ICP procedure will interchange $\mathcal{O}(n^2)$ pairs of jobs. On the other hand, the necessary operations for interchanging two jobs result in a linear overhead. Hence, the complexity of ICP is bounded by $\mathcal{O}(n^3)$.

4.4. Hybrid Approach

In order to benefit from all the methods described above, this section proposes a hybrid approach, termed HYB, that combines the CB and the ICP procedures.

Given a feasible schedule S as input, the new method works as follows: first a cover of C-paths $\mathcal{C} = \{P_1, \ldots, P_k\}$ is computed and, as CB does, each $P_i \in \mathcal{C}$ is processed by means of the *ProcessPath+* procedure, resulting in the improved C-path P'_i. This produces a (potentially) better schedule S', and the cover $\mathcal{C}' = \{P'_1, \ldots, P'_k\}$. Then, the schedule undergoes the loop of ICP, by which jobs are interchanged between different C-paths in the cover $\mathcal{C}'$.

By the properties of CB and ICP, the final schedule will never have a greater total tardiness than that of the original one. Furthermore, the complexity of HYB is bounded by $\mathcal{O}(n^3)$, given by the ICP component of the method.

5. New Memetic Algorithms

The efficiency of the local search methods proposed in the previous section makes them well-suited for working in combination with the genetic algorithm described in Section 3.2, each of them giving rise to a different memetic algorithm.

The new algorithms have the same structure as the MA proposed in [32] (described in Section 3.4), only differing in the local search procedure used. After a schedule is computed by the schedule builder (Algorithm 1) in the evaluation phase of the GA, a local search procedure is issued with the aim of improving it.

As a consequence, five new memetic algorithms for the $(1, Cap(t)||\sum T_i)$ problem are proposed: MA_{iSCP}, $\text{MA}_{\text{SCP+}}$, MA_{CB}, MA_{ICP} and MA_{HYB}. MA_{iSCP} uses the SCP procedure (Algorithm 3) with the slack-aware improvement condition proposed in Section 4.1.1. $\text{MA}_{\text{SCP+}}$ combines the GA with the hill-climbing approach SCP+ (Algorithm 6) described in Section 4.1.2. MA_{CB} integrates the cover-based local search method CB (Algorithm 8) introduced in Section 4.2. MA_{ICP} uses the local search procedure ICP (Algorithm 9), based on interchanging jobs between C-paths as described in Section 4.3. Finally, MA_{HYB} exploits the hybrid local search procedure presented in Section 4.4.

As the MA proposed in [32], the new memetic algorithms instrument a Lamarckian evolution model, by which the characteristics of the improved schedules are transmitted to the population. This is done by a simple procedure that swaps two job indices in the chromosomes whenever the local search methods swap or interchange a pair of jobs. This way, the improved schedules are coded back into the chromosomes, what facilitates that the MAs converge to high-quality solutions. Notice that running the schedule builder on the resulting chromosomes would result in the improved schedules computed by the local search methods.

The complexity of the memetic algorithms depends on the population size ($PopSize$), the number of generations ($\#Gen$) and the complexity of the local search method used. Since the complexity of the genetic operators is not greater than that of the local search procedures, the complexity of MA_{iSCP}, $\text{MA}_{\text{SCP+}}$ and MA_{CB} is given by $\mathcal{O}(PopSize \times \#Gen \times n^2)$, whereas MA_{ICP} and MA_{HYB} have a complexity of $\mathcal{O}(PopSize \times \#Gen \times n^3)$.

6. Results

This section reports the results from an experimental study carried out to assess the performance of the algorithms proposed in this paper.

The experiments were carried out over a set of instances built for this purpose, using the random generation procedure proposed in [35]. Given a number of jobs (n) and the maximum capacity of the machine (MC), the generator produces instances where $Cap(t)$ is

an unimodal step function, making them similar in structure to those expected to arise in the context of electric vehicles charging [29].

The generation procedure works as follows ($U(a,b)$ denotes a random integer from a uniform distribution in $[a,b]$, and $N(\mu,\sigma)$ denotes a random integer from a normal distribution with mean μ and standard deviation σ): First, each job i is assigned a processing time $p_i = U(20,100)$, and $min\_p_i = min\{p_i | i = 1,\ldots,n\}$ and $sum\_p_i = \sum_{i=1}^{n} p_i$ are defined. The initial capacity of the machine is $IC = U(1,MC)$, and its final capacity is $FC = 2$. Then, $Cap(t)$ is defined by means of different capacity intervals, first increasing the capacity in one unit from IC to MC, and then reducing it one by one until reaching FC. Each interval has a duration of $max\{min\_p_i/4, N(R, 0.2 \times R)\}$, with $R = sum\_p_i/S$, and $S = \sum_{j=IC}^{MC-1} j + \sum_{j=FC}^{MC} j$. This seeks that the jobs are distributed over all the capacity intervals. Finally, each job i is assigned a due date $d_i = U(p_i, B)$, where $B = R \times (2 \times MC - IC - 1)$ is an approximation of completion time of all the jobs.

Using the approach above, 10 instances with each of the following configurations of n and MC were generated: $n = 120$ and $MC \in \{3,5,7,10\}$; $n = 250$ and $MC \in \{10,20,30\}$; $n = 500$ and $MC \in \{10,20,30\}$; $n = 750$ and $MC \in \{10,20,30,50\}$ and $n = 1000$ and $MC \in \{10,20,30,50,100\}$. In all, the benchmark set consists of 190 instances. Note that earlier work [32] considered instances with up to 120 jobs in the experimental study. Herein, most of the instances are (much) larger, and so (much) more challenging what serves to evaluate how the different methods scale in practice.

A prototype was coded in C++ and all the experiments were run on a Linux cluster (Intel Xeon 2.26 GHz. 128 GB RAM).

