Author's Accepted Manuscript

Scatter search with path relinking for the job shop with time lags and setup times

Miguel A. González, Angelo Oddi, Riccardo Rasconi, Ramiro Varela

www.elsevier.com/locate/caor

PII: S0305-0548(15)00036-2
DOI: http://dx.doi.org/10.1016/j.cor.2015.02.005
Reference: CAOR3726

To appear in: *Computers & Operations Research*

Cite this article as: Miguel A. González, Angelo Oddi, Riccardo Rasconi, Ramiro Varela, Scatter search with path relinking for the job shop with time lags and setup times, *Computers & Operations Research,* http://dx.doi.org/10.1016/j.cor.2015.02.005

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting galley proof before it is published in its final citable form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

ACCEPTED MANUSCRIPT

Scatter Search with Path Relinking for the Job Shop with Time Lags and Setup Times

Miguel A. González[a,*], Angelo Oddi[b], Riccardo Rasconi[b], Ramiro Varela[a]

[a] *Department of Computing, University of Oviedo, Campus de Gijón, 33204, Gijón (Spain)*
[b] *Institute of Cognitive Sciences and Technologies, ISTC-CNR, Via San Martino della Battaglia 44, 00185, Rome (Italy)*

Abstract

This paper addresses the job shop scheduling problem with time lags and sequence-dependent setup times. This is an extension of the job shop scheduling problem with many applications in real production environments. We propose a scatter search algorithm which uses path relinking and tabu search in its core. We consider both feasible and unfeasible schedules in the execution, and we propose effective neighborhood structures with the objectives of reducing the makespan and regain feasibility. We also define new procedures for estimating the quality of the neighbors. We conducted an experimental study to compare the proposed algorithm with the state-of-the-art, in benchmarks both with and without setups. In this study, our algorithm has obtained very competitive results in a reduced run time.

Keywords: Job Shop, Setup Times, Time Lags, Scatter Search, Tabu Search, Path Relinking

1. Introduction

The Job Shop Scheduling Problem (JSP) is a simple model of many real production processes. It is one of the most classical and difficult scheduling problems and it has been studied for several decades, from the seminal work of Giffler and Thompson [1] to the most recent papers, for example [2].

However, in many environments the production model has to consider additional characteristics or complex constraints. For example, in automobile, printing, semiconductor, chemical or pharmaceutical industries, setup operations such as cleaning up or changing tools are required between two consecutive jobs on the same machine. These setup operations depend on both the outgoing and incoming jobs, so they cannot be considered as being part of any of these jobs. Additionally, time lag constraints arise in many real-life scheduling applications. For example, in the steel industry, the time lag between the heating of a piece of steel and its moulding should be small [3]. Similarly when scheduling chemical reactions, the reactive usually cannot be stored for a long time between two stages of a process to avoid interactions with external elements [4]. When these two factors are considered simultaneously, the problem is known as the job shop scheduling problem with time lags and sequence-dependent setup times (SDST-JSPTL).

The classical JSP with makespan minimization has been intensely studied, and many results have been established that have given rise to very efficient algorithms, as for example the $i-$TSAB algorithm proposed by Nowicki and Smutnicki in [5].

Incorporating sequence-dependent setup times changes the nature of scheduling problems, and the well-known results and techniques for the JSP are not directly applicable anymore. The job shop with setups (SDST-JSP) is studied in [6], where Brucker and Thiele developed a branch and bound algorithm. Cheung and Zhou in [7] use a genetic algorithm combined with two dispatching rules. Also, in [8] Balas et al. propose a shifting bottleneck heuristic. One of the most efficient proposals is the branch and bound algorithm proposed by Artigues and Feillet in [9]. More recently, in [10] and [11] Vela et al. and Gonzalez et al. design two hybrid approaches that combine a genetic algorithm with local search procedures, taking some ideas proposed by Van Laarhoven et al. in [12] as a basis for new neighborhood structures. The maximum lateness minimization was also considered by some researchers [13] [14].

The addition of time lag constraints between successive job operations makes difficult even the usually simple task of finding a feasible schedule. The job shop with time lags is a very challenging problem for local search metaheuristics, because the classical neighborhood structures for the standard job shop lead to unfeasible schedules most of the time. Very few articles tackle time-lag constraints. For example, Wikum et al. in [15] study problems with a single machine considering minimum and maximum distances between jobs and prove that these problems are NP-hard, although some particular cases are polynomially solvable.

*Corresponding author. Phone: +34 985182493
Email addresses: mig@uniovi.es (Miguel A. González), angelo.oddi@istc.cnr.it (Angelo Oddi), riccardo.rasconi@istc.cnr.it (Riccardo Rasconi), ramiro@uniovi.es (Ramiro Varela)

Accepted manuscript

ACCEPTED MANUSCRIPT

Brucker et al. in [16] show many examples of scheduling problems that can be modelled as single-machine problems with time-lags, like multi-processor tasks, multi-purpose machines or problems with changeover costs. They also propose a branch-and-bound method to solve the problem. Hurink and Keuchel, in [17], propose a local search approach for the single machine problem. Fondrevelle et al. in [18] and Dhouib et al. in [19] solve permutation flow-shop scheduling problems with maximal and minimal time lags. Botta-Genoulaz [20] tackles the maximum lateness minimization in the hybrid flow shop scheduling with precedence constraints and time lags. Another example is the thesis of Zhang [21], where several variants of online and offline problems with time-lags are studied.

The job shop with minimum and maximum time lags (JSPTL) was tackled by Caumond et al. [22], where they propose a list scheduling heuristic for generating initial solutions and a memetic algorithm based on a disjunctive graph model. Artigues et al. [23] study the JSPTL as well. They propose a job insertion heuristic for generating initial solutions and generalized resource constraint propagation mechanisms, which are embedded in a branch-and bound algorithm. Recently, Grimes and Hebrard [24] propose a constraint programming approach that uses a number of generic SAT and AI techniques such as weighted degree variable ordering, solution guided value ordering, geometric restarting and nogood recording from restarts. Their approach was adapted to solve several variants of the job shop scheduling problem, including the JSPTL. In [25] and[26], the authors propose constraint propagation methods for the job shop with generic time-lags, which is a generalized version of the JSPTL. Additionally, since the JSPTL is a particular case of the resource-constrained project scheduling problem with time lags (RCPSP/max), literature on this problem is also worth mentioning, for example [27] [28] [29] and [30].

We have already seen that there are several papers dealing with each of the two factors separately (setups and time lags). However, very few papers have considered both minimum and maximum time lags and sequence-dependent setup times at the same time (SDST-JSPTL). As far as we know, the only approach is that reported in [31], where Oddi et al. solve this problem by means of a constraint-based iterative sampling procedure.

Hybrid metaheuristics usually perform very well at solving combinatorial optimization problems, since they allow algorithm designers to combine the advantages of different search techniques. In particular, they have a long track of success with scheduling problems. Even for the classical JSP researchers continue to propose hybrid metaheuristics that outperform previous ones. For example, in [32] an Hybrid Genetic Tabu Search for solving the JSP is proposed. Also, the algorithms proposed in [33] and in [34] are probably the most efficient approaches to the JSP with makespan minimization. These algorithms combine the $i-$TSAB algorithm proposed in [5] with a simulated annealing in the first case and with a solution-guided search method in the second case.

In this paper we propose to solve the SDST-JSPTL with a scatter search algorithm which uses path relinking and tabu search in its core. This hybrid metaheuristic can obtain the best results in some scheduling problems, e.g. [35]. We propose a novel neighborhood structure, establish feasibility and non improving conditions, and give an algorithm for fast estimation of neighbors' quality. We consider both feasible and unfeasible solutions through the search and the neighborhood structure defines a different subset of moves depending on whether the current solution is feasible or unfeasible. We also extend the heuristic proposed in [23] for generating initial solutions and the longest path algorithm defined in [22]. We conducted an experimental study in which we first analyzed the proposed algorithm, and then we compared it with the state-of-the-art in both the SDST-JSPTL and the standard JSPTL, showing that our results are very competitive.

The remainder of the paper is organized as follows. In Section 2 we formulate the problem and describe the solution graph model. In Section 3 we describe the heuristic used to generate the initial schedules. Section 4 details the algorithm used to calculate a schedule from a processing order. In Section 5 we define the proposed neighborhoods. Section 6 details the metaheuristics used. In Section 7 we report the results of the experimental study, and finally Section 8 summarizes the main conclusions of this paper.

2. The Job Shop Scheduling Problem with Time Lags and Setup Times

In this section we introduce the SDST-JSPTL. Firstly, we give a formal definition of the problem as an extension of the classic JSP and then we propose a disjunctive model that allow us to represent problem instances and schedules, as well as to formalize the neighborhood structures.

2.1. Problem formulation

In the SDST-JSPTL, we are given a set of n jobs, $J = \{J_1, \ldots, J_n\}$ that must be processed on a set of m machines or resources, $M = \{M_1, \ldots, M_m\}$, subject to a set of constraints. There are *precedence constraints*, so each job J_i, $i = 1, \ldots, n$, consists of N_i operations $O_i = \{o_{i1}, \ldots, o_{iN_i}\}$ to be sequentially scheduled. Also, there are *capacity constraints*, upon which an operation o_{ij} requires the uninterrupted and exclusive use of given machine $m_{ij} \in M$ during p_{ij} time units.

We also consider the addition of sequence-dependent setup times which depend on both the outgoing and incoming operations. After an operation o_{ij} leaves the machine and before an operation o_{kl} enters the same machine, a setup operation is required to adjust the machine, with duration $s_{o_{ij}o_{kl}}$. We also define an initial setup time $s_{0o_{ij}}$ required when o_{ij} is the first operation scheduled on its machine. We consider that the setup times verify the triangle inequality, i.e., $s_{uv} + s_{vw} \geq s_{uw}$ holds for any operations

ACCEPTED MANUSCRIPT

u, v and w requiring the same machine. This condition is usually assumed in the literature as it usually happens in real scenarios.

Finally, we consider the addition of minimum and maximum time lags between consecutive operations of the same job. If we have the operation o_{ij} (which is not the last operation of its job), we denote by $TL_{min}(o_{ij})$ and $TL_{max}(o_{ij})$ the minimum and maximum difference between the ending time of o_{ij} and the starting time of o_{ij+1}.

A solution to this problem consists of an allocation of starting times for each operation in the set $O = \cup_{1 \leq i \leq n} O_i$ which is *feasible* (i.e. all constraints hold). The objective is to find an *optimal* solution according to some criterion, most commonly that the *makespan*, which is the completion time of the last operation to finish, is minimal.

2.2. Disjunctive graph model

We propose a disjunctive model for the SDST-JSPTL which extends the model proposed in [22] to cope with setup times. In accordance with this model, we have a directed graph $G = (V, A \cup E \cup I_1 \cup I_2 \cup M)$ where

- Each node in set V represents an operation of the problem, with the exception of the dummy nodes *start* and *end*, which represent fictitious operations with processing time 0 that do not require any machine.
- A is the set of *conjunctive arcs* and represents precedence relations. It contains arcs of the form (v, w), where v is the predecessor of w in its job sequence, weighted with $p_v + TL_{min}(v)$.
- Arcs in E are called *disjunctive arcs* and represent capacity constraints. E is partitioned into subsets E_i, with $E = \cup_{i=1,\ldots,m} E_i$, where E_i corresponds to machine M_i and includes two directed arcs (v, w) and (w, v) for each pair v, w of operations requiring that resource. Each arc (v, w) of E is weighted with the processing time of the operation in the source node plus the corresponding setup time, $p_v + s_{vw}$.
- Set I_1 includes arcs of the form $(start, v)$ for each operation of the problem, weighted with s_{0v}.
- Set I_2 contains arcs (v, end), where v is the last operation of a job, weighted with p_v.
- Finally, set M includes arcs of the form (w, v), for all operation w not being the first operation in a job sequence, where v is the predecessor of w in its job sequence, weighted with $-(p_v + TL_{max}(v))$.

A schedule S is represented by a subgraph of G, $G_S = (V, A \cup H \cup K_1 \cup I_2 \cup M)$, where

- $H = \cup_{i=1\ldots m} H_i$, H_i being a minimal subset of arcs of E_i defining a processing order for all operations requiring M_i.
- K_1 consists of arcs $(start, v_i), i = 1, \ldots, m$, v_i being the first operation of H_i.

G_S must not have positive length cycles for the schedule to be feasible, but zero or negative length cycles are admitted. Finding a solution can then be reduced to discovering compatible orderings H_i, or partial schedules, that translate into a solution graph G_S without positive length cycles [22].

Figure 1 shows a feasible schedule for a problem with 3 jobs and 3 machines; dotted arcs belong to H and K_1, while continuous arcs belong to A, M and I_2. In the figure we also show the Gantt chart of the schedule, which is a visual representation that indicates the starting and ending times for each operation of the instance. On the other hand, Figure 2 shows an example of a solution graph with a positive length cycle, hence representing an unfeasible schedule.

Given a feasible schedule S, we define the concepts of critical path, node, arc and block as follows.

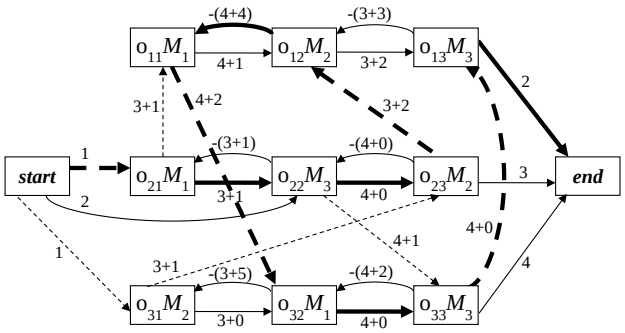
Definition 1. *A critical path is a directed path from node start to node end in G_S having maximum cost. Nodes and arcs in a critical path are also termed critical. A critical path is a sequence of the form $start, B_1, \ldots, B_r, end$, where each $B_k, 1 \leq k \leq r$, is a maximal subsequence of consecutive operations requiring the same machine which is termed critical block.*

The makespan of a schedule S is the cost of a critical path in G_S. In Figure 1, bold-face arcs represent a critical path whose cost is 22. Notice that critical paths in the SDST-JSPTL may contain negative cost arcs, unlike what happens in the classical JSP. The concepts of critical path and critical block are of major importance for job scheduling problems due to the fact that most formal properties and solution methods rely on them. Some of these properties have given rise to a number of neighborhood structures for the classic JSP, which are based on exchanging the order of operations of a given critical block [12], [36], [37], [38], [39].

