GRASP algorithm for the unrelated parallel machine scheduling problem with setup times and additional resources

Juan C. Yepes-Borrero[a,*], Fulgencia Villa[a], Federico Perea[a], Juan Pablo Caballero-Villalobos[b]

[a] *Grupo de Sistemas de Optimización Aplicada, Instituto Tecnológico de Informática, Ciudad Politécnica de la Innovación, Edifico 8G, Acc. B. Universitat Politècnica de València, Camino de Vera s/n, 46021, València, Spain.*

[b] *Departamento de Ingeniería Industrial, Pontificia Universidad Javeriana, Calle 40 Nº 5-50 Ed. José Gabriel Maldonado, S.J. Bogotá, Colombia*

Abstract

This paper provides practitioners with new approaches for solving realistic scheduling problems that consider additional resources, which can be implemented on expert and intelligent systems and help decision making in realistic settings. More specifically, we study the unrelated parallel machine scheduling problem with setup times and additional limited resources in the setups (UPMSR-S), with makespan minimization criterion. This is a more realistic extension of the traditional problem, in which the setups are assumed to be done without using additional resources (e.g. workers). We propose three metaheuristics following two approaches: a first approach that ignores the information about additional resources in the constructive phase, and a second approach that takes into account this information about the resources. Computational experiments are carried out over a benchmark of small and large instances. After the computational analysis we conclude that the second approach shows an excellent performance, overcoming the first approach.

Keywords: Unrelated parallel machines, Scheduling, Sequence dependent setup times, Makespan, Additional resources, GRASP.

*Corresponding author. Tel: +34 96 387 70 00. Ext: 74914. Fax: +34 96 387 74 99

Email addresses: juancamiloyepes@gmail.com (Juan C. Yepes-Borrero), mfuvilju@eio.upv.es (Fulgencia Villa), perea@eio.upv.es (Federico Perea), juan.caballero@javeriana.edu.co (Juan Pablo Caballero-Villalobos)

1. Introduction

Nowadays, companies face more changing and volatile environments, where increased competitiveness and personalization of products play a key role. Therefore, companies need smart tools that help their decision-making process to become more efficient and effective. In this paper, we propose intelligent methods to solve hard decision-making problems using optimization techniques, in order to contribute to the smart factory concept[1], that is, sustainable and intelligent industries. Scheduling plays a very important role in industry. There are many different scheduling problems modeling different types of production processes. In many cases, factories need flexibility in their productive processes to achieve a higher personalization of their products. This flexibility may include the need of additional resources, which makes scheduling problems much more difficult to solve.

Among the scheduling problems that appear in industrial processes, one of them is the so called Unrelated Parallel Machines scheduling problem (UPM), where a set of jobs have to be processed by a set of parallel machines. As the machines are unrelated, the processing time of a job may be different depending on the machine the job is assigned to. Recently, many studies have been conducted on the Unrelated Parallel Machine scheduling problem with Setup times between jobs (UPMS), which is an extension of the UPM problem. The UPMS arises when machines need to be reconfigured after the processing of one job, and before the processing of the next one.

In the literature of the UPMS, no constraint is normally assumed made on the number of setups that can be done at the same time. In other words, at any point in time one may do as many setups as needed. Arguably, it is common that machines process jobs automatically, without the help of extra resources. However, we want to underline that in manufacturing environments, the machine setups between jobs is usually done by additional resources (e.g. workers). Since the number of these available resources is typically limited, the number of setups that can be done at the same time is limited. Therefore, an extension, and more realistic approach to the UPMS is the Unrelated Parallel Machine scheduling problem with setup times and additional Resources in the Setups (UPMSR-S), which is the problem introduced in this paper.

[1] https://www.capgemini.com/resources/preparing-for-smart-factories/

Among the variety of objectives considered in scheduling, one of the most studied is the minimization of the makespan, denoted by $C_{\max}$. The makespan is defined as the completion time of the schedule. In other words, the makespan is the latest completion time of a job. In this paper, we address the UPMSR-S, with the objective of minimizing the makespan.

The rest of the paper is organized as follows: In Section 2, an overview of the related literature is presented. In Section 3 the formal definition of the problem and a mathematical model are presented. Sections 4 and 5 introduce the heuristics and metaheuristics designed for solving the UPMSR-S. Section 6 shows the experimental campaign to assess the algorithms proposed. Finally, in Section 7 some conclusions and directions for future research are given.

2. Literature review

Unrelated parallel machine scheduling problems have been widely studied in the past years (see e.g. Fanjul-Peyro and Ruiz (2010), Fanjul-Peyro and Ruiz (2011), Arroyoa and Leung (2017)). The consideration of sequence dependent setup times between jobs (UPMS) has also received lot of attention. The interested reader in the UPMS problem is referred to Vallada and Ruiz (2011), Kurz and Askin (2001), Kim et al. (2002), among others. Allahverdi (2015) presents a review of scheduling problems of parallel machines with setup times.

However, the problem with additional resources has been the focus of far fewer studies in the research community, especially when the additional resources are needed to do the setups between jobs. In this section we focus our attention on the most recent algorithms for the parallel machine scheduling problems considering setup times with the objective to minimize makespan. Besides, we also review the available algorithms for parallel machine scheduling problems that consider additional resources.

Kurz and Askin (2001) present a mathematical programming model and several heuristics for the parallel machines scheduling problem with sequence-dependent set-up times. Rabadi et al. (2006) present a heuristic for the unrelated machine case. Helal et al. (2006) propose a tabu search algorithm to minimize the makespan. De-Paula et al. (2007) propose a method based on the VNS strategy for identical and unrelated parallel machines to minimize the makespan. Arnaout et al. (2010) propose a two-stage ant colony optimization algorithm. Vallada and Ruiz (2011) propose a genetic algorithm. More recently, Avalos-Rosales et al. (2015) propose a metaheuristic algorithm for the unrelated parallel machine problem with sequence and machine-dependent setup times and Diana et al. (2014) propose an immune-inspired algorithm for the same problem. Fanjul-Peyro et al. (2019) propose a new mixed integer linear program and a mathematical programming based algorithm for the UPMS. Although the minimization of the makespan is one of the most studied optimization criterion in scheduling, other objectives have been analyzed. For example, Expósito-Izquierdo et al. (2019) propose a metaheuristic to study the effect of learning or tiredness on the setup times in a scheduling problem with identical parallel machines.

As stated earlier, there are fewer studies for scheduling problems with additional resources. Ruiz-Torres et al. (2007) study a uniform parallel machines problem subject to a secondary resource in order to minimize the number of tardy jobs, where the speed of the machines depends on the allocation of the secondary resource. Ruiz and Andrés-Romano (2011) propose heuristics for the unrelated parallel machines problem with resource-assignable sequence dependent setup times, where the resources are not limited and with the objective of minimizing a linear combination of the total resources assigned and the total completion time. Afzalirad and Rezaeian (2016) propose an integer mathematical model and a genetic algorithm for an unrelated parallel machine scheduling problem with sequence dependent setup times, resource constraints on the processing times, precedence constraints and machine eligibility restrictions.

Some other works of different variations of parallel machines problems with additional resources can be found in Chen (2004), Edis and Oguz (2012), Edis and Ozkarahan (2012), Edis et al. (2013) and Bitar et al. (2016). More recently, Fanjul-Peyro et al. (2017) present models and matheuristics for the unrelated parallel machine scheduling problem with additional resources. For the same problem, Arbaoui and Yalaoui (2018) use constraint programming, Villa et al. (2018) present some heuristics and Fleszar and Hindi (2018) present different algorithms, including mathematical programming models and constraint programming techniques.

The GRASP algorithm (Greedy Randomized Adaptive Search Procedure) was introduced by Feo and Resende (1989). Ever since then, this algorithm has successfully been applied to solve real combinatorial problems. Different examples of applications can be found in Resende and Ribeiro (2014). Scheduling problems is one of the topics in which GRASP has been applied. Feo et al. (1991) propose a GRASP algorithm to solve a single machine scheduling problem with flow time and earliness penalties. Feo et al. (1996) use a GRASP algorithm to solve a single machine scheduling with sequence dependent setup costs and linear delay penalties. For the job shop scheduling problem, Aiex et al. (2003) and Binato et al. (2002) design GRASP algorithms. Rajkumar et al. (2011) present a GRASP algorithm to solve the flexible job-shop scheduling problem with limited resource constraints. Laguna and Velarde (1991) solve the just-in-time scheduling problem in parallel machines and they propose an approach that combines elements of GRASP algorithm and Tabu Search. Finally, some other fields in which GRASP algorithms have been successfully applied are project scheduling (see Alvarez-Valdes et al. (2008)), cutting and packing (see Parreño et al. (2010)), and industrial applications (see Anticona (2006)).