To assess the performance of the proposed methods six memetic algorithms are compared, termed MA_{SCP}, MA_{iSCP}, MA_{SCP+}, MA_{CB}, MA_{ICP} and MA_{HYB}. Recall that MA_{SCP} is the memetic algorithm proposed in [32], whereas the other MAs are the new ones proposed in this work, described in Section 5.

For each instance, 30 independent runs of each method were performed, recording the best and average total tardiness of the solutions found. Following [32], the considered MAs were run with a population size of 250 individuals, and crossover and mutation probabilities of 0.9 and 0.1, respectively. In addition, to make the comparison fair, the termination condition is a given time limit, which was set depending on the size of the instances. Specifically, for an instance with n jobs, the time limit is set to $n/2$ s, which in most cases is sufficient for the algorithms to converge and, arguably, it constitutes a reasonable time in practice given the complexity of the problem.

In order to evaluate the quality of the solutions, the error in percentage w.r.t. the best solution found across all the experiments is calculated for the solutions reached by each method on each instance. Specifically, if for a given instance the best known solution has a total tardiness T_{best} and an algorithm finds a solution with a total tardiness T (with $T_{best} \leq T$), the error in percentage is computed as $100 \times (T - T_{best})/T_{best}$. This way, the error in percentage of a solution represents its deviation from the best solution known for a given problem instance.

The experimental study is organized as follows: First, the slack-aware improvement condition is analyzed by comparing MA_{SCP} and MA_{iSCP}. Then, the performance of the memetic algorithms MA_{SCP+}, MA_{CB}, MA_{ICP} and MA_{HYB} is assessed. Finally, the study provides a detailed comparison of the best memetic algorithm with both the state-of-the-art approach [32] and a constraint programming model.

6.1. Analyzing the Slack-Aware Improvement Condition

The objective of the first series of experiments is to measure the effectiveness of the slack-aware improvement condition described in Section 4.1.1, and assess the gains it brings in terms of the quality of the schedules computed. To this end, MA_{SCP} and MA_{iSCP} were run on all the instances. Recall that MA_{SCP} exploits the SCP local search procedure using the original improvement condition for swapping a pair of consecutive jobs in a C-path, whereas MA_{iSCP} integrates the iSCP procedure, which uses the new slack-aware condition.

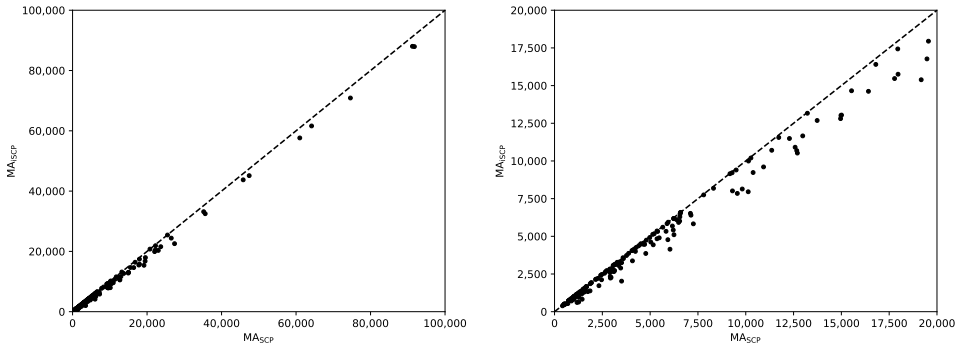
Table 1 shows a summary of the results. For each method, it reports the error in percentage of the best and average solutions, averaged for groups of instances with the same size n and maximum capacity of the machine MC. As can be observed, MA_{iSCP} yields (much) better results than MA_{SCP} on all the groups of instances, in both the best and average solutions found. The improvements in the quality of the solutions are significant. On average, the error of the best and average solutions computed by MA_{iSCP} is about 48.67% and 40.34% of that of the solutions found by MA_{SCP}. The greatest improvements are observed for the instances with $n = 500$, where the ratios are 35.76% and 26.62%. On the other hand, the results show that the errors tend to increase with MC. This is always the case for MA_{iSCP}, whereas MA_{SCP} follows this trend for most values, with the exception of $MC = 10$ for $n \geq 250$.

Figure 7 shows two scatter plots with the average total tardiness of the solutions computed by the methods. Notice that the plots show total tardiness values, instead of errors in percentage as reported in Table 1. Figure 7a shows these values for all the instances (notice that the axes are limited to 100,000) and, to get a closer view, Figure 7b shows the values below 20,000, which contains most of the instances (with the exception of a few outliers). In these plots, the points below (resp. above) the diagonal line represent instances for which MA_{iSCP} returned better (resp. worse) solutions on average than MA_{SCP}. As can be observed, there is a clear superiority of MA_{iSCP} over MA_{SCP}.

The results confirm that the slack-aware improvement condition is beneficial. As a consequence, this condition will be used in all the memetic algorithms evaluated in the next subsection.

Table 1. Summary of results from MA_{SCP} and MA_{iSCP} after evolving a population of 250 individuals for $n/2$ s. Errors in percentage of the best and average solutions over 30 runs are reported.

