Iterative Improvement Algorithms for the Blocking Job Shop

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 provides an analysis of the efficacy of a known iterative improvement meta-heuristic approach from the AI area in solving the Blocking Job Shop Scheduling Problem (BJSSP) class of problems. The BJSSP is known to have significant fallouts on practical domains, and differs from the classical Job Shop Scheduling Problem (JSSP) in that it assumes that there are no intermediate buffers for storing a job as it moves from one machine to another; according to the BJSSP definition, each job has to wait on a machine until it can be processed on the next machine. In our analysis, two specific variants of the iterative improvement meta-heuristic are evaluated: (1) an adaptation of an existing scheduling algorithm based on the Iterative Flattening Search and (2) an off-the-shelf optimization tool, the IBM ILOG CP Optimizer, which implements Self-Adapting Large Neighborhood Search. Both are applied to a reference benchmark problem set and comparative performance results are presented. The results confirm the effectiveness of the iterative improvement approach in solving the BJSSP; both variants perform well individually and together succeed in improving the entire set of benchmark instances.

Introduction

Over the past decades, several extensions of the classical job shop problem have been proposed in the literature. One such extension is the so-called *job shop with blocking constraints* or Blocking Job Shop Scheduling Problem (BJSSP). The BJSSP is a particularly meaningful problem as it captures the essence of a range of practical applications. It differs from the classical Job Shop Scheduling Problem (JSSP) in that it assumes that there are no intermediate buffers for storing a job as it moves from one machine to another. According to the BJSSP definition, each job has to wait on the machine that has just processed it until it can be processed on the next machine; the objective remains that of minimizing *makespan*.

The BJSSP actually relates fairly strongly to automated manufacturing trends and systems that have emerged in recent years. Modern manufