Exploiting Iterative Flattening Search to Solve Job Shop Scheduling Problems with Setup Times

Angelo Oddi [1] and **Riccardo Rasconi** [1] and **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 heuristic algorithm for solving a job-shop scheduling problem with sequence dependent setup times (SDST-JSSP). This strategy, known as Iterative Flattening Search (IFS), iteratively applies two steps: (1) a relaxation-step, in which a subset of scheduling decisions are randomly retracted from the current solution; and (2) a solving-step, in which a new solution is incrementally recomputed from this partial schedule. The algorithm relies on a core constraint-based search procedure, which generates consistent orderings of activities that require the same resource by incrementally imposing precedence constraints on a temporally feasible solution. Key to the effectiveness of the search procedure is a conflict sampling method biased toward selection of the most critical conflicts. The efficacy of the overall heuristic optimization algorithm is demonstrated empirically on a set of well known SDST-JSSP benchmarks.

Introduction

This paper describes an iterative improvement approach to solve job-shop scheduling problems with ready times, deadlines, and *sequence dependent* setup times (SDST-JSSP). Over the last ten years, there has been an increasing interest in solving scheduling problems with setup times (Allahverdi and Soroush 2008; Allahverdi et al. 2008). 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 this paper we focus on one family of techniques referred to as Iterative Flattening Search (IFS). IFS was first introduced in (Cesta, Oddi, and Smith 2000) 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*). The original IFS procedure was extended in two subsequent works (Michel and Van Hentenryck 2004; Godard, Laborie, and Nuitjen 2005) such that additional optimal solutions and improvements to known upper-bounds for the reference benchmark problems were obtained. More recently (Oddi et al. 2008; 2010) initiated a systematic study aimed at evaluating the effectiveness of single *component* strategies within the same uniform software framework.

The IFS algorithm proposed in this work relies on a *core* constraint-based search procedure, which generates a consistent ordering of activities that require the same resource by incrementally adding precedence constraints between activity pairs in a temporally feasible solution. Specifically, the algorithm we propose in this work is an extension of the SP-PCP procedure proposed in (Oddi and Smith 1997) applied to the case of scheduling problems without setup times.

In the current literature there are other examples of procedures for solving scheduling problems with setup times that are extensions of the counterpart procedures used to solve the same (or similar) scheduling problem without setup times. This is the case of the work by (Brucker and Thiele 1996), for example, which relies on an earlier solutions introduced in (Brucker, Jurisch, and Sievers 1994). Another example is the more recent work of (Vela, Varela, and González 2009) and (González, Vela, and Varela 2009), which proposes effective heuristic procedures based on genetic algorithms and local search. In these works, the local search procedures that are introduced extend a procedure originally proposed by (Nowicki and Smutnicki 2005) for the classical job-shop scheduling problem to the setup times case by introducing a neighborhood structure that similarly properties relating to critical paths in an underlying disjunctive graph formulation of the problem. A third example is the work of (Balas, Simonetti, and Vazacopoulos 2008), which extends the well-know *shifting bottleneck* procedure (Adams, Balas, and Zawack 1988) to the SDST-JSSP case. Both (Balas, Simonetti, and Vazacopoulos 2008) and (Vela, Varela, and González 2009) have produced reference results with their techniques on a previously studied benchmark set of SDST-JSSP problems initially proposed by (Brucker and Thiele 1996). We use this benchmark problem set as a basis for direct comparison to our solution procedure in the experimental section of this paper.

This paper is organized as follows. An introductory section defines the reference SDST-JSSP problem and its representation. A central section describes the iterative improvement search, the adopted relaxation straegies and the core constraint-based search procedure. An experimental section describes the performance of our algorithm on the benchmark problem set of (Brucker and Thiele 1996) and

the most interesting results are explained. Some conclusions and a discussion of future work end the paper.

The Scheduling Problem with Setup Times

In this section we provide a definition of the job-shop scheduling problem with sequence dependent setup times (SDST-JSSP). The SDST-JSSP entails synchronizing the use of a set of 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 m activities $a_{ik} \in J_k$ $(i = 1, \ldots, m)$, and the execution of each activity a_{ik} is subject to the following constraints:

- *resource availability* - each activity a_i requires the exclusive use of a single resource r_{a_i} for its entire duration; no *preemption* is allowed and all the activities included in a job J_k require distinct resources.
- *processing time constraints* - each a_i has a fixed processing time p_i such that $e_i - s_i = p_i$, where the variables s_i and e_i represent the start and end time of a_i.
- *sequence dependent setup times* - for each resource r, the value st^r_{ij} represents the setup time between two generic activities a_i and a_j requiring the same resource r, such that $e_i + st^r_{ij} \leq s_j$. The setup times st^r_{ij} verify the so-called *triangular inequality* (see (Brucker and Thiele 1996; Artigues and Feillet 2008)). The triangle inequality (traditionally in literature, this property is always considered verified) imposes that, for each 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.
- *job release dates* - each Job J_k has a release date rd_k, which specifies the earliest time that the any activity in J_k can be started.

A *solution* $S = \{S_1, S_2, \ldots, S_n\}$ is an assignment S_i to the activities start-times s_i such that 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}. We observe as the proposed optimization problems is *NP-hard*, because is an extensions of the well-known job-shop scheduling problem $J||C_{max}$ (Sotskov and Shakhlevich 1995).

A CSP Representation

There are different ways to formulate this problem as a *Constraint Satisfaction Problem* (CSP) (Montanari 1974). Analogously to (Cheng and Smith 1994; Oddi and Smith 1997), the problem is treated as one of establishing *precedence constraints* between pairs of activities that require the same resource, so as to eliminate all possible conflicts in the resource use. Such representation is close to the idea of *disjunctive graph* initially used for the classical job shop scheduling without setup times and also used in the extended case of setup times (Brucker and Thiele 1996; Balas, Simonetti, and Vazacopoulos 2008; Vela, Varela, and González 2009; Artigues and Feillet 2008).

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. J is a set of directed edges (a_i, a_j) representing the precedence constraints among the activities (job precedences constraints) and are weighted with the processing time p_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 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, a decision variable x_{ijr} is defined for each pair of activities a_i and a_j requiring resource 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 the current case 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, in case the setup times verify the triangle inequality, previous decisions on the x_{ijr} can be represented as the 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_{ijr}, for any SDST-JSSP we define the directed graph G_d , called *distance graph*, which is an extended version of the disjunctive 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 represent its start and end time points respectively. The set of edges E represents all the imposed temporal constraints, i.e., precedences, durations and setup times. 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-JSSP 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 (see (Dechter, Meiri, and Pearl 1991) for more details on the STP). Moreover, any time point tp_i is associated to a given *feasibility interval* $[lb_i, ub_i]$, which determines the current set of feasible values for tp_i. Thus, a search for a solution to a SDST-JSSP 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. Given a Simple Temporal Problem, the problem is consistent if and only if no closed paths with negative length (or negative cycles) are contained in the graph G_d.

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 (see (Dechter, Meiri, and Pearl 1991)). Hence, the *minimal* allowed distance between tp_j and tp_i is $-d(tp_j, tp_i)$ and the maximal distance is $d(tp_i, tp_j)$. Given that d_{i0} is the length

of the shortest path on G_d from the time point tp_i to the origin point tp_0 and d_{0i} is the length of the shortest path from the origin point tp_0 to the time point tp_i, the interval $[lb_i, ub_i]$ of time values associated to the generic time variable tp_i is computed on the graph G_d as the interval $[-d(tp_i, tp_0), d(tp_0, tp_i)]$ (see (Dechter, Meiri, and Pearl 1991)). 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.

A Precedence Constraint Posting Procedure

The proposed procedure for solving instances of SDST-JSSP is an extension of the SP-PCP scheduling procedure (Shortest Path-based Precedence Constraint Posting) proposed in (Oddi and Smith 1997), which utilizes shortest path information in $G_d(V, E)$ for guiding the search process. Similarly to the original SP-PCP procedure, shortest path information is utilized in a twofold fashion to enhance the search process.

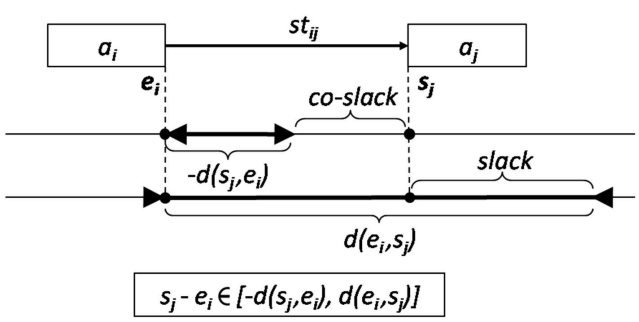

Figure 1: $slack(e_i, s_j) = d(e_i, s_j) - st^r_{ij}$ Vs. $co\text{–}slack(e_i, s_j) = -d(s_j, e_i) - st^r_{ij}$

The first way in which shortest path information is exploited is by introducing new *dominance conditions* (which adapt to the setup times case those presented in (Oddi and Smith 1997)), through which problem constraints are *propagated* and unconditional decisions for promoting early pruning of alternatives are identified. The concepts of $slack(e_i, s_j)$ and $co\text{–}slack(e_i, s_j)$ (complementary slack) play a central role in the definition of such new dominance conditions. Given two activities a_i, a_j and the related interval of distances $[-d(s_j, e_i), d(e_i, s_j)]$ [1] and $[-d(s_i, e_j), d(e_j, s_i)]$ [2] on the graph G_d, they are defined as follows (see Figure 1):

- $slack(e_i, s_j) = d(e_i, s_j) - st^r_{ij}$ is the difference between the maximal distance $d(e_i, s_j)$ and the setup time st^r_{ij}. 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 $slack(e_i, s_j) < 0$, then the ordering $a_i \preceq a_j$ is not feasible.
- $co\text{–}slack(e_i, s_j) = -d(s_j, e_i) - st^r_{ij}$ 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 $co\text{–}slack(e_i, s_j) \geq 0$ (in Figure 1 a *negative* co-slack is represented), 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.

For any pair of activities a_i and a_j that are competing for the same resource r, the new dominance conditions describing the four 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 the search has reached an inconsistent state.

Conditions 2, and 3, alternatively, distinguish *uniquely resolvable conflicts*, i.e., there is only one feasible ordering of a_i and a_j, and the decision of which constraint to post is thus unconditional. 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}; therefore, we still need to impose the constraint $e_j + st^r_{ji} \leq s_i$. Condition 3 works similarly, and entails that only the $a_i \preceq a_j$ ordering is feasible. Finally, Condition 4 designates a class of *resolvable conflicts*; in this case, both orderings of a_i and a_j remain feasible, and it is therefore necessary to perform a *search decision*.

The second way in which shortest path information is exploited is by defining *variable* and *value* ordering heuristics to select and resolve conflicts in the set characterized by Condition 4. 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. The *variable* ordering heuristic attempts to focus first on the conflict with the least amount of sequencing flexibility (i.e., the conflict that is closest to previous Condition 1). More precisely, the conflict (a_i, a_j) with the overall minimum value of $VarEval(a_i, a_j) = min\{bd_{ij}, bd_{ji}\}$ is always selected for resolution, where[4]:

$$bd_{ij} = \frac{slack(e_i, s_j)}{\sqrt{S}}, \quad bd_{ji} = \frac{slack(e_j, s_i)}{\sqrt{S}}$$

[1] between the end-time e_i of a_i and the start-time s_j of a_j

[2] between the end-time e_j of a_j and the start-time s_i of a_i

[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

[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 sequencing flexibility.