		MA_{SCP}		**MA_{iSCP}**	
n	***MC***	**Best**	**Avg.**	**Best**	**Avg.**
120	3	**0.01**	0.49	**0.01**	**0.18**
	5	0.08	0.98	**0.07**	**0.29**
	7	0.35	1.50	**0.08**	**0.48**
	10	0.65	1.97	**0.30**	**1.13**
	Avg.	0.27	1.23	**0.11**	**0.52**
250	10	1.46	4.09	**0.27**	**0.99**
	20	1.64	3.81	**0.59**	**1.95**
	30	1.98	4.71	**1.51**	**3.39**
	Avg.	1.70	4.20	**0.79**	**2.11**
500	10	8.54	20.00	**0.53**	**1.40**
	20	4.35	11.55	**2.01**	**3.46**
	30	6.92	13.41	**4.55**	**7.11**
	Avg.	6.60	14.99	**2.36**	**3.99**
750	10	7.88	16.21	**0.36**	**1.79**
	20	6.08	14.81	**0.97**	**2.17**
	30	12.97	21.98	**6.59**	**9.70**
	50	18.58	25.21	**11.62**	**14.65**
	Avg.	11.38	19.55	**4.88**	**7.08**
1000	10	15.72	41.02	**0.80**	**2.88**
	20	5.86	11.63	**1.61**	**3.03**
	30	21.08	31.39	**9.42**	**13.50**
	50	27.97	41.59	**12.55**	**17.17**
	100	43.66	50.27	**36.71**	**42.45**
	Avg.	22.86	35.18	**12.22**	**15.81**
All		9.78	16.66	**4.76**	**6.72**



(**a**) (**b**)

Figure 7. Comparison of MA_{SCP} and MA_{iSCP}. (**a**) Total tardiness limit: 100,000. (**b**) Total tardiness limit: 20,000.

6.2. Analyzing MA_{SCP+}, MA_{CB}, MA_{ICP} and MA_{HYB}

The second part of the experimental study is devoted to the evaluation of the memetic algorithms MA_{SCP+}, MA_{CB}, MA_{ICP} and MA_{HYB}, which use the local search procedures SCP+, CB, ICP and HYB respectively.

As in the previous experiments, 30 independent runs of the methods were performed on each instance in the benchmark set. The results are summarized in Table 2. As can be observed, on average all the algorithms improve the results obtained by MA_{SCP} and MA_{iSCP} (shown in Table 1), some of them very substantially. For most groups of instances the hill-climbing approach used by SCP+ leads to solutions of similar quality than those computed by MA_{iSCP}, with the notable exception of the largest (and hardest) instances with $n = 1000$, where the errors are much smaller. MA_{CB} reaches significantly better solutions than MA_{SCP+} in all cases, indicating a remarkable effectiveness of the cover-based local search method. On the other hand, MA_{ICP} lays behind all the other memetic algorithms in the table, what suggests that the ICP procedure is not as effective as the other local search methods. However, the combination of CB and ICP is beneficial in practice, as it allows MA_{HYB} to achieve the best results by a wide margin. In most cases, both the best and average solutions computed by MA_{HYB} have an error between 0% and 1%, and the average error of the solutions computed by MA_{HYB} is smaller than that of the best solutions computed by any other method.

Figure 8 shows the evolution over time of the average error of the solutions computed the memetic algorithms on the sets of instances with 250, 500, 750 and 1000 jobs. Although the errors in the plots are different, there are similarities shared by all of them. First of all, it is clear that MA_{HYB} and MA_{CB} are in the first and second positions from the beginning for all the considered sizes, and these methods keep their lead for the whole duration of the experiments. In addition, MA_{ICP} exhibits a faster convergence pattern than MA_{SCP+}, allowing it to compute better solutions during a large fraction of the given time. However MA_{SCP+} is eventually able to compute solutions of similar quality and even outperform MA_{ICP} on the largest instances. Noticeably, the differences in favor of MA_{HYB} and MA_{CB} over the other algorithms grow with n. In the case of $n = 250$, all the MAs are close, even though the mentioned ranking among them is already observable; with $n = 500$ and $n = 750$, the superiority of MA_{HYB} and MA_{CB} becomes clearer and finally, with $n = 1000$, it is evident. In addition, these two algorithms (especially MA_{HYB}) are able to compute high-quality solutions in short time (using just a small portion of the given time limit), what represents an important advantage over the other methods.

Table 2. Summary of results from MA_{SCP+}, MA_{CB}, MA_{ICP} and MA_{HYB} after evolving a population of 250 individuals for $n/2$ s. Errors in percentage of the best and average solutions over 30 runs are reported.

		MA_{SCP+}		MA_{CB}		MA_{ICP}		MA_{HYB}	
n	***MC***	**Best**	**Avg.**	**Best**	**Avg.**	**Best**	**Avg.**	**Best**	**Avg.**
120	3	**0.00**	0.16	**0.00**	0.11	0.12	2.20	**0.00**	**0.01**
	5	0.07	0.27	0.05	0.20	**0.00**	1.10	**0.00**	**0.03**
	7	0.09	0.42	0.07	0.32	0.06	0.75	**0.00**	**0.07**
	10	0.33	1.19	0.15	0.68	0.16	1.06	**0.00**	**0.17**
	Avg.	0.12	0.51	0.07	0.33	0.08	1.28	**0.00**	**0.07**
250	10	0.39	1.18	0.15	0.56	1.05	2.12	**0.00**	**0.12**
	20	0.81	2.10	0.25	0.80	0.51	1.17	**0.00**	**0.14**
	30	1.61	3.53	0.34	1.45	1.10	1.85	**0.07**	**0.37**
	Avg.	0.94	2.27	0.25	0.94	0.89	1.71	**0.02**	**0.21**
500	10	0.49	1.26	0.21	0.58	3.28	5.31	**0.00**	**0.12**
	20	2.08	3.63	0.65	1.21	2.71	3.92	**0.12**	**0.37**
	30	4.83	7.48	0.98	2.38	3.28	5.08	**0.16**	**0.38**
	Avg.	2.47	4.12	0.61	1.39	3.09	4.77	**0.09**	**0.29**
750	10	0.39	1.48	**0.06**	0.67	7.82	10.16	0.14	**0.62**
	20	1.13	2.27	**0.16**	0.70	2.71	3.84	0.17	**0.28**
	30	6.64	10.28	1.53	2.87	4.09	5.93	**0.00**	**0.27**
	50	11.23	14.00	1.79	3.23	7.13	8.78	**0.00**	**0.29**
	Avg.	4.85	7.01	0.89	1.87	5.44	7.18	**0.08**	**0.36**
1000	10	0.42	2.15	**0.00**	**0.78**	21.25	27.28	0.64	1.07
	20	1.48	2.40	**0.12**	0.90	6.21	7.56	0.23	**0.39**
	30	7.45	10.05	**0.20**	1.40	19.00	22.91	0.34	**0.83**
	50	11.45	14.98	2.72	4.21	4.07	5.89	**0.08**	**0.14**
	100	21.18	25.20	2.86	4.03	5.53	6.44	**0.00**	**0.20**
	Avg.	8.40	10.95	1.18	2.26	11.21	14.02	**0.26**	**0.53**
All		3.79	5.47	0.65	1.42	4.74	6.49	**0.10**	**0.31**