Most related works in the literature, dealing with scheduling and setups, do not consider scarce resources. We strongly believe that neglecting the need of resources is not always realistic, since in most manufacturing processes machine setups are typically performed (or at least controlled) by workers. We therefore consider this paper as an attempt to close the gap between academic research and real scheduling in parallel machine problems with setups.

3. Problem formulation

In this section we formally introduce the UPMSR-S, for which the following sets and parameters are needed:

- Set $N = \{1, \dots, n\}$ of jobs to be scheduled, indexed by j, k and ℓ.
- Set $M = \{1, \dots, m\}$ of unrelated parallel machines, indexed by i.
- Set $T = \{1, \dots, t_{\max}\}$ of time units, indexed by t. Parameter $t_{\max}$ is a large value, which is an upper bound for the makespan.
- Parameter p_{ij} is the processing time of job j on machine i.
- Parameter s_{ijk} is the setup time of machine i between the processing of jobs j and k, in this order.
- Parameter r_{ijk} is the necessary number of renewable resources to do the setup on machine i between job j and job k, in this order.
- Parameter $R_{\max}$ is the number of available resources, needed for the setups.

The m machines are always available, and each machine can process only one job at a time and without preemption. Additionally, there is no precedence restriction in the sequence of jobs and all machines are available from time 0. The setup times and resources are both sequence and machine dependent. That is, the setup time on machine i between jobs j and k may be different from the setup time on the same machine between jobs k and j. Furthermore, the setup time between jobs j and k on machine i may be different from the setup time between jobs j and k on other machines.

Having limited resources to do the setups, the feasibility of the solution obtained depends on the number of resources used at any point in time. For instance, if we want to do setups on two or more machines at the same time, it is necessary that the sum of the resources required by these setups is not greater than $R_{\max}$. If this restriction can not be accomplished, it is necessary to rearrange one or more setups, possibly generating idle times in the machines.

The following definition will be needed in the rest of the paper.

Definition 3.1. *Job k is the successor of job j if the two jobs are processed by the same machine i and between j and k, the machine i does not process another job. In the same way, job j is the predecessor of k if k is the successor of j.*

3.1. MILP model formulation

In order to present a mixed integer linear ($MILP$) formulation for the UPMSR-S problem, we define the following variables.

- Binary variable Y_{ij} takes value 1 if job j is processed on machine i, 0 otherwise.
- Binary variable X_{ijk} takes value 1 if job k is the successor of job j on machine i, 0 otherwise.
- Binary variable H_{ijkt} takes value 1 if the setup on machine i, between the successive jobs j and k, ends at instant t, 0 otherwise.
- $C_{\max}$ is the maximum completion time of the schedule or makespan.

Additionally, it is necessary to define the set $N_0 = N \cup \{0\}$, where 0 is a dummy job in which all machines start and end. We set $s_{i0k} = s_{ik0} = r_{i0k} = r_{ik0} = p_{i0} = 0, \forall\ i \in M; \forall\ k \in N_0$.

A model for the UPMSR-S is:

$$\min C_{\max} \tag{1}$$

$$\text{s.t.} \sum_{k \in N} X_{i0k} \leq 1, \ i \in M \tag{2}$$

$$\sum_{i \in M} Y_{ij} = 1, \ j \in N \tag{3}$$

$$Y_{ij} = \sum_{k \in N_0, j \neq k} X_{ijk}, \ i \in M, j \in N \tag{4}$$

$$Y_{ik} = \sum_{j \in N_0, j \neq k} X_{ijk}, \ i \in M, k \in N \tag{5}$$

$$\sum_{t \leq t_{\max}} H_{ijkt} = X_{ijk}, \forall i \in M, j \in N_0, k \in N, k \neq j \tag{6}$$

$$\sum_{t} tH_{ijkt} \geq \sum_{\ell \in N_0} \sum_{t \leq t_{\max}} H_{i\ell jt}(t + s_{ijk} + p_{ij}) - \bar{M}(1 - X_{ijk}),$$
$$\forall \ i \in M, j \in N_0, k \in N, k \neq j \tag{7}$$

$$\sum_{i \in M, j \in N_0, k \in N, k \neq j, t' \in \{t, \ldots, t+s_{ijk}-1\}} r_{ijk} H_{ijkt'} \leq R_{\max}, \forall \ t \leq t_{\max} \tag{8}$$

$$\sum_{t \leq t_{\max}} tH_{ijkt} \leq C_{\max}, \forall i \in M, j \in N_0, k \in N_0, k \neq j \tag{9}$$

$$X_{ijk} \geq 0, \ Y_{ij} \geq 0, H_{ijkt} \in \{0, 1\}.$$

The objective (1) minimizes the makespan of the solution. Constraints (2) establish that at most one job is assigned to the first position of the sequence of each machine. Constraints (3) ensure that each job is assigned to one and only one machine. Constraints (4) ensure that every job j that is processed on machine i has a unique successor k. Constraints (5) ensure that each job k that is processed on machine i, has a unique predecessor j. Constraints (6) ensure that for every machine i and for each pair of successive jobs j and k on machine i, the setup between j and k must end in one and only one moment before $t_{\max}$. Constraints (7) ensure that the setup between two successive jobs j and k on machine i, has to end at the earliest, when the previous setup ends plus the process time of job j on corresponding i, plus the setup time between jobs j and k on machine i. Here, $\bar{M}$ is a sufficiently large value. Constraints (8) ensure that for any instant of time, the number of resources used does not exceed $R_{\max}$. Finally, constraints (9) impose that the makespan must be greater than or equal to the final instant of all the setups done, including the final fictitious setup between the last job processed and the dummy job 0. Note that, due to the structure of the problem, X and Y can be relaxed. Since H are binary, (6) implies that X will be integer. Therefore (2) implies that X are binary. Analogously, (4)-(5) imply that Y is integer, and adding (3) force Y to be binary.

As we will see in the experiments, this model can only solve instances of small size. Therefore, in the next sections we propose more efficient approaches.

4. Heuristics

Solving a UPMSR-S problem involves deciding the following three subproblems:

1. Assignment problem. Decide which jobs should be processed on each machine.
2. Sequencing problem. Decide the order in which the jobs must be processed.
3. Timing problem. Decide the time on which the jobs and setups are processed.

Due to the complexity of the problem, we divide the algorithms proposed in this section into two phases: constructive phase and repairing phase. In the first one, jobs are assigned and sequenced on machines (decision 1 and decision 2). In the second phase, the solution obtained in the constructive phase is analyzed in order to check if the resource constraints are satisfied. If the solution is unfeasible, a procedure to repair the solution is carried out and setups are rearranged (decision 3). Figure 1 shows the general procedure of the proposed heuristic algorithms to solve the UPMSR-S. In the rest of this section we detail both the constructive phase and the repairing phase.

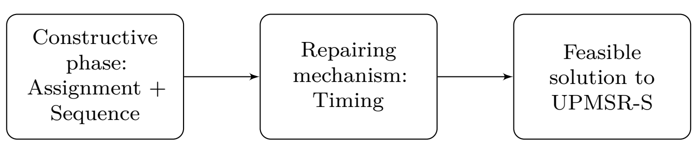


Figure 1: Heuristics flowchart.

4.1. Constructive phase

For the constructive phase, three algorithms following two different approaches have been developed. The first approach consists of building a solution regardless all the information about the resource constraints. In other words, we look for solutions to the UPMS problem. For this approach we have adapted two algorithms from the existing literature on the UPMS. In the second approach we do consider the information about the number of resources used while the solution is built. The algorithm designed following this approach is not based on any previous research. Since the problem is new, we cannot compare with other algorithms in the literature. However, we do re-implement the best algorithms found for the UPMS problem and adapt them to the UPMSR-S (Constructive 1 and Constructive 2, defined below), so they can be compared with the original algorithm that we propose (Constructive 3, defined below).

4.1.1. First approach constructive

For this approach, two constructive algorithms are proposed. Both are based on the two most efficient algorithms we found for the UPMS problem.