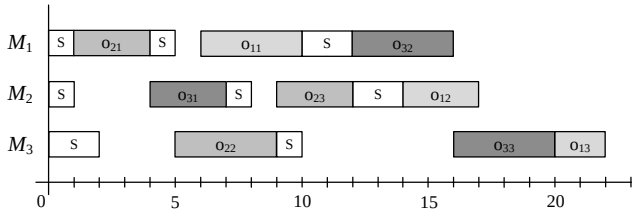
We define the following notation. PJ_v and SJ_v denote the operations just before and after operation v in the job sequence, and PM_v and SM_v denote the operations just before and after v in its machine sequence. To formalize the description of the neighborhood structures, we introduce the concepts of head and tail of an operation v, denoted r_v and q_v respectively. The head of an operation v is the longest path from node *start* to v (i.e. the starting time of v in the schedule represented by G_S), and the tail of an operation v is the longest path from v to node *end*. Hence, $r_v + q_v$ is the length of the longest path from node *start* to node *end* through node v, and therefore it is a lower bound of the makespan. Moreover, $r_v + q_v$ is the makespan if node v belongs to a critical path. Heads and tails are calculated by means of equations (1) and (2).

Accepted manuscript

ACCEPTED MANUSCRIPT



(a) Solution graph. The sequence or arcs in bold is a critical path with length 22.



(b) Gantt chart. S denotes setup operations. The makespan is 22.

Figure 1: A feasible schedule to a problem with 3 jobs and 3 machines.

$$
\begin{aligned}
r_{start} =& q_{end} = 0 \\
r_v =& \max(r_{PJ_v} + p_{PJ_v} + TL_{min}(PJ_v), \\
& r_{PM_v} + p_{PM_v} + s_{PM_v v}, \\
& r_{SJ_v} - p_v - TL_{max}(v)) \\
r_{end} =& \max_{v \in PJ_{end}} \{r_v + p_v\}
\end{aligned}
\tag{1}
$$

$$
\begin{aligned}
q_v =& \max(q_{SJ_v} + p_v + TL_{min}(v), \\
& q_{SM_v} + p_v + s_{vSM_v}, \\
& q_{PJ_v} - p_{PJ_v} - TL_{max}(PJ_v)) \\
q_{start} =& \max_{v \in SM_{start}} \{q_v + s_{0v}\}
\end{aligned}
\tag{2}
$$

Abusing notation, SM_{start} denotes the set consisting of the first operation processed in each of the m machines and PJ_{end} denotes the set consisting of the last operation processed in each of the n jobs. The completion time of an operation v is denoted by c_v and can be calculated as $c_v = r_v + p_v$. The objective is to find a solution S that minimizes the makespan, i.e., the completion time of the last operation, denoted as $C_{max}(S) = \max_{v \in PJ_{end}} c_v$, which is also r_{end}. Note that r_v and q_v could not be calculated if the solution graph had some positive length cycle involving operation v.

Finally, in order to deal with unfeasible schedules, let us define an additional type of schedule.

Definition 2. *A relaxed schedule S is a schedule represented by a subgraph of the form $G_S = (V, A \cup H \cup K_1 \cup I_2 \cup M')$, where $M' \subset M$, such that G_S does not have positive length cycles.*

So, a relaxed schedule may not fulfil a number of maximum time lag constraints, those of the form $r_v + p_v + TL_{max}(v) \leq r_w$, for $w \in M \setminus M'$. For example, if the negative arc (o_{33}, o_{32}) were removed from the graph of Figure 2, then the resulting graph would represent a relaxed schedule.

3. Calculating feasible solutions to SDST-JSPTL

For the classical JSP, approximate solutions can be obtained in a fast way by greedy heuristics based on priority rules. These heuristics usually build a solution by scheduling one operation at a time. In each step, they select one out of a set of candidate operations, which are ordered by means of a priority rule, to be scheduled next. In this process, an insertion point has to be found for the selected operation such that this operation can be scheduled to obtain a new feasible partial schedule. With the consideration of maximum time lags, classical JSP heuristics cannot be easily extended, and finding a non-trivial feasible schedule for the SDST-JSPTL is not simple. In particular, with the aforementioned method, it often happens that at some step finding an insertion point that does not violate the maximum time lag constraints is not possible.

The JSPTL is a particular case of the general RCPSP with time-lags. For this last problem, answering the question whether there exists a feasible solution is itself an NP-complete problem [27]. However, for the JSPTL and SDST-JSPTL only time lags between successive operations of the same job are considered, and a trivial solution called *canonical schedule* can be obtained by the following greedy algorithm [22]. From an arbitrary order of jobs, the operations in each job are scheduled from the first to the last so that each operation is assigned the earliest starting time at the end of the partial schedule built so far. So a number of $n!$ canonical schedules can be built with makespan equal to the sum of processing times for all operations.

As it may be expected, canonical schedules have very poor performance. For this reason, Caumond et al. in [22] have designed a list-based heuristic combined with a method for repairing partial solutions.

Afterwards, Artigues et al. in [23] proposed a job insertion heuristic, based on the idea that only time lag constraints between consecutive operations of the same job exist. The jobs are considered in some order, which can be random or calculated with some heuristic rule (see Section 6.1 for some examples). At each step, the operations of the current job are all inserted in the partial schedule. Once a job has been scheduled, it is considered as fixed and the start times of its operations cannot be modified in the next steps. The heuristic tries to schedule the operations

Accepted manuscript

ACCEPTED MANUSCRIPT

in a job using the idle-time intervals (insertion positions) of the machines.

When no insertion position exists for the current operation, it is necessary to do backtracking to reschedule some of the previous operations of the current job. In the worst case, a canonical schedule is generated. In the experimental study done in [23], this heuristic is shown to produce better solutions than the list-based heuristic from [22], especially in instances with tight maximum time lag constraints.

From all the above, we adopted the heuristic from [23] to generate the initial solutions, and we extended it to deal with sequence-dependent setup times. In our case, to decide if the insertion of an operation in an interval is feasible we have to take into account the setup times with respect to the machine predecessor and successor of the operation, and therefore we need to add setup times in the appropriate places of the algorithm (conditions (18) to (22) of the algorithm described [23]).

4. Encoding solutions and decoding algorithm

Two of the key points in designing population based metaheuristics are finding a proper solution encoding and then devising an efficient decoding algorithm to obtain feasible solutions. As we have seen in the previous section, obtaining even a simple feasible schedule to the SDST-JSPTL is not easy, so devising a suitable coding schema and a decoder to produce feasible schedules is even more difficult. For these reasons, we opted to dealing with unfeasible schedules in our algorithms. Bearing this in mind, we choose permutations with repetition as encoding schema and use a decoding algorithm that may build infeasible schedules.

4.1. Coding schema

We represent a schedule by a vector that contains a topological order of the graph, disregarding negative cost arcs. The topological order is represented by means of a permutation with repetition [40]. This is a permutation of the set of operations, each being represented by its job number. For example, for a problem with 3 jobs: $J_1 = \{o_{11}, o_{12}\}, J_2 = \{o_{21}, o_{22}, o_{23}, o_{24}\}, J_3 = \{o_{31}, o_{32}, o_{33}\}$, the sequence (2 1 2 3 2 3 3 2 1) is a valid vector that represents the topological order $\{o_{21}, o_{11}, o_{22}, o_{31}, o_{23}, o_{32}, o_{33}, o_{24}, o_{12}\}$. With this representation, canonical schedules are represented by permutations where all operations of the same job appear consecutively.

4.2. Decoding algorithm

To build schedules from a vector of operations, we consider the procedure used by Caumond et al. in [22] and extend it to deal with setup times, to calculate tails (as they are needed for the neighbor estimation algorithm), and to record cycles (as this information is used by the proposed neighborhood structure). The resulting decoder is given in Algorithm 1. It starts from a schedule in which all maximum time lags are relaxed (Algorithm 2). Then, it iterates over the operations in the order given by the vector. For an operation v, it tries to check the constraint represented by the negative arc (SJ_v, v). So, if $r_v + p_v + TL_{max}(v) < r_{SJ_v}$ then the heads and tails of some operations have to be updated by constraint propagation (Algorithm 3). If no positive length cycle is created after adding this new constraint, the new heads and tails remain in the schedule; otherwise, the maximum time lag constraint is relaxed and added to the list of violated constraints, the cycle registered, and the heads and tails of the operations are not modified.

Therefore, the algorithm returns the heads and tails of every operation of the instance and a list with the relaxed maximum time lag constraints and the cycles generated by them, which will be of use in the neighborhood structure N_{NF} defined in Section 5. Notice that the set of relaxed constraints is not unique, but it depends on the insertion order of the maximum time lag constraints. In our proposal this order is given by the vector.

The algorithm also calculates the vectors P and Q such that $P(i)$ and $Q(i)$ represent the predecessor and successor of operation i respectively in the longest path from *start*

Algorithm 1 DecodingAlgorithm

Require: A SDST-JSPTL instance and a *vector* with a ordering of the N operations
  Uncheck all maximum time lags;
  Call EvaluateWithoutTimeLags;
  $Save\_r \leftarrow r$; $Save\_P \leftarrow P$;
  for each v in *vector* **do**
    check time lag of operation v; // fix (SJ_v, v)
    if $r_v + p_v + TL_{max}(v) < r_{SJ_v}$ **then**
      Call AddMaximumTimeLagConstraint(v);
      if there is a positive length cycle **then**
        $r \leftarrow Save\_r$; $P \leftarrow Save\_P$;
        uncheck time lag of v; // remove (SJ_v, v)
      else
        $Save\_r \leftarrow r$; $Save\_P \leftarrow P$;
      end if
    end if
  end for
  $Makespan \leftarrow \max\{r_{o_{iN_i}} + p_{iN_i}, 1 \leq i \leq n\}$;
  return The schedule and the list of unsatisfied (unchecked) time lag constraints if the schedule is relaxed;

This algorithm produces a schedule from a topological order vector. If the schedule is feasible, every negative arc (SJ_v, v) is included in the solution graph. If the schedule is unfeasible, it returns the list of constraints that are not satisfied. For each unsatisfied constraint, the corresponding negative arc is not included in the solution graph, therefore it is a relaxed schedule. Initially r is the set of heads and $P(i)$ is the predecessor of operation i respectively in the longest path from *start* to i in the schedule calculated in *EvaluateWithoutTimeLags*.

Accepted manuscript

ACCEPTED MANUSCRIPT

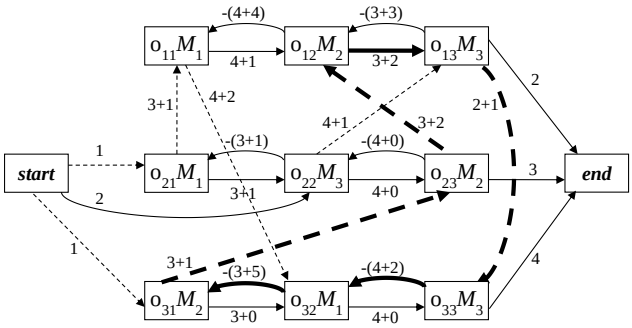
Algorithm 2 EvaluateWithoutTimeLags

Require: A SDST-JSPTL instance and a *vector* with a ordering of the N operations
// First we calculate the heads and the vector P
for $i \leftarrow 1$ to N **do**
  $v \leftarrow vector[i]$; $r_v \leftarrow 0$; $P(v) \leftarrow start$;
  if $PJ_v \neq start$ **then**
    $r_v \leftarrow r_{PJ_v} + p_{PJ_v} + TL_{min}(PJ_v)$;
    $P(v) \leftarrow PJ_v$;
  end if
  $PM_v \leftarrow$ last operation in *vector* before v requiring the same machine, *start* if there is no one;
  Fix the arc (PM_v, v) in the solution graph;
  if $PM_v = start$ and $s_{0v} > r_v$ **then**
    $r_v \leftarrow s_{0v}$;
    $P(v) \leftarrow start$;
  end if
  if $PM_v \neq start$ and $r_{PM_v} + p_{PM_v} + s_{PM_v v} > r_v$ **then**
    $r_v \leftarrow r_{PM_v} + p_{PM_v} + s_{PM_v v}$;
    $P(v) \leftarrow PM_v$;
  end if
end for
// Now we calculate the tails and the vector Q
for $i \leftarrow N$ to 1 **do**
  $v \leftarrow vector[i]$; $q_v \leftarrow p_v$; $Q(v) \leftarrow end$;
  if $SM_v \neq end$ **then**
    $q_v \leftarrow q_{SM_v} + p_v + s_{vSM_v}$;
    $Q(v) \leftarrow SM_v$;
  end if
  if $SJ_v \neq end$ and $q_{SJ_v} + p_v + TL_{min}(v) > q_v$ **then**
    $q_v \leftarrow q_{SJ_v} + p_v + TL_{min}(v)$;
    $Q(v) \leftarrow SJ_v$;
  end if
end for
return A schedule built without considering maximum time lag constraints, the heads and tails of every operation, and the vectors P and Q;

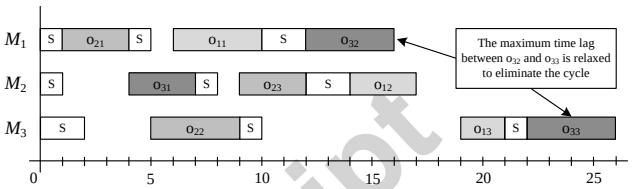
This algorithm builds a schedule so that the order of operations in each machine is the same as the order of these operations in the given vector. The maximum time lag constraints are not considered so the solution graph includes no negative cost arc, i.e., all these constraints are unchecked. It returns heads and tails for all operations in r and q, and the vectors P and Q such that $P(i)$ and $Q(i)$ represent the predecessor and successor of operation i respectively in the longest path from *start* to *end* through i in the solution graph.

to *end* through node i in the solution graph. P and Q are updated each time a head or a tail is modified, and they will be used for cycle detection (as shown in Algorithm 3) and for neighbors' estimation, as we will see in Section 5.4.

Figure 2 represents the schedule generated by the algorithm from the sequence (2 3 2 1 2 1 1 3 3), which corresponds to the topological ordering (o_{21}, o_{31}, o_{22}, o_{11}, o_{23}, o_{12}, o_{13}, o_{32}, o_{33}). Bold-face arcs show a positive length cycle that makes this schedule unfeasible. This cycle was created in the last iteration of the algorithm, when trying to add the maximum time lag constraint from o_{32} to o_{33}. Therefore, this constraint has to be relaxed and so the built schedule is relaxed.