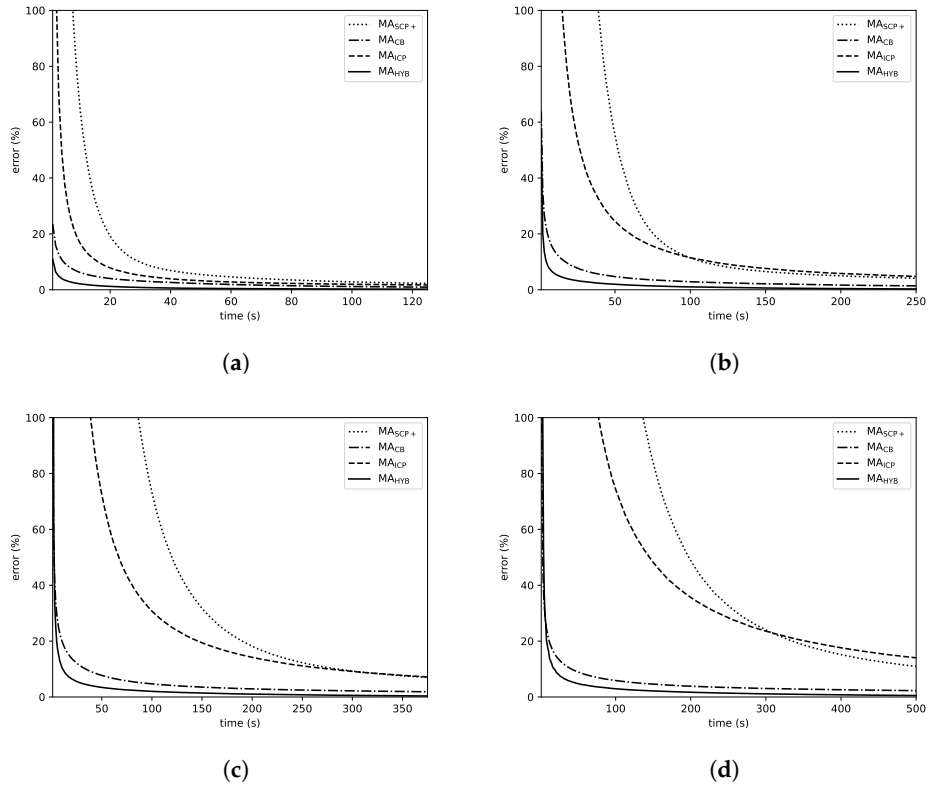


Figure 8. Evolution of the average error over time. (**a**) $n = 250$; (**b**) $n = 500$; (**c**) $n = 750$; (**d**) $n = 1000$.

Figure 9 shows a boxplot with the average errors of MA_{SCP+}, MA_{CB}, MA_{ICP} and MA_{HYB} over the whole set of instances. It confirms the previously made points: MA_{HYB} is ahead in terms of results, followed by MA_{CB}, MA_{SCP+} and MA_{ICP}.

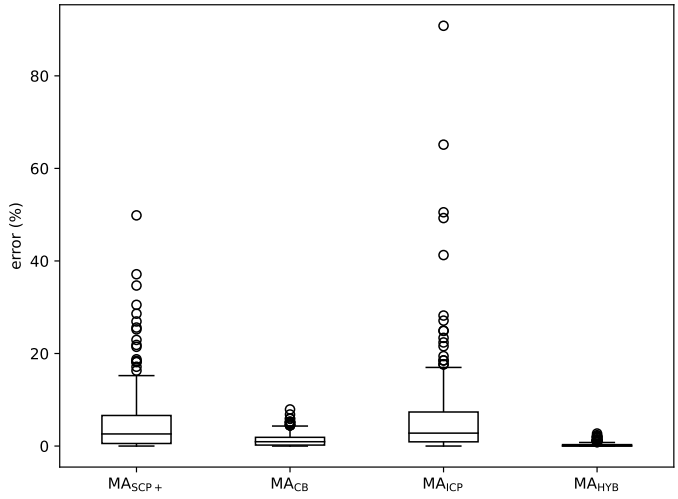


Figure 9. Boxplot comparing the average results yielded by MA_{SCP+}, MA_{CB}, MA_{ICP} and MA_{HYB}.

Some conclusions can be drawn from the experiments. First of all, the cover-based methods get the best results, since MA_{HYB} and MA_{CB} are ahead of the other approaches in terms of the quality of solutions they reach. In addition, as already mentioned, ICP does not seem to be a good stand-alone local search for a memetic algorithm considering that MA_{ICP} is outperformed by every other method in this comparison; however, its combination with CB in MA_{HYB} leads to the best overall results, so, all things considered, it is a useful technique.

6.3. Final Comparison

The experimental study concludes comparing the best memetic algorithm among the ones proposed in this paper (MA_{HYB}) and two other methods, namely, the memetic algorithm MA_{SCP} proposed in [32] and a constraint programming approach.

To our best knowledge, MA_{SCP} is the current best-performing approach in the literature. As pointed out, this method was shown to outperform the genetic algorithm previously proposed in [31], as well as priority rules, both classical ones and others built by means of genetic programming approaches [34,35] in terms of the quality of the solutions computed.

