Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence

Iterative Flattening Search for the Flexible Job Shop Scheduling Problem

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 meta-heuristic algorithm for solving the Flexible Job Shop Scheduling Problem (FJSSP). This strategy, known as Iterative Flattening Search (IFS), iteratively applies a relaxation-step, in which a subset of scheduling decisions are randomly retracted from the current solution; and a solving-step, in which a new solution is incrementally recomputed from this partial schedule. This work contributes two separate results: (1) it proposes a constraint-based procedure extending an existing approach previously used for classical Job Shop Scheduling Problem; (2) it proposes an original relaxation strategy on feasible FJSSP solutions based on the idea of randomly breaking the execution orders of the activities on the machines and opening the resource options for some activities selected at random. The efficacy of the overall heuristic optimization algorithm is demonstrated on a set of well-known benchmarks.

1 Introduction

The paper focuses on a family of solving techniques referred to as Iterative Flattening Search (IFS). IFS was first introduced in [Cesta *et al.*, 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*). Extensions to the original IFS procedure were made in two subsequent works [Michel and Van Hentenryck, 2004; Godard *et al.*, 2005] both of which substantially improved its performance on reference benchmark problems and established additional new best solutions. More recently [Oddi *et al.*, 2010] have performed a systematic study aimed at evaluating the effectiveness of single *component strategies* within the same uniform software framework.

In this paper we develop and evaluate an IFS procedure for solving a scheduling problem with a different structure than the multi-capacity job-shop problem. We focus specifically on the Flexible Job Shop Scheduling Problem (FJSSP), a generalization of the classical JSSP where a given activity may be processed on any one of a designated set of available machines. The FJSSP is more difficult than the classical Job Shop Scheduling Problem (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, it is also necessary to choose a *routing policy*, that is which machine will process each activity. The objective remains that of minimizing *makespan*. The problem is motivated by interest in developing Flexible Manufacturing Systems (FMS), as underscored in [Rossi and Dini, 2007]; an effective synthesis of the existing solving approaches is proposed in [Ben Hmida *et al.*, 2010]. The core set of procedures which generates the best results include the genetic algorithm (GA) proposed in [Gao *et al.*, 2008], the tabu search (TS) approaches of [Mastrolilli and Gambardella, 2000; Bozejko *et al.*, 2010] and the discrepancy-based method, called climbing depth-bounded discrepancy search (CDDS), defined in [Ben Hmida *et al.*, 2010]. We use the results produced by these procedures as our evaluation reference point in this paper.

The IFS variant that we propose relies on a *core* constraint-based search procedure as its solver. This procedure is an extension of the SP-PCP procedure proposed in [Oddi and Smith, 1997]. 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 incorporate an additional set of decision variables relating to resource choice. To provide a basis for embedding this core solver within an IFS optimization framework, we also specify a new metaheuristic procedure for relaxing a feasible solution by randomly disrupting the activity sequences on various machines and reintroducing resource choice. Empirical analysis of our algorithm shows that it is generally comparable in performance to the best algorithms published over the last 10 years.

The paper is organized as follows. Section 2 defines the 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 metaheuristics. An experimental section describes the performance of our algorithm on a set of benchmark problems, and explains the most interesting results. Some conclusions end the paper.

2 Flexible Job Shop Scheduling Problem

The 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. 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 FJSSP is *NP-hard* since it is an extension of the JSSP problem [Garey and Johnson, 1979].

3 A CSP Representation

There are different ways to model the problem as a *Constraint Satisfaction Problem* (CSP), we use an approach similar to [Oddi and Smith, 1997]. In particular, we focus on *assigning resources to activities*, a distinguishing aspect of FJSSP and on *establishing precedence 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_{ir} ($r \in R_i$) 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 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 $a_j \preceq a_i$.

To support the search for a consistent assignment to the set of decision variables x_i and o_{ijr}, for any 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 represent its start and end time points respectively. 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 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 [Dechter *et al.*, 1991]. Thus, a search for a solution to a 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 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. 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 [Dechter *et al.*, 1991]. 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 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 [Oddi and Smith, 1997], 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, similarly to what proposed in [Huguet and Lopez, 2000]. 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 (R_i is the domain of the decision variable x_i). As soon as the search progresses and 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 $p_{ir} \notin [-d(s_i, e_i), d(e_i, s_i)]$ are removed from the domain of x_i and a new interval $[p_i^{min}, p_i^{max}]$ is recomputed accordingly. In the case the domain of the decision variable x_i becomes *empty*, then the search is reached a *failure* state.

The second way to exploit shortest path is by introducing *Dominance Conditions*, through which problem constraints are *propagated* and mandatory decisions for promoting early pruning of alternatives are identified. 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).

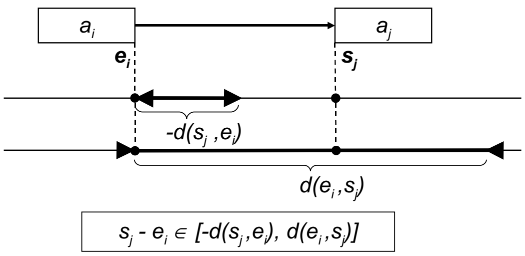

Figure 1: Max distance $d(e_i, s_j)$ vs. the min distance $-d(s_j, e_i)$

We observe as $d(e_i, s_j)$ is the maximal distance between a_i and a_j, it provides a measure of the degree of *sequencing flexibility* between a_i and a_j [3]. In addition, $-d(s_j, e_i)$ is the minimum possible distance between a_i and a_j (see Figure 1), then there is no need to separate a_i and a_j when $-d(s_j, e_i) \geq 0$. 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. $d(e_i, s_j) < 0 \wedge d(e_j, s_i) < 0$
2. $d(e_i, s_j) < 0 \wedge d(e_j, s_i) \geq 0 \wedge -d(s_i, e_j) < 0$
3. $d(e_i, s_j) \geq 0 \wedge d(e_j, s_i) < 0 \wedge -d(s_j, e_i) < 0$
4. $d(e_i, s_j) \geq 0 \wedge d(e_j, s_i) \geq 0$

Condition 1 represents an *unresolvable resource conflict*. There is no way to order a_i and a_j when they require the same resource r 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. In the particular case where $|R_i| = |R_j| = 1$ the decision $a_j \preceq a_i$ is mandatory. In the 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 of a_i and a_j remain feasible, and 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 assignments 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.

Initially, all the pairs of activities (a_i, a_j), such that ($|R_i| \geq 1 \vee |R_j| \geq 1$ 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 $FR_{ij} = 2(|R_i| + |R_j|) - |R_i \cap R_j|$[4], and the secondary key is a heuristic evaluation based on temporal flexibility and computed as $FT_{ij} = min_{r \in R_i \cap R_j}\{VarEval_r(a_i, a_j)\}$, where the $VarEval_r(a_i, a_j)$ heuristic is an extension to the FJSSP of the heuristic proposed in [Oddi and Smith, 1997], and it is computed as follows. As stated above, in this context $d(e_i, s_j)$ and $d(e_j, s_i)$ provide measures of the degree of *sequencing flexibility* between a_i and a_j. More precisely, given an activity pair (a_i, a_j), both assigned to resource r, the related heuristic evaluation is $VarEval_r(a_i, a_j) =$

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

where $S = \frac{min\{d(e_i,s_j), d(e_j,s_i)\}}{max\{d(e_i,s_j), d(e_j,s_i)\}}$ [5]. The pair (a_i^*, a_j^*) with the lowest value $\langle FR_{ij}, FT_{ij} \rangle$ (double-key sorting) is firstly selected and then it is associated to a resource decision variable

[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 resource flexibility FR_{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.