Constructive 1: The first constructive is based on the algorithm proposed by Diana et al. (2014). This algorithm is based on the Dynamic Job Assignment with Setups Resource Assignment, proposed by Ruiz and Andrés-Romano (2011), and Multiple Insertion, proposed by Kurz and Askin (2001). The idea in this algorithm is, for each job not assigned (jobs not assigned are referred to as pending jobs), to evaluate the increases in makespan due to its possible inclusion at each of the positions of the partial solution, and to assign the job that generates the lowest makespan increase (this will be the "best" position). This constructive procedure is summarized in Algorithm 1, where C_i is the completion time of machine i, C'_{ijk} is the completion time of machine i in the partial solution after the insertion of job j in position k, and N^* is the set of pending jobs to be assigned.

Algorithm 1 Constructive 1

```
N* ← N
while N* ≠ ∅ do
    foreach j ∈ N* do
        foreach i ∈ M do
            Find the best position k to insert j and save C'_ijk;
        end
    end
    (i*, j*, k*) = arg min_{i,j,k} {C'_ijk};
    Insert j* on i* in position k* and update C_i of machine i*;
    N* ← N* \ {j*};
end
```

Constructive 2: The second constructive is based on the algorithm proposed by Avalos-Rosales et al. (2015) for the UPMS. The idea in this algorithm is to sort the jobs in a non-increasing order according to its average processing time over all machines, defined as $\bar{p_j} = \sum_{i \in M} p_{ij}/m$. Afterwards, take the first job of that list to calculate the increases in makespan C'_i due to its possible inclusion at each of the positions of machine i in the partial solution. Then the job is assigned to the position that generates the lowest makespan increase (this will be the "best" position). This constructive procedure is summarized in Algorithm 2.

Algorithm 2 Constructive 2

```
N* ← N
while N* ≠ ∅ do
    Calculate p̄_j = Σ_{i∈M} p_ij/m ∀j ∈ N*;
    j* = arg max_{j∈N*} {p̄_j};
    foreach i ∈ M do
        Find the best position k to insert j*, and save C'_ijk;
    end
    (i*, k*) = arg min_{i,k} {C'_ij*k};
    Insert j* on i* in position k* and update C_i of machine i*;
    N* ← N* \ {j*};
end
```

4.1.2. Second approach constructive

As opposed to the first approach, in which the information about resource constraints is not taken into account, the second approach does consider the information about the resources. A new constructive is proposed following this approach.

Constructive 3: The idea in this constructive is to take into account, not only the completion time of the machines, but also the number of resources needed to do a setup when a job is assigned. Note that, if we have a sequence $(j_1, j_2, \dots, j_{k-1}, j_k, \dots, j_\ell)$, and we insert a new job j in position k, in general the new sequence is $(j_1, j_2, \dots, j_{k-1}, j, j_k, \dots, j_\ell)$. Then, we no longer do the setup between j_{k-1} and j_k, and we have two new setups: the setup between j_{k-1} and j and the setup between j and j_k.

For this purpose, we define a coefficient that takes into account all the factors that are affected when we insert a job j in some position k of machine i, in the partial solution. We call this coefficient $\lambda_{i,j,k}$, which measures not only the completion time on a machine when a new job is assigned, but also the extra resources needed. This coefficient is defined as:

$$\lambda_{i,j,k} = C'_i + p_{ij} + (\theta_{s(i,k-1,k)} * \theta_{r(i,k-1,k)}) + (\theta_{s(i,k,k+1)} * \theta_{r(i,k,k+1)}) - (\gamma_{s(i,k)} * \gamma_{r(i,k)})$$

where:

- C'_i is the completion time, in the partial solution, of the machine where the job j is inserted.

- $\theta_{s(i,k-1,k)}$ is the time needed for the new setup that we have to do between the jobs in positions $k-1$ and k, when we insert job j in position k on machine i ($\theta_{s(i,k,k+1)}$ is defined analogously).

- $\theta_{r(i,k-1,k)}$ is the number of resources that we need for the new setup between the jobs in positions $k-1$ and k, when we insert job j in position k on machine i ($\theta_{r(i,k,k+1)}$ is defined analogously).

- $\gamma_{s(i,k)}$ is the time needed for the setup that we no longer have to do, when we insert the new job in position k on machine i.

- $\gamma_{r(i,k)}$ is the number of resources that we needed to do the setup that we no longer have to do, when we insert the new job in position k on machine i.

This constructive inserts each pending job at each position of the partial solution. Afterwards, we calculate the λ value and assign the job that generates the lowest value of λ. This algorithm follows the strategy proposed by Diana et al. (2014). The novelty we introduce consists of considering the information about the new resources constraint, to build solutions that need less resources (possibly allowing an increase in makespan). This fact makes the repairing mechanism of phase 2 easier, because the solution built in the constructive phase is closer to feasibility.

Algorithm 3 summarizes this constructive procedure.

Algorithm 3 Constructive 3

```
N* ← N
while N* ≠ ∅ do
    foreach j ∈ N* do
        foreach i ∈ M do
            Find the best position k to insert j and save λ_{i,j,k};
        end
    end
    (i*, j*, k*) = arg min_{i,j,k}{λ_{i,j,k}};
    Insert j* on i* in position k*;
    N* ← N* \ {j*};
end
```

It is important to clarify that the solutions obtained by any of the three constructive algorithms proposed in this section may be non feasible, in the sense that more than $R_{\max}$ resources may be needed at some points in time. Therefore, the repairing mechanism in Section 4.2 is implemented for all three constructive algorithms, which aims at ensuring that the resources used at any point in time do not exceed $R_{\max}$.

4.2. Repairing phase

Once all jobs are assigned and sequenced, it is necessary to evaluate the solution obtained in order to verify if the resource constraints are satisfied. In case more than $R_{\max}$ resources are needed at one point in time, the solution must be repaired. In this section, these evaluation and repairing methods are explained.

The evaluation method consists of calculating the total resources needed at each time instant. If the resource constraints are satisfied at one instant, we evaluate the next time instant. This process is repeated until all the sequence is evaluated or until we find an instant at which the resource constraints are not satisfied. Figure 2 illustrates an example of the repairing mechanism in a solution with 6 jobs, 3 machines and 3 resources available to do the setups. Grey boxes represent jobs being processed, the number inside them being the job index. White boxes represent machine setups. Inside them we see the setup times and resources needs. In Figure 2(a), we can see that the resource constraints are satisfied until instant 4. In instant 5, 5 resources in total are needed to do the setups in machines 1 and 2. Since $R_{\max} = 3$, this solution needs to be repaired.

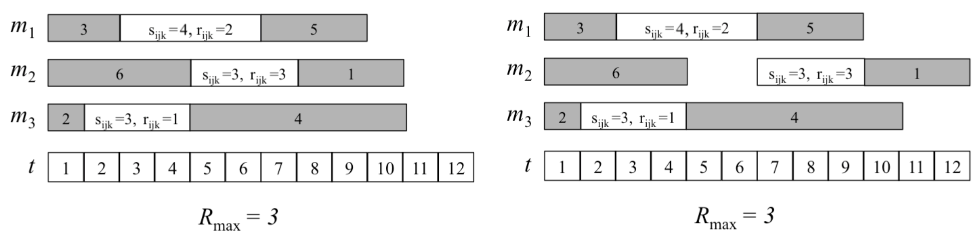
The proposed repairing mechanism consists of postponing the beginning of the setup that starts the latest, out of the setups that overlap at this time instant, until the completion time of the setup finishing first. Then, the consumption of resources is re-evaluated and if the resource constraints are satisfied, we evaluate the next time instant. In this case, the setup on machine 2 is postponed two time units until the setup on machine 1 ends (Figure 2(b)). It is important to mention that if there are several such setups that start at the same time, the rule to break ties is to postpone the setup that is done in the machine with lowest completion time C_i. Algorithm 4 summarizes this repairing procedure.

Algorithm 4 Repairing mechanism

```
for t < t_max do
    Evaluate the consumption of resources at instant t;
    if consumption of resources > R_max then
        Postpone the beginning of the setup in the machine that begins the
        latest out of those that overlap at instant t;
    end
end
```

Hereinafter, we denote the three heuristics algorithms as follows:

- Heuristic 1: Constructive 1 + Repairing mechanism.
- Heuristic 2: Constructive 2 + Repairing mechanism.
- Heuristic 3: Constructive 3 + Repairing mechanism.



(a) Non feasible solution. (b) Feasible solution.

Figure 2: Example Repairing mechanism.

5. GRASP Algorithm

As we will see in the experiments section, the results of the heuristics proposed in Section 4 are far from the optimal solutions. Therefore, in this section we propose multi-start methods based on the heuristics above, in order to find a greater variety of solutions. Multi-start methods are well-know algorithms to diversify the solutions found, in order to overcome local optimality. More specifically, we propose a GRASP (Greedy Randomized Adaptive Search Procedure) algorithm. As stated in the literature review, this type of algorithm is one of the most commonly used multi-start methods. A complete GRASP iteration has two phases: one phase that consists of constructing a partial solution (see Section 5.1), and a second phase that consists of applying a local search procedure in order to improve the solution found in the constructive phase (see Section 5.2).

5.1. Randomization of the constructive phase

Randomization in the constructive phase is widely used in combinatorial optimization in order to avoid local optimality. In this section, we propose the following randomization of the constructive algorithms proposed in Section 4. During the assignment process, instead of choosing the best candidate according to the assignment rule defined, we assign at random one candidate from a restricted candidate list (RCL). The size of the RCL depends on an α value ($\alpha \in [0, 1]$) that we calibrate in the experiments section. The closer α is to 1, the larger the size of RCL.

5.2. Local search

In order to improve the makespan of the sequences obtained by the constructive phase of the GRASP algorithms, a local search consisting of three different phases is proposed. These three phases follow the same philosophy as the second approach (See Section 4.1). They seek for changes in the sequence that take into account not only the completion times on the machines, but also the amount of resources needed. Once all jobs are assigned and sequenced in the constructive phase, the next three local search phases are applied in the following order:

1. Internal swap
2. External swap
3. External insertion

Before and between these operations, the solution is evaluated (and repaired by the repairing mechanism, if necessary) in order to keep the current best solution. After applying the repairing mechanism, the solution may have idle times as we can see in the Figure 2 b). However, we justify to the left this solution before applying the next local search, a procedure we call "shiftleft()", which deletes idle times. This operation possibly introduces infeasibility into the partial solution. If the external swap or the external insertion find a better solution than the current solution, the whole process is repeated after the completion of the external insertion. Algorithm 5 shows a pseudocode of the local search.

Algorithm 5 Pseudocode Local search.

```
Current solution ← Initial solution
Current solution ← Apply Repairing mechanism
Best solution ← Current solution
StopCriteria ← False
while StopCriteria = False do
    StopCriteria ← True
    ShiftLeft(Current solution)
    Current solution ← Apply Internal swap
    Current solution ← Apply Repairing mechanism
    if Current solution < Best solution then
        Best solution ← Current solution
    end
    ShiftLeft(Current solution)
    Current solution ← Apply External swap
    Current solution ← Apply Repairing mechanism
    if Current solution < Best solution then
        Best solution ← Current solution
        StopCriteria ← False
    end
    ShiftLeft(Current solution)
    Current solution ← Apply External insertion
    Current solution ← Apply Repairing mechanism
    if Current solution < Best solution then
        Best solution ← Current solution
        StopCriteria ← False
    end
end
```

We now explain each of the three local search phases more in detail.

5.2.1. Internal swap

This operation is widely used in scheduling problems, as for example in Vallada and Ruiz (2011), Diana et al. (2014) and Arnaout et al. (2010). In this operation, for each job j on each machine i, we test a swap between job j and any other job k processed on the same machine. Note that, after such swap, in general, there will be two setups that we no longer do, and two new setups. For each such swap, we compute a coefficient that considers, not only the completion time of the machines, but also the number of resources needed. We call this coefficient $S_{(j,k)}$ and is defined as:

$$S_{(j,k)} = (\gamma_{s(j,k)} * \gamma_{r(j,k)}) - (\theta_{s(j,k)} * \theta_{r(j,k)}),$$

where:

- $\gamma_{s(j,k)}$ is the time needed for the setups that we no longer have to do when we apply the internal swap.
- $\gamma_{r(j,k)}$ is the number of resources needed to do the setups that we no longer have to do, when we apply the internal swap.
- $\theta_{s(j,k)}$ is the time needed for the new setups that we have to do when we apply the internal swap.
- $\theta_{r(j,k)}$ is the number of resources that we need for the new setups that we have to do when we apply the internal swap.

After evaluating all the possible swaps, we keep the swap that generates the largest $S_{(j,k)}$. We repeat this process while we improve the solution. Algorithm 6 summarizes the internal swap process. For the sake of brevity, $j \in i$ means that job j is assigned to machine i.

Algorithm 6 Internal swap.

StopCriteria $\leftarrow$ *False*
while *StopCriteria* $=$ *False* **do**
    StopCriteria $\leftarrow$ *True*
    foreach $i \in M$ **do**
        Best Swap $\leftarrow 0$
        foreach $j \in i$ **do**
            foreach $k \in i$ *and* $k \neq j$ **do**
                Test the swap job j with job k and compute $S_{(j,k)}$
                if $S_{(j,k)} >$ *Best Swap* **then**
                    Best Swap $\leftarrow S_{(j,k)}$
                    StopCriteria $\leftarrow$ *False*
                end
            end
        end
        Do *Best Swap*
    end
end

5.2.2. External swap

To explain the external swap, we define i' as the machine yielding the makespan. In the external swap, we try to swap each job j previously assigned on the machine i', with each job k of each of the other machines $i \neq i'$. Note that, after each such external swap, in general, there will be two setups in each machine that we no longer have to do, and two new setups on each of the two machines. When we test a swap, we compute a coefficient that follows the same idea as the previous internal swap, defined as:

$$S_{(i,j,k)} = (\rho_{(i,j,k)} + \gamma_{s(i,j,k)} * \gamma_{r(i,j,k)}) - (\phi_{(i,j,k)} + \theta_{s(i,j,k)} * \theta_{r(i,j,k)}),$$

where:

- $\rho_{(i,j,k)}$ is the sum of the processing times of the swapped jobs in the original sequence.
- $\phi_{(i,j,k)}$ is the sum of the processing times of the swapped jobs after the swap.
- $\gamma_{s(i,j,k)}$ is the time needed for the setups that we no longer have to do (on the two machines) when we apply the external swap.

- $\gamma_{r(i,j,k)}$ is the number of resources needed for the setups that we no longer have to do (on the two machines), when we apply the external swap.

- $\theta_{s(i,j,k)}$ is the time needed for the new setups (on the two machines).

- $\theta_{r(i,j,k)}$ is the number of resources needed for the new setups (on the two machines).

When all swaps are tested, we keep the swap that generates the largest $S_{(i,j,k)}$. We repeat this process while we improve the solution. Algorithm 7 summarizes the external swap operation.

Algorithm 7 External swap.

$StopCriteria \leftarrow False$
while $StopCriteria = False$ **do**
    $StopCriteria \leftarrow True$
    $Best\ Swap \leftarrow 0$
    $i' \leftarrow Makespan\ Machine$
    foreach $j \in i'$ **do**
        foreach $i \in M \setminus \{i'\}$ **do**
            foreach $k \in i$ **do**
                Test the swap $j - k$ and compute $S_{(i,j,k)}$
                if $S_{(i,j,k)} > Best\ Swap$ **then**
                    $Best\ Swap \leftarrow S_{(i,j,k)}$
                    $StopCriteria \leftarrow False$
                end
            end
        end
    end
    Do $Best\ Swap$
end

5.2.3. External insertion

This operation consists of testing the insertion of each job scheduled on the machine i' that defines the makespan, in each position on the other machines. Note that, after one such insertion, in general, on machine i' there are two setups that we no longer do, and one new setup. Besides, on the machine where the job is inserted, there will be two new setups, and one of the original setups is no longer done. As in the internal and external swaps, we compute a coefficient that considers the completion time on the machines and the amount of resources needed in the sequence, seeking to reduce this consumption of resources (without significantly increasing the completion time). By abuse of notation, we call this coefficient $S_{(i,j,k)}$ defined as:

$$S_{(i,j,k)} = (C_{max} + \gamma_{s(i,j,k)} * \gamma_{r(i,j,k)}) - (C^*_{(i,j,k)} + \theta_{s(i,j,k)} * \theta_{r(i,j,k)}),$$

where:

- $C_{\max}$ is the makespan in the original sequence.
- $C^*_{(i,j,k)}$ is the completion time of the machine i after job j is inserted in position k.
- $\gamma_{s(i,j,k)}$ is the time needed for the setups that we no longer have to do (on the two machines) when we apply the external insertion.
- $\gamma_{r(i,j,k)}$ is the number of resources needed for the setups that we no longer have to do (on the two machines).
- $\theta_{s(i,j,k)}$ is the time needed for the new setups (on the two machines).
- $\theta_{r(i,j,k)}$ is the number of resources needed for the new setups (on the two machines).