(a) Solution graph. Bold-face arcs show a positive length cycle that makes this schedule unfeasible.



(b) Gantt chart. If the time lag constraint between o_{32} and o_{33} were relaxed, the schedule would be feasible.

Figure 2: An unfeasible schedule to a problem with 3 jobs and 3 machines built from the vector (2 3 2 1 2 1 1 3 3).

In Algorithm 3, w_{uv} represents the weight of the arc (u,v). We also define the set $IN(v)$ which includes every operation u such that the arc (u,v) exists in the graph. Analogously, the set $OUT(v)$ includes every operation u such that the arc (v,u) exists in the graph. It is important to remark that, if some maximum time lag constraint $TL_{max}(u)$ is not checked, then $SJ_u \notin IN(u)$ and $u \notin OUT(SJ_u)$.

Moreover, in Algorithm 3, a FIFO data structure is used to implement the following methods: *Empty_the_queue* removes everything from the queue, *EmptyQueue* returns *true* if the queue is empty and *false* otherwise. *Enqueue*(v) inserts operation v in the queue, and finally *Dequeue*() extracts an element from the queue. The advantage of using FIFO over LIFO structures in calculating longest paths is already known in the literature [41]. Experimentally, we have also obtained better results with a FIFO than with a LIFO, both in runtime and quality of results. We have found that the average length (measured in number of operations) of the positive length cycles that we encounter through the search is about 17% lower with the queue, and that has a positive effect in the neighborhood structure N_{NF} defined in Section 5.

Accepted manuscript

ACCEPTED MANUSCRIPT

Algorithm 3 AddMaximumTimeLagConstraint

Require: A SDST-JSPTL instance and a schedule with the maximum time lag constraint i, i.e., that due to the arc (SJ_i, i) that we just checked
// Firstly, updates heads and P and checks for cycles
$TestCycle \leftarrow false$; $Empty\_the\_queue$; $Enqueue(SJ_i)$;
while $(EmptyQueue = false)$ and $(TestCycle = false)$ **do**
  $v = Dequeue()$;
  for all u in the set $OUT(v)$ **do**
    if $r_v + w_{vu} > r_u$ **then**
      $r_u \leftarrow r_v + w_{vu}$; $Enqueue(u)$; $P(u) \leftarrow v$;
      if $u = SJ_i$ **then**
        $TestCycle \leftarrow true$;
        Annotate positive cycle from $P(SJ_i)$;
      end if
    end if
  end for
end while
// If no cycles, tails and Q are calculated
if $TestCycle = false$ **then**
  $Empty\_the\_queue$; $Enqueue(i)$;
  while $(EmptyQueue = false)$ **do**
    $v = Dequeue()$;
    for all u in the set $IN(v)$ **do**
      if $q_v + w_{uv} > q_u$ **then**
        $q_u \leftarrow q_v + w_{uv}$; $Enqueue(u)$; $Q(u) \leftarrow v$;
      end if
    end for
  end while
end if
return A modified schedule that may or may not satisfy the maximum time lag constraint i. If it is satisfied, heads, tails and vectors P and Q are updated accordingly. If not, it returns a positive length cycle involving the negative arc (SJ_i, i);

This algorithm starts from a schedule with some maximum time lag constraints unchecked and the constraint i just checked, and tries to modify the heads and tails of the schedule accordingly. If it is not possible, it detects and annotates a positive length cycle that the negative arc (SJ_i, i) introduces in the solution graph.

5. Neighborhood structure

It is well-known that local search metaheuristics have problems facing maximum time lag constraints, because it is very difficult to design neighborhood structures that produce feasible schedules without using costly repairing procedures. To deal with this issue, we propose combining a neighborhood structure, termed N_F that reverses arcs on the critical path, with the purpose of reducing the makespan, with a new neighborhood, termed N_{NF}, that reverses arcs on the cycles associated to the relaxed constraints, with the objective of transforming an unfeasible schedule into a feasible one. N_F and N_{NF} will be applied to feasible and relaxed schedules, respectively, and as we will see both of them may lead to relaxed schedules.

5.1. N_F structure

N_F is an extension of the neighborhood structure N^S used by Gonzalez et al. in [42] and [11] to deal with the job shop scheduling with setups. N^S extends some of the structures proposed by Van Laarhoven et al. in [12] that have given rise to some of the best methods for the JSP such as, for example, the algorithms proposed in [5], [33], [38] or [39]. The structure N_F is based on the following results.

Proposition 1. *Let S and S' be two feasible schedules such that S' is obtained from S by reversing an arc (u, v) which does not belong to any critical path in S. If the setup times fulfill the triangular inequality, then $C_{max}(S') \geq C_{max}(S)$.*

Proof. If neither u nor v are critical, then every critical path in S will be also a path in S', so $C_{max}(S') \geq C_{max}(S)$. If v is in a critical path of the form $P = (b_1, x, v, y, b_2)$ in S, where b_1 and b_2 are sequences of operations, and u is not, if y requires different machine than v then P is a path in S'. If y and v require the same machine then in S' there will be a path $P' = (b_1, x, v, u, y, b_2)$; as $s_{vu} + s_{uy} \geq s_{vy}$ then P' is larger than P. Analogous reasoning may be done if u is critical and v is not. So, in any case $C_{max}(S') \geq C_{max}(S)$. □

Therefore, as we suppose the triangular inequality holds, we will consider reversing processing orders of operations in the critical path in order to obtain improving schedules. Regarding feasibility, we will accept neighbors that may have positive length cycles only if these cycles go through at least one negative cost arc; i.e., neighbors can only have positive length cycles because of the maximum time lag constraints, which can be easily detected by the procedure *AddMaximumTimeLagConstraint* as we have seen in Section 4. These cycles are solved by creating relaxed schedules. On the other hand, cycles formed exclusively by positive cost arcs are more problematic, because they cannot be repaired by creating a relaxed schedule, as these type of cycles do not violate necessarily a maximum time lag constraint.

The left side of Figure 3 shows a portion of the schedule with a path of the form $(v\ u_1\ \ldots\ u_n\ w)$ such that all these operations belong to the same machine. Discontinuous arrows represent potential alternative paths formed only by positive cost arcs. So, a neighbor can be obtained by inserting the operation w just before v if none of these alternative paths exist; otherwise, this move would lead to a positive length cycle composed exclusively by positive cost arcs, as shown in the right side of Figure 3.

The following result establishes a sufficient condition to ensure that the neighbor do not have any of these cycles. Notice that it is valid for both feasible and relaxed schedules, as it only tries to avoid cycles not involving negative cost arcs.

Accepted manuscript

ACCEPTED MANUSCRIPT

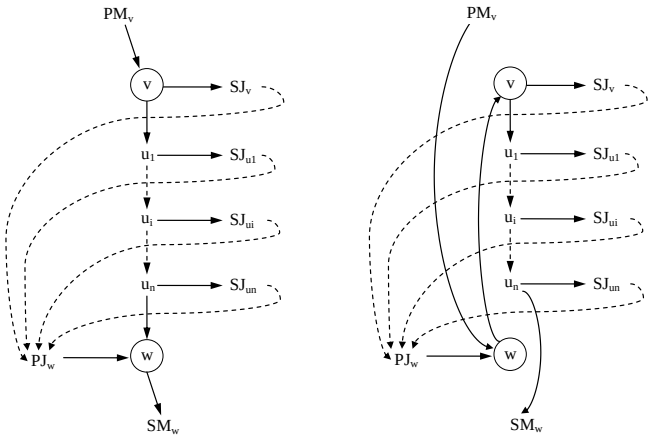


Figure 3: Potential alternative paths (formed only by positive cost arcs) between two operations v and w belonging to the same machine that would create a positive length cycle formed only by positive cost arcs in the neighbor created by inserting w before v. The initial schedule is shown in the left side while the neighbor is in the right side.

Proposition 2. *Let S be either a feasible or a relaxed schedule and $(b'\ v\ b\ w\ b'')$ a sequence of consecutive operations of the same machine, where b, b' and b'' are in itself sequences of operations of the form $b = (u_1, \ldots, u_n)$, $b' = (u'_1, \ldots, u'_{n'})$ and $b'' = (u''_1, \ldots, u''_{n''})$ with $n, n', n'' \geq 0$. Let S' be a schedule created from S by inserting the operation w just before the operation v. If S' has a positive length cycle, this cycle goes through at least one negative cost arc if the following condition holds*

$$r_{PJ_w} < r_{SJ_u} + p_{SJ_u} + C \qquad \forall u \in \{v\} \cup b \tag{3}$$

where $C = s_{zPJ_w}$ if $SM_z = PJ_w$, and $C = min\{TL_{min}(z) + p_{SJ_z}, s_{zSM_z} + p_{SM_z}\}$ otherwise, being $z = SJ_u$.

Proof. If condition (3) holds, then a path from u to PJ_w in G_S through only positive cost arcs cannot exist, for all $u \in \{v\} \cup b$ (see left side of Figure 3). Therefore, after moving w before v no cycle will exist through only positive arcs (see right side of Figure 3). □

A similar result can be stated regarding the insertion of an operation v just after an operation w. With these results we can define the following neighborhood structure.

Definition 3. N_F^* ***structure.*** *Let S be a feasible schedule, B a critical block of that schedule (see Section 2.2), and v an operation in B. The neighborhood $N_F^*(S)$ is built with solutions such that v is moved to another position in B, provided that the sufficient condition of feasibility stated in Proposition 2 is preserved.*

Clearly, the neighborhood size $\|N_F^*\|$ may be very large. So, it would be convenient to discard some neighbors, preferably non-improving ones. To this end, we establish the following sufficient condition for non improvement, which can be efficiently evaluated on each candidate neighbor.

Proposition 3. *Let S be a feasible schedule and $(b'\ v\ b\ w\ b'')$ a critical block, where b, b' and b'' are sequences of operations of the form $b = (u_1, \ldots, u_n)$, $b' = (u'_1, \ldots, u'_{n'})$ and $b'' = (u''_1, \ldots, u''_{n''})$ with $n', n'' \geq 1$, $n \geq 0$. Even if the schedule S' obtained from S by inserting w just before v is feasible, S' does not improve S if the following condition holds*

$$s_{u'_{n'}v} + s_{u_n w} + s_{wu''_1} \leq s_{u'_{n'}w} + s_{wv} + s_{u_n u''_1} \tag{4}$$

If $n = 0$, u_n should be substituted by v in the equation 4.

Proof. Condition (4) derives easily from a single comparison of the length of the critical path before the move with that of the new path through nodes $u'_{n'}$ and u''_1 after the move. Notice that after the move the paths from node *start* to $u'_{n'}$ and from u''_1 to node *end* do not change. The only changes take place in the path from v to w and, as the tasks involved are the same, the difference in length is only due to the added and removed setup times. Therefore, if condition (4) holds, the makespan of S' cannot be better than that of S. □

An analogous result can be established for the non-improvement of a neighbor created by inserting an operation v after an operation w. Hence, we can define a reduced neighborhood structure N_F based on this last result.

Definition 4. N_F ***structure.*** *Let S be a feasible schedule, B a critical block of that schedule, and v an operation of B. The neighborhood $N_F(S)$ is built with solutions such that v is moved to another position in B, provided that the non-improvement condition stated in Proposition 3 does not hold and that the sufficient condition stated in Proposition 2 is preserved.*

As an example, consider the schedule represented in Figure 4. There is only one critical block of length higher than one in the critical path: (o_{21}, o_{32}, o_{11}). The neighborhood structure N_F would consider the following four neighbors: inserting o_{21} after o_{32}, inserting o_{21} after o_{11}, inserting o_{32} after o_{11} and inserting o_{11} before o_{21}. In particular, Figure 1 shows the neighbor created by inserting o_{32} after o_{11}. Notice that the makespan is reduced from 24 to 22.

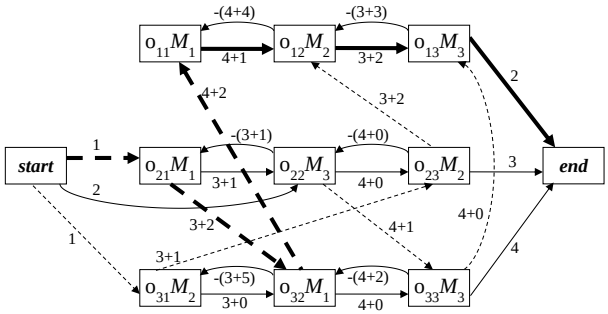
5.2. N_{NF} structure

As we have pointed, we propose another type of neighborhood structure which is applied to schedules not satisfying some maximum time lag constraint, and so it defines several types of moves that try to bring schedules towards feasibility.

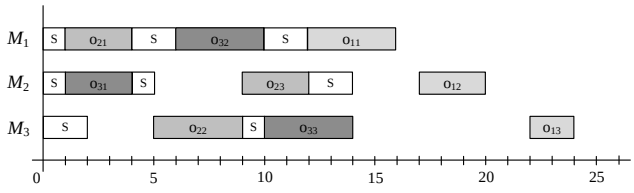
The first type of moves is based on trying to get closer two consecutive operations in a job that violate a maximum time lag constraint. Let v be an operation that

Accepted manuscript

ACCEPTED MANUSCRIPT



(a) Solution graph. The sequence or arcs in bold is a critical path with length 24.



(b) Gantt chart. The makespan is 24.

Figure 4: Another feasible schedule to the problem with 3 jobs and 3 machines.

violates the maximum time lag with respect to his job predecessor. Then we consider two neighbors: reversing the arc (PM_v,v) and reversing the arc (PJ_v,SM_{PJ_v}). In these neighbors the difference between the ending time of PJ_v and the starting time of v has the chance to be lower and so these neighbors may not violate that maximum time lag constraint.

The second type of moves is based on reversing an arc in a positive length cycle associated to a relaxed maximum time lag constraint. For each unchecked constraint and its stored cycle, we consider reversing each arc between two consecutive operations that require the same machine. The intuition behind these moves is that the addition of the constraint that produced the positive length cycle in the current schedule may not produce that cycle in any of these neighbors.

As an example we can consider the schedule of Figure 2. There is one relaxed maximum time lag from o_{32} to o_{33}, and the positive length cycle detected if we try to add this maximum time lag is (o_{33}, o_{32}, o_{31}, o_{23}, o_{12}, o_{13}). In this case, the first type of moves defined by N_{NF} would consider the reversal of the arc ($PM_{o_{33}}$, o_{33}) (i.e. the arc (o_{13}, o_{33})).