[5] The $\sqrt{S}$ bias is introduced to take into account cases where a first conflict with the overall $min\{d(e_i, s_j), d(e_j, s_i)\}$ has a

```
PCP(Problem, C_max)
1. S ← 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 ←ChooseDecisionVariable(S)
11.          if (C = nil)
12.            then return(S)
13.            else begin
14.              vc ← ChooseValueConstraint(S, C)
15.              PostConstraint(S, vc)
16.            end
17.        end
18. end-loop
19. return(S)
```

Figure 2: The PCP one-shot algorithm

or to an ordering decision variable depending on the cardinalities $|R_i|$ and $|R_j|$.

Resource decision variables. In case the condition $|R_i| > 1 \lor |R_j| > 1$ holds for the selected (a_i^*, a_j^*) pair, the chosen resource decision variable between x_i^* and x_j^* will be the one whose domain of values has the lowest cardinality (i.e., the MCF choice). As opposed to variable ordering, the *value* ordering heuristic is accomplished so as to retain the highest temporal flexibility. 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 $F_{min}(r) = min_{a_k \in A_r}\{VarEval_r(a_i, a_k)\}$. Hence, 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.

Ordering decision variables. In case the condition $|R_i| = 1 \land |R_j| = 1$ holds, the (a_i^*, a_j^*) pair is directly selected to be ordered, as it represents the conflict with the least amount of sequencing flexibility (i.e., the conflict that is closest to previous Condition 1 sub-case a). As in the previous case, the *value* ordering heuristic attempts to resolve the selected conflict (a_i, a_j) by simply choosing the precedence constraint that retains the highest amount of sequencing flexibility (least constrained value). Specifically, $a_i \preceq a_j$ is selected if $d(e_i, s_j) > d(e_j, s_i)$ and $a_j \preceq a_i$ is selected otherwise.

4.3 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 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

very large $max\{d(e_i, s_j), d(e_j, s_i)\}$, 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.

```
IFS(S, MaxFail, γ)
1. S_best ← S
2. counter ← 0
3. while (counter ≤ MaxFail) do
4.     RELAX(S, γ)
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)
```

Figure 3: The IFS schema

all the jobs. The PCP algorithm shown in Figure 2 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 precedence constraint (i.e., $a_i \preceq a_j$ or $a_j \preceq a_i$, see Section 3). 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. 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 of 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 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 three 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 terminating; (3) a parameter γ explained in Section 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 2 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 number of scheduling decisions. Here we use a strategy similar to the one in [Godard *et al.*, 2005] and called *chain-based relaxation*. Given the graph representation described above, each scheduling decision is either a *precedence 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 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. First, a subset of activities a_i is randomly selected from the input solution S, with each activity having an uniform probability $\gamma \in (0, 1)$ 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. Second, a procedure similar to CHAINING – used in [Policella *et al.*, 2007] – is applied to the set of unselected activities. This operation is accomplished in three steps: (1) all previously posted precedence 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 a precedence constraint (p, 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 precedence 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.

6 Experimental Analysis

To empirically evaluate the IFS algorithm, we have considered a well known FJSSP benchmark set described in the literature and available on the Internet at `http://www.idsia.ch/~monaldo/fjsp.html`. The set is composed of 21 instances initially provided by Barnes and Chambers (in the literature and in the rest of the paper this benchmark is referred to as *BCdata*), with the objective of minimizing the *makespan*. The benchmark is briefly described in the appendix of the work [Mastrolilli and Gambardella, 2000]. The IFS 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. In our experiments the $MaxFail$ parameter (see algorithm in Figure 3) was set to 100000; however, a maximum CPU time limit of 3200 seconds was set for each run.