When all insertions are tested, we keep the insertion that yields the largest $S_{(i,j,k)}$. When an insertion is done, this operation is completed. Algorithm 8 summarizes the external insertion process.

Algorithm 8 External Insertion.

$Best\ Insertion \leftarrow 0$
$i' \leftarrow Makespan\ Machine$
foreach $j \in i'$ **do**
  foreach $i \in M \setminus \{i'\}$ **do**
    foreach $k \in i$ **do**
      Test the insertion of job j, in position k of machine i, and compute $S_{(i,j,k)}$
      if $S_{(i,j,k)} > Best\ Swap$ **then**
        $Best\ Insertion \leftarrow S_{(i,j,k)}$
      end
    end
  end
end
Do *Best Insertion*

6. Computational experiments

In order to assess the efficiency and quality of the algorithms proposed in this paper, we test them on a randomly generated benchmark. The benchmark consists of two sets of small and large instances, and is based on the one used in Vallada and Ruiz (2011). Since those are instances for the problem without resources (UPMS), they are here completed by adding the resource needs (r_{ijk}) and the number of available resources ($R_{\max}$), as will be explained later. The set of small instances has 640 instances, with $n \in \{6, 8, 10, 12\}$ and $m \in \{2, 3, 4, 5\}$. The set of large instances has 1000 instances with $n \in \{50, 100, 150, 200, 250\}$ and $m \in \{10, 15, 20, 25, 30\}$. For both groups of instances, the setup times were generated by an integer uniform distribution in the ranges: $\{1-9\}$, $\{1-49\}$, $\{1-99\}$ and $\{1-124\}$. The processing times for both groups of instances were generated by an integer uniform distribution between 1 and 99. By combining the different values of n, the different values of m and the four different distributions for the setup times, we have: 1) 4x4x4 = 64 different configurations for the small instances. 2) 5x5x4 = 100 different configurations for the large instances. Each such configuration has been randomly replicated 10 times, having in total 640 small instances and 1000 large instances. Instances and complete results are available from the authors upon request.

Over these instances we have added the following input data. For small instances, the maximum number of available resources ($R_{\max}$) was generated by an integer uniform distribution between 1 and 2. For large instances, $R_{\max}$ was generated by an integer uniform distribution between 3 and 4. For both instances, the resource needs r_{ijk} were generated by an integer uniform distribution between 1 and $R_{\max}$.

The experiments were run on a Pentium core i7 PC running at 2.60 GHz and 8 GB of RAM memory under Windows 10 64 bit. The platform used for the codes is Microsoft Visual Studio 2013 and the methods were coded in C# under the same .NET Framework.

In order to compare the proposed algorithms, the Relative Percentage Deviation (RPD) is computed for each algorithm and instance, according to the following expression:

$$RPD = \frac{C_{\max}(alg) - C_{\max}(best)}{C_{\max}(best)} \cdot 100,$$

where $C_{\max}(alg)$ is the makespan of the solution obtained with the algorithm tested and $C_{\max}(best)$ is the best known makespan for the instance.

6.1. Heuristics in solutions solved to optimality

The $MILP$ model was implemented using CPLEX 12.6. Only the instances with $n = 6$ were tested, as for larger values of n the $MILP$ seldom returns the optimal solution.

The solver was allowed to run for 1 hour. After this time, the solver was able to find the optimal solution for 140 of these 160 instances. In this section, the three proposed deterministic algorithms (Section 4.1) are compared with these optimal solutions.

Table 1 shows the average RPD between the solutions obtained by each heuristic and the optimal solutions, on these instances. Columns "Av. $t(ms)$" show the average CPU times, in milliseconds, of each heuristic, for each value of m. Column "% optimal" shows the percentage of optimal solutions found by the $MILP$ model, for each value of m. Column "Avg. $t(s)$" shows the average CPU times, in seconds, of the $MILP$ model.

We observe that the solutions obtained by the heuristics of the first approach yield lower RPD than the heuristic of the second approach. We underline that Heuristic 2 yields the lowest average RPD and seems to be the fastest, in this set of instances.

	First Approach				Second Approach			
	Heuristic 1		Heuristic 2		Heuristic 3		$MILP$ Model	
Size	RPD	Av. $t(ms)$	RPD	Av. $t(ms)$	RPD	Av. $t(ms)$	% of optimal	Avg. $t(s)$
6x2	13.43	5.33	16.09	4.21	12.56	5.45	52.5	1881.90
6x3	23.46	5.35	20.45	4.35	24.40	5.65	87.5	1069.10
6x4	28.02	5.28	14.61	4.01	29.51	5.61	100	716.61
6x5	27.61	5.21	8.00	4.15	28.47	5.41	77.5	1093.03
Average	23.13	5.29	14.79	4.18	23.74	5.53	79.37	1190.16

Table 1: Average Relative Percentage Deviation (RPD) in instances solved to optimality for deterministic algorithms.

6.2. Heuristics in small instances

We continue with the comparison among the three heuristics in all 640 small instances. Table 2 shows the RPD and the average time in milliseconds of each algorithm, for each group of instances. We can see that Heuristic 2 yields slightly better RPD than the other algorithms. We can also see that the CPU times of the three algorithms are similar.

	First Approach				Second Approach	
	Heuristic 1		Heuristic 2		Heuristic 3	
Size	RPD	$t(ms)$	RPD	$t(ms)$	RPD	$t(ms)$
6x2	8.06	5.32	9.58	5.41	8.67	6.01
6x3	12.67	5.35	8.57	5.49	13.85	7.01
6x4	18.86	5.21	6.07	5.21	20.20	7.13
6x5	18.30	5.23	1.46	4.45	19.03	6.91
8x2	3.13	5.42	9.32	5.53	3.96	7.13
8x3	9.56	6.3	11.71	5.62	9.75	7.34
8x4	17.03	6.32	10.08	5.74	15.47	7.41
8x5	16.05	5.92	11.04	5.69	19.62	7.03
10x2	5.26	6.52	7.42	6.58	5.40	7.29
10x3	8.45	6.6	5.85	6.9	8.29	7.35
10x4	11.31	6.65	12.02	6.88	12.94	7.37
10x5	13.34	5.98	8.92	5.59	13.28	7.01
12x2	5.18	9.01	8.93	9.45	3.58	8.56
12x3	8.03	8.89	10.88	9.23	6.51	9.12
12x4	8.03	7.9	13.85	8.53	6.93	9.23
12x5	12.88	7.84	18.98	7.86	9.74	8.03
Average	11.01	6.53	9.67	6.51	11.07	7.49

Table 2: Comparison between deterministic algorithms in small instances.

In order to verify if such differences are maintained when the size of the instances increase, the results over the large set are analyzed in the next section.

6.3. Heuristics in large instances

In Table 3 the RPD and average CPU time in seconds, for the three heuristics are shown. As opposed to small instances, in the large instances we can see greater differences between the heuristics. It is specially interesting to note that Heuristic 1 and Heuristic 2 (which do not consider information about resources in the constructive phase) perform much worse than Heuristic 3 (which does consider the information about the resources). We observe that the RPD of Heuristic 3 is less than 1%, while the other heuristics have RPD close to 40% and 70%, respectively. These large differences in the performances of the heuristics proposed are due to the fact that Heuristic 3 considers the resources during the constructive phase, whereas Heuristics 1 and 2 do not. These results also empirically prove that modifying algorithms so that resources are considered in the constructive phase, really improves the quality of the solutions returned. A reason for this is that the repairing phase is easier for Heuristic 3, as the solution obtained during the constructive phase is closer to being feasible.

