Job Shop Scheduling with Routing Flexibility and Sequence Dependent Setup-Times

Angelo Oddi[1], Riccardo Rasconi[1], Amedeo Cesta[1], and Stephen F. Smith[2]

[1] Institute of Cognitive Science and Technology, CNR, Rome, Italy
angelo.oddi,riccardo.rasconi,amedeo.cesta@istc.cnr.it
[2] Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, USA sfs@cs.cmu.edu

Abstract. This paper presents a meta-heuristic algorithm for solving a job shop scheduling problem involving both sequence dependent setup-times and the possibility of selecting alternative routes among the available machines. The proposed strategy is a variant of the Iterative Flattening Search (IFS) schema. This work provides three separate results: (1) a constraint-based solving procedure that extends an existing approach for classical Job Shop Scheduling; (2) a new variable and value ordering heuristic based on temporal flexibility that take into account both sequence dependent setup-times and flexibility in machine selection; (3) an original relaxation strategy based on the idea of randomly breaking the execution orders of the activities on the machines with a activity selection criteria based on their proximity to the solution's critical path. The efficacy of the overall heuristic optimization algorithm is demonstrated on a new benchmark set which is an extension of a well-known and difficult benchmark for the Flexible Job Shop Scheduling Problem.

1 Introduction

This paper describes an iterative improvement approach to solve job-shop scheduling problems involving both *sequence dependent* setup-times and the possibility of selecting *alternative routes* among the available machines. Over the last years there has been an increasing interest in solving scheduling problems involving both setup-times and flexible shop environments [3, 2]. This fact stems mainly from the observation that in various real-word industry or service environments there are tremendous savings when setup times are explicitly considered in scheduling decisions. In addition, the possibility of selecting alternative routes among the available machines is motivated by interest in developing Flexible Manufacturing Systems (FMS) [25] able to use multiple machines to perform the same operation on a job's part, as well as to absorb large-scale changes, in volume, capacity, or capability.

The proposed problem, called in the rest of the paper Flexible Job Shop Scheduling Problem with Sequence Dependent Setup Times (SDST-FJSSP) is a generalization of the classical Job Shop Scheduling Problem (JSSP) where a given activity may be

Proceedings of the 18^{th} RCRA workshop on *Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (RCRA 2011).
In conjunction with IJCAI 2011, Barcelona, Spain, July 17-18, 2011.

processed on any one of a designated set of available machines and there are no setup-times. This problem is more difficult than the classical JSSP (which is itself *NP-hard*), since it is not just a sequencing problem; in addition to deciding how to sequence activities that require the same machine (involving sequence-dependent setup-times), it is also necessary to choose a *routing policy*, i.e., deciding which machine will process each activity. The objective remains that of minimizing *makespan*.

Despite this problem is often met in real manufacturing systems, not many papers take into account sequence dependent setup-times in flexible job-shop environments. On the other hand, a richer literature is available when setup-times and flexible job-shop environments are considered separately. In particular, on the side of setup-times a first reference work is [7], which relies on an earlier proposal presented in [6]. More recent works are [28] and [13], which propose effective heuristic procedures based on genetic algorithms and local search. In these works, the introduced local search procedures extend an approach originally proposed by [19] for the classical job-shop scheduling problem to the setup times case. A last noteworthy work is [5], which extends the well-known *shifting bottleneck* procedure [1] to the setup-time case. Both [5] and [28] have produced reference results on a previously studied benchmark set of JSSP with sequence dependent setup-times problems initially proposed by [7]. About the Flexible Job Shop Scheduling FJSSP an effective synthesis of the existing solving approaches is proposed in [14]. The core set of procedures which generate the best results include the genetic algorithm (GA) proposed in [10], the tabu search (TS) approach of [16] and the discrepancy-based method, called climbing depth-bound discrepancy search (CDDS), defined in [14]. Among the papers dealing with both sequence dependent setup times and flexible shop environments there is the work [23], which considers a shop type composed of pools of identical machines as well as two types of setup times: one modeling the transportation times between different machines (sequence dependent) and the other one modeling the required reconfiguration times (not sequence dependent) on the machines. The other work that deals with sequence dependent setup times and routing flexibility is [24], which considers a flow-shop environment with multi-purpose machines such that each stage of a job can be processed by a set of unrelated machines (the processing times of the jobs depend on the machine they are assigned to). [26] considers a problem similar to the previous one, where the jobs are composed by a single step, but setup-times are both sequence and machine dependent. Finally, [27] considers a job-shop problem with parallel identical machines, release times and due dates but sequence independent setup-times.

This paper focuses on a family of solving techniques referred to as Iterative Flattening Search (IFS). IFS was first introduced in [8] as a scalable procedure for solving multi-capacity scheduling problems. IFS is an iterative improvement heuristic designed to minimize schedule makespan. Given an initial solution, IFS iteratively applies two-steps: (1) a subset of solving decisions are randomly retracted from a current solution (*relaxation-step*); (2) a new solution is then incrementally recomputed (*flattening-step*). Extensions to the original IFS procedure were made in two subsequent works [17, 12] and more recently [20] have performed a systematic study aimed at evaluating the effectiveness of single *component strategies* within the same uniform software framework. The IFS variant that we propose relies at its core on a constraint-based solver. This

procedure is an extension of the SP-PCP procedure proposed in [21]. SP-PCP generates consistent orderings of activities requiring the same resource by imposing precedence constraints on a temporally feasible solution, using *variable* and *value* ordering heuristics that discriminate on the basis of temporal flexibility to guide the search. We extend both the procedure and these heuristics to take into account both sequence dependent setup-times and flexibility in machine selection. To provide a basis for embedding this core solver within an IFS optimization framework, we also specify an original relaxation strategy based on the idea of randomly breaking the execution orders of the activities on the machines with a activity selection criteria based on their *proximity* to the solution's critical path.