In order to make the comparison more comprehensive, MA_{HYB} is also compared to a constraint programming approach that was built using the commercial solver IBM ILOG CP Optimizer (version 12.9). This solver is specialized in scheduling and has been shown to be very effective in different problems [9,57]. The $(1, Cap(t)||\sum T_i)$ problem was modeled in a conventional way, using interval variables (`IloIntervalVar`) to represent the jobs. The machine was modeled as a non-renewable resource using a cumulative function defined as a sum of pulse functions (`IloPulse`), enforcing that its capacity is never exceeded by an `IloAlwaysIn` constraint for each capacity interval. Finally, the objective function was set to minimize the total tardiness, defined as a numerical expression (`IloNumExpr`) that sums the positive differences between the completion time of the jobs and their due dates.

The constraint programming approach, referred to as CPO, was implemented in C++. Besides, 30 independent runs were performed on each instance (since it has some stochasticity), setting a time limit of $n/2$ s, as for the other methods. In all these experiments, CPO was run using one worker and the default values for CP Optimizer's parameters.

Table 3 shows the results obtained by MA_{SCP}, CPO and MA_{HYB} in terms of the error of the solutions reached by each algorithm. As can be observed, MA_{HYB} conclusively

outperforms MA_{SCP} and CPO. For all groups of instances, the errors yielded by MA_{HYB} are at least one order of magnitude smaller than the errors of the solutions computed by the two other methods. MA_{SCP} performs (much) better than CPO on the smaller instances with $n \in \{120, 250\}$, whereas for the largest instances CPO gets better results. This suggests that MA_{SCP} suffers from some scalability issues, what indicates that the SCP local search method becomes less effective as the size of the instances grows. However, the new local search methods proposed in this paper lead MA_{HYB} to scale to much larger instances, always finding solutions very close to the best known ones.

Figure 10 depicts a boxplot with the best and average errors of the three methods over the whole set of instances. It confirms that MA_{HYB} is clearly outperforms both MA_{SCP} and CPO, with no overlapping between its boxes and those of the other two methods. In addition, the plot shows that even the few outliers from MA_{HYB} correspond to solutions with small errors.

Table 3. Summary of results from MA_{SCP}, MA_{HYB} and CPO after being run with a time limit of $n/2$ s. Errors in percentage of the best and average solutions over 30 runs are reported.

		MA_{SCP}		CPO		MA_{HYB}	
n	***MC***	**Best**	**Avg.**	**Best**	**Avg.**	**Best**	**Avg.**
120	3	0.01	0.49	3.82	3.85	**0.00**	**0.01**
	5	0.08	0.98	4.54	4.54	**0.00**	**0.03**
	7	0.35	1.50	5.47	5.47	**0.00**	**0.07**
	10	0.65	1.97	6.43	6.63	**0.00**	**0.17**
	Avg.	0.27	1.23	5.06	5.12	**0.00**	**0.07**
250	10	1.46	4.09	6.29	6.36	**0.00**	**0.12**
	20	1.64	3.81	9.45	9.46	**0.00**	**0.14**
	30	1.98	4.71	12.74	12.74	**0.07**	**0.37**
	Avg.	1.70	4.20	9.49	9.52	**0.02**	**0.21**
500	10	8.54	20.00	8.97	8.97	**0.00**	**0.12**
	20	4.35	11.55	9.89	9.94	**0.12**	**0.37**
	30	6.92	13.41	16.90	17.24	**0.16**	**0.38**
	Avg.	6.60	14.99	11.92	12.05	**0.09**	**0.29**
750	10	7.88	16.21	6.93	6.95	**0.14**	**0.62**
	20	6.08	14.81	10.62	10.63	**0.17**	**0.28**
	30	12.97	21.98	15.70	16.14	**0.00**	**0.27**
	50	18.58	25.21	13.46	13.46	**0.00**	**0.29**
	Avg.	11.38	19.55	11.68	11.79	**0.08**	**0.36**
1000	10	15.72	41.02	9.23	9.39	**0.64**	**1.07**
	20	5.86	11.63	5.64	5.67	**0.23**	**0.39**
	30	21.08	31.39	14.79	14.85	**0.34**	**0.83**
	50	27.97	41.59	14.46	14.46	**0.08**	**0.14**
	100	43.66	50.27	8.77	8.77	**0.00**	**0.20**
	Avg.	22.86	35.18	10.58	10.63	**0.26**	**0.53**
All		9.78	16.66	9.69	9.76	**0.10**	**0.31**

A series of statistical inference tests was performed to the results shown in Table 3 with the objective of making the experimental study more robust. The tests are aimed at the detection of statistically significant differences among the average errors of the solutions yielded by the methods, and they were conducted over all the instances in the benchmark set.

In order to know if there are significant differences between the results achieved by the algorithms and rank them, an Aligned Friedman Rank Test was performed, following [46,58,59]. This is a multiple-comparison non-parametric test, whose null hypothesis considers that there are no differences in the rankings of the algorithms. If the null hypothesis is rejected, the method at the top of the ranking will be compared against the rest of them by means of a collection of post-hoc procedures. The considered post-hoc

procedures (Bonferroni–Dunn, Holm, Hochberg, Hommel, Holland, Rom, Finn, Finner and Li) are described in [58].

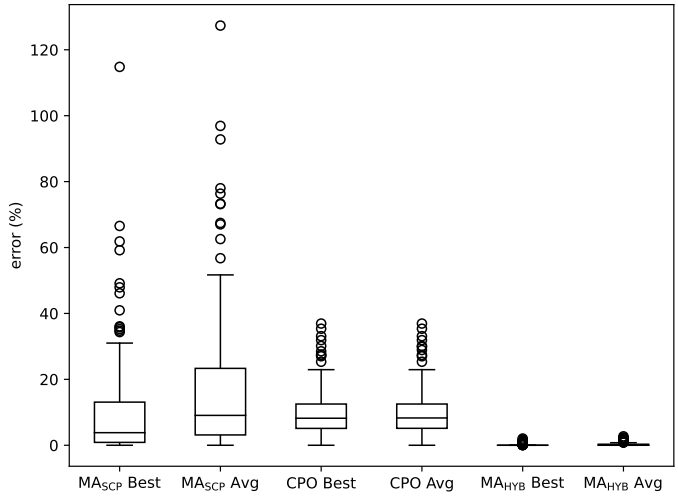


Figure 10. Boxplot comparing best and average results yielded by MA_{SCP}, CPO and MA_{HYB}.

The average ranking calculated by the Aligned Friedman Rank Test (distributed according to χ^2 with 2 degrees of freedom: 135.68) is shown in Table 4. As can be observed, MA_{HYB} appears in first place, followed by CPO and MA_{SCP} in second and third place, respectively. The test returned a p-value of 9.59×10^{-11}, meaning that the differences among the results yielded by the algorithms are statistically significant.

Table 4. Average rankings of MA_{HYB}, CPO and MA_{SCP} obtained with the Aligned Friedman Rank Test.

Position	Algorithm	Ranking
1	MA_{HYB}	121.12
2	CPO	335.75
3	MA_{SCP}	399.62

In order to compare MA_{HYB} with the other methods (MA_{SCP} and CPO), a series of post-hoc procedures were carried out. These procedures share the same null hypothesis, which states that the distributions of the results obtained by MA_{HYB} and the comparing methods are equal. Table 5 shows the adjusted p-values obtained by each procedure, p_{Bonf} (Bonferroni–Dunn), p_{Holm} (Holm), p_{Hoch} (Hochberg), p_{Homm} (Hommel), p_{Holl} (Holland), p_{Rom} (Rom), p_{Finn} (Finner) and p_{Li} (Li). The results lead to the conclusion that the differences in favor of MA_{HYB} are statistically significant, since all the p-values are very close to 0, rejecting the null hypothesis in all cases.

Table 5. Adjusted p-values given by each post-hoc procedure, comparing MA_{HYB} against CPO and MA_{SCP}.

Algorithm	p_{Bonf}	p_{Holm}	p_{Hoch}	p_{Homm}	p_{Holl}	p_{Rom}	p_{Finn}	p_{Li}
MA_{SCP}	1.34×10^{-48}	1.34×10^{-48}	1.34×10^{-48}	1.34×10^{-48}	0.0	1.34×10^{-48}	0.0	6.72×10^{-49}
CPO	3.31×10^{-45}	1.65×10^{-45}	1.65×10^{-45}	1.65×10^{-45}	0.0	1.65×10^{-45}	0.0	1.65×10^{-45}

7. Conclusions

The new local search methods developed in this paper have been shown to give rise to very effective memetic algorithms for solving the $(1, Cap(t)||\sum T_i)$ problem. The results from the experimental study confirm that each of the local search techniques brings improvements to the memetic algorithms in practice. MA_{HYB}, which exploits the hybrid local search procedure, stands out as the best method overall, computing high-quality solutions in short time and showing a remarkable ability to scale to large and challenging

problem instances. As a result, this algorithm clearly outperforms existing approaches, including the memetic algorithm proposed in [32] and a constraint programming approach.

Although the memetic algorithms proposed in this paper are able to converge in short time, they may not be fast enough if real-time requirements are considered. However, the low worst-case complexities of the new local search procedures encourages further research in this direction, as deploying these methods in online settings. Future work will investigate their application in combination with real-time approaches, as schedule builders guided by priority rules. Hopefully, the local search procedures will be able to improve the quality of the solutions obtained by priority rules. In addition, another interesting topic for future research would be exploring the use of mathematical programming methods for solving the $(1, Cap(t)||\sum T_i)$ problem, as well as their combination with the algorithms proposed in this paper.