The second type of moves defined by N_{NF} would consider the possible neighbors created by reversing a machine arc that belongs to the positive length cycle detected. Hence, the options are (o_{31}, o_{23}), (o_{23}, o_{12}) and (o_{13}, o_{33}). Figure 4 represents the neighbor created by reversing the arc (o_{13}, o_{33}), which is completely feasible as there is no need to relax any maximum time lag constraint.

Therefore, in this simple case N_{NF} is able to create a feasible neighbor from an unfeasible one with a single move. However, several moves may be necessary if we need to restore the feasibility of an unfeasible schedule with many relaxed maximum time lags.

Formally, the neighborhood structure N_{NF} is defined as follows.

Definition 5. N_{NF} ***structure.*** *Let S be a relaxed schedule and let $U = (u_1, \ldots, u_D)$ be the set of D operations such that $r_{u_i} > r_{PJ_{u_i}} + p_{PJ_{u_i}} + TL_{max}(PJ_{u_i})$, $\forall i \in 1, \ldots, D$. Let $C = (C_1, \ldots, C_D)$ be the set of D positive length cycles that are detected if the corresponding relaxed maximum time lag constraint is added to S. Each C_i is of the form $C_i = (C_{i1}, \ldots, C_{il_i})$. Then, the following moves generate a neighbor in $N_{NF}(S)$:*

1. *Reversing the processing order of PM_v and v, $\forall v \in U$.*
2. *Reversing the processing order of PJ_v and SM_{PJ_v}, $\forall v \in U$.*
3. *Reversing the processing order of two operations v and w such that $w = SM_v$, provided that both v and w belong to the same C_i, $i \in 1, \ldots, D$.*

provided that the condition stated in Proposition 2 holds.

5.3. N_{TL} structure

We can now define the neighborhood structure we propose to use in our tabu search for the SDST-JSPTL.

Definition 6. N_{TL} ***structure.*** *Let S be a relaxed schedule with D relaxed maximum time lag constraints. Then we define $N_{TL}(S) = N_F(S)$ if $D = 0$, i.e., if S is feasible, and $N_{TL}(S) = N_{NF}(S)$ if $D > 0$.*

Once a neighbor is generated, it has to be evaluated to take part in the selection process. In the following section we describe how we are evaluating neighbors. It is also important to remark that when one of the neighbors is finally selected we need to reconstruct the vector containing the topological order of the graph. However, it is necessary to reconstruct only a portion of that ordering. For example, if the neighbor is created with N_F by inserting operation v after w, or by inserting w before v, we only need to reconstruct the vector starting from the position of operation v and finishing in the position of operation w (a similar reconstruction is detailed by Mattfeld in [43] for the classical JSP).

5.4. Neighbors estimate

As exact evaluation is computationally expensive, usually some estimation procedure is used instead to evaluate the neighbors. In particular, for neighborhood structures involving arc reversing, the procedure *lpath* proposed by Taillard in [36] for the classical JSP is commonly used as a base for designing an efficient estimation algorithm.

We propose here the procedure *lpathTL* which is an extension of *lpath* to cope with minimum and maximum

Accepted manuscript

ACCEPTED MANUSCRIPT

time lags and setup times. This procedure takes as input a sequence of operations requiring the same machine $(x, Q_1, \ldots, Q_q, y)$ after a move, where $Q_1, \ldots, Q_q$ is a permutation of operations $O_1, \ldots, O_q$ appearing as $(x, O_1, \ldots, O_q, y)$ before the move. For each $i = 1, \ldots, q$, $lpathTL$ estimates the cost of the longest path from *start* to *end* through Q_i. The maximum of these values is taken as the makespan estimation for the neighboring schedule, even though it may not be a lower bound on the makespan.

This procedure is suitable for all the moves we have defined for N_{TL}, by taking the appropriate input sequence $(Q_1, \ldots, Q_q)$ in each case. Hence, for N_F, if w is moved before v in a block of the form $(b'\ v\ b\ w\ b'')$, the input sequence is $(w\ v\ b)$, and if v is moved after w in a block of the form $(b'\ v\ b\ w\ b'')$, the input sequence is $(b\ w\ v)$. For N_{NF}, if (v, w) is the reversed arc, the input sequence is $(w\ v)$.

The algorithm starts estimating the heads of the operations involved in the move, from the beginning to the end, and then estimates the tails of the same operations, from the end to the beginning. To calculate this estimate, if $x = PM_{O_1}$ (notice that x may be the node *start*) and $y = SM_{O_q}$ (notice that y may be the node *end*) before the move, the heads and tails for the operations $Q_1, \ldots, Q_q$ are estimated as indicated in Algorithm 4. In this algorithm, C_{max} denotes the makespan of the original schedule, $P(v)$ denotes the predecessor of operation v in the longest path from node *start* to node v, and $Q(v)$ denotes the successor of operation v in the longest path from node v to node *end*.

To estimate the head of an operation v notice that, if the longest path from node *start* to node PJ_v goes through v, we are in a special case and we take into account the starting time of PJ_v, but only after temporarily relaxing the maximum time lag constraints of the previous operations of the job. The motivation is that in this case the starting time of PJ_v is set by v, hence considering r_{PJ_v} without any change would likely produce an estimation larger than the real value. On the other hand, recalculating a modified head $r^M_{PJ_v}$ by taking into account only its job and machine predecessors will lead to a better estimate for the head of v.

Additionally, notice that for estimating the head of an operation v we only take into account the head of SJ_v if the longest path from node *start* to node SJ_v does not go through v. The motivation is that, if $P(SJ_v) \neq v$ then it is possible that the starting time of SJ_v does not change in the new schedule, hence the new head of v may be set by the maximum time lag constraint between v and SJ_v. Otherwise, if $P(SJ_v) = v$ notice that the old head of SJ_v can not give information about the new head of v.

Similar considerations apply for calculating the tails of an operation v: we only consider the tail of PJ_v if the longest path from node PJ_v to node *end* does not go through node v. And in case that the longest path from node SJ_v to node *end* goes through node v we calculate a modified tail of SJ_v relaxing the maximum time lag constraints of all job successors of v, as similarly done in the head calculation.

Notice that in the moves defined for N_{NF} it may happen that no processing order of critical operations is reversed. In this case the makespan of the neighbor is at least the makespan of the original schedule (see Proposition 1), and therefore we can refine the estimate with this value.

In order to evaluate the accuracy of the proposed estimation algorithm, we estimated and evaluated the actual makespan of about 50 million neighbors for instances with different sizes and time lags. We observed that the estimate was in average 4.48% higher than the actual makespan, but the estimate coincided with the exact value of the makespan in 39% of the neighbors in average. This number highly depends on the tightness of the maximum time lags. In fact, in instances with very loose time lags the estimates coincided with the makespan in 94% of the neighbors in some cases, whereas for instances with tight time lags the estimates coincided with the makespan in only 20% of the neighbors.

We have tried to run the algorithm without using this procedure, evaluating exactly each one of the neighbors by applying the decoding algorithm described in Section 4. However, the running time in this case was between 10 and 20 times higher depending on the instance, while the results were not better, and in fact were worse in some cases. Therefore, we can conclude that this estimate is really efficient and appropriate.

5.4.1. An example

Let us consider, for example, how the makespan of the schedule in Figure 4 is estimated when this schedule is obtained as neighbor of the schedule in Figure 1 by reversing the arc (o_{11}, o_{32}).

- The first step is estimating the head of o_{32}, and to this end we have to consider the estimations through the job predecessor (Est_{PJ}), the machine predecessor (Est_{PM}) and the job successor (Est_{SJ}). To calculate Est_{PJ} we take into account that $PJ_{o_{32}} = o_{31}$ and $P(o_{31}) = o_{32}$, which means that the head (the starting time) of task o_{31} is set by the maximum time lag constraint between o_{31} and o_{32}. Therefore, we relax the maximum time lags of the job predecessors of o_{32}. We calculate the modified head $r^M_{o_{31}}$ (i.e. the starting time if its job had no maximum time lag constraints), which is 1. So, Est_{PJ} will be 1 plus the duration of task o_{31} plus the minimum time lag between o_{31} and o_{32}; i.e., $Est_{PJ} = 1 + 3 + 0 = 4$.

  Then, Est_{PM} is calculated as the head of the machine predecessor plus its duration and the corresponding setup time, with a total of 6. Finally, Est_{SJ} is 0, as $P(o_{33}) = o_{32}$, so in this case the starting time of o_{33} is set by o_{32}, and therefore it is not useful to consider its head. Finally, the head of o_{32} is estimated as:

  $$r'_{o_{32}} = \max\{4, 6, 0\} = 6$$

ACCEPTED MANUSCRIPT

Algorithm 4 Neighbors makespan estimate (lpathTL)

Require: A sequence of operations $(x, Q_1, \ldots, Q_q, y)$ as they appear after a move
// First we estimate the heads of the operations
$r'_x \leftarrow r_x$; $v \leftarrow x$;
for $i \leftarrow 1$ to q **do**
  $w \leftarrow Q_i$; $Est_{PJ} \leftarrow 0$, $Est_{PM} \leftarrow s_{0w}$, $Est_{SJ} \leftarrow 0$;
  if $PJ_w \neq start$ **then**
    if $P(PJ_w) = w$ **then**
      // In this case, to estimate the head of w we relax the maximum time lags of its job predecessors
      $j \leftarrow$ the job of operation w; $k \leftarrow$ position of w inside its job; $r^M_{o_{j1}} \leftarrow r_{PM_{o_{j1}}} + p_{PM_{o_{j1}}} + s_{PM_{o_{j1}}o_{j1}}$;
      for $l \leftarrow 2$ to $k-1$ **do** $r^M_{o_{jl}} \leftarrow \max\{r^M_{o_{jl-1}} + p_{o_{jl-1}} + TL_{min}(o_{jl-1}), r_{PM_{o_{jl}}} + p_{PM_{o_{jl}}} + s_{PM_{o_{jl}}o_{jl}}\}$ **end for**
      $Est_{PJ} \leftarrow r^M_{PJ_w} + p_{PJ_w} + TL_{min}(PJ_w)$;
    else
      $Est_{PJ} \leftarrow r_{PJ_w} + p_{PJ_w} + TL_{min}(PJ_w)$;
    end if
  end if
  if $PM_w \neq start$ **then** $Est_{PM} \leftarrow r'_v + p_v + s_{vw}$; **end if**
  if $SJ_w \neq end$ and $P(SJ_w) \neq w$ **then** $Est_{SJ} \leftarrow r_{SJ_w} - p_w - TL_{max}(w)$; **end if**
  $r'_w \leftarrow \max\{Est_{PJ}, Est_{PM}, Est_{SJ}\}$;
  $v \leftarrow w$;
end for
// Now we estimate the tails of the operations
$q'_y \leftarrow q_y$; $w \leftarrow y$;
for $i \leftarrow q$ to 1 **do**
  $v \leftarrow Q_i$; $Est_{SJ} \leftarrow p_v$, $Est_{SM} \leftarrow 0$, $Est_{PJ} \leftarrow 0$;
  if $SJ_v \neq end$ **then**
    if $Q(SJ_v) = v$ **then**
      // In this case, to estimate the tail of v we relax the maximum time lags of its job successors
      $j \leftarrow$ the job of operation v; $k \leftarrow$ position of v inside its job; $q^M_{o_{jN_j}} \leftarrow q_{SM_{o_{jN_j}}} + p_{o_{jN_j}} + s_{o_{jN_j}SM_{o_{jN_j}}}$;
      for $l \leftarrow N_j - 1$ to $k+1$ **do** $q^M_{o_{jl}} \leftarrow \max\{q^M_{o_{jl+1}} + p_{o_{jl}} + TL_{min}(o_{jl}), q_{SM_{o_{jl}}} + p_{o_{jl}} + s_{o_{jl}SM_{o_{jl}}}\}$ **end for**
      $Est_{SJ} \leftarrow q^M_{SJ_v} + p_v + TL_{min}(v)$;
    else
      $Est_{SJ} \leftarrow q_{SJ_v} + p_v + TL_{min}(v)$;
    end if
  end if
  if $SM_v \neq end$ **then** $Est_{SM} \leftarrow q'_w + p_v + s_{vw}$; **end if**
  if $PJ_v \neq start$ and $Q(PJ_v) \neq v$ **then** $Est_{PJ} \leftarrow q_{PJ_v} - p_{PJ_v} - TL_{max}(PJ_v)$; **end if**
  $q'_v \leftarrow \max\{Est_{SJ}, Est_{SM}, Est_{PJ}\}$;
  $w \leftarrow v$;
end for
// And finally we calculate the estimate
$FinalEst \leftarrow \max_{i=1,\ldots,q}\{r'_{Q_i} + q'_{Q_i}\}$;
if no processing order of critical operations was reversed to create the neighbor **then**
  $FinalEst \leftarrow \max\{FinalEst, C_{max}\}$;
end if
return The estimate $FinalEst$;

which is the actual value of the new head of o_{32}. Notice that if the maximum time lag constraints were not relaxed, then we would obtain $Est_{PJ} = 7$.

- Then we estimate the head of o_{11}. In this case, $Est_{PJ} = 0$ because $PJ_{o_{11}} = start$; $Est_{PM} = 12$, as the estimated tail of o_{32} was 6, plus its duration and the corresponding setup time; and $Est_{SJ} = 6$ because the head of o_{12} is 14, minus its duration (4) and the corresponding maximum time lag (4). Therefore, the head of o_{11} is estimated as:

$$r'_{o_{11}} = \max\{0, 12, 6\} = 12$$

which again coincides with the actual value of the new

Accepted manuscript

ACCEPTED MANUSCRIPT

head.

- The next step is estimating the tail of o_{11}. In this case we have to consider the tail through the job successor (Est_{SJ}), the machine successor (Est_{SM}) and the job predecessor (Est_{PJ}). First, we see that $Est_{SM} = 0$ because $SM_{o_{11}} = end$. Also, $Est_{PJ} = 0$ because $PJ_{o_{11}} = start$. However, for calculating Est_{SJ} we see that $o_{12} \neq end$ and $Q(o_{12}) = o_{11}$. As done with the heads, to compute Est_{SJ} we calculate the modified tails of the job successors of o_{11} relaxing the maximum time lag constraints of the job. In this case we obtain a value of 12 for Est_{SJ}, and thus the estimated tail of o_{11} is:

  $q'_{o_{11}} = \max\{12, 0, 0\} = 12$

  which coincides with the actual value of the tail of o_{11} in the neighbor. If we had calculated Est_{SJ} without doing the relaxation, the final estimate of the tail would be 13.