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** UniquelyResolvableConflict(S)
8. **then** PostUnconditionalConstraints(S)
9. **else begin**
10. $Con \leftarrow$ChooseResolvableConflict(S)
11. **if** ($Con = nil$)
12. **then return**(S)
13. **else begin**
14. $Prec \leftarrow$ ChoosePrecConstraint(S, Con)
15. PostConstraint(S, $Prec$)
16. **end**
17. **end**
18. **end-loop**
19. **return**(S)

Figure 2: Basic PCP algorithm

and

$$S = \frac{min\{slack(e_i, s_j), slack(e_j, s_i)\}}{max\{slack(e_i, s_j), slack(e_j, s_i)\}}$$

As opposed to variable ordering, the *value* ordering heuristic attempts to resolve the selected conflict (a_i, a_j) simply by choosing the precedence constraint that retains the highest amount of sequencing flexibility. Specifically, $a_i \preceq a_j$ is selected if $bd_{ij} > bd_{ji}$ and $a_j \preceq a_i$ is selected otherwise.

The PCP Algorithm

Figure 2 gives the basic overall PCP solution procedure, which starts from an empty solution (Step 1) where the graphs G_d is initialized according to the previous section on the CSP representation of the problem. 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 2 analyses all pairs (a_i, a_j) of activities that require the same resource (i.e., the *decision variables* of the corresponding CSP problem), and decides their *values* in terms of precedence ordering (i.e., $a_i \preceq a_j$ or $a_j \preceq a_i$, see Section), on the basis of the response provided by the *dominance conditions*.

In broad terms, the procedure in Figure 2 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 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 constraint) in the graph G_d. In particular, `Propagate`(S) updates the shortest path distances on the graph G_d. We observe that within the main loop of the procedure PCP shown in Figure 2 new constraints are added incrementally (one-by-one) to G_d, hence the complexity of this step [5] is in the worst case $O(n^2)$.

A solution is found when the PCP algorithm finds a feasible assignment to the activity start times such that all resource conflicts are resolved (i.e., all the setup times st_{ij} are satisfied), according to the following proposition:

Proposition 1 *A solution S is found when none of the four dominance conditions is verified on S.*

The previous assertion can be demonstrated *by contradiction*. Let us suppose that the PCP procedure exits with success (none of the four dominance conditions is verified on S) and that at least two sequential activities a_i and a_j, requiring the same resource r do not satisfy the setup constraints $e_i + st^r_{ij} \leq s_j$ or $e_j + st^r_{ji} \leq s_i$. Since the *triangle inequality* holds for the input problem, it is guaranteed that the length of the *direct* setup transition $a_i \preceq a_j$ between two generic activities a_i and a_j is the shortest possible (i.e., no *indirect* transition $a_i \rightsquigarrow a_k \rightsquigarrow a_j$ having a shorter overall length can exist). This fact is relevant for the PCP approach, because the solving algorithm proceeds by checking/imposing either the constraint $e_i + st^r_{ij} \leq s_j$ or the constraint $e_j + st^r_{ji} \leq s_i$ for each pair of activities. Hence, when none of the four dominance conditions is verified, each subset of activities A^r requiring the same resource r is totally ordered over time. Clearly, for each pair (a_i, a_j), such that $a_i, a_j \in A^r$, either $co\text{–}slack(e_i, s_j) \geq 0$ or $co\text{–}slack(e_j, s_i) \geq 0$; hence, all pairs of activities (a_i, a_j) requiring the same resource r *satisfy* the setup constraints $e_i + st^r_{ij} \leq s_j$ or $e_j + st^r_{ji} \leq s_i$. In fact, by definition $co\text{–}slack(e_i, s_j) \geq 0$ implies $-d(s_j, e_i) \geq st^r_{ij}$ and together the condition $s_j - e_i \geq -d(s_j, e_i)$ (which holds because G_d is consistent), we have $e_i + st^r_{ij} \leq s_j$ (a similar proof is given for $co\text{–}slack(e_j, s_i) \geq 0$).

To wrap up, when none of the four dominance conditions is verified and the PCP procedure exits with success, the G_d graph represents a consistent Simple Temporal Problem and, as described in the previous section on the CSP representation of the problem, one possible solution of the problem is the so-called *earliest-time solution*, such that $S_{est} = \{S_i = -d(tp_i, tp_0) : i = 1 \ldots n\}$.