Author Contributions: Conceptualization, R.M. and C.M.; methodology, R.M and C.M.; software, R.M.; validation, R.M. and C.M; formal analysis, R.M. and C.M.; investigation, R.M. and C.M.; writing—original draft preparation, R.M. and C.M.; writing—review and editing, R.M. and C.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Spanish Government under grant PID2019-106263RB-I00.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The problem instances used in the experimental study and detailed results are available at https://github.com/raulmencia/One-Machine-Scheduling-with-Time-Dependent-Capacity-via-Efficient-Memetic-Algorithms.git (accessed on 19 November 2021).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Pinedo, M.L. *Planning and Scheduling in Manufacturing and Services*; Springer: New York, NY, USA, 2009. [CrossRef]
2. Zhan, Z.H.; Liu, X.F.; Gong, Y.J.; Zhang, J.; Chung, H.S.H.; Li, Y. Cloud Computing Resource Scheduling and a Survey of Its Evolutionary Approaches. *ACM Comput. Surv.* **2015**, *47*, 1–33. [CrossRef]
3. Garey, M.R.; Johnson, D.S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*; W. H. Freeman & Co.: New York, NY, USA, 1979.
4. Brucker, P. *Scheduling Algorithms*, 4th ed.; Springer: Berlin/Heidelberg, Germany, 2004.
5. Ganian, R.; Hamm, T.; Mescoff, G. The Complexity Landscape of Resource-Constrained Scheduling. In Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, Yokohama, Japan, 7–15 January 2021; IJCAI: Los Angeles, CA, USA, 2021; pp. 1741–1747. [CrossRef]
6. Knust, S.; Brucker, P. Complexity Results for Scheduling Problems. Available online: http://www.informatik.uni-osnabrueck.de/knust/class/ (accessed on 19 November 2021).
7. Carlier, J. The one-machine sequencing problem. *Eur. J. Oper. Res.* **1982**, *11*, 42–47. [CrossRef]
8. Brucker, P.; Jurisch, B.; Sievers, B. A Branch and Bound Algorithm for the Job-Shop Scheduling Problem. *Discret. Appl. Math.* **1994**, *49*, 107–127. [CrossRef]
9. Laborie, P.; Rogerie, J.; Shaw, P.; Vilím, P. IBM ILOG CP optimizer for scheduling - 20+ years of scheduling with constraints at IBM/ILOG. *Constraints Int. J.* **2018**, *23*, 210–250. [CrossRef]
10. Ku, W.; Beck, J.C. Mixed Integer Programming models for job shop scheduling: A computational analysis. *Comput. Oper. Res.* **2016**, *73*, 165–173. [CrossRef]
11. Mustu, S.; Eren, T. The single machine scheduling problem with sequence-dependent setup times and a learning effect on processing times. *Appl. Soft Comput.* **2018**, *71*, 291–306. [CrossRef]
12. Soares, L.C.R.; Carvalho, M.A.M. Biased random-key genetic algorithm for scheduling identical parallel machines with tooling constraints. *Eur. J. Oper. Res.* **2020**, *285*, 955–964. [CrossRef]
13. Gonçalves, J.F.; Resende, M.G.C. An extended Akers graphical method with a biased random-key genetic algorithm for job-shop scheduling. *Int. Trans. Oper. Res.* **2014**, *21*, 215–246. [CrossRef]
14. Gonçalves, J.; Mendes, J.; Resende, M. A genetic algorithm for the resource constrained multi-project scheduling problem. *Eur. J. Oper. Res.* **2008**, *189*, 1171–1190. [CrossRef]
15. Queiroga, E.; Pinheiro, R.G.S.; Christ, Q.; Subramanian, A.; Pessoa, A.A. Iterated local search for single machine total weighted tardiness batch scheduling. *J. Heuristics* **2021**, *27*, 353–438. [CrossRef]
16. Nowicki, E.; Smutnicki, C. An Advanced Tabu Search Algorithm for the Job Shop Problem. *J. Sched.* **2005**, *8*, 145–159. [CrossRef]

17. Chen, Y.; Lu, J.; He, R.; Ou, J. An Efficient Local Search Heuristic for Earth Observation Satellite Integrated Scheduling. *Appl. Sci.* **2020**, *10*, 5616. [CrossRef]
18. França, P.M.; Mendes, A.; Moscato, P. A memetic algorithm for the total tardiness single machine scheduling problem. *Eur. J. Oper. Res.* **2001**, *132*, 224–242. [CrossRef]
19. Abdel-Basset, M.; Mohamed, R.; Abouhawwash, M.; Chakrabortty, R.K.; Ryan, M.J. A Simple and Effective Approach for Tackling the Permutation Flow Shop Scheduling Problem. *Mathematics* **2021**, *9*, 270. [CrossRef]
20. Onwubolu, G.; Davendra, D. Scheduling flow shops using differential evolution algorithm. *Eur. J. Oper. Res.* **2006**, *171*, 674–692. [CrossRef]
21. Merkle, D.; Middendorf, M.; Schmeck, H. Ant colony optimization for resource-constrained project scheduling. *IEEE Trans. Evol. Comput.* **2002**, *6*, 333–346. [CrossRef]
22. Zhou, H.; Pang, J.; Chen, P.K.; Chou, F.D. A modified particle swarm optimization algorithm for a batch-processing machine scheduling problem with arbitrary release times and non-identical job sizes. *Comput. Ind. Eng.* **2018**, *123*, 67–81. [CrossRef]
23. Malakar, S.; Ghosh, M.; Bhowmik, S.; Sarkar, R.; Nasipuri, M. A GA based hierarchical feature selection approach for handwritten word recognition. *Neural Comput. Appl.* **2020**, *32*, 2533–2552. [CrossRef]
24. Bacanin, N.; Stoean, R.; Zivkovic, M.; Petrovic, A.; Rashid, T.A.; Bezdan, T. Performance of a Novel Chaotic Firefly Algorithm with Enhanced Exploration for Tackling Global Optimization Problems: Application for Dropout Regularization. *Mathematics* **2021**, *9*, 2705. [CrossRef]
25. Hall, N.G.; Potts, C.N. Supply Chain Scheduling: Batching and Delivery. *Oper. Res.* **2003**, *51*, 566–584. [CrossRef]
26. Wang, X.; Ren, T.; Bai, D.; Ezeh, C.; Zhang, H.; Dong, Z. Minimizing the sum of makespan on multi-agent single-machine scheduling with release dates. *Swarm Evol. Comput.* **2021**, 100996. [CrossRef]
27. Jin, F.; Song, S.; Wu, C. A simulated annealing algorithm for single machine scheduling problems with family setups. *Comput. Oper. Res.* **2009**, *36*, 2133–2138. [CrossRef]
28. Adams, J.; Balas, E.; Zawack, D. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Manag. Sci.* **1988**, *34*, 391–401. [CrossRef]
29. Hernández-Arauzo, A.; Puente, J.; Varela, R.; Sedano, J. Electric vehicle charging under power and balance constraints as dynamic scheduling. *Comput. Ind. Eng.* **2015**, *85*, 306–315. [CrossRef]
30. Graham, R.; Lawler, E.; Lenstra, J.; Kan, A. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Ann. Discret. Math.* **1979**, *5*, 287–326.
31. Mencía, C.; Sierra, M.R.; Mencía, R.; Varela, R. Genetic Algorithm for Scheduling Charging Times of Electric Vehicles Subject to Time Dependent Power Availability. In *International Work-Conference on the Interplay Between Natural and Artificial Computation*; Springer International Publishing: Cham, Switzerland, 2017; pp. 160–169.
32. Mencía, C.; Sierra, M.R.; Mencía, R.; Varela, R. Evolutionary one-machine scheduling in the context of electric vehicles charging. *Integr. Comput. Aided Eng.* **2019**, *26*, 49–63. [CrossRef]
33. Vepsalainen, A.P.J.; Morton, T.E. Priority Rules for Job Shops with Weighted Tardiness Costs. *Manag. Sci.* **1987**, *33*, 1035–1047. [CrossRef]
34. Gil-Gala, F.J.; Mencía, C.; Sierra, M.R.; Varela, R. Evolving priority rules for on-line scheduling of jobs on a single machine with variable capacity over time. *Appl. Soft Comput.* **2019**, *85*, 105782. [CrossRef]
35. Gil-Gala, F.J.; Sierra, M.R.; Mencía, C.; Varela, R. Combining hyper-heuristics to evolve ensembles of priority rules for on-line scheduling. *Nat. Comput.* **2020**. [CrossRef]
36. Koulamas, C. The total tardiness problem: Review and extensions. *Oper. Res.* **1994**, *42*, 1025–1041. [CrossRef]
37. Giffler, B.; Thompson, G.L. Algorithms for Solving Production Scheduling Problems. *Oper. Res.* **1960**, *8*, 487–503. [CrossRef]
38. Kolisch, R. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *Eur. J. Oper. Res.* **1996**, *90*, 320–333. [CrossRef]
39. Artigues, C.; Lopez, P.; Ayache, P. Schedule Generation Schemes for the Job Shop Problem with Sequence-Dependent Setup Times: Dominance Properties and Computational Analysis. *Ann. Oper. Res.* **2005**, *138*, 21–52. [CrossRef]
40. Palacios, J.J.; Vela, C.R.; Rodríguez, I.G.; Puente, J. Schedule Generation Schemes for Job Shop Problems with Fuzziness. In Proceedings of the 21st European Conference On Artificial Intelligence, Prague, Czech Republic, 18–22 August 2014; pp. 687–692.
41. Sierra, M.R.; Mencía, C.; Varela, R. New schedule generation schemes for the job-shop problem with operators. *J. Intell. Manuf.* **2015**, *26*, 511–525. [CrossRef]
42. Mencía, R.; Sierra, M.R.; Mencía, C.; Varela, R. Schedule Generation Schemes and Genetic Algorithm for the Scheduling Problem with Skilled Operators and Arbitrary Precedence Relations. In Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, 7–11 June 2015; AAAI Press: Palo Alto, CA, USA, 2015; pp. 165–173.
43. Sprecher, A.; Kolisch, R.; Drexl, A. Semi-active, active, and non-delay schedules for the resource-constrained project scheduling problem. *Eur. J. Oper. Res.* **1995**, *80*, 94–102. [CrossRef]
44. Holland, J. *Adaptation in Natural and Artificial Systems*; University of Michigan Press: Ann Arbor, MI, USA, 1975.
45. Guo, W.; Xu, P.; Zhao, Z.; Wang, L.; Zhu, L. Scheduling for airport baggage transport vehicles based on diversity enhancement genetic algorithm. *Nat. Comput.* **2020**, *19*, 663–672. [CrossRef]