	First Approach				Second Approach	
	Heuristic 1		Heuristic 2		Heuristic 3	
Size	RPD	$t(s)$	RPD	$t(s)$	RPD	$t(s)$
50x10	39.84	0.010	49.11	0.009	2.39	0.014
50x15	38.51	0.009	57.37	0.007	0.19	0.010
50x20	38.52	0.012	73.44	0.010	0.12	0.021
50x25	38.98	0.013	67.99	0.008	0.52	0.023
50x30	39.30	0.018	55.02	0.009	0.58	0.020
100x10	37.53	0.042	42.85	0.012	0.78	0.045
100x15	38.08	0.039	65.05	0.014	0.12	0.047
100x20	37.43	0.041	75.89	0.020	0.00	0.052
100x25	38.15	0.040	74.99	0.019	0.00	0.049
100x30	37.72	0.043	80.11	0.020	0.00	0.050
150x10	37.37	0.081	57.64	0.070	0.01	0.091
150x15	37.54	0.080	68.98	0.075	0.00	0.089
150x20	37.87	0.078	76.29	0.071	0.00	0.084
150x25	38.35	0.083	75.82	0.070	0.00	0.094
150x30	38.21	0.089	78.88	0.078	0.00	0.124
200x10	39.07	0.120	53.04	0.090	0.07	0.353
200x15	40.44	0.140	70.78	0.099	0.00	0.362
200x20	41.20	0.138	74.14	0.109	0.00	0.342
200x25	42.18	0.233	81.30	0.098	0.00	0.456
200x30	42.23	0.288	85.32	0.094	0.00	0.488
250x10	42.95	0.399	59.09	0.204	0.03	0.501
250x15	43.48	0.322	73.25	0.284	0.00	0.531
250x20	44.27	0.343	80.12	0.293	0.00	0.553
250x25	44.34	0.464	82.01	0.286	0.00	0.609
250x30	45.61	0.589	83.12	0.400	0.00	0.700
Average	39.97	0.148	69.66	0.098	0.19	0.228

Table 3: Comparison between deterministic algorithms in large instances.

6.4. GRASP in instances solved to optimality

In this section we show the results of the GRASP algorithms introduced in Section 5. These algorithms will stop when a time limit is reached as explained later. Table 4 shows the RPD of the three GRASP algorithms for different values of α and for different values of m in the instances solved to optimality. Column "$t(s)$" shows the time limits for all algorithms for each combination of m and n. We observe that for GRASP 1 and GRASP 2, the results are better with larger α, while for GRASP 3, the results are better with smaller α. Note that there is a small difference between GRASP 2 and GRASP 3: GRASP 2 with $\alpha = 0.75$ has an average RPD of 3.35%, while GRASP 3 with $\alpha = 0.25$ has an average RPD of 2.77%. It seems that, as opposed to the heuristics, the GRASP in which the information about resources is considered in the constructive phase (GRASP 3) yields lower RPD, in the instances solved to optimality.

		First Approach						Second Approach		
		GRASP 1			GRASP 2			GRASP 3		
Size	$t(s)$	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.75$	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.75$	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.75$
6x2	3	8.17	7.65	6.08	8.48	2.89	1.91	3.53	1.66	1.53
6x3	3	12.48	8.92	8.13	12.23	7.73	5.97	3.57	3.83	4.65
6x4	3	15.49	10.90	7.91	11.95	5.74	2.53	2.40	4.53	8.34
6x5	3	18.00	9.16	10.82	6.43	4.67	2.99	1.59	2.51	3.73
Av. RPD		13.53	9.16	8.24	9.77	5.26	3.35	2.77	3.13	4.56

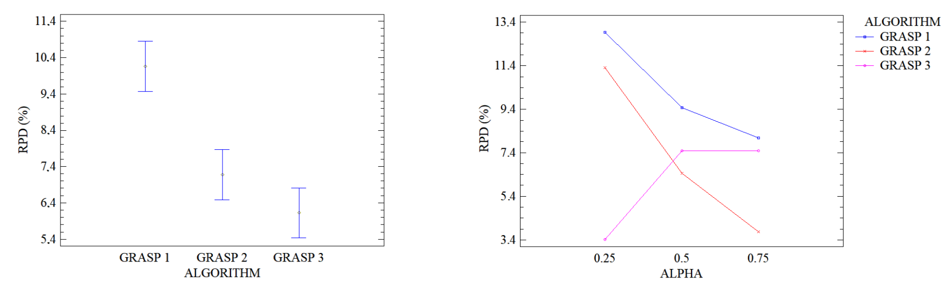
Table 4: Average Relative Percentage Deviation (RPD) in instances solved to optimality for GRASP algorithms.

6.5. GRASP in small instances

Table 5 shows the average RPD for the three GRASP algorithms with different values of α in the small instances. We observe that the algorithms with the first approach (GRASP 1 and GRASP 2) perform better with higher value of α, while GRASP 3 performs better with lower value of α. In order to check if the differences in the average RPD are statistically significant, an analysis of variance (ANOVA), Montgomery (2012) is applied. We consider RPD as the response variable. Two factors are analyzed: ALGORITHM $\in$ {GRASP 1, GRASP 2, GRASP 3}, and ALPHA $\in \{0.25, 0.5, 0.75\}$. Figure 3(a) shows the means plot with LSD intervals at the 95% confidence level for factor ALGORITHM. We observe that there are statistically significant differences between GRASP 1 and the other GRASP algorithms. However, there are no statistically significant differences (overlapped intervals) between GRASP 2 and GRASP 3, although the average RPD of GRASP 3 is lower. Figure 3(b) shows the interaction plot between the two factors considered. We observe that GRASP 3 performs better with lower α (less randomness), while the algorithms with the first approach perform better with larger α (more randomness).

		First Approach						Second Approach		
		GRASP 1			GRASP 2			GRASP 3		
Size	$t(s)$	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.75$	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.75$	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.75$
6x2	3	7.32	5.78	4.76	9.57	2.79	1.16	2.65	1.35	1.58
6x3	3	12.08	7.64	6.91	10.37	6.12	4.45	2.43	2.05	3.37
6x4	3	15.32	10.73	7.74	11.76	5.56	2.35	2.23	4.36	8.19
6x5	3	17.83	9.20	10.53	6.27	4.69	2.71	1.51	2.27	3.37
8x2	3	7.21	4.44	3.73	10.22	5.32	2.55	1.19	1.14	2.42
8x3	3	13.37	9.65	8.63	17.13	5.46	2.53	4.05	3.95	7.72
8x4	3	19.23	12.57	11.77	18.12	9.00	3.41	3.88	6.36	10.06
8x5	3	18.43	16.75	18.15	12.83	9.77	7.14	5.31	8.26	9.54
10x2	5	6.64	5.55	3.99	5.53	2.99	1.97	2.08	2.82	3.34
10x3	5	12.28	9.36	7.29	11.76	6.50	4.30	3.70	4.54	7.55
10x4	5	13.38	11.44	7.00	15.66	9.79	5.44	2.18	5.31	10.96
10x5	5	19.49	12.72	10.97	12.79	8.20	6.23	5.41	7.08	10.21
12x2	5	5.89	3.98	3.55	5.47	4.26	3.18	3.06	4.02	4.92
12x3	5	8.62	6.69	5.51	9.61	7.59	4.46	3.43	6.77	9.77
12x4	5	12.73	9.51	7.16	13.59	5.76	4.18	6.23	11.13	12.29
12x5	5	17.01	15.56	11.51	10.32	9.32	4.09	5.28	14.79	14.37
Av. RPD		12.93	9.47	8.07	11.31	6.44	3.76	3.41	5.39	7.48

Table 5: Average Relative Percentage Deviation (RPD) for GRASP algorithms in small instances.



(a) Means Plot and LSD intervals.

(b) Interaction Plot.

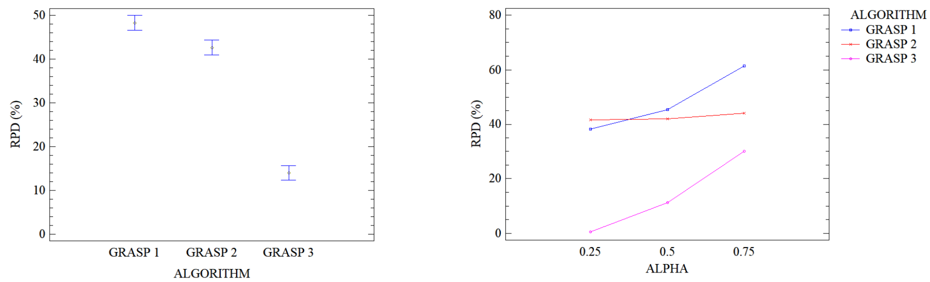
Figure 3: ANOVA in small instances.