The paper is organized as follows. Section 2 defines the SDST-FJSSP problem and Section 3 introduces a CSP representation. Section 4 describes the core constraint-based search procedure while Section 5 introduces details of the IFS meta-heuristics. An experimental section (Section 6) describes the performance of our algorithm on a set of benchmark problems, and explains the most interesting results. Some conclusions end the paper.

2 The Scheduling Problem

The SDST-FJSSP entails synchronizing the use of a set of machines (or resources) $R = \{r_1, \ldots, r_m\}$ to perform a set of n activities $A = \{a_1, \ldots, a_n\}$ over time. The set of activities is partitioned into a set of nj jobs $\mathcal{J} = \{J_1, \ldots, J_{nj}\}$. The processing of a job J_k requires the execution of a strict sequence of n_k activities $a_i \in J_k$ and cannot be modified. All jobs are released at time 0. Each activity a_i requires the exclusive use of a *single resource* r_i for its entire duration chosen among a *set of available resources* $R_i \subseteq R$. No *preemption* is allowed. Each machine is available at time 0 and can process more than one operation of a given job J_k (*recirculation* is allowed). The processing time p_{ir} of each activity a_i depends on the selected machine $r \in R_i$, such that $e_i - s_i = p_{ir}$, where the variables s_i and e_i represent the start and end time of a_i. Moreover, for each resource r, the value st^r_{ij} represents the setup time between two generic activities a_i and a_j (a_j is scheduled immediately after a_i) requiring the same resource r, such that $e_i + st^r_{ij} \leq s_j$. As is traditionally assumed in the literature, the setup times st^r_{ij} satisfy the so-called *triangular inequality* (see [7, 4]). The triangle inequality states that, for any three activities a_i, a_j, a_k requiring the same resource, the inequality $st^r_{ij} \leq st^r_{ik} + st^r_{kj}$ holds. A *solution* $S = \{(\overline{s_1}, \overline{r_1}), (\overline{s_2}, \overline{r_2}), \ldots, (\overline{s_n}, \overline{r_n})\}$ is a set of pairs $(\overline{s_i}, \overline{r_i})$, where $\overline{s_i}$ is the assigned start time of a_i, $\overline{r_i}$ is the selected resource for a_i and all the above constraints are satisfied. Let C_k be the completion time for the job J_k, the *makespan* is the value $C_{max} = max_{1 \leq k \leq nj}\{C_k\}$. An *optimal* solution S^* is a solution S with the minimum value of C_{max}. The SDST-FJSSP is *NP-hard* since it is an extension of the JSSP problem [11].

3 A CSP Representation

There are different ways to model the problem as a *Constraint Satisfaction Problem* (CSP) [18]; here we use an approach similar to [21]. In particular, we focus on *assigning*

resources to activities, a distinguishing aspect of SDST-FJSSP and on *establishing sequence dependent setup time constraints* between pairs of activities that require the same resource, so as to eliminate all possible conflicts in the resource usage.

Let $G(A_G, J, X)$ be a graph where the set of vertices A_G contains all the activities of the problem together with two dummy activities, a_0 and a_{n+1}, respectively, representing the beginning (reference) and the end (horizon) of the schedule. Each activity a_i is labelled with the set of available resource choices R_i. J is a set of directed edges (a_i, a_j) representing the precedence constraints among the activities (job precedences constraints) and are labelled with the set of processing times $p_i r$ ($r \in R_i$) of the edge's source activity a_i. The set of undirected edges X represents the *disjunctive constraints* among the activities requiring the same resource r; there is an edge for each pair of activities a_i and a_j requiring the same resource r ($R_i = R_j = \{r\}$) and the related label represents the set of possible ordering between a_i and a_j: $a_i \preceq a_j$ or $a_j \preceq a_i$. Hence, in CSP terms, there are *two sets of decision variables*: (1) a variable x_i is defined for each activity a_i to select one resource for its execution, the domain of x_i is the set of available resource R_i: (2) A variable o_{ijr} is defined for each pair of activities a_i and a_j requiring the same resource r ($x_i = x_j = r$), which can take one of two values $a_i \preceq a_j$ or $a_j \preceq a_i$. It is worth noting that in considering either ordering we have to take into account the presence of sequence dependent setup times, which must be included when an activity a_i is executed on the same resource *before* another activity a_j. As we will see in the next sections, if the setup times satisfy the triangle inequality, the previous decisions for x_{ijr} can be represented as the following two temporal constraints: $e_i + st^r_{ij} \leq s_j$ (i.e. $a_i \preceq a_j$) or $e_j + st^r_{ji} \leq s_i$ (i.e. $a_j \preceq a_i$).

To support the search for a consistent assignment to the set of decision variables x_i and o_{ijr}, for any SDST-FJSSP we define the directed graph $G_d(V, E)$, called *distance graph*, which is an extended version of the graph $G(A_G, J, X)$. The set of nodes V represents time points, where tp_0 is the *origin* time point (the reference point of the problem), while for each activity a_i, s_i and e_i have their usual meaning. The set of edges E represents all the imposed temporal constraints, i.e., precedences and durations. In particular, for each activity a_i we impose the interval duration constraint $e_i - s_i \in [p_i^{min}, p_i^{max}]$, such that p_i^{min} (p_i^{max}) is the minimum (maximum) processing time according to the set of available resources R_i. Given two time points tp_i and tp_j, all the constraints have the form $a \leq tp_j - tp_i \leq b$, and for each constraint specified in the SDST-FJSSP instance there are two weighted edges in the graph $G_d(V, E)$; the first one is directed from tp_i to tp_j with weight b and the second one is directed from tp_j to tp_i with weight $-a$. The graph $G_d(V, E)$ corresponds to a *Simple Temporal Problem* (STP) and its consistency can be efficiently determined via shortest path computations; the problem is consistent if and only if no closed paths with negative length (or negative cycles) are contained in the graph G_d [9]. Thus, a search for a solution to a SDST-FJSSP instance *can proceed by repeatedly adding new precedence constraints into* $G_d(V, E)$ and recomputing shortest path lengths to confirm that $G_d(V, E)$ remains consistent.