- Then, we estimate the tail of o_{32} as:

  $q'_{o_{32}} = \max\{10, 18, 9\} = 18$

- And the final estimate is calculated as:

  $\max\{r'_{o_{11}} + q'_{o_{11}}, r'_{o_{32}} + q'_{o_{32}}\} = \max\{24, 24\} = 24$

  which in this case is the exact makespan of the neighbor.

6. Scatter search for the SDST-JSPTL

Scatter Search (SS) is a population-based evolutionary metaheuristic that was first proposed by Glover in [44]. SS is recognized as an excellent method at achieving a proper balance between intensification and diversification in the search. It has been successfully applied to a number of problems, in particular to scheduling [45] [46] [47] [48], [49].

The SS five-method template proposed by Glover in [50] has been the main reference for most SS implementations to date, including our proposal. This template consists of 1) A diversification-generation method, 2) An improvement method, 3) A reference-set update method, 4) A subset-generation method and 5) A solution-combination method.

Our proposal is shown in Algorithm 5. It starts by creating an initial set of solutions P which are improved using Tabu Search (TS). Then, a reference set $RefSet$ is obtained selecting the best solutions from P. The algorithm iterates until a stopping condition is met, this being a given number of iterations without improvement. At each iteration, a pair of solutions from $RefSet$ is combined using Path Relinking (PR) to generate a new solution, which is also improved by TS. Then, the reference set update method is applied. Additionally, if all possible pairs of solutions in $RefSet$ have already been combined without introducing any new solution to the set, a diversification phase is applied. Further detail on the algorithm is given in the following subsections.

Algorithm 5 Scatter Search

Require: A SDST-JSPTL instance
  Create the set P with $PSize$ random solutions;
  Apply Tabu Search to every solution of P;
  Build $RefSet$ taking the best solutions from P;
  while not Stop Condition **do**
    if All possible pairs of solutions in $RefSet$ were already combined **then**
      Apply diversification phase;
    end if
    Choose two solutions S_{ini} and S_{end} from $RefSet$ not combined yet;
    Combine S_{ini} and S_{end} with PR to obtain S^*;
    Apply the improvement method (TS) to S^*;
    Update $RefSet$, if necessary, with the improved S^*;
  end while
  return The best solution in $RefSet$;

6.1. Initial reference set construction

The reference set should contain a collection of high quality and diverse solutions. To achieve this, an initial set P is generated with $PSize$ random solutions which are all improved using tabu search. We create random initial solutions by means of the job insertion heuristic described in Section 3. As suggested in [23] by Artigues et al., we consider the next orderings of the jobs for the first 8 solutions of P:

1. Lexicographical and Anti-lexicographical order
2. Ascending and descending order of job durations
3. Ascending and descending order of time lags
4. Ascending and descending order of time lags plus durations

The remaining solutions of P are built using random job orderings. Notice that all solutions created to build P are feasible, and therefore all solutions included in the reference set $RefSet$ will also be feasible. In particular, $RefSet$ is built by selecting the $RSSize$ best solutions from P in terms of makespan, provided that the distance between each pair of solutions is higher than a parameter $MinDist$, which indicates the minimum distance allowed between solutions in $RefSet$.

The distance between two solutions S_1 and S_2 (denoted by $D(S_1, S_2)$) is given by the disjunctive graph distance, or Hamming distance, defined by Mattfeld in [43] as the number of pairs of operations requiring the same machine which are processed in different order in S_1 and S_2. The maximum possible distance $maxDist$ between two solutions depends on the instance size, and can be calculated as

$maxDist = \sum_{i=1}^{m} \frac{K_i \times K_i - 1}{2}$

Accepted manuscript

ACCEPTED MANUSCRIPT

where K_i is the total number of operations that require machine M_i.

6.2. Subset generation method and diversification phase

For subset generation, we use a simple method that consists in selecting all possible pairs of solutions in $RefSet$ in a given order. Each pair of solutions is combined to obtain a new solution (as explained in section 6.3), unless they have not changed since the last time they were selected together. Tabu Search is then applied to the new solution and $RefSet$ is updated in accordance with the reference set updating method (see section 6.5).

If no new solution is added to $RefSet$ after combining all possible pairs of solutions, the diversification process is applied: set P is rebuilt starting with the best solution so far, and new $PSize - 1$ solutions are generated at random. In this case we only use random job orderings in the job-insertion heuristic, to avoid repeating previously generated solutions. These schedules are then improved by tabu search, and finally a new $RefSet$ is obtained from P.

6.3. Solution-combination method

As solution-combination method we use Path Relinking (PR). This metaheuristic has been successfully applied to the classical job shop scheduling problem. For example in [51], Aiex et al. apply path relinking within a GRASP algorithm as an intensification strategy. Nowicki and Smutnicki in [5] improve their TSAB metaheuristic by introducing a new initial solution generator based on path relinking. Nasiri and Kianfar in [52] combine tabu search and path relinking using two different neighborhoods. Recently, Jia and Hu in [53] propose a combination of path relinking and tabu search for the flexible job shop scheduling problem.

The Path Relinking procedure combines two solutions, referred to as initial (S_{ini}) and guiding (S_{end}) solutions, to obtain a new solution. In our algorithm, the initial solution is always the best of the two solutions taken from $RefSet$. The procedure starts from the initial solution and repeatedly applies moves so each single move produces a solution which is closer to the guiding solution than the current one. The search finishes when reaching the guiding solution or after a maximum number of $maxDist$ iterations. Then, the best solution located between 1/4 and 3/4 of the created trajectory is returned, as it is similarly done by Nowicki and Smutnicki in [46]. The best solution is defined as the solution with the fewest relaxed maximum time lag constraints, using the makespan for tie-breaking.

It is clear that in this algorithm our priority is not only to reduce the makespan but also to reduce the distance between two solutions. For this reason, in this case we are considering the less restrictive neighborhood structure N_F^* instead of N_F, as some of the neighbors discarded by N_F may be very attractive in terms of distance. Therefore in the path relinking we propose to use in principle the following structure.

Definition 7. N_{TL}^* ***structure.*** *Let S be a relaxed schedule with D relaxed maximum time lag constraints. Then we define $N_{TL}^*(S) = N_F^*(S)$ if $D = 0$, i.e., if S is feasible, and $N_{TL}^*(S) = N_{NF}(S)$ if $D > 0$.*

A single move yields the initial solution one unit closer to or farther from the guiding solution when using N_{NF}, but can get several units closer to or farther from the guiding solution when using N_F^*. Since several neighbors of one solution may be at the same distance of the guiding solution, the estimated makespan is used as tie-breaking rule.

In order to escape from local optima, similarly to Tabu Search, a neighbor created by reversing an arc already reversed in the last iterations is discarded, unless it becomes the closest neighbor so far to the guiding solution. Even with this mechanism, local optima may be so deep that it is very difficult to find a complete path from the initial to the guiding solution using N_{TL}^*. For this reason, we are also going to define another less restrictive structure N_{PR} that consists of all single moves leading to a feasible schedule. We define a single move as a reversal of the processing order of two consecutive operations of the same machine, even if they are not critical operations.

Definition 8. N_{PR} ***structure.*** *Let v and w be consecutive operations of the same machine. In a neighboring solution the processing order of v and w is reversed, provided that the sufficient condition of feasibility stated in Proposition 2 is preserved.*

Thus, as soon as N_{TL}^* fails to reach a solution closer to the guiding solution $maxFails$ times, the neighborhood structure changes to N_{PR}, and for the remaining of the path relinking procedure the best neighbor that is closer to the guiding solution is chosen.

An alternative to the above would be to use only N_{PR} in the path relinking algorithm. However, it is better to start with N_{TL}^*, since it is a more refined structure which guides the path through promising neighbors, and use N_{PR} only to complete the path when N_{TL}^* gets stuck into local optima.

The proposed PR method is detailed in Algorithm 6, where TL denotes the Tabu List used in the algorithm, and $\neg Tabu(S', TL)$ means that the move from S to S' is not in TL.

In summary, the algorithm starts from a schedule $S = S_{ini}$ and in each iteration it selects a schedule S^* among the neighbors of S, such that S^* is the closest schedule to S_{end} that is not tabu or it is the closest to S_{end} found so far, breaking ties by the estimated makespan. The neighborhood is in principle N_{TL}^*; however, after $maxFails$ iterations without reaching a new schedule S^* closer to S_{end} than the previous one, the neighborhood swaps to N_{PR} as explained before. When we choose a new schedule S^*, we update the variables that register the minimum distance to S_{end}, the number of fails in reaching a schedule better than the previous one, the distance of the current solution

Accepted manuscript

ACCEPTED MANUSCRIPT

Algorithm 6 Path Relinking

Require: A SDST-JSPTL instance, an initial schedule S_{ini} and a guiding schedule S_{end}
$S \leftarrow S_{ini}$; $dist_{min}, dist_{cur} \leftarrow D(S, S_{end})$;
$numIters \leftarrow 0$; $numFails \leftarrow 0$; $TL \leftarrow \emptyset$;
while $dist_{cur} > 0 \wedge numIters \leq maxDist$ **do**
  if $numFails < maxFails$ **then**
    $\mathbf{N} = N^*_{TL}$
  else
    $\mathbf{N} = N_{PR}$;
  end if
  $S^* \leftarrow argmin\{D(S', S_{end}), S' \in \mathbf{N}(S) \wedge (D(S', S_{end}) < dist_{min} \vee \neg Tabu(S', TL))\}$;
  Update TL accordingly;
  if $D(S^*, S_{end}) < dist_{min}$ **then**
    $dist_{min} \leftarrow D(S^*, S_{end})$;
  end if
  if $D(S^*, S_{end}) > dist_{cur}$ **then**
    $numFails \leftarrow numFails + 1$;
  end if
  $dist_{cur} \leftarrow D(S^*, S_{end})$;
  $S \leftarrow S^*$;
  Apply *DecodingAlgorithm* to S;
  $numIters \leftarrow numIters + 1$;
end while
return the best solution between 1/4 and 3/4 of the length of the created trajectory;

to S_{end} and also the tabu list. Finally, the new schedule S^* is stored in S and must be evaluated by means of the *DecodingAlgorithm*. Here, it is important to clarify that before this point, we only know the vector of symbols that represents the topological ordering of the operations and the estimated makespan. Hence, we have to apply the *DecodingAlgorithm* to calculate the heads and tails of all operations and so determine the actual makespan and whether or not it is a relaxed or a feasible schedule. The procedure finishes either when reaching S_{end} or after $maxDist$ iterations; however in our experimental study the second condition was never met and it always finished by reaching the guiding solution S_{end}. The algorithm returns the best solution located between 1/4 and 3/4 of the created trajectory, i.e. the solution with the fewest relaxed maximum time lag constraints, now using the actual makespan for breaking ties.

6.4. Improvement method

As improvement method we use tabu search (TS). TS is an advanced local search technique proposed by Glover in [54, 55] that can escape from local optima by selecting non-improving neighbors. To avoid revisiting recently visited solutions and explore new promising regions of the search space, it maintains a tabu list with a set of moves which are not allowed when generating the new neighborhood. TS has a solid record of good empirical performance in problem solving, in particular in scheduling. For example, the $i-$TSAB algorithm from Nowicki and Smutnicki [5] is one of the best approaches for the classical JSP. Also, in [11] Gonzalez et al. propose a TS algorithm that obtains the best results so far for the SDST-JSP with lateness minimization. TS is often used in combination with other metaheuristics such as genetic algorithms [56] or scatter search and path relinking [57].

Algorithm 7 shows the tabu search algorithm considered herein, where TL and CL denote, respectively, the Tabu List and the Cycle List. The general scheme is similar to other tabu search algorithms from the literature [38] [37]. In the first step the initial solution is evaluated. Then, it iterates over a number of steps. In each iteration, the neighborhood of the current solution is built and one of the neighbors is selected for the next iteration. The tabu search finishes after a number of *maxImproveIter* iterations without improvement, returning the best solution reached so far.

The selection rule chooses the neighbor with the best estimated makespan, discarding suspect-of-cycle and tabu neighbors. Instead of storing actual solutions in the tabu list, those arcs which have been reversed to generate a neighbor are stored. Thus a new neighbor is marked as tabu if it requires reversing at least one arc included in the tabu list. Unlike usual, we do not use an aspiration criterion. The reason is that the makespan estimation procedure for the problem with time lags is not as accurate as it is, for example, in the case of the classical job shop, hence some of the times a tabu move leads to a solution with better estimate than the makespan of the best solution reached, it results to be worse. This choice was supported by some experimental results not reported here.

The length of the tabu list is usually of critical importance, since it allows for equilibrium between intensification and diversification. All TS algorithms try to keep this equilibrium with different strategies that control the number of iterations a solution can keep its tabu status. Several studies (see for example [58]) show that a dynamic management usually yields better results than a static one. Indeed, we have conducted some preliminary experiments to confirm this. Here, we use the dynamic length schema and the cycle checking mechanism based on witness arcs used, among others, by Dell' Amico and Trubian in [38].

In order to keep the overall time taken by the scatter search from becoming prohibitive, tabu search must take low time in each run. For this reason, we do not include any other diversification techniques, as restarts or variable neighborhood search, which are often used when TS is run alone.

6.5. Reference set update

If the solution S returned by TS is feasible (i.e. with no relaxed maximum time lags), it can be considered for inclusion in $RefSet$. In this case, S replaces the worst one in $RefSet$ (denoted by S_W), either if it is better than the current best solution or if $C_{max}(S) < C_{max}(S_W)$ and the

Accepted manuscript

ACCEPTED MANUSCRIPT

Algorithm 7 Tabu Search

Require: A SDST-JSPTL instance and a schedule S^0
  $S \leftarrow S^0$; $S^B \leftarrow S$; $C^B_{max} \leftarrow C_{max}(S^B)$;
  $improveIter \leftarrow 0$;
  $TL, CL \leftarrow \emptyset$;
  while $improveIter < maxImproveIter$ **do**
    $improveIter \leftarrow improveIter + 1$;
    $S^* \leftarrow argmin\{C_{max}(S'), S' \in N_{TL}(S) \wedge \neg Tabu(S', TL) \wedge \neg Cycle(S', CL))\}$;
    Update TL and CL accordingly;
    $S \leftarrow S^*$;
    Apply $DecodingAlgorithm$ to S;
    if $C_{max}(S) < C^B_{max}$ **then**
      $S^B \leftarrow S$; $C^B_{max} \leftarrow C_{max}(S)$;
      $improveIter \leftarrow 0$;
    end if
  end while
  return The solution S^B with makespan C^B_{max};

distance from S to all the solutions in $RefSet$ exceeds a given minimum distance $MinDist$, in order to avoid introducing very similar solutions in $RefSet$. Notice that in the proposed algorithm, all solutions in $RefSet$ must be feasible.