The Optimization Algorithm

Figure 3 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 two parameters as input: (1) an initial solution S; (2) a positive integer $MaxFail$ which specifies the maximum number of non-makespan improving moves that the algorithm will tolerate before ter-

[5] Let us suppose we have a consistent G_d, in the case we add a new edge (tp_x, tp_y) with weight w_{xy}, if $w_{xy} + d(tp_y, tp_x) \geq 0$ (G_d remains consistent, because no negative cycle is added), then the generic shortest path distance can be updated as $d(tp_i, tp_j) = min\{d(tp_i, tp_j), d(tp_i, tp_x) + w_{xy} + d(tp_y, tp_j).\}$

```
IFS(S,MaxFail)
begin
1.  S_best ← S
2.  counter ← 0
3.  while (counter ≤ MaxFail) do
4.      RELAX(S)
5.      S ←PCP(S, C_max(S_best))
6.      if C_max(S) < C_max(S_best) then
7.          S_best ← S
8.          counter ← 0
9.      else
10.         counter ← counter + 1
11. return (S_best)
end
```

Figure 3: The IFS schema

minating. After initialization (Steps 1-2), a solution is repeatedly modified within the while loop (Steps 3-10) by the application of the RELAX procedure, as explained in the following section, and the PCP procedure (see Figure 2). On each iteration, the RELAX step reintroduces the possibility of resource contention, and the PCP step then restores resource feasibility by removing detected resource conflicts. 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 in $MaxFail$ moves, the algorithm terminates and returns the best solution found.

The algorithms we are describing here are all based on a representation of the basic scheduling problem as a precedence graph $G(A_G, J, X)$ introduced above. We remember that G is 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}. J is a set of directed edges (a_i, a_j) representing the job precedences constraints. 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 and the related label represents the set of possible ordering between a_i and a_j: $a_i \preceq a_j$ or $a_j \preceq a_i$. 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 possible ordering between a_i and a_j: $a_i \preceq a_j$ or $a_j \preceq a_i$. In general the directed graph G_S represents 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.

In the next two subsection we define two different relaxation procedures based on the graph G_S.

Relaxation Procedures

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 a number of scheduling decisions. Given the graph representation described above, each such decision is a *precedence constraint* between a pair of activities that are competing for the same resource capacity. The first strategy we present, used in (Cesta, Oddi, and Smith 2000; Michel and Van Hentenryck 2004) for problems without setup times, removes precedence constraints between pair of activities belonging to the solution *critical path*, and hence is called *cp-based relaxation*. The second strategy, similar to the one proposed in (Godard, Laborie, and Nuitjen 2005) again for problems without setup times, starts from a G_S solution and randomly *breaks* some total orders (or chains) imposed on the subset of activities requiring the same resource r, and hence is given the name *chain-based relaxation*.

```
CPRELAX(S, p_r, n_r)
begin
1. for k = 1 to n_r
2.     forall (a_i, a_j) ∈ CriticalPath(S) ∩ X_S
3.         if random(0,1) < p_r
4.             then S ← S \ (a_i, a_j)
end
```

Figure 4: Relaxation procedure based on removal from critical path

Precedence Relaxation The *cp-based relaxation* strategy is centered on the solution's *critical path*. A *path* in $G_S(A, J, X_S)$ is a sequence of activities $a_1, a_2, \ldots, a_k$, such that, $(a_i, a_{i+1}) \in J \cup X_S$ with $i = 1, 2, \ldots, (k-1)$. The length of a path is the sum of the activities processing times and a *critical path* is a path from a_0 to a_{n+1} which determines the solution's makespan C_{max}. Any makespan improvement will necessarily require a modification to a subset of precedence constraints laying on the *critical path*, since these constraints collectively determine the solution's current makespan. From this observation, the relaxation step is designed to retract some posted precedence constraints on the solution's critical path. Fig. 4 shows the CPRELAX procedure. Steps 2-4 consider the set of posted precedence constraints which belong to the current critical path. A subset of these constraints is randomly selected on the basis of the parameter $p_r \in (0, 1)$ and then removed from the current solution. These steps are iterated n_r times (effective values range from 2 to 6), such that, a new critical path of S is computed at each iteration. Notice that at each relaxation, the new critical path can be completely different from the previous one. This allows the relaxation step to also take into account those paths whose length is very close to the critical one.