A solution S is given as a affine graph $G_S(A_G, J, X_S)$, such that each undirected edge (a_i, a_j) in X is replaced with a directed edge representing one of the possible orderings between a_i and a_j: $a_i \preceq a_j$ or $a_j \preceq a_i$. In general the directed graph G_S rep-

resents a set of temporal solutions $(S_1, S_2, \ldots, S_n)$ that is, a set of assignments to the activities' start times which are consistent with the set of imposed constraints X_S. Let $d(tp_i, tp_j)$ ($d(tp_j, tp_i)$) designate the shortest path length in graph $G_d(V, E)$ from node tp_i to node tp_j (from node tp_j to node tp_i); then, the constraint $-d(tp_j, tp_i) \leq tp_j - tp_i \leq d(tp_i, tp_j)$ is demonstrated to hold [9]. Hence, the interval $[lb_i, ub_i]$ of time values associated with a given time variable tp_i respect to the *reference* point tp_0 is computed on the graph G_d as the interval $[-d(tp_i, tp_0), d(tp_0, tp_i)]$. In particular, given a STP, the following two sets of value assignments $S_{lb} = \{-d(tp_1, tp_0), -d(tp_2, tp_0), \ldots, -d(tp_n, tp_0)\}$ and $S_{ub} = \{d(tp_0, tp_1), d(tp_0, tp_2), \ldots, d(tp_0, tp_n)\}$ to the STP variables tp_i represent the so-called *earliest-time solution* and *latest-time solution*, respectively.

4 Basic Constraint-based Search

The proposed procedure for solving instances of SDST-FJSSP integrates a Precedence Constraint Posting (PCP) one-shot search for generating sample solutions and an Iterative Flattening meta-heuristic that pursues optimization. The one-shot step, similarly to the SP-PCP scheduling procedure (Shortest Path-based Precedence Constraint Posting) proposed in [21], utilizes shortest path information in $G_d(V, E)$ to guide the search process. Shortest path information is used in a twofold fashion to enhance the search process: to propagate problem constraints and to define variable and value ordering heuristics.

4.1 Propagation Rules

The first way to exploit shortest path information is by introducing conditions to remove infeasible values from the domains of the decision variables x_i, representing the assignment of resources to activities. Namely, for each activity a_i we relax the disjunctive duration constraint into the interval constraint $e_i - s_i \in [p_i^{min}, p_i^{max}]$, such that p_i^{min} (p_i^{max}) is the minimum (maximum) processing time according to the set of available resources R_i (i.e., the domain of the decision variable x_i). As the search proceeds, as soon as the interval of distance between the start time and the end time of a_i $[-d(s_i, e_i), d(e_i, s_i)]$ is updated, the duration $p_{ir} \notin [-d(s_i, e_i), d(e_i, s_i)]$ is removed from the domain of x_i and a new interval $[p_i^{min}, p_i^{max}]$ is recomputed accordingly. In case the domain of the decision variable x_i becomes *empty*, the search reaches a *failure* state.

The second way to exploit shortest path is by introducing new *Dominance Conditions* (which adapt those presented in [21] to the setup times case), through which problem constraints are *propagated* and mandatory decisions for promoting early pruning of alternatives are identified. The following concepts of $slack(e_i, s_j)$ and *co-*$slack(e_i, s_j)$ (complementary slack) play a central role in the definition of such new dominance conditions. Given two activities a_i, a_j both assigned to resource r, and the related interval of distances $[-d(s_j, e_i), d(e_i, s_j)]$ on the graph G_d, they are defined as follows:

- $slack^r(e_i, s_j) = d(e_i, s_j) - st_{ij}^r$ is the difference between the maximal distance $d(e_i, s_j)$ and the setup time st_{ij}^r. Hence, it provides a measure of the degree of

sequencing flexibility between a_i and a_j [3] taking into account the setup time constraint $e_i + st^r_{ij} \leq s_j$. If $slack^r(e_i, s_j) < 0$, then the ordering $a_i \preceq a_j$ is not feasible.

- $co\text{-}slack^r(e_i, s_j) = -d(s_j, e_i) - st^r_{ij}$ is the difference between the minimum possible distance between a_i and a_j, $-d(s_i, e_j)$, and the setup time st^r_{ij}; if $co\text{-}slack^r(e_i, s_j) \geq 0$, then there is no need to separate a_i and a_j, as the setup time constraint $e_i + st^r_{ij} \leq s_j$ is already satisfied.

In order not to overload the notation, in the rest of the paper the $slack$ and $co-slack$ elements will be presented without the resource r superscript.

For any pair of activities a_i and a_j that can **compete for the same resource** r ($R_i \cap R_j \neq \emptyset$), given the corresponding durations p_{ir} and p_{jr}, the Dominance Conditions, describing the four main possible cases of conflict, are defined as follows:

1. $slack(e_i, s_j) < 0 \wedge slack(e_j, s_i) < 0$
2. $slack(e_i, s_j) < 0 \wedge slack(e_j, s_i) \geq 0 \wedge co\text{-}slack(e_j, s_i) < 0$
3. $slack(e_i, s_j) \geq 0 \wedge slack(e_j, s_i) < 0 \wedge co\text{-}slack(e_i, s_j) < 0$
4. $slack(e_i, s_j) \geq 0 \wedge slack(e_j, s_i) \geq 0$

Condition 1 represents an *unresolvable conflict*. There is no way to order a_i and a_j taking into account the setup times st^r_{ij} and st^r_{ji}, without inducing a negative cycle in the graph $G_d(V, E)$. When Condition 1 is verified there are four different interesting sub-cases generated on the basis of the cardinality of the domain sets R_i and R_j.

a. $|R_i| = |R_j| = 1$: the search has reached a *failure* state;
b. $|R_i| = 1 \wedge |R_j| > 1$: the resource requirement r can be removed from R_j;
c. $|R_i| > 1 \wedge |R_j| = 1$: the resource requirement r can be removed from R_i;
d. $|R_i| > 1 \wedge |R_j| > 1$: the activities a_i and a_j cannot use the same resource r.