6.6. *Comparison between GRASP algorithms in large instances*

Regarding the large instances, Table 6 shows the results obtained by each GRASP algorithm for different values of α. Note that the CPU time $t(s)$ increases with the size of the instance. Similarly as the small instances, the algorithms following the first approach (GRASP 1 and GRASP 2) perform better with higher value of α, while GRASP 3 performs better with lower value of α. Besides, in this group of instances, we observe large differences between the two approaches. GRASP 3 with $\alpha = 0.25$ yields an average RPD of 0.52%, while the other GRASP algorithms yields an average RPD greater than 40%. As stated earlier, these large differences among the two approaches may be because the second approach (GRASP 3) generates solutions closer to feasibility and the repairing mechanism modifies less the initial solution. As in the small instances group, an ANOVA is applied in order to validate if the differences are statistically significant with the same response variable and factors. Figure 4(a) shows the means plot with LSD intervals at the 95% confidence level for large instances and factor ALGORITHM. We confirm that the algorithm following the second approach (GRASP 3) yields significantly lower RPD than the algorithms following the first approach. As in small instances, in the interaction plot in Figure 4(b) we observe that GRASP 3 performs better with lower α. Nevertheless, as opposed to the small instances, GRASP 1 performs better with lower α.

		First Approach						Second Approach		
		GRASP 1			GRASP 2			GRASP 3		
Size	$t(s)$	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.75$	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.75$	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.75$
50x10	10	16.81	21.67	32.81	21.15	21.14	19.19	2.87	8.78	19.81
50x15	10	37.09	44.62	50.67	32.22	36.33	34.03	1.33	9.65	25.15
50x20	10	37.12	47.70	65.06	36.33	40.71	39.99	0.64	14.34	28.14
50x25	10	43.06	56.78	78.76	34.55	33.88	36.85	0.00	18.66	34.77
50x30	10	32.54	57.67	88.57	22.30	28.58	19.71	3.57	17.89	40.12
100x10	20	31.10	28.46	42.27	25.50	27.23	31.10	0.55	9.47	25.10
100x15	20	45.36	45.29	65.65	38.23	40.61	45.36	0.21	11.18	26.71
100x20	20	52.56	51.37	76.33	50.75	48.52	52.56	0.00	13.88	29.33
100x25	20	54.66	56.08	86.27	53.03	54.03	54.66	0.18	13.53	30.69
100x30	20	57.46	55.90	90.54	50.56	48.12	57.46	0.72	13.74	34.33
150x10	30	19.85	26.15	32.20	29.06	27.26	30.31	0.23	7.21	18.99
150x15	30	34.90	42.65	49.82	44.09	43.83	45.04	0.34	8.52	21.87
150x20	30	41.71	48.01	57.41	51.76	51.74	57.31	0.69	9.18	24.49
150x25	30	42.26	51.79	62.05	55.45	51.44	59.72	0.31	12.59	27.20
150x30	30	40.46	50.12	62.34	57.66	53.36	55.72	0.07	11.09	27.03
200x10	40	20.29	27.01	33.98	26.67	26.22	28.80	0.35	8.03	22.11
200x15	40	35.46	42.55	55.23	39.99	41.22	44.24	0.17	8.96	27.18
200x20	40	37.34	44.51	56.87	44.89	44.70	46.88	0.05	9.37	28.73
200x25	40	41.78	50.75	63.60	50.28	48.61	51.46	0.06	11.11	32.67
200x30	40	46.51	57.51	74.97	56.40	58.54	58.47	0.17	11.71	37.18
250x10	50	22.32	27.67	38.89	28.10	28.44	30.20	0.03	7.89	24.13
250x15	50	35.97	43.54	55.64	39.79	39.98	43.21	0.11	9.32	31.28
250x20	50	39.93	51.05	66.01	48.06	48.69	50.41	0.11	11.20	38.97
250x25	50	42.85	51.80	70.78	50.88	51.93	52.52	0.08	12.32	42.76
250x30	50	43.88	54.82	77.56	55.84	55.88	59.00	0.21	13.38	51.41
Av. RPD		38.13	45.42	61.37	41.74	42.04	44.17	0.52	11.32	30.01

Table 6: Average Relative Percentage Deviation (RPD) for GRASP algorithms in large instances.



(a) Means Plot and LSD intervals. (b) Interaction Plot.

Figure 4: ANOVA in large instances.

6.7. Effect of local search

In order to verify that the local search phase contributes to the GRASP algorithms proposed, a sample of 100 large instances has been solved with each GRASP algorithm without the local search. More precisely, we select one instance of each possible configuration in the large set. For each such instance, we run each GRASP algorithm with the local search, and without the local search. In both cases, the maximum time allowed is as explained in Table 6. Table 7 shows the percentage difference between the solutions obtained by the algorithms with local search and the algorithms without local search. This difference is calculated for each GRASP algorithm and for each α value. We observe that, since all values are positive, the algorithms with local search find better solutions. Moreover, to verify if there are significant differences between the solutions, an ANOVA was implemented, obtaining statistically significant differences.

	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.75$
GRASP 1	4.70	4.85	3.21
GRASP 2	6.56	5.17	5.10
GRASP 3	6.88	3.68	2.38

Table 7: Differences (in %) between solutions with local search and solutions without local search.

7. Conclusions and future work

In this paper, we reduce the gap between academic research and real scheduling in parallel machine problems. We have designed efficient smart tools to solve the unrelated parallel machine scheduling problem with setup times and additional resources in the setups (UPMSR-S) with makespan minimization. Therefore, we have proposed a mathematical model and three metaheuristics for the UPMSR-S. The first two metaheuristics ignore the information about the resource constraints during the constructive phase (first approach), and then, the solution obtained is evaluated and repaired (if the resource constraints are not satisfied) with a repairing mechanism. The third metaheuristic takes into account the information about the resource constraints during the constructive phase (second approach) and, as with the first approach, the solution obtained is evaluated and repaired if necessary, with the same repairing mechanism. A local search algorithm consisting of three swap and insertion operations is also proposed to try to improve the initial solution. An exhaustive comparative evaluation between the proposed algorithms is carried out under an extensive benchmark of small and large instances. After a deep analysis, we conclude that there are no statistically significant difference between the best metaheuristic of the first approach (GRASP 2) and the metaheuristic of the second approach (GRASP 3) in small instances. In large instances, the differences between the two approaches are larger and the second approach metaheuristic is much better than the other metaheuristics. This confirms that algorithms in which the resource constraints are considered in the constructive phase, are expected to yield better results. Besides, we also proved empirically that the local search phase significantly contributes to all GARSP algorithms proposed.

We have empirically proved that, if the scarce resources are really bounding (which happens in our large size instances), including knowledge of the problem in the constructive phase significantly improves the algorithm. However, in instances in which the resources are not as limiting (which happens in our small size instances), including information about the resources in the constructive phase does not significantly improve the algorithm's performance. Then, the main strengths of our method rely on its capability for finding good solutions, with short CPU time, to large instances of the proposed problem, when the scarce resources are really binding. Note that, this type of instances is more common in manufacturing environments.

Future research on this topic will focus on different lines. First of all, we want to address the problem from a bi-objective perspective, in which both the makespan and the maximum number of resources are minimized simultaneously. Secondly, another future line is the stochastic version of the problem here introduced. In particular, setup times and processing times could be considered as non deterministic parameters, to provide a more realistic approach for instances in which high variability appears in one or both of these family of parameters. Therefore, we believe that simheuristic algorithms are a good strategy in this complex problem (see Juan et al. (2014)). Thirdly, a game theory analysis would be useful when considering situations in which the different resources are owned by different agents, with conflicting objectives. Lastly, it would also be interesting to extend this research to other scheduling problems such as the flowshop.

Acknowledgments

The first three authors would like to acknowledge the support from Spanish "Ministerio de Economia y competitividad" throughout grant number MTM2016-74983 and grant "SCHEYARD – Optimization of Scheduling Problems in Container Yards" (No. DPI2015-65895-R) financed by FEDER funds and the *Universitat Politècnica de València* under grant SP20180164 of the program *Primeros Proyectos de Investigación (PAID-06-18), Vicerrectorado de Investigación, Innovación y Transferencia*. Juan C. Yepes-Borrero acknowledges financial support by the *El Instituto Colombiano de Crédito Educativo y Estudios Técnicos en el Exterior – ICETEX* under program *Pasaporte a la ciencia – Doctorado*. Special thanks are due to two anonymous referees for their valuable comments.

Afzalirad, M. and Rezaeian, J. (2016). Resource-constrained unrelated parallel machine scheduling problem with sequence dependent setup times, precedence constraints and machine eligibility restrictions. *Computers & Industrial Engineering*, 98:40 – 52.