Chain Relaxation The *chain-based relaxation* strategy requires an input solution as a graph $G_S(A, J, X_S)$. As explained above, a solution is a modification of the original precedence graph G that represents the input scheduling problem. G_S contains a set of additional 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. Given a generic activity a_i, let $pred(a_i)$ be its predecessor activity and $succ(a_i)$ its successor activity.

The CHAINRELAX procedure proceeds in two steps. First, a subset of activities from the input solution S are ran-

domly *selected* on the basis of the parameter $p_r \in (0, 1)$. Second, a procedure similar to CHAINING - used in (Policella et al. 2007) - is applied to the set of unselected activities. The modified CHAINING procedure (that takes into account setup times) can be accomplished in three steps: (1) all the previously posted levelling constraints X_S are removed from the solution S; (2) the unselected activities are sorted by increasing earliest start times; (3) for each resource and for each activity a_i (according to the increasing order of start times), a_i's predecessors p is considered and a precedence constraint (p, a_i) is posted, so as to impose the related setup time from p to a_i. The last step is iterated until all the activities are linked by precedence constraints. It is worth observing 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 thus re-creating *conflict* is resource usages.

Experimental Analysis

In this section we propose a set of empirical evaluations of the IFS algorithm. We have considered a well known benchmark set described in the literature and available on the Internet. This benchmark was originally proposed in (Brucker and Thiele 1996) with the objective of minimizing the *makespan* C_{max}. The IFS algorithm has been implemented in CMU Common Lisp Ver. 20a and run on a AMD Phenom II X4 Quad 3.5 Ghz under Linux Ubuntu 9.0.

The Benchmark Set

This set is composed of 15 instances initially provided by (Brucker and Thiele 1996) and later integrated with other 10 instances; they are available at `http://www.andrew.cmu.edu/user/neils/tsp/t2ps/`. Each instance is characterized by the configuration $(nJ \times nA)$ where for every instance, nJ is the number of present jobs and nA is the number of activities per job. The original benchmark of 15 problems is divided in sets of 5 instances each, composed as follows: the first set contains 10×5 problems, the second set contains 15×5 problems, and the third set contains 20×5 problems. The 10 problems successively added are divided in two sets of 5 instances each: the first set contains 15×5 problems, while the second set contains 20×5 problems. Hence, our benchmark is therefore composed of 25 instances that range from 50 to 100 activities; in the remainder of this work, this benchmark will be referred to as BTS25.

Results

We propose two different set of experiments, one for the *cp-based relaxation*, and another one for the *chain-based relaxation*. In order to give a clear idea of the strength of the proposed IFS procedure, we report the results obtained with the combinations of input parameters which gave the best performance.

Table 1 shows the results for the IFS procedure using the *cp-based relaxation* obtained from the values of the parameters n_r (the number of operated relaxation steps) and p_r (the removal probability on the current critical path). The column labeller Δ^0 shows the average percentage variation from the infinite capacity makespan, while N_i represents the number of improved solutions with respect to the best known solutions for BTS25. The latter are selected as the union of the best results proposed in the papers (Balas, Simonetti, and Vazacopoulos 2008), (Vela, Varela, and González 2009) and (Artigues and Feillet 2008), for the first 15 BTS25 instances; for the last 10 instances, the best results are the ones proposed in (Balas, Simonetti, and Vazacopoulos 2008). The column labelled Δ^{bests} contains the average percentage variation from the best solutions, while the last column N_{iFlat} represents the average number IFS cycles performed over all the BTS25 instances within the imposed cpu bound of 3200 seconds.