Results. Table 1 shows the results obtained running our IFS algorithm on the *BCdata* set. The table is composed of eight columns and 24 rows, one row per instance plus 3 data wrap-up rows. The ***best*** column contains the best results known in current literature to the best of our knowledge; our results will therefore be compared against such values. In particular, each value in the ***best*** column represents the best makespan obtained with at least one of the approaches described in [Ben Hmida *et al.*, 2010; Mastrolilli and Gambardella, 2000; Gao *et al.*, 2008; Bozejko *et al.*, 2010]. The columns labeled $\gamma = 0.2$ to $\gamma = 0.7$ (see Section 4) contain the results obtained running the IFS procedure with a different value for the *relaxing factor* γ. The bold values in Table 1 represent the best results known in literature for each instance; the underlined bold values indicate the most significant improvements obtained by our IFS procedure (the relative instances have also been underlined). For each γ run, the last three rows of the table show respectively: (1) the number B of best solutions found (and, between brackets, the number **N** of *most recent* best solutions found in [Bozejko *et al.*, 2010]), (2) the average number of utilized solving cycles (Av.C.), and (3) the average mean relative error ($Av.MRE$)[6] respect to the lower bounds reported in [Mastrolilli and Gambardella, 2000]. The imposed CPU limit of 3200 seconds may appear very high, especially when compared with the limits imposed on some of the competing procedures; yet, the attention should be focused on the relatively low number of solving cycles our procedure requires to converge to a good solution. In fact, while the IFS metaheuristic generally requires a number of *relaxation/solving* cycles on the order of the tens of thousands, in our experimentation the best results were obtained with a number of cycles ranging from 6000 to 15000 (approximately), indicating the effectiveness of the inner PCP procedure.

Table 1: Results on the *BCdata* benchmark

inst.	***best***	γ					
		0.2	0.3	0.4	0.5	0.6	0.7
mt10x	**918**	980	936	936	934	**918**	**918**
mt10xx	**918**	936	929	936	933	**918**	926
mt10xxx	**918**	936	929	936	926	926	926
mt10xy	**905**	922	923	923	915	**905**	909
mt10xyz	**847**	878	858	851	862	**847**	851
mt10c1	**927**	943	937	986	934	934	**927**
mt10cc	**908**	926	923	919	919	910	911
setb4x	**925**	967	945	930	**925**	937	937
setb4xx	**925**	966	931	933	**925**	937	929
setb4xxx	**925**	941	930	950	950	942	935
setb4xy	**910**	**910**	941	936	936	916	914
setb4xyz	**905**	928	909	**905**	**905**	**905**	**905**
setb4c9	**914**	926	937	926	926	920	920
setb4cc	**907**	929	917	**907**	914	**907**	909
seti5x	**1199**	1210	**1199**	**1199**	1205	1207	1209
seti5xx	**1198**	1216	1199	1205	1211	1207	1206
seti5xxx	**1197**	1205	1206	1206	1199	1206	1206
seti5xy	**1136**	1175	1171	1175	1166	1156	1148
seti5xyz	**1125**	1165	1149	1130	1134	1144	1131
seti5c12	**1174**	1196	1209	1200	1198	1198	1175
seti5cc	**1136**	1177	1155	1162	1166	1138	1150
B (**N**)		1(**1**)	1(**1**)	3(**2**)	3(**0**)	6(**1**)	3(**0**)
Av.C.		15158	13521	10086	7994	7064	5956
Av.MRE		25.48	24.25	24.44	23.96	23.28	23.09

[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

As Table 1 shows, our procedure is competitive with the state of the art; more in detail, the three underlined instances in the table show exactly the same results recently obtained in [Bozejko *et al.*, 2010] with a *specialized* tabu search algorithm using a high performance GPU with 128 processors, while 10 previously known bests on a total of 21 instances are confirmed. Analyzing the $Av.MRE$ results, the best performances are obtained with γ values lying in the $[0.5, 0.7]$ range, which is confirmed by the higher concentration of confirmed best solutions. It is interesting to notice how the average number of solving cycles exhibits a steadily decreasing trend as γ increases; the more the solution is disrupted at each cycle, the greater the effort to find an alternative solution and hence the smaller number of solving cycles within the same CPU time limit. Nevertheless, the results show that increasing the disruption level in the range $[0.5, 0.7]$ helps in obtaining higher quality solutions, while increasing further triggers the opposite effect. As part of our ongoing work, we are testing our procedure on some of the biggest instances (i.e., *la36-la40*) of another well known FJSSP dataset, namely the Hurink *edata* dataset. This preliminary test is yielding promising results, as we improved two instances (namely, *la36*, from 1162 to 1160 and *la38*, from 1144 to 1143) w.r.t. the results published in [Mastrolilli and Gambardella, 2000].