Aiex, R., Binato, S., and Resende, M. (2003). Parallel GRASP with path-relinking for job shop scheduling. *Parallel Computing*, 29(4):393 – 430.

Allahverdi, A. (2015). The third comprehensive survey on scheduling problems with setup times/costs. *European Journal of Operational Research*, 246(2):345 – 378.

Alvarez-Valdes, R., Crespo, E., Tamarit, J., and Villa, F. (2008). GRASP and path relinking for project scheduling under partially renewable resources. *European Journal of Operational Research*, 189(3):1153 – 1170.

Anticona, M. T. (2006). A GRASP algorithm to solve the problem of dependent tasks scheduling in different machines. In Bramer, M., editor, *Artificial Intelligence in Theory and Practice*, pages 325–334, Boston, MA. Springer US.

Arbaoui, T. and Yalaoui, F. (2018). Solving the unrelated parallel machine scheduling problem with additional resources using constraint programming. In Nguyen, N. T., Hoang, D. H., Hong, T.-P., Pham, H., and Trawiński, B., editors, *Intelligent Information and Database Systems*, pages 716–725, Cham. Springer International Publishing.

Arnaout, J.-P., Rabadi, G., and Musa, R. (2010). A two-stage ant colony optimization algorithm to minimize the makespan on unrelated parallel machines with sequence-dependent setup times. *Journal of Intelligent Manufacturing*, 21(6):693–701.

Arroyoa, J. E. C. and Leung, J. Y.-T. (2017). Scheduling unrelated parallel batch processing machines with non-identical job sizes and unequal ready times. *Computers & Operations Research*, 78:117–128.

Avalos-Rosales, O., Angel-Bello, F., and Alvarez, A. (2015). Efficient metaheuristic algorithm and re-formulations for the unrelated parallel machine scheduling problem with sequence and machine-dependent setup times. *International Journal of Advanced Manufacturing Technology*, 76(9-12):1705–1718.

Binato, S., Hery, W. J., Loewenstern, D. M., and Resende, M. G. C. (2002). *A Grasp for Job Shop Scheduling*, pages 59–79. Springer US, Boston, MA.

Bitar, A., Dauzère-Pérès, S., Yugma, C., and Roussel, R. (2016). A memetic algorithm to solve an unrelated parallel machine scheduling problem with auxiliary resources in semiconductor manufacturing. *Journal of Scheduling*, 19(4):367–376.

Chen, J.-F. (2004). Unrelated parallel machine scheduling with secondary resource constraints. *International Journal of Advanced Manufacturing Technology*, 26:285–292.

De-Paula, M. R., Ravetti, M. G., Mateus, G. R., and Pardalos, P. M. (2007). Solving parallel machines scheduling problems with sequence-dependent setup times using variable neighbourhood search. *IMA Journal of Management Mathematics*, 18:101–115.

Diana, R. O. M., F., d. M., de Souza, S. R., and de Almeida Vitor, J. F. (2014). An immune-inspired algorithm for an unrelated parallel machines' scheduling problem with sequence and machine dependent setup-times for makespan minimisation. *Neurocomputing*, 163:94–105.

Edis, E. B. and Oguz, C. (2012). Parallel machine scheduling with flexible resources. *Computers & Industrial Engineering*, 63(2):433 – 447.

Edis, E. B., Oguz, C., and Ozkarahan, I. (2013). Parallel machine scheduling with additional resources: Notation, classification, models and solution methods. *European Journal of Operational Research*, 230(3):449 – 463.

Edis, E. B. and Ozkarahan, I. (2012). Solution approaches for a real-life resource-constrained parallel machine scheduling problem. *The International Journal of Advanced Manufacturing Technology*, 58(9):1141–1153.

Expósito-Izquierdo, C., Angel-Bello, F., Melián-Batista, B., Alvarez, A., and Báez, S. (2019). A metaheuristic algorithm and simulation to study the effect of learning or tiredness on sequence-dependent setup times in a parallel machine scheduling problem. *Expert Systems with Applications*, 117:62 – 74.

Fanjul-Peyro, L., Perea, F., and Ruiz, R. (2017). Models and matheuristics for the unrelated parallel machine scheduling problem with additional resources. *European Journal of Operational Research*, 260(2):482–493.

Fanjul-Peyro, L. and Ruiz, R. (2010). Iterated greedy local search methods for unrelated parallel machine scheduling. *European Journal of Operational Research*, 207:55–69.

Fanjul-Peyro, L. and Ruiz, R. (2011). Size-reduction heuristics for the unrelated parallel machines scheduling problem. *Computers & Operations Research*, 38:301–309.

Fanjul-Peyro, L., Ruiz, R., and Perea, F. (2019). Reformulations and an exact algorithm for unrelated parallel machine scheduling problems with setup times. *Computers & Operations Research*, 101:173–182.

Feo, T. A. and Resende, M. G. (1989). A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67 – 71.

Feo, T. A., Sarathy, K., and McGahan, J. (1996). A grasp for single machine scheduling with sequence dependent setup costs and linear delay penalties. *Computers & Operations Research*, 23(9):881 – 895.

Feo, T. A., Venkatraman, K., and Bard, J. F. (1991). A GRASP for a difficult single machine scheduling problem. *Computers & Operations Research*, 18(8):635 – 643.

Fleszar, K. and Hindi, K. S. (2018). Algorithms for the unrelated parallel machine scheduling problem with a resource constraint. *European Journal of Operational Research*, 271(3):839 – 848.

Helal, M., Rabadi, G., and Al-Salem, A. (2006). A tabu search algorithm to minimize the makespan for the unrelated parallel machines scheduling problem with setup times. *International Journal of Operations Research*, 3(3):182–192.

Juan, A. A., Barrios, B. B., Vallada, E., Riera, D., and Jorba, J. (2014). A simheuristic algorithm for solving the permutation flow shop problem with stochastic processing times. *Simulation Modelling Practice and Theory*, 46:101 – 117. Simulation-Optimization of Complex Systems: Methods and Applications.

Kim, D.-W., Kim, K.-H., Jang, W., and Chen, F. F. (2002). Unrelated parallel machine scheduling with setup times using simulated annealing. *Robotics and Computer-Integrated Manufacturing*, 18:223–231.

Kurz, M. E. and Askin, R. G. (2001). Heuristic scheduling of parallel machines with sequence-dependent set-up times. *International Journal of Production Research*, 39:3747–3769.

Laguna, M. and Velarde, J. L. G. (1991). A search heuristic for just-in-time scheduling in parallel machines. *Journal of Intelligent Manufacturing*, 2(4):253–260.

Montgomery, D. C. (2012). *Design and Analysis of Experiments*. John Wiley & Sons, New York, eigth edition.

Parreño, F., Alvarez-Valdes, R., Oliveira, J. F., and Tamarit, J. M. (2010). A hybrid GRASP/VND algorithm for two- and three-dimensional bin packing. *Annals of Operations Research*, 179(1):203–220.

Rabadi, G., Moraga, R. J., and Al-Salem, A. (2006). Heuristics for the unrelated parallel machine scheduling problem with setup times. *Journal of Intelligent Manufacturing*, 17(1):85–97.

Rajkumar, M., Asokan, P., Anilkumar, N., and Page, T. (2011). A GRASP algorithm for flexible job-shop scheduling problem with limited resource constraints. *International Journal of Production Research*, 49(8):2409–2423.

Resende, M. G. C. and Ribeiro, C. C. (2014). *GRASP: Greedy Randomized Adaptive Search Procedures*, pages 287–312. Springer US, Boston, MA.

Ruiz, R. and Andrés-Romano, C. (2011). Scheduling unrelated parallel machines with resource-assignable sequence-dependent setup times. *The International Journal of Advanced Manufacturing Technology*, 57(5):777–794.

Ruiz-Torres, A. J., López, F. J., and Ho, J. C. (2007). Scheduling uniform parallel machines subject to a secondary resource to minimize the number of tardy jobs. *European Journal of Operational Research*, 179(2):302 – 315.

Vallada, E. and Ruiz, R. (2011). A genetic algorithm for the unrelated parallel machine scheduling problem with sequence dependent setup times. *European Journal of Operational Research*, 211:612–622.

Villa, F., Vallada, E., and Fanjul-Peyro, L. (2018). Heuristic algorithms for the unrelated parallel machine scheduling problem with one scarce additional resource. *Expert Systems with Applications*, 93:28 – 38.