7. Experimental study

We have conducted an experimental study across benchmarks of common use for the SDST-JSPTL. Furthermore, to strengthen the conclusions of this study we have also considered the same problem without setup times, i.e., the JSPTL. It is our purpose to analyze the proposed Scatter Search with Path Relinking (SSPR) algorithm and to compare it with the state-of-the-art. The considered benchmarks and performance metrics are given in Sections 7.1 and 7.2 respectively. Then, in Section 7.3 we analyze the behavior of our algorithm and fix some of the running parameters. In Section 7.4 we compare SSPR against other algorithm such as TS running alone and a memetic algorithm. Finally, we have compared our approach with the state of the art in both the SDST-JSPTL (Section 7.5) and the JSPTL (Section 7.6).

7.1. Benchmark sets

For the SDST-JSPTL we have considered the instances defined by Oddi et al. in [31], which are derived from the 25 instances defined by Brucker and Thiele in [6] for the SDST-JSP by adding minimum and maximum time lags. Two benchmarks of 25 instances each were generated, denoted by BTS25-TW15 and BTS25-TW20. These benchmarks are available online [1]. In the first benchmark, the allowed slack between any two contiguous activities of the same job is smaller, and intuitively the benchmark should be more difficult to solve. These instances have sizes of 10×5, 15×5 and 20×5 (number of jobs $\times$ number of machines). The setup times are sequence dependent and they fulfill the triangle inequality.

For the JSPTL without setup times we use the instances generated by Caumond et al. in [22] by adding maximum time lag constraints to the forty Lawrence classical JSP instances of the OR-library [59]. Given a job shop instance, we can create a new instance by assigning each operation a maximum time lag calculated as $\beta \times AvgT$, where $AvgT$ is the average processing time over operations of its job, and β is a parameter used to create the new instance. We consider β values of 0, 0.25, 0.5, 1, 2, 3 and 10. The minimum time lags are all 0. This is justified by the fact that only maximum time lags increase the problem complexity, since it is always possible to integrate the value of a minimum time lag in the conjunctive arc between an operation and its job successor. Hence, we have a total of $40 \times 7 = 280$ instances in all.

7.2. Performance metric

To compare the results of the algorithms, we report the best and average makespan, and also the Relative Percentage Deviation (RPD) which is defined as:

$$RPD = ((Alg_{sol} - Min_{sol})/Min_{sol}) \times 100$$

where Alg_{sol} is the value of the objective function obtained by a given algorithm for a given instance, and Min_{sol} is the best-known solution for the instance for all the algorithms considered, including possible new best solutions found with SSPR.

7.3. Parameter tuning

For parameter tuning, we have performed a preliminary study, considering different values of the parameters and we used similar run times for each configuration tested.

$RSSize$ defines the number of solutions of $RefSet$ and is usually lower than 20, as indicated by Glover in [50]. Here we tried 8, 10 and 12, as they are typical values in the literature (see for example [46] and [47]).

$PSize$ defines the number of solutions of P. In [60] it is suggested that $PSize = max(100, 5 * RSSize)$. However, the diversification phase we have proposed needs rebuilding the set P several times during an execution. As this process is computationally expensive, we also considered a smaller size for P. In particular we tested sizes 20 and 100.

$maxFails$ is a parameter of the path relinking algorithm which defines the number of times that we can choose with N^*_{TL} a neighbor further to the guiding solution before switching to N_{PR}. We considered the values 1 (so it immediately changes to N_{PR} when a local optimum is reached), 5 and 20.

$maxImproveIter$ establishes the stopping condition of the tabu search algorithm. As TS is embedded in the

[1] `http://pst.istc.cnr.it/~angelo/bts25tw/`

Accepted manuscript

ACCEPTED MANUSCRIPT

Table 1: Values tested in the parameter tuning. Bold values indicate the best configuration found.

Parameter	Values tested
$RSSize$	8, 10, **12**
$PSize$	**20**, 100
$maxFails$	1, **5**, 20
$maxImproveIter$	100, 200, **500**, 1000
$MinDist$	5, **20**, 50

scatter search core and as it is executed many times during an execution, we cannot choose so large values as when TS algorithm runs alone. We considered 100, 200, 500 and 1000.

$MinDist$ is a parameter that controls the minimum distance allowed between solutions in $RefSet$, and hence it should ensure the diversity of the set. We tested 5, 20 and 50.

Table 1 shows a summary of the tested values, indicating in bold the configuration that achieved the best average results.

As stopping criterion for the algorithm we used a maximum of 500 iterations of SSPR without improvement, since this results in a good convergence pattern. This is shown in Figure 5, which details the evolution of the best and average makespan using this configuration for one run of the dataps15Max12 instance. We can also notice the 4 times that the diversification phase is activated with a sudden increase in the average makespan of the solutions in $RefSet$.

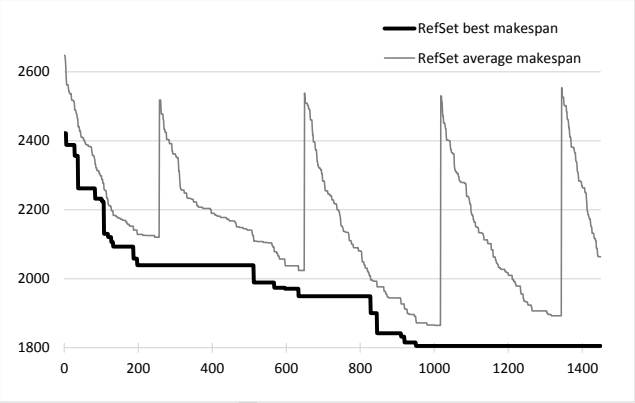


Figure 5: Evolution of the best and average makespan of $RefSet$ depending on the iteration number, for one run of the dataps15Max12 instance.

7.4. Comparison of SSPR with a memetic algorithm and TS alone

It is well known that the tabu search metaheuristic is very efficient in solving scheduling problems (see for example [5], [33] or [11]). Here we have carried out some experiments to assess if the scatter search with path relinking shell is capable of improving the performance of the tabu search alone. To do this, we compared the results of the SSPR algorithm (with PR and TS) with those from TS alone. For running TS alone we have opted to set the same stopping criterion as in SSPR (500 iterations without improvement), and to launch TS starting from random solutions (created with the heuristic described in Section 3) as many times as possible in the run time used by SSPR.

Table 2: Comparison of SSPR, GA+TS and TS. The algorithms were applied to two sets with 25 instances each and then best and average RPD values are reported.

Instances	TS		GA+TS		SSPR	
	best	avg	best	avg	best	avg
BTS25-TW15	12.5	17.3	10.3	15.4	0.4	4.7
BTS25-TW20	10.0	14.2	5.4	8.1	0.0	1.7

We also compare SSPR with a genetic algorithm combined with tabu search. To this end, we designed a memetic algorithm (GA+TS) with the following characteristics: the encoding schema is based on permutations with repetition as it was detailed in Section 4. The initial population is generated at random with the heuristic described in Section 3, and then improved with TS. In order to build schedules from chromosomes, we use the decoding algorithm detailed in Section 4. In each generation of the algorithm, we group the chromosomes randomly in pairs and we apply the crossover operator to them, generating two offspring solutions from each two parents. We use the Job Order Crossover described in [40]. The next step is to improve every new generated chromosome with TS. For selecting chromosomes for the next generation, we choose the two best individuals from each group of two parents and their two offspring. To achieve similar running times, GA+TS was given a population of 70 chromosomes and the stopping criterion was set at 10 generations without improving the best solution. We have tried several configurations for running GA+TS and TS, with similar or worse results than those described here.

Table 2 shows the results of these experiments, comparing SSPR, GA+TS and TS. We group the instances in two blocks of 25 instances, and for each method we show the RPD of the best and average results in 10 runs.

Overall, GA+TS obtains lower RPD than TS in both benchmarks, especially in BTS25-TW20. The best method is SSPR, obtaining the lowest RPD by far in both benchmarks. The average makespan obtained by SSPR is equal or better than that obtained by GA+TS or by TS alone in every instance. TS obtained an average makespan about 11% worse than that of SSPR, and GA+TS about 8% worse. From these results, we concluded that the combination of the three metaheuristics is better than both TS and GA+TS.

In these runs we have observed that path relinking is much more efficient than traditional crossover operators in the presence of maximum time lags. This is because

ACCEPTED MANUSCRIPT

solutions created with standard crossover operators usually violate many maximum time lag constraints and it is difficult to restore the feasibility of these schedules. On the opposite, with path relinking we have control over the generated solutions along the trajectory, and we can select at the end of the procedure a feasible solution, or at least one that violates a small number of maximum time lag constraints.

7.5. Comparison with the state-of-the-art in the SDST-JSPTL

As we have pointed, there are few papers that tackle the SDST-JSPTL. To our knowledge, the most representative approach is the Constraint-based Iterative Sampling Procedure (ISP) proposed by Oddi et al. in [31]. So we choose this method to compare with SSPR. ISP is implemented in CMU Common Lisp Ver.20a and run on a AMD Phenom II X4 Quad 3.5 Ghz under Linux Ubuntu 10.04.1., with a maximum CPU time limit set to 800 seconds for each run. Detailed results are provided online using three different configurations for their algorithm[2]. Here we are considering the best and the average makespan from those three runs.

SSPR is implemented in C++ and our experiments were carried out on a Intel Core i5-2450M CPU at 2.50GHz with 4GB of RAM, running in Windows 8 Professional. Since SSPR is a stochastic algorithm, we have run it 10 times on each instance, recording the best and average solutions of the 10 runs. The average number of iterations of SSPR in the 50 instances was 976 (the stop criterion was set at 500 consecutive iterations without improvement).

Table 3 summarizes the results of the experiments from the BTS25-TW15 and BTS25-TW20 benchmarks. The best and average makespan values for each instance are reported, from 3 and 10 runs with ISP and SSPR respectively. The time taken (in seconds) of a single run of SSPR is also given. Additionally, we report the average RPD (Relative Percentage Error) for each method, calculated using the best known result from both ISP and SSPR. We mark in bold the best known solutions, and we mark with a "(*)" when SSPR improves the previous best known solution.

Overall, compared to ISP, SSPR establishes new best solutions for 27 instances, reaches the same best known solution for 21 instances and only for the instances dataps15Max10 and dataps15Max14 the solution reached is worse than the current best known solution. Regarding the average makespan, SSPR is better than ISP in 27 instances, worse in 9 instances and they are equal in 14. Moreover, the average value obtained by SSPR in 25 instances is better than the best solution reached by ISP.

In the first benchmark, ISP achieved a RPD of 4.64% for the best results and of 5.77% for the average results, while SSPR achieved a RPD of 0.35% for the best solutions and 4.71% for the average values. In the second benchmark, ISP achieved a RPD of 4.41% for the best results and of 5.15% for the average results, while SSPR achieved a RPD of 0.00% for the best solutions and 1.66% for the average values.

To further assess the quality of the proposed method, we have done some statistical tests to analyze differences between SSPR and ISP. As we have multiple-problem analysis, we used non-parametric statistical tests. First, we have run a Shapiro-Wilk test to confirm the non-normality of the data. Then we have used paired Wilcoxon signed rank test to compare the medians of the RPD values between SSPR and ISP. In these tests, the level of confidence used was 95% and the alternative hypothesis was "the difference between the errors of the SSPR and ISP is smaller than 0". The p-value obtained with the test is 0.0006912, therefore we can conclude that there exist statistically significant differences between SSPR and ISP in the set of 50 instances considered.

Additionally, the CPU time of SSPR (between 4 and 37 seconds per run depending on the instance) is lower than that of ISP (800 seconds per run). In any case, CPU times are not directly comparable due to the differences in programming languages, operating systems and target machines.

7.6. Comparison with the state-of-the-art in the JSPTL

In the JSPTL it is easier to compare with the state-of-the-art, since there exists more works. However, there are few results reported for benchmarks with positive maximum time lag constraints, as most papers focus on the "no wait" case (all minimum and maximum time lags with value 0).

Caumond et al. introduced a memetic algorithm (MA) in [22]. It is implemented on Delphi 6.0 package and the experiments were carried out on a 1.8GHz computer under Windows XP with 512MB of memory. They perform 4 runs for each instance.

Artigues et al. proposed a branch and bound (BB) procedure [23]. Detailed results of BB are presented in the technical report [61]. It is coded in Ada 95 using GNAT 4.4.1 compiler, and the experiments were performed on a 2.33 GHz computer with 4GB of RAM running under Linux Red Hat 4.4.1-2. As the method is deterministic, the authors perform only one run for each instance.

Grimes and Hebrard proposed a constraint-programming approach (CP) in [24]. Detailed results are reported in the web[3] [4]. In particular, they report results from their complete algorithm and from the initial dichotomic search phase for finding the initial solution. In this second case the run time is much more reduced, and comparable to that of SSPR. CP was run on an Intel Xeon

[2] http://pst.istc.cnr.it/∼angelo/bts25tw/

[3] http://homepages.laas.fr/ehebrard/Experiment_(CP_2011)/Results/Results.html

[4] http://4c.ucc.ie/∼dgrimes/JSSP/tljspres.htm

Accepted manuscript

ACCEPTED MANUSCRIPT

Table 3: Comparison with the state-of-the-art in the SDST-JSPTL.