7 Conclusions

In this paper we have proposed the use of Iterative Flattening Search (IFS) as a means of effectively solving the FJSSP. The proposed algorithm uses as its core solving procedure an extended version of the SP-PCP procedure originally proposed by [Oddi and Smith, 1997] and a new relaxation strategy targeted to the case of FJSSP. The effectiveness of the procedure was demonstrated on the *BCdata* benchmark set of FJSSPs. More specifically, the main contributions of this work are: (1) an extension of the slack-based value and variable ordering heuristics of [Oddi and Smith, 1997] for the FJSSP which, together with the propagation rules, constitute the basic components of the greedy PCP procedure; (2) a new relaxation strategy for the FJSSP; and (3) an evaluation of the full IFS algorithm against a challenging benchmark set for which many optimal solutions are still unknown. On average, the performance of IFS in this setting is found to be in line with the best-known algorithms published over the last 10 years. The fact that it was possible to adapt the original IFS procedure to the FJSSP problem without altering its core strategy, once again has proven the overall efficacy of the logic at the heart of IFS, as well as its versatility.

Further improvement of the current algorithm may be possible by incorporating additional heuristic information and search mechanisms. This will be the focus of future work. One possible direction is analysis of the effects on performance produced by a randomized version of the PCP algorithm. Another is the study of landscape analysis methods to balance exploration and exploitation search phases.

Acknowledgments

CNR authors are partially supported by EU under the ULISSE project (Contract FP7.218815), and MIUR under the PRIN project 20089M932N (funds 2008). S.F. Smith is funded in part by DARPA (Contract W911NF-11-2-0009).

References

[Ben Hmida *et al.*, 2010] A. Ben 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.

[Bozejko *et al.*, 2010] W. Bozejko, M. Uchronski, and M. Wodecki. Parallel meta^{2}heuristics for the flexible job shop problem. In L. Rutkowski et al. (Eds.), editor, *Proceedings of ICAISC 2010, Part II*, number 6114 in LNAI, pages 395–402. Springer-Verlag, Berlin Heidelberg, 2010.

[Cesta *et al.*, 2000] Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems. In *Proceedings of AAAI 2000*, pages 742–747, 2000.

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

[Gao *et al.*, 2008] 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.

[Garey and Johnson, 1979] 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.

[Godard *et al.*, 2005] Daniel Godard, Philippe Laborie, and Wim Nuitjen. Randomized Large Neighborhood Search for Cumulative Scheduling. In *Proceedings of ICAPS-05*, pages 81–89, 2005.

[Huguet and Lopez, 2000] M.-J. Huguet and P. Lopez. Mixed task scheduling and resource allocation problems. In *Proceedings of CP-AI-OR'00, Paderborn, Germany*, pages 71–79, March 2000.

[Mastrolilli and Gambardella, 2000] M. Mastrolilli and L. M. Gambardella. Effective neighbourhood functions for the flexible job shop problem. *Journal of Scheduling*, 3:3–20, 2000.

[Michel and Van Hentenryck, 2004] Laurent Michel and Pascal Van Hentenryck. Iterative Relaxations for Iterative Flattening in Cumulative Scheduling. In *Proceedings of ICAPS-04*, pages 200–208, 2004.

[Oddi and Smith, 1997] A. Oddi and S.F. Smith. Stochastic Procedures for Generating Feasible Schedules. In *Proceedings of AAAI-97*, pages 308–314, 1997.

[Oddi *et al.*, 2010] Angelo Oddi, Amedeo Cesta, Nicola Policella, and Stephen F. Smith. Iterative flattening search for resource constrained scheduling. *J. Intelligent Manufacturing*, 21(1):17–30, 2010.

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

[Rossi and Dini, 2007] 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.