Table 1: the table shows the average percentage variation from the infinite capacity makespan (Δ^0), the number of improved solutions (N_i), the average percentage deviations from the best-known solutions for the BTS25 (Δ^{bests}), and the average number of IFS cycles (N_{iFlat}) performed over all the BTS25 instances within the imposed cpu bound of 3200 seconds, for different values of n_r and p_r (percentage value).

n_r	p_r	N_i	Δ^0	Δ^{bests}	N_{iFlat}
5	10	1	168.9	2.1	2682.4
	15	2	168.7	1.9	2696.2
	20	1	168.4	2.0	2749.8
	25	1	168.8	2.0	2784.1
	30	3	167.5	1.6	2785.4
6	10	2	169.1	2.2	2722.4
	15	2	168.9	2.1	2785.6
	20	1	167.2	1.5	2758.9
	25	2	168.7	1.9	2766.6
	30	1	168.9	2.0	2782.7
7	10	3	168.5	2.0	2715.9
	15	2	167.7	1.6	2822.0
	20	2	168.0	1.6	2793.0
	25	1	167.2	1.3	2816.7
	30	2	168.0	1.6	2810.6
8	10	1	168.6	2.0	2729.8
	15	2	167.8	1.5	2777.8
	20	2	168.1	1.7	2812.0
	25	2	167.6	1.5	2830.3
	30	2	168.3	1.6	2842.7
BESTS	-	5	163.9	0.2	-

About the results shown in in Table 1, we note that the best performances are obtained for the value $n_r = 7$, and each single run has performance quite close to the best known results (the value Δ^{bests} ranges from 1.3 to 2.0). However, as we are running a random algorithm, we can consider the best results obtained over the set of performed runs and to evaluate the overall performance (the row BESTS). In particular, the average percentage deviation from the best known results is 0.2 and we want to highlight that in this analysis, 5 problems have been improved.

Table 2: the table shows the average percentage variation from the infinite capacity makespan (Δ^0), the number of improved solutions (N_i), the average percentage deviations from the best-known solutions for the BTS25 (Δ^{bests}), the average number of IFS cycles (N_{iFlat}) performed over all the BTS25 instances within the imposed cpu bound of 3200 seconds, for different values of p_r (percentage value).

p_r	N_i	Δ^0	Δ^{bests}	N_{iFlat}
10	0	171.2	3.1	2567.6
15	3	168.3	1.9	2613.7
20	3	168.5	2.1	2610.6
25	3	168.3	1.9	2716.4
30	2	167.7	1.7	2742.1
35	3	166.5	1.1	2726.5
40	3	166.9	1.3	2775.4
45	2	167.7	1.5	2769.3
50	2	167.5	1.4	2779.9
55	2	167.1	1.3	2776.9
60	3	167.0	1.2	2778.7
65	4	167.9	1.5	2769.3
70	2	168.1	1.6	2766.8
75	2	168.6	1.7	2793.7
BESTS	7	164.2	0.3	-

Table 2 shows the results for the IFS procedure using the *chain-based relaxation* obtained from the values of the parameter p_r (the probability for the selection of activities). About the results shown in Table 2 the other parameters have the same meaning of previous Table 1, and we again impose on each run a cpu bound of 3200 seconds. We note that the best performance are obtained within the range of values of the parameter p_r from 35% to 60%. Each single run has performance quite close to the best known results (the value Δ^{bests} ranges from 1.1 to 1.5). Likewise the previous case, we can consider the best results obtained over the set of performed runs and to evaluate the overall performance (the row BESTS). In this second case, the average percentage deviation from the best known results is 0.3 and we want to highlight that 7 problems have been improved. Hence, from the previous exploration we can draw the conclusion that on average, our algorithm's performances are in line with the best known algorithms. In some cases, the best known results have been improved.

Conclusions

Building from prior research, in particular (Oddi and Smith 1997; Godard, Laborie, and Nuitjen 2005; Oddi et al. 2010), in this paper we have investigated the use of iterative improving and precedence constraint posting algorithm as a means of effectively solving scheduling problems with sequence dependent setup times. The proposed iterative sampling algorithm uses as its core solving procedure an extended version of the SP-PCP procedure proposed by (Oddi and Smith 1997) and two already known relaxation strategies (Oddi et al. 2010) extended in this paper to the case of scheduling problems with setup times. To demonstrate the effectiveness of the procedure, a set of experiments were performed on a well known benchmark set of job shop scheduling problems with setup times. In the average, our algorithm's performances are in line with the known best algorithms, and in some cases it is able to improve the best known results.