Conditions 2, and 3, alternatively, distinguish *uniquely resolvable conflicts*, i.e., there is only one feasible ordering of a_i and a_j when both the activities require r, and the decision of which constraint to post is thus unconditional. In the particular case where $|R_i| = |R_j| = 1$ the decision $a_j \preceq a_i$ is mandatory; if Condition 2 is verified, only $a_j \preceq a_i$ leaves $G_d(V, E)$ consistent. It is worth noting that the presence of the condition $co\text{-}slack(e_j, s_i) < 0$ entails that the minimal distance between the end time e_j and the start time s_i is shorter than the minimal required setup time st^r_{ji}; hence, we still need to impose the constraint $e_j + st^r_{ji} \leq s_i$. In other words, the $co\text{-}slack$ condition avoids the imposition of unnecessary precedence constraints for trivially solved conflicts. Condition 3 works similarly, and entails that only the $a_i \preceq a_j$ ordering is feasible. In case there is at least one activity with more than one resource option ($|R_i| > 1 \vee |R_j| > 1$), it is still possible to choose different resource assignments for a_i and a_j, and avoid posting a precedence constraint. Condition 3 works similarly, and entails that only the $a_i \preceq a_j$ ordering is feasible when $|R_i| = |R_j| = 1$.

Condition 4 designates a class of *resolvable conflicts* with more search options. In this case, when $|R_i| = |R_j| = 1$ both orderings between a_i and a_j remain feasible, and

[3] Intuitively, the higher is the degree of *sequencing flexibility*, the larger is the set of feasible assignments to the start times of a_i and a_j.

it is therefore necessary to perform a *search decision*. When there is at least one activity a_i or a_j with more than one resource option ($|R_i| > 1 \vee |R_j| > 1$), then there is also the possibility of choosing different resource assignment to a_i and a_j, and avoid to post a precedence constraint.

4.2 Heuristic Analysis

Shortest path information in G_d can also be exploited to define *variable* and *value* ordering heuristics for the decision variables x_i and o_{ijr} in all cases where no mandatory decisions are deduced from the propagation phase. The idea is to evaluate both types of decision variables (x_i and o_{ijr}) and select the one (independently of type) with the minimum heuristic evaluation. The selection of the variables is based on the *most constrained first* (MCF) principle and the selection of values follows the *least constraining value* (LCV) heuristic.

Ordering decision variables. We start to analyze the case of selecting an ordering decision variables o_{ijr}, under the hypothesis that both the activity a_i and a_j use the same resource $r \in R_i \cap R_j$. As stated above, in this context $slack(e_i, s_j)$ and $slack(e_j, s_i)$ provide measures of the degree of *sequencing flexibility* between a_i and a_j. More precisely, given a variable o_{ijr}, related to the pair (a_i, a_j), its heuristic evaluation is $VarEval(a_i, a_j) =$

$$\begin{cases} min\{\frac{slack(e_i,s_j)}{\sqrt{S}}, \frac{slack(e_j,s_i)}{\sqrt{S}}\} & \text{if } slack(e_i, s_j) \geq 0 \wedge slack(e_j, s_i) \geq 0 \\ slack(e_j, s_i) & \text{if } slack(e_i, s_j) < 0 \wedge slack(e_j, s_i) \geq 0 \\ slack(e_i, s_j) & \text{if } slack(e_i, s_j) \geq 0 \wedge slack(e_j, s_i) < 0. \end{cases}$$

where $S = \frac{min\{slack(e_i,s_j),slack(e_j,s_i)\}}{max\{slack(e_i,s_j),slack(e_j,s_i)\}}$ [4]. The *variable* ordering heuristic attempts to focus first on the most constrained conflict (a_i, a_j), that is, on the conflict with the least amount of temporal flexibility (i.e., the conflict that is closest to previous Condition 1.a).

As opposed to variable ordering, the *value* ordering heuristic attempts to resolve the selected conflict (a_i, a_j) by simply choosing the activity pair sequencing that retains the highest amount of temporal flexibility (least constrained value). Specifically, $a_i \preceq a_j$ is selected if $slack(e_i, s_j) > slack(e_j, s_i)$ and $a_j \preceq a_i$ is selected otherwise.

Resource decision variables. Decision variables x_i are also selected according to the MCF principle. Initially, all pairs of activities (a_i, a_j), such that ($|R_i| > 1 \vee |R_j| > 1$ and $R_i \cap R_j \neq \emptyset$)) undergo a double-key sorting, where the primary key is a heuristic evaluation based on resource flexibility and computed as $F_{ij} = 2(|R_i| + |R_j|) - |R_i \cap R_j|$, while the secondary key is the known $VarEval(a_i, a_j)$ heuristic, based on temporal flexibility[5]. Then, we select the pair (a_i^*, a_j^*) with the lowest value of the

[4] The $\sqrt{S}$ bias is introduced to take into account cases where a first conflict with the overall $min\{slack(e_i, s_j), slack(e_j, s_i)\}$ has a very large $max\{slack(e_i, s_j), slack(e_j, s_i)\}$, and a second conflict has two shortest path values just slightly larger than this overall minimum. In such situations, it is not clear which conflict has the least temporal flexibility.