Benchmark BTS25-TW15						Benchmark BTS25-TW20					
Instance	ISP		SSPR			Instance	ISP		SSPR		
	Best	Avg.	Best	Avg.	t(s)		Best	Avg.	Best	Avg.	t(s)
dataps15Max01	**808**	808	**808**	808	8	dataps2Max01	**808**	808	**808**	808	5
dataps15Max02	**795**	795	**795**	795	7	dataps2Max02	**795**	795	**795**	795	4
dataps15Max03	**771**	816	**771**	771	7	dataps2Max03	**771**	771	**771**	771	5
dataps15Max04	**730**	730	**730**	730	6	dataps2Max04	**730**	730	**730**	730	5
dataps15Max05	**715**	715	**715**	715	6	dataps2Max05	**715**	715	**715**	715	4
dataps15Max06	1325	1328	**1303(*)**	1320	22	dataps2Max06	1191	1197	**1076(*)**	1109	16
dataps15Max07	**973**	973	**973**	1001	23	dataps2Max07	**973**	973	**973**	973	8
dataps15Max08	**1006**	1006	**1006**	1182	19	dataps2Max08	**999**	999	**999**	999	12
dataps15Max09	**1101**	1142	**1101**	1119	16	dataps2Max09	**1101**	1101	**1101**	1101	8
dataps15Max10	**1122**	1122	1196	1215	17	dataps2Max10	**1068**	1084	**1068**	1068	11
dataps15Max11	2234	2244	**2018(*)**	2152	37	dataps2Max11	1945	1965	**1864(*)**	1906	28
dataps15Max12	1922	1926	**1805(*)**	1875	35	dataps2Max12	1656	1661	**1486(*)**	1582	33
dataps15Max13	2039	2049	**1768(*)**	1889	34	dataps2Max13	1790	1800	**1579(*)**	1624	27
dataps15Max14	**1927**	2023	1968	2025	31	dataps2Max14	1726	1736	**1619(*)**	1626	21
dataps15Max15	2317	2349	**2106(*)**	2249	30	dataps2Max15	2073	2111	**1909(*)**	2013	29
datapss15Max06	1369	1371	**1156(*)**	1211	23	datapss2Max06	1236	1256	**1156(*)**	1156	11
datapss15Max07	1170	1205	**1100(*)**	1248	19	datapss2Max07	1162	1169	**1100(*)**	1118	14
datapss15Max08	1230	1230	**1123(*)**	1299	25	datapss2Max08	**1123**	1149	**1123**	1189	12
datapss15Max09	**1162**	1187	**1162**	1201	14	datapss2Max09	**1162**	1162	**1162**	1162	8
datapss15Max10	**1170**	1170	**1170**	1204	16	datapss2Max10	1179	1203	**1168(*)**	1168	11
datapss15Max11	1895	1908	**1775(*)**	1817	35	datapss2Max11	1749	1758	**1512(*)**	1605	33
datapss15Max12	1573	1600	**1456(*)**	1515	34	datapss2Max12	1477	1483	**1389(*)**	1407	21
datapss15Max13	1754	1765	**1606(*)**	1661	29	datapss2Max13	1599	1606	**1463(*)**	1491	25
datapss15Max14	1769	1781	**1680(*)**	1714	30	datapss2Max14	1606	1644	**1560(*)**	1600	21
datapss15Max15	1874	1893	**1732(*)**	1818	31	datapss2Max15	1739	1758	**1619(*)**	1643	24
Average RPD	4.64	5.77	0.35	4.71			4.41	5.15	0.00	1.66	

Values in **bold** are best known solutions, **(*)** improves previous best known solution.

Accepted manuscript

ACCEPTED MANUSCRIPT

Table 4: Comparison of BB, MA and SSPR on JSPTL instances (I)

Instance	Opt.	BB			MA		SSPR		
		LB	UB	t(s)	best	t(s)	best	avg	t(s)
la01_0_0,5	758	758	**758**	436	867	149	**758**	758	7
la01_0_1	683	683	**683**	89	723	164	**683**	683	5
la01_0_2	666	666	**666**	80	**666**	86	**666**	666	4
la02_0_0,5	742	742	**742**	661	872	80	**742**	751.6	7
la02_0_1	686	686	**686**	390	723	150	**686**	686	5
la02_0_2	673	673	**673**	788	683	167	**673**	673	5
la03_0_0,5	679	679	**679**	930	685	211	**679**	679.6	8
la03_0_1	640	640	**640**	282	641	206	**640**	640.0	7
la03_0_2	630	630	**630**	223	648	150	631	634.6	5
la04_0_0,5	703	703	**703**	724	769	104	**703**	712.7	8
la04_0_1	646	646	**646**	373	662	151	**646**	646.6	7
la04_0_2	619	619	**619**	61	631	83	**619**	619	4
la05_0_0,5	622	622	**622**	417	678	135	**622**	623.1	6
la05_0_1	593	593	**593**	366	615	92	**593**	593	5
la05_0_2	593	593	**593**	96	**593**	36	**593**	593	4
la06_0_0,5	1003	926	1471	526	1153	545	**1006**	1052.0	14
la06_0_1	926	926	1391	524	1086	1117	**926**	947.3	18
la06_0_3	926	926	1391	524	1101	405	**926**	926	7
la06_0_10	926	926	927	707	**926**	14	**926**	926	4
la07_0_0,5	953	869	1430	529	1132	573	**982**	1003.1	12
la07_0_1	896	869	1065	754	1009	532	**906**	923.1	16
la07_0_3	890	869	1079	659	975	477	**890**	890	6
la07_0_10	890	869	1123	518	**890**	39	**890**	890	6
la08_0_0,5	984	863	1454	529	1124	1199	**1014**	1042.1	16
la08_0_1	892	863	1052	587	1013	541	**897**	921.1	13
la08_0_3	863	863	1052	587	1013	544	**863**	863	6
la08_0_10	863	863	**863**	260	**863**	17	**863**	863	5

Values in **bold** indicate the best result of the algorithms considered in the table.

Accepted manuscript

ACCEPTED MANUSCRIPT

Table 5: Comparison of BB, MA and SSPR on JSPTL instances (II)

Instances	BKS	BB			MA			SSPR		
		LB	UB	t(s)	best	avg	t(s)	best	avg	t(s)
la01_0_0	971	971	**971**	1877	975	1015.25	602	**971**	983.10	8
la02_0_0	937	859	1082	765	**937**	968.25	476	961	968.60	5
la03_0_0	820	805	834	1441	**820**	872.50	405	**820**	829.00	6
la04_0_0	887	537	1027	507	911	923.25	676	**888**	896.10	6
la05_0_0	777	715	836	746	818	842.75	235	**777**	781.60	7
la06_0_0	1248	926	1770	25	1305	1399.25	1817	**1279**	1349.80	14
la07_0_0	1172	869	1536	530	1282	1332.25	2594	**1172**	1238.00	11
la08_0_0	1244	863	1640	528	1312	1382.75	2369	**1306**	1339.80	11
la09_0_0	1358	951	1859	530	1547	1574.25	2181	**1358**	1439.30	11
la10_0_0	1287	958	1666	526	1333	1429.75	1386	**1287**	1351.00	10
la11_0_0	1627	1222	2323	572	1875	1937.25	6185	**1760**	1812.40	19
la12_0_0	1414	1039	2011	577	1594	1661.00	5809	**1483**	1577.60	21
la13_0_0	1592	1150	2219	583	1799	1851.50	8888	**1737**	1779.20	21
la14_0_0	1578	1292	2146	585	1960	2022.00	5279	**1749**	1833.80	18
la15_0_0	1689	1207	2276	573	1928	2011.50	7722	**1773**	1850.20	19
la16_0_0	1575	1285	1908	1030	1833	1918.25	1503	**1604**	1661.30	20
la17_0_0	1371	683	1776	583	1591	1657.75	1540	**1398**	1452.60	20
la18_0_0	1417	623	2005	578	1790	1806.75	1710	**1492**	1578.90	15
la19_0_0	1482	685	2066	579	1831	1897.00	1069	**1482**	1537.80	20
la20_0_0	1526	744	2300	582	1828	1867.25	1739	**1526**	1606.00	17

Values in **bold** indicate the best result of the algorithms considered in the table.

2.66GHz machine with 12GB of RAM on Fedora 9. They perform 10 runs for each instance. Additionally, Grimes and Hebrard in [24] also adapted a model written by Chris Beck for Ilog Scheduler (version 6.3) to problems featuring time lag constraints. This model was used to showcase the SGMPCS algorithm [62]. They used two strategies, denoted as Texture-Luby and Texture-Geom+Guided. In this case they use the same machine but they only report 5 runs for each instance.

As in [22] Caumond et al. only report results on some particular instances with positive time lags, we firstly show detailed results on these instances in table 4. In the first two columns we show the name of the instance and the optimum value. Then the results from BB: lower bound, upper bound and time taken in seconds. And finally results from MA and SSPR: best makespan, average makespan and time taken. We mark in bold the best solution obtained for an instance with the three methods. We can see that BB obtains the optimum solution in the smaller la01 to la05 instances, but obtains worse results in the bigger ones. Our method outperforms BB and MA, as we were able to obtain the best solution of the three algorithms in 26 of the 27 instances, while BB reached it in 16 instances, and MA in only 5 instances. Moreover, the time taken by SSPR is much lower than that of BB or MA.

Table 5 shows results on some instances of the special "no wait" case. We report the best known makespan, not the optimum value, as for some of these instances the optimum is not known. As we can see SSPR was able to obtain the best solution of the three algorithms in 19 of the 20 instances, while BB reached it in only 1 instance, and MA in 2 instances. Again, the the time taken by SSPR is much lower than that of BB and MA.

Table 6: Comparison of BB and SSPR on JSPTL instances (III)

Instances	BB		SSPR			
	best	t(s)	best	avg	worst	t(s)
la[1,25]_0_0	32.13	730	3.89	8.52	13.57	18
la[1,25]_0_0.5	41.83	642	3.78	6.80	10.20	23
la[1,25]_0_1	32.82	603	0.97	2.73	4.71	23
la[1,25]_0_3	17.16	676	0.00	0.04	0.13	8
la[1,25]_0_10	12.44	614	0.00	0.02	0.06	5
Averages	27.27	653	1.73	3.62	5.74	16

In [61] Artigues et al. report detailed results for BB in 125 instances of the benchmark, therefore it is easier to make a comparison. Table 6 shows RPD values averaged for each group of instances with the same value of β. SSPR obtains much better results than BB, even in the worst of the 10 runs, with a RPD of 5.74, against the RPD of 27.27 obtained by BB. Moreover, the average run time of 16 seconds used by SSPR is much lower than the average run time of 653 seconds used by BB.

To further compare SSPR with BB and MA, as done in Section 7.5, we have run a Shapiro-Wilk test to confirm the non-normality of the data and a then a paired Wilcoxon signed rank test to compare the medians of the RPD val-

Accepted manuscript

ACCEPTED MANUSCRIPT

Table 7: Comparison of SSPR with Texture and CP methods on JSPTL instances (IV)

Instance Sets	Texture						CP						SSPR		
	Luby			Geom+Guided			Complete			Dich.phase					
	best	avg	t(s)	best	avg	t(s)	best	avg	t(s)	best	avg	t(s)	best	avg	t(s)
la[1,40]_0_0	26.15	32.37	3246	16.89	23.81	3181	0.18	2.10	2888	1.47	4.50	114	9.39	15.07	49
la[1,40]_0_0.25	23.67	29.75	3151	12.64	18.31	2907	0.37	2.03	2364	1.99	4.56	90	12.64	17.97	64
la[1,40]_0_0.5	20.49	25.51	2877	6.05	10.00	2729	0.13	1.75	2254	2.05	4.82	95	7.25	10.94	65
la[1,40]_0_1	16.28	23.88	2593	1.60	4.10	2310	0.45	2.03	2271	2.57	5.32	63	1.95	4.58	72
la[1,40]_0_2	7.55	10.51	1555	0.29	1.22	1228	0.44	1.45	1280	1.89	3.50	45	0.36	1.35	47
la[1,40]_0_3	3.48	4.45	1154	0.003	0.38	720	0.32	0.66	1102	0.99	1.88	37	0.07	0.28	31
la[1,40]_0_10	0.11	0.12	371	0.002	0.06	229	0.01	0.09	347	0.13	0.32	21	0.01	0.07	10
Averages	13.96	18.09	2135	5.35	8.27	1901	0.27	1.44	1786	1.58	3.56	66	4.53	7.18	48

ues between SSPR and the other methods. For comparing SSPR with BB we have used the set of 125 instances for which the authors provide results (see Table 6). The p-value obtained from the test is $<$ 2.2e-16, therefore we can conclude that there exist statistically significant differences between SSPR and BB. For comparing with MA we have used the set of 47 instances reported in Tables 4 and 5. In this case the p-value obtained is 2.442e-08, hence there also exist statistically significant differences between SSPR and MA.

Finally we also compare SSPR with the Texture-Luby, Texture-Geom+Guided and the CP method of Grimes and Hebrard [24], both with the complete algorithm and with the results after the dichotomic phase alone. Table 7 shows the RPD values averaged for each group of 40 instances. For each method we show the RPD of the best makespan, the RPD of the average makespan and the running time in seconds of a single run. SSPR outperforms Texture-Luby, obtaining better RPD values in all 7 benchmarks using a runtime 40 times more reduced. Additionally, SSPR obtains an average RPD comparable to that of Texture-Geom+Guided in a much more reduced run time. Considering the average results in 10 runs, the average RPD for the 280 instances is 7.18 for SSPR and 8.27 for Texture-Geom+Guided, in an average runtime of 48 and 1901 seconds respectively.

However, the complete constraint programming approach of [24] outperforms SSPR in terms of average RPD in 4 of the 7 benchmarks, but using an average run time more than 30 times higher. If we compare our method with the dichotomic phase alone of the constraint programming approach (arguably the most fair comparison in terms of running time), SSPR outperforms CP in 4 of the 7 benchmarks. In particular SSPR obtains a better average RPD in the instances with $\beta = 1, 2, 3, 10$ and a worse average RPD with $\beta = 0, 0.25, 0.5$. Therefore CP is better in the special "no wait" case and when maximum time lags are very tight, while SSPR is better with looser time lags.

In this case, to avail these results with statistical tests, as the number of instances is very high and the results depend highly on the set of instances considered, we have decided to run a paired Wilcoxon signed rank test for each set of 40 instances and for each method, therefore a total of 28 tests using a level of confidence of 95%. The results of these tests show that SSPR is significantly better than Texture-Luby in sets with $\beta = 0, 0.25, 0.5, 1, 3$, significantly better than Texture-Geom+Guided in the set with $\beta = 0$ and significantly better than the dichotomic phase alone of CP in sets with $\beta = 1, 2, 3$. If we decrease the level of confidence to 90% we find that SSPR is also significantly better than Texture-Luby in the set with $\beta = 2$, better than the Complete CP approach in the set with $\beta = 3$ and also better than the dichotomic phase alone of CP in the set with $\beta = 10$.