Acknowledgments. CNR authors are partially supported by CNR internal funds under project RSTL (funds 2007), EU under the ULISSE project (Contract FP7.218815), and MIUR under the PRIN project 20089M932N (funds 2008).

References

Adams, J.; Balas, E.; and Zawack, D. 1988. The shifting bottleneck procedure for job shop scheduling. *Management Science* 34(3):391–401.

Allahverdi, A., and Soroush, H. 2008. The significance of reducing setup times/setup costs. *European Journal of Operational Research* 187(3):978–984.

Allahverdi, A.; Ng, C.; Cheng, T.; and Kovalyov, M. 2008. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research* 187(3):985–1032.

Artigues, C., and Feillet, D. 2008. A branch and bound method for the job-shop problem with sequence-dependent setup times. *Annals OR* 159(1):135–159.

Balas, E.; Simonetti, N.; and Vazacopoulos, A. 2008. Job shop scheduling with setup times, deadlines and precedence constraints. *Journal of Scheduling* 11(4):253–262.

Brucker, P., and Thiele, O. 1996. A branch & bound method for the general-shop problem with sequence dependent setup-times. *OR Spectrum* 18(3):145–161.

Brucker, P.; Jurisch, B.; and Sievers, B. 1994. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics* 49(1-3):107–127.

Cesta, A.; Oddi, A.; and Smith, S. F. 2000. Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems. In *AAAI/IAAI.* 17th *National Conference on Artificial Intelligence*, 742–747.

Cheng, C., and Smith, S. 1994. Generating Feasible Schedules under Complex Metric Constraints. In *Proceedings 12th National Conference on AI (AAAI-94).*

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

Godard, D.; Laborie, P.; and Nuitjen, W. 2005. Randomized Large Neighborhood Search for Cumulative Scheduling. In *ICAPS-05. Proceedings of the* 15th *International Conference on Automated Planning & Scheduling*, 81–89.

González, M. A.; Vela, C. R.; and Varela, R. 2009. 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*.

Michel, L., and Van Hentenryck, P. 2004. Iterative Relaxations for Iterative Flattening in Cumulative Scheduling. In *ICAPS-04. Proceedings of the* 14th *International Conference on Automated Planning & Scheduling*, 200–208.

Montanari, U. 1974. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences* 7:95–132.

Nowicki, E., and Smutnicki, C. 2005. An advanced tabu search algorithm for the job shop problem. *Journal of Scheduling* 8(2):145–159.

Oddi, A., and Smith, S. 1997. Stochastic Procedures for Generating Feasible Schedules. In *Proceedings 14th National Conference on AI (AAAI-97)*, 308–314.

Oddi, A.; Cesta, A.; Policella, N.; and Smith, S. F. 2008. Combining Variants of Iterative Flattening Search. *Journal of Engineering Applications of Artificial Intelligence* 21:683–690.

Oddi, A.; Cesta, A.; Policella, N.; and Smith, S. F. 2010. Iterative flattening search for resource constrained scheduling. *J. Intelligent Manufacturing* 21(1):17–30. DOI:10.1007/s10845-008-0163-8.

Policella, N.; Cesta, A.; Oddi, A.; and Smith, S. 2007. From Precedence Constraint Posting to Partial Order Schedules. *AI Communications* 20(3):163–180.

Sotskov, Y. N., and Shakhlevich, N. V. 1995. Np-hardness of shop-scheduling problems with three jobs. *Discrete Appl. Math.* 59(3):237–266.

Vela, C. R.; Varela, R.; and González, M. A. 2009. Local search and genetic algorithm for the job shop scheduling problem with sequence dependent setup times. *Journal of Heuristics*.