[5] The resource flexibility F_{ij} increases with the size of the domains R_i and R_j, and decreases with the size of the set $R_i \cap R_j$, which is correlated to the possibility of creating resource conflicts.

pair $\langle F_{ij}, VarEval(a_i, a_j)\rangle$, where $VarEval(a_i, a_j)$ is computed for each possible resource $r \in R_i \cap R_j$. Finally, between x_i^* and x_j^* we select the variable whose domain of values has the lowest cardinality.

Value ordering on the decision variables x_i is also accomplished by using temporal flexibility measures. If R_i is the domain of the selected decision variable x_i, then for each resource $r \in R_i$, we consider the set of activities A_r already assigned to resource r and calculate the value $F_{min}(r) = min_{a_k \in A_r}\{VarEval(a_i, a_k)\}$. Then, for each resource r we evaluate the flexibility associated with the most critical pair (a_i, a_k), under the hypothesis that the resource r is assigned to a_i. The resource $r^* \in R_i$ which maximizes the value $F_{min}(r)$, and therefore allows a_i to retain maximal flexibility, is selected.

PCP(*Problem,* C_{max})
1. $S \leftarrow$ InitSolution(*Problem*, C_{max})
2. **loop**
3. Propagate(S)
4. **if** UnresolvableConflict(S)
5. **then return**(nil)
6. **else**
7. **if** UniquelyResolvableDecisions(S)
8. **then** PostUnconditionalConstraints(S)
9. **else begin**
10. $C \leftarrow$ChooseDecisionVariable(S)
11. **if** ($C = nil$)
12. **then return**(S)
13. **else begin**
14. $vc \leftarrow$ ChooseValueConstraint(S, C)
15. PostConstraint(S, vc)
16. **end**
17. **end**
18. **end-loop**
19. **return** S

Fig. 1. The PCP one-shot algorithm

4.3 The PCP Algorithm

Figure 1 gives the basic overall PCP solution procedure, which starts from an empty solution (Step 1) where the graphs G_d is initialized according to Section 3. Also, the procedure accepts a *never-exceed* value (C_{max}) of the objective function of interest, used to impose an initial *global* makespan to all the jobs. The PCP algorithm shown in Figure 1 analyses the decision variables x_i and o_{ijr} and, respectively, decides their *values* in terms of imposing a duration constraint on a selected activity or a setup time constraint (i.e., $a_i \preceq a_j$ or $a_j \preceq a_i$, see Section 3). In broad terms, the procedure in Figure 1 interleaves the application of Dominance Conditions (Steps 4 and 7) with variable and value ordering (Steps 10 and 14 respectively) and updating of the solution graph G_d (Steps 8 and 15) to conduct a single pass through the search tree. At each

IFS(S,$MaxFail$, γ)
begin
1. $S_{best} \leftarrow S$
2. $counter \leftarrow 0$
3. **while** ($counter \leq MaxFail$) **do**
4. RELAX(S, γ)
5. $S \leftarrow$PCP($S, C_{max}(S_{best})$)
6. **if** $C_{max}(S) < C_{max}(S_{best})$ **then**
7. $S_{best} \leftarrow S$
8. counter $\leftarrow$ 0
9. **else**
10. counter $\leftarrow$ counter + 1
11. **return** (S_{best})

end

Fig. 2. The IFS schema

cycle, a propagation step is performed (Step 3) by the function `Propagate`(S), which propagates the effects of posting a new solving decision (i.e., a setup time constraint) in the graph G_d. In particular, `Propagate`(S) updates the shortest path distances on the graph G_d. A solution S is found when the PCP algorithm finds a feasible assignment of resources $\overline{r_i} \in R_i$ to activities a_i ($i = 1 \ldots n$) and when none of the four dominance conditions is verified on S. In fact, when none of the four Dominance Conditions is verified (and the PCP procedure exits with success), for each resource r, the set of activities A_r assigned to r represents a total execution order. In addition, as the graph G_d represents a consistent Simple Temporal Problem (see Section 3), one possible solution to the problem is the *earliest-time solution*, such that $S = \{(-d(s_1, tp_0), \overline{r_1}), (-d(s_2, tp_0), \overline{r_2}), \ldots, (-d(s_n, tp_0), \overline{r_n})\}$.

5 The Optimization Metaheuristic

Figure 2 introduces the generic IFS procedure. The algorithm basically alternates relaxation and flattening steps until a better solution is found or a maximal number of iterations is executed. The procedure takes three parameters as input: (1) an initial solution S; (2) a positive integer $MaxFail$, which specifies the maximum number of consecutive non makespan-improving moves that the algorithm will tolerate before terminating; (3) a parameter γ, representing the selection probability of an activity for removal (*relaxing factor*), as explained in 5.1. After the initialization (Steps 1-2), a solution is repeatedly modified within the while loop (Steps 3-10) by applying the RELAX procedure (as explained in the following section), and the PCP procedure shown in Figure 1 used as flattening step. At each iteration, the RELAX step reintroduces the possibility of resource contention, and the PCP step is called again to restore resource feasibility. In the case a better makespan solution is found (Step 6), the new solution is saved in S_{best} and the *counter* is reset to 0. If no improvement is found within $MaxFail$ moves, the algorithm terminates and returns the best solution found.

5.1 Relaxation Procedure

The first part of the IFS cycle is the *relaxation step*, wherein a feasible schedule is relaxed into a possibly resource infeasible, but precedence feasible, schedule by retracting some scheduling decisions. Here we use a strategy similar to the one in [12] and called *chain-based relaxation*. Given the graph representation described above, each scheduling decision is either a *setup time constraint* between a pair of activities that are competing for the same resource capacity and/or a *resource assignment* to one activity. The strategy starts from a solution S and randomly *breaks* some total orders (or *chains*) imposed on the subset of activities requiring the same resource r. The relaxation strategy requires an input solution as a graph $G_S(A, J, X_S)$ which (Section 3) is a modification of the original precedence graph G that represents the input scheduling problem. G_S contains a set of additional *general* precedence constraints X_S which can be seen as a set of *chains*. Each chain imposes a total order on a subset of problem activities requiring the same resource.