Table 8: New best solutions found with SSPR in JSPTL instances

Instance	New best solution
la21_0_1	1178
la23_0_1	1164
la27_0_2	1303
la28_0_2	1234
la29_0_2	1241
la33_0_2	1801
la33_0_3	1719
la29_0_10	1159

SSPR established new best solutions for 6 instances of this benchmark, which is a remarkable performance considering that most of the methods considered in this experimental study used much larger computation times. Moreover, in preliminary experiments with different configurations we have also found 2 additional new best solutions. These new upper bounds are summarized in Table 8 and were taken into account for calculating the RPDs.

Additionally, regarding run times in the JSPTL benchmarks we have to notice that SSPR is at disadvantage, as it does all the necessary setup calculations even when they are zero, as it occurs in these instances. Detailed results of SSPR over these instances are available on the web[5].

[5]Repository section in `http://www.di.uniovi.es/iscop`

Accepted manuscript

ACCEPTED MANUSCRIPT

8. Conclusions

We have proposed a new algorithm termed SSPR to minimize the makespan in the job shop scheduling problem with minimum and maximum time lags and sequence-dependent setup times (SDST-JSPTL). This algorithm combines scatter search, path relinking and tabu search. First, we have defined a graph model for the solutions and extended the algorithm proposed by Caumond et al. in [22] for calculating schedules. We have also extended the heuristic proposed by Artigues et al. in [23] for creating initial solutions. Based on the graph model, we propose and analyze a novel neighborhood structure which is embedded into the tabu search and considers a different set of moves depending on the feasibility of the current solution. When it is feasible it considers moves designed to improve the makespan, otherwise it makes moves aimed to restore feasibility. This structure is complemented with an algorithm to estimate the neighbors' makespan, which is also proposed in this paper.

The SSPR algorithm has been extensively evaluated on conventional instances on both the SDST-JSPTL and the JSPTL (i.e., without setup times), and compared with the state-of-the-art. The results are very competitive in a reduced computational time, establishing new upper bounds for a number of instances in the benchmarks considered. In the SDST-JSPTL, SSPR obtained significantly better results than the Constraint-based Iterative Sampling Procedure proposed in [31]. In the JSPTL, SSPR also obtained significantly better results than the memetic algorithm proposed in [22] and the branch and bound proposed in [23] in a more reduced computational time. However, compared to the constraint programming method proposed by Grimes and Hebrard in [24], SSPR outperforms it in instances with loose time lags, but shows lower efficiency when all the maximum time lags of the instance are very tight. Overall, we can conclude that local search based metaheuristics may be very efficient in solving problems with maximum time lags, which are typically very challenging problems.

We believe that the main reasons for the good performance of SSPR are the combination of the diversification provided by the scatter search and path relinking shell combined with the intensification provided by the tabu search, strengthened with the fact that the proposed neighborhood was specifically tailored to dealing with setup times and minimum and maximum time lags at the same time. We have also seen that the path relinking procedure is very effective in this particular problem compared to standard crossover operators.

Acknowledgements

We would like to thank Diarmuid Grimes for providing us with his detailed results and insights. Miguel A. González and Ramiro Varela were supported by grant MEC-FEDER TIN2010-20976-C02-02 and FICYT grant FC-13-COF13-035. Miguel A. González was also supported by grant "José Castillejo" (reference CAS12/00125). Angelo Oddi and Riccardo Rasconi were supported by MIUR Flagship Initiative FdF-SP1-T2.1 Project GECKO Genetic Evolutionary Control Knowledge-based mOdule. Also, we thanks the anonymous referees by their comments and suggestions which have contributed to make the paper clearer.

References

[1] B. Giffler, G.L. Thompson, Algorithms for solving production scheduling problems, Operations Research 8 (1960) 487–503.

[2] R. Qing-dao-er-ji, Y. Wang, A new hybrid genetic algorithm for job shop scheduling problem, Computers & Operations Research 39 (2012), 2291–2299.

[3] D.A. Wismer, Solution of the flowshop-scheduling Problem with no intermediate queues, Operations Research 20 (3) (1972), 689–697.

[4] C. Rajendran, A no-wait flowshop scheduling heuristic to minimize makespan, The Journal of the Operational Research Society 45 (4) (1994), 472–478.

[5] E. Nowicki, C. Smutnicki, An advanced tabu search algorithm for the job shop problem, Journal of Scheduling 8 (2005), 145–159.

[6] P. Brucker, O. Thiele, A branch and bound method for the general-job shop problem with sequence-dependent setup times, Operations Research Spektrum 18 (1996), 145–161.

[7] W. Cheung, H. Zhou, Using genetic algorithms and heuristics for job shop scheduling with sequence-dependent setup times, Annals of Operations Research 107 (2001), 65–81.

[8] E. Balas, N. Simonetti, A. Vazacopoulos, Job shop scheduling with setup times, deadlines and precedence constraints, Journal of Scheduling 11 (2008), 253–262.

[9] C. Artigues, D. Feillet, A branch and bound method for the job-shop problem with sequence-dependent setup times, Annals of Operations Research 159 (1) (2008), 135–159.

[10] C.R. Vela, R. Varela, M.A. González, Local search and genetic algorithm for the job shop scheduling problem with sequence dependent setup times, Journal of Heuristics 16 (2010), 139–165.

[11] M.A. González, C.R. Vela, R. Varela, A competent memetic algorithm for complex scheduling, Natural Computing 11 (2012), 151–160.

[12] P. Van Laarhoven, E. Aarts, K. Lenstra, Job shop scheduling by simulated annealing, Operations Research 40 (1992), 113-125.

[13] A. Oddi, R. Rasconi, A. Cesta, S.F. Smith, Iterative-sampling search for job shop scheduling with setup times, in: COPLAS 2009 Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (2009), 27–33.

[14] M.A. González, C.R. Vela, I. González-Rodríguez, R. Varela, Lateness minimization with tabu search for job shop scheduling problem with sequence dependent setup times, Journal of Intelligent Manufacturing 24 (4) (2013), 741–754.

[15] E.D. Wikum, D.C. Llewellyin, G.L. Nemhauser, One-machine generalized precedence constrained scheduling problem, Operations Research Letters 16 (1994), 87–99.

[16] P. Brucker, T. Hilbig, J. Hurink, A branch and bound algorithm for a single machine scheduling with positive and negative time-lags, Discrete Applied Mathematics 94 (1999), 77–99.

[17] J. Hurink, J. Keuchel, Local search algorithms for a single-machine scheduling problem with positive and negative time-lags, Discrete Applied Mathematics 112 (1-3) (2001), 179–197.

[18] J. Fondrevelle, A. Oulamara, M.C. Portmann, Permutation flowshop scheduling problems with maximal and minimal time lags, Computers & Operations Research 33 (6) (2006), 1540–1556.

Accepted manuscript

ACCEPTED MANUSCRIPT

[19] E. Dhouib, J. Teghem, T. Loukil, Lexicographic optimization of a permutation flow shop scheduling problem with time lag constraints, International Transactions in Operational Research 20 (2) (2013), 213–232.
[20] V. Botta-Genoulaz, Hybrid flow shop scheduling with precedence constraints and time lags to minimize maximum lateness, International Journal of Production Economics 64 (2000), 101–111.
[21] X. Zhang, Scheduling with Time Lags, PhD Thesis, Erasmus Research Institute of Management (ERIM). Erasmus University Rotterdam (EPS-2010-206-LIS) (2010).
[22] A. Caumond, P. Lacomme, N. Tchernev, A memetic algorithm for the job-shop with time-lags, Computers & Operations Research 35 (2008), 2331–2356.
[23] C. Artigues, M.J. Huguet, P. Lopez, Generalized disjunctive constraint propagation for solving the job shop problem with time lags, Engineering Applications of Artificial Intelligence 24 (2011), 220–231.
[24] D. Grimes, E. Hebrard, Models and strategies for variants of the job shop scheduling problem, in: CP 2011. LNCS 6876, Springer-Verlag (2011), 356–372.
[25] P. Lacomme, N. Tchernev, M.J. Huguet, Dedicated constraint propagation for job-shop problem with generic time-lags, in: Proceedings of the IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA 2011) (2011), 1–7.
[26] M.J. Huguet, P. Lacomme, N. Tchernev, Job-shop with generic time lags: an heuristic based approach, Research Report LIMOS/RR-12-06, France (2012).
[27] M. Bartusch, R.H. Mohring, F.J. Rademacher, Scheduling project networks with resource constraints and time windows, Annals of Operations Research 16 (1988), 201–240.
[28] K. Brinkmann, K. Neumann, Heuristic procedures for resource-constrained project scheduling with minimal and maximal time lags: the minimum project-duration and resource-levelling problems, Journal of Decision Systems 5 (1996), 129–156.
[29] R. Kolisch, R. Padman, An integrated survey of project scheduling, OMEGA International Journal of Management Science 29 (3) (2001), 249–272.
[30] K. Neumann, C. Schwindt, J. Zimmermann, Project scheduling with time windows and scarce resources, Springer (2002).
[31] A. Oddi, R. Rasconi, A. Cesta, S.F. Smith, Solving job shop scheduling with setup times through constraint-based iterative sampling: an experimental analysis, Annals of Mathematics and Artificial Intelligence 62 (2011), 371–402.
[32] S. Meeran, M.S. Morshed, A hybrid genetic tabu search algorithm for solving job shop scheduling problems: a case study, Journal of Intelligent Manufacturing 23 (2012), 1063–1078.
[33] C.Y. Zhang, P.G. Li, Y.Q. Rao, Z.L. Guan, A very fast TS/SA algorithm for the job shop scheduling problem, Computers & Operations Research 35 (2008), 282–294.
[34] J.C. Beck, T.K. Feng, J.P. Watson, Combining constraint programming and local search for job-shop scheduling, Informs Journal on Computing 23 (2011), 1–14.
[35] M.A. González, C.R. Vela, R. Varela, I. González-Rodríguez, An advanced scatter search algorithm for solving job shops with sequence dependent and non-anticipatory setups, AI Communications, In press (2014).
[36] E. Taillard, Parallel taboo search techniques for the job shop scheduling problem, ORSA Journal on Computing 6 (1993), 108–117.
[37] E. Nowicki, C. Smutnicki, A fast taboo search algorithm for the job shop scheduling problem, Management Science 42 (1996), 797–813.
[38] M. Dell'Amico, M. Trubian, Applying tabu search to the job-shop scheduling problem, Annals of Operational Research 41 (1993), 231-252.
[39] E. Balas, A. Vazacopoulos, Guided local search with shifting bottleneck for job shop scheduling, Management Science 44 (2) (1998), 262–275.
[40] C. Bierwirth, A generalized permutation approach to jobshop scheduling with genetic algorithms, OR Spectrum 17 (1995), 87–92.
[41] G. Gallo, S. Pallotino, Shortest path methods: a unifying approach, Mathematical Programming Study 26 (1986), 38–64.
[42] M.A. González, C.R. Vela, R. Varela, A new hybrid genetic algorithm for the job shop scheduling problem with setup times, in: Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-2008), AIII Press (2008), 116–123.
[43] D.C. Mattfeld, Evolutionary search and the job shop: investigations on genetic algorithms for production scheduling, Springer-Verlag (1995).
[44] F. Glover, Heuristics for integer programming using surrogate constraints, Decision Sciences 8 (1) (1977), 156–166.
[45] A.S. Jain, A multi-level hybrid framework for the deterministic job-shop scheduling problem, PhD Thesis, Dept. of APEME, University of Dundee (1998).
[46] E. Nowicki, C. Smutnicki, Some aspects of scatter search in the flow-shop problem, European Journal of Operational Research 169 (2006), 654–666.
[47] T. Yamada, R. Nakano, Scheduling by genetic local search with multi-step crossover, in: Proceedings of Fourth International Conference On Parallel Problem Solving from Nature (PPSN IV) (1996), 960–969.
[48] A. Manikas, Y.L. Chang, A scatter search approach to sequence-dependent setup times job shop scheduling, International Journal of Production Research 47 (18) (2009), 5217–5236.
[49] M. Saravanan, A.N. Haq, A scatter search algorithm for scheduling optimisation of job shop problems, International Journal of Product Development 10 (2010), 259–272.
[50] F. Glover, A template for scatter search and path relinking, in: Artificial Evolution. Lecture Notes in Computer Science 1363, Springer (1998), 13–54.
[51] R.M. Aiex, S. Binato, M.G.C. Resende, Parallel GRASP with path-relinking for job shop scheduling, Parallel Computing 29 (2003), 393–430.
[52] M.M. Nasiri, F. Kianfar, A guided tabu search/path relinking algorithm for the job shop problem, International Journal on Advanced Manufacturing Technologies 58 (2012), 1105–1113.
[53] S. Jia, Z.H. Hu, Path-relinking tabu search for the multi-objective flexible job shop scheduling problem, Computers & Operations Research 47 (2014), 11–26.
[54] F. Glover, Tabu search–part I, ORSA Journal on Computing 1 (3) (1989), 190–206.
[55] F. Glover, Tabu search–part II, ORSA Journal on Computing 2 (1) (1989), 4–32.
[56] M.A. González, I. González-Rodríguez, C.R. Vela, R. Varela, An efficient hybrid evolutionary algorithm for scheduling with setup times and weighted tardiness minimization, Soft Computing 16 (12) (2012), 2097–2113.
[57] M. Resende, C. Ribeiro, F. Glover, R. Martí, in: Scatter search and path-relinking: fundamentals, advances, and applications. Handbook of Metaheuristics, International Series in Operations Research & Management Science 146, Springer (2010), 87–107.
[58] J.K. Hao, R. Dorne, P. Galinier, Tabu search for frequency assignment in mobile radio networks, Journal of Heuristics 4 (1) (1998), 47–62.
[59] S. Lawrence, Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement), Technical Report, Graduate School of Industrial Administration, Carnegie Mellon University (1984).
[60] F. Glover, M. Laguna, R. Marti, in: Advances in evolutionary computation: theory and applications. Chapter: Scatter search, Springer (2003), 519–537.
[61] C. Artigues, M.J. Huguet, P. Lopez, Generalized disjunctive constraint propagation for solving the job shop problem with time lags, Technical Report N.10373, LAAS-CNRS, Toulouse, France (2010).
[62] J.C. Beck, Solution-guided multi-point constructive search for job shop scheduling, Computers & Operations Research 35 (3) (2008), 906–915.

Accepted manuscript