46. Mencía, R.; Mencía, C.; Varela, R. Efficient repairs of infeasible job shop problems by evolutionary algorithms. *Eng. Appl. Artif. Intell.* **2021**, *104*, 104368. [CrossRef]
47. Davis, L. Applying Adaptive Algorithms to Epistatic Domains. In Proceedings of the 9th International Joint Conference on Artificial Intelligence, Los Angeles, CA, USA, 18–23 August 1985; pp. 162–164.
48. Kirkpatrick, S.; Gelatt, C.D.; Vecchi, M.P. Optimization by Simulated Annealing. *Science* **1983**, *220*, 671–680. [CrossRef]
49. Glover, F.W.; Laguna, M. *Tabu Search*; Kluwer: Alphen aan den Rijn, The Netherlands, 1997. [CrossRef]
50. Idzikowski, R.; Rudy, J.; Gnatowski, A. Solving Non-Permutation Flow Shop Scheduling Problem with Time Couplings. *Appl. Sci.* **2021**, *11*, 4425. [CrossRef]
51. Du, J.; Leung, J.Y.T. Minimizing Total Tardiness on One Machine Is NP-Hard. *Math. Oper. Res.* **1990**, *15*, 483–495. [CrossRef]
52. Talbi, E. *Metaheuristics—From Design to Implementation*; Wiley: Hoboken, NJ, USA, 2009.
53. Gao, L.; Zhang, G.; Zhang, L.; Li, X. An efficient memetic algorithm for solving the job shop scheduling problem. *Comput. Ind. Eng.* **2011**, *60*, 699–705. [CrossRef]
54. Mencía, R.; Mencía, C.; Varela, R. A memetic algorithm for restoring feasibility in scheduling with limited makespan. *Nat. Comput.* **2020**. [CrossRef]
55. Machado-Domínguez, L.F.; Paternina-Arboleda, C.D.; Vélez, J.I.; Sarmiento, A.B. A memetic algorithm to address the multi-node resource-constrained project scheduling problem. *J. Sched.* **2021**, *24*, 413–429. [CrossRef]
56. Williams, J.W.J. Algorithm 232 - Heapsort. *Commun. ACM* **1964**, *7*, 347–348. [CrossRef]
57. Vilím, P.; Laborie, P.; Shaw, P. Failure-Directed Search for Constraint-Based Scheduling. In Proceedings of the International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research; CPAIOR 2015, Barcelona, Spain, 18–22 May 2015; Springer: Cham, Switzerland, 2015; pp. 437–453. [CrossRef]
58. Derrac, J.; García, S.; Molina, D.; Herrera, F. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm Evol. Comput.* **2011**, *1*, 3–18. [CrossRef]
59. Gallardo, J.E.; Cotta, C. A GRASP-based memetic algorithm with path relinking for the far from most string problem. *Eng. Appl. Artif. Intell.* **2015**, *41*, 183–194. [CrossRef]