The *chain-based relaxation* proceeds in two steps. Firstly, a subset of activities a_i is randomly selected from the input solution S, according to some criteria that will be explained below. The selection process is generally driven by a parameter $\gamma \in (0, 1)$ that indicates the probability that each activity has to be selected (γ is called the *relaxing factor*). For each selected activity, the resource assignment is removed and the original set of available options R_i is re-estabilished. Secondly, a procedure similar to CHAINING – used in [22] – is applied to the set of unselected activities. This operation is in its turn accomplished in three steps: (1) all previously posted setup time constraints X_S are removed from the solution S; (2) the unselected activities are sorted by increasing earliest start times of the input solution S; (3) for each resource r and for each unselected activity a_i assigned to r (according to the increasing order of start times), a_i's predecessor $p = pred(a_i, r)$ is considered and the setup time constraint related to the sequence $p \preceq a_i$ is posted (the dummy activity a_0 is the first activity of all the chains). This last step is iterated until all the activities are linked by the correct setup time constraints. Note that this set of unselected activities still represents a feasible solution to a scheduling sub-problem, which is represented as a graph G_S in which the randomly selected activities *float* outside the solution and thus re-create *conflict* in resource usage.

As anticipated above, we implemented two different mechanisms to perform the random activity selection process, respectively called *Random* and a *Slack-based*.

Random selection According to the random selection approach, at each solving cycle of the IFS algorithm in Figure 2, a subset of activities a_i is randomly selected from the input solution S, with each activity having an uniformly distributed selection probability equal to γ. It is of great importance to underscore that according to this approach, the activities to be relaxed are randomly picked up from the solution S *with the same probability* which, as we will see shortly, entails a relaxation characterized by a greater disruption on S, compared to the following selection approach.

Critical Path-biased selection As opposed to the random selection, at each iteration the critical path-biased selection approach restricts the pool of the relaxable activities to the subset containing those activities that are closer to the *critical path condition* (*critical*

path set). As known, an activity a_i belongs to the critical path (i.e., meets the critical path condition) when, given a_i's end time e_i and its feasibility interval $[lb_i, ub_i]$, the condition $lb_i = ub_i$ holds. For each activity a_i, the smaller the difference $ub_i - lb_i$ computed on e_i, the closer is a_i to the critical path condition. At each IFS iteration, the critical path set is built so as to contain any activity a_i with a probability directly proportional to the γ parameter and inversely proportional to the $ub_i - lb_i$ value. For obvious reasons, the critical path-biased relaxation entails a smaller disruption on the solution S, as it operates on a smaller set of activities; the activities that are farther from the critical path condition will have a minimum probability to be selected. As explained in the following section, this difference has important consequences on the experimental behavior.

6 Experimental Analysis

The empirical evaluation has been carried out on a SDST-FJSSP benchmark set synthesized on purpose out of the first 20 instances of the *edata* subset of the FJSSP *HUdata* testbed from [15], and will therefore be referred to as *SDST-HUdata*. Each one of the *SDST-HUdata* instances has been created by adding to the original *HUdata* instance one Setup-Time matrix $st^r(nJ \times nJ)$ for each present machine r, where nJ is the number of present jobs. Without loss of generality, the same randomly generated Setup-Time matrix was added for each machine of all the benchmark instances. Each value $st^r_{i,j}$ in the Setup-Time matrix models the setup time necessary to reconfigure the r-th machine to switch from job i to job j. Note that machine reconfiguration times are sequence dependent: setting up a machine to process a product of type j after processing a product of type i can generally take a different amount of time than setting up the same machine for the opposite transition. The elements $st^r_{i,j}$ of the Setup-Time matrix satisfy the *triangle inequality* [7, 4], that is, for each three activities a_i, a_j, a_k requiring the same machine, the inequality $st^r_{ij} \leq st^r_{ik} + st^r_{kj}$ holds. The 20 instances taken from *HUdata* (namely, the instances *la01-la20*) are divided in four groups of five $(nJ \times nA)$ instances each, where nJ is the number of jobs and nA is the number of activities per job for each instance. More precisely, group *la01-la05* is (10×5), group *la06-la10* is (15×5), group *la11-la15* is (20×5), and group *la16-la20* is (10×10). In all instances, the processing times on machines assignable to the same activity are identical, as in the original *HUdata* set. The algorithm used for these experiments has been implemented in Java and run on a AMD Phenom II X4 Quad 3.5 Ghz under Linux Ubuntu 10.4.1.

Results. Table 1 and table 2 show the obtained results running our algorithm on the *SDST-HUdata* set using the *Random* or *Slack-based* procedure in the IFS relaxation step, respectively. Both tables are composed of 10 columns and 23 rows (one row per problem instance plus three data wrap-up rows). The ***best*** column lists the shortest makespans obtained in the experiments for each instance; underlined values represent the best values obtained from both tables (global bests). The columns labeled $\gamma = 0.2$ to $\gamma = 0.9$ (see Section 4) contain the results obtained running the IFS procedure with a different value for the *relaxing factor* γ. For each problem instance (i.e., for each row) the values in bold indicate the best makespan found among all the tested γ values (γ runs).

Table 1. Results with random selection procedure

inst.	***best***	γ							
		0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
la01	**726**	772	731	728	**726**	729	**726**	729	740
la02	**749**	785	785	**749**	**749**	**749**	**749**	**749**	768
la03	**652**	677	658	658	658	**652**	**652**	658	675
la04	**673**	**673**	**673**	**673**	689	689	680	680	690
la05	**603**	613	613	**603**	605	605	606	607	632
la06	**950**	965	**950**	954	954	971	997	995	1020
la07	**916**	946	**916**	925	919	947	950	987	1000
la08	**954**	973	961	964	**954**	963	958	1000	1001
la09	**1002**	1039	**1002**	1039	1020	1042	1020	1045	1068
la10	**977**	1017	**977**	1022	**977**	1027	1008	1042	1048
la11	**1265**	**1265**	1312	1285	1282	1345	1332	1372	1368
la12	**1088**	**1088**	1114	1130	1167	1165	1199	1209	1198
la13	**1255**	**1255**	**1255**	**1255**	1300	1280	1300	1316	1315
la14	**1292**	**1292**	1315	1344	1346	1362	1351	1345	1372
la15	**1298**	**1298**	1302	1338	1355	1352	1367	1388	1429
la16	**1012**	1028	**1012**	**1012**	**1012**	**1012**	**1012**	**1012**	1023
la17	**864**	881	885	885	**864**	888	**864**	**864**	902
la18	**985**	1021	1007	1029	999	**985**	**985**	**985**	**985**
la19	**956**	1006	992	975	**956**	**956**	978	959	981
la20	**997**	1008	1010	**997**	**997**	**997**	**997**	**997**	999
B (**N**)	**12**	**6(1)**	**7(5)**	**6(4)**	**8(5)**	**6(5)**	**7(5)**	**5(3)**	**1(1)**
Av.C.		20149	17579	14767	11215	10950	9530	7782	7588
Av.MRE		19.34	18.29	18.66	18.37	19.42	19.43	20.60	22.44

For each γ run, the last three rows of both tables show respectively (up-bottom): (1) the number B of best solutions found *locally* (i.e., within the current table) and, underlined within round brackets, the number **N** of best solutions found *globally* (i.e., between both tables); (2) the average number of utilized solving cycles (Av.C.), and (3) the average mean relative error ($Av.MRE$)[6] with respect to the lower bounds of the original *HUdata* set (i.e., without setup times), reported in [16]. For all runs, a maximum CPU time limit was set to 800 seconds.

One significant result that the tables show is the difference in the average of utilized solving cycles (Av.C. row) between the *random* and the *slack-based* relaxation procedure. In fact, it can be observed that on average the slack-based approach uses more solving cycles in the same allotted time than its random counterpart (i.e., the slack-based relaxation heuristic is faster in the solving process). This is explained by observing that the slack-based relaxation heuristic entails a less severe disruption of the current solution at each solving cycle compared to the random heuristic, as the former generally relaxes a lower number of activities (given the same γ value). The lower the disruption level of the current solution in the relaxation step, the easier it is to re-gain solution feasibility in the flattening step. In addition of this efficiency issue, the slack-based relaxation approach also provides the extra effectiveness deriving from operating in the vicinity of the critical path of the solution, as demonstrated in [8].

The good performance exhibited by the slack-based heuristic can be also observed by inspecting the B(**N**) rows in both tables. Clearly, the slack-based approach finds a

[6] The individual MRE of each solution is computed as follows: $MRE = 100 \times (C_{max} - LB)/LB$, where C_{max} is the solution makespan and LB is the instance's lower bound

Table 2. Results with slack-based selection procedure

inst.	***best***	γ							
		0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
la01	**726**	739	736	**726**	**726**	**726**	**726**	**726**	**726**
la02	**749**	785	**749**	**749**	**749**	**749**	**749**	**749**	**749**
la03	**652**	658	658	658	658	658	**652**	658	658
la04	**673**	686	686	686	**673**	686	680	**673**	680
la05	**603**	613	**603**	613	605	**603**	604	**603**	605
la06	**960**	963	963	971	**960**	963	962	970	970
la07	**925**	941	966	941	**925**	931	946	972	1000
la08	**948**	983	963	**948**	964	993	967	994	973
la09	**1002**	1020	1020	**1002**	**1002**	1040	1069	1052	1042
la10	**985**	993	991	1007	1022	1022	1017	**985**	1024
la11	**1256**	**1256**	1257	1295	1295	1308	1318	1324	1332
la12	**1082**	**1082**	1097	1098	1159	1152	1188	1163	1207
la13	**1215**	1222	1240	1240	1223	**1215**	1311	1301	1311
la14	**1285**	1308	**1285**	1285	1311	1295	1335	1372	1345
la15	**1291**	1333	**1291**	1330	1302	1311	1383	1389	1412
la16	**1007**	1012	1012	1012	**1007**	1012	1012	1012	1012
la17	**858**	889	868	893	895	888	**858**	859	872
la18	**985**	1019	1025	1021	1007	**985**	**985**	**985**	**985**
la19	**956**	1006	976	987	984	**956**	980	**956**	959
la20	**997**	**997**	1033	**997**	**997**	**997**	1003	**997**	**997**
B (**N**)	**17**	**3**(**3**)	**4**(**4**)	**5**(**5**)	**8**(**6**)	**7**(**7**)	**5**(**5**)	**8**(**7**)	**4**(**4**)
Av.C.		21273	18068	15503	13007	10643	10653	8639	8575
Av.MRE		18.67	18.09	18.26	18.19	18.14	19.58	19.44	20.16

higher number of best solutions (17 against 12), which is confirmed by comparing the number of locally found bests (B) with the global ones (**N**), for each γ value, and for both heuristics.

Another interesting aspect can be found analyzing the γ values range where the best performances are obtained ($Av.MRE$ row). Inspecting the $Av.MRE$ values, the following can in fact be stated: (1) the slack-based heuristic finds solutions of higher quality w.r.t. the random heuristic over the complete γ variability range; (2) in the random case, the best results are obtained in the $[0.3, 0.5]$ γ range, while in the slack-based case the best γ range is wider ($[0.3, 0.6]$).

7 Conclusions

In this paper we have proposed the use of Iterative Flattening Search (IFS) as a means of effectively solving the SDST-FJSSP. The proposed algorithm uses as its core solving procedure an extended version of the SP-PCP procedure proposed by [21] and a new relaxation strategy targeted to the case of SDST-FJSSP. The effectiveness of the procedure was demonstrated on 20 modified instances of the *edata* subset of the FJSSP *HUdata* testbed from [15], a well known and difficult Flexible Job Shop Scheduling benchmark set. In particular, we show as the new slack-based relaxation strategy exhibits better performance than the random selection one. Further improvement of the current algorithm may be possible by incorporating additional heuristic information and search mechanisms. One of the next steps will be the collection of the benchmarks proposed in the cited works [23, 24, 26, 27], although no one of the problems proposed

in these papers coincides with the SDST-FJSSP, basically they can be seen as slight variations of this problem, hence the proposed IFS procedure can be adapted to solve an interesting and large class of flexible manufacturing scheduling problems. This will be the focus of our future work together the realization of a web repository to collect all the interesting benchmark sets.

Acknowledgments

CNR authors are partially supported by EU under the ULISSE project (Contract FP7.218815), and MIUR under the PRIN project 20089M932N (funds 2008).

References

1. J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.
2. A. Allahverdi, C. Ng, T. Cheng, and M. Kovalyov. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, 187(3):985–1032, 2008.
3. A. Allahverdi and H. Soroush. The significance of reducing setup times/setup costs. *European Journal of Operational Research*, 187(3):978–984, 2008.
4. C. Artigues and D. Feillet. A branch and bound method for the job-shop problem with sequence-dependent setup times. *Annals OR*, 159(1):135–159, 2008.
5. E. Balas, N. Simonetti, and A. Vazacopoulos. Job shop scheduling with setup times, deadlines and precedence constraints. *Journal of Scheduling*, 11(4):253–262, 2008.
6. P. Brucker, B. Jurisch, and B. Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49(1-3):107–127, 1994.
7. P. Brucker and O. Thiele. A branch & bound method for the general-shop problem with sequence dependent setup-times. *OR Spectrum*, 18(3):145–161, 1996.
8. A. Cesta, A. Oddi, and S. F. Smith. Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems. In *AAAI/IAAI. 17^{th} National Conference on Artificial Intelligence*, pages 742–747, 2000.
9. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.
10. J. Gao, L. Sun, and M. Gen. A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems. *Computers & Operations Research*, 35:2892–2907, 2008.
11. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.
12. D. Godard, P. Laborie, and W. Nuitjen. Randomized Large Neighborhood Search for Cumulative Scheduling. In *Proceedings of ICAPS-05*, pages 81–89, 2005.
13. M. A. González, C. R. Vela, and R. Varela. A Tabu Search Algorithm to Minimize Lateness in Scheduling Problems with Setup Times. In *Proceedings of the CAEPIA-TTIA 2009 13th Conference of the Spanish Association on Artificial Intelligence*, 2009.
14. A. B. Hmida, M. Haouari, M.-J. Huguet, and P. Lopez. Discrepancy search for the flexible job shop scheduling problem. *Computers & Operations Research*, 37:2192–2201, 2010.
15. J. Hurink, B. Jurisch, and M. Thole. Tabu search for the job-shop scheduling problem with multi-purpose machines. *OR Spectrum*, 15(4):205–215, February 1994.
16. M. Mastrolilli and L. M. Gambardella. Effective neighbourhood functions for the flexible job shop problem. *Journal of Scheduling*, 3:3–20, 2000.

17. L. Michel and P. Van Hentenryck. Iterative Relaxations for Iterative Flattening in Cumulative Scheduling. In *Proceedings of ICAPS-04*, pages 200–208, 2004.
18. U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences*, 7:95–132, 1974.
19. E. Nowicki and C. Smutnicki. An advanced tabu search algorithm for the job shop problem. *Journal of Scheduling*, 8(2):145–159, 2005.
20. A. Oddi, A. Cesta, N. Policella, and S. F. Smith. Iterative flattening search for resource constrained scheduling. *J. Intelligent Manufacturing*, 21(1):17–30, 2010.
21. A. Oddi and S. Smith. Stochastic Procedures for Generating Feasible Schedules. In *Proceedings 14th National Conference on AI (AAAI-97)*, pages 308–314, 1997.
22. N. Policella, A. Cesta, A. Oddi, and S. Smith. From Precedence Constraint Posting to Partial Order Schedules. *AI Communications*, 20(3):163–180, 2007.
23. A. Rossi and G. Dini. Flexible job-shop scheduling with routing flexibility and separable setup times using ant colony optimisation method. *Robotics and Computer-Integrated Manufacturing*, 23(5):503–516, 2007.
24. R. Ruiz and C. Maroto. A genetic algorithm for hybrid flowshops with sequence dependent setup times and machine eligibility. *European Journal of Operational Research*, 169(3):781 – 800, 2006.
25. A. K. Sethi and S. P. Sethi. Flexibility in manufacturing: A survey. *International Journal of Flexible Manufacturing Systems*, 2:289–328, 1990. 10.1007/BF00186471.
26. E. Vallada and R. Ruiz. A genetic algorithm for the unrelated parallel machine scheduling problem with sequence dependent setup times. *European Journal of Operational Research*, 211(3):612 – 622, 2011.
27. V. Valls, M. A. Perez, and M. S. Quintanilla. A tabu search approach to machine scheduling. *European Journal of Operational Research*, 106(2-3):277 – 300, 1998.
28. C. R. Vela, R. Varela, and M. A. González. Local search and genetic algorithm for the job shop scheduling problem with sequence dependent setup times. *Journal of Heuristics*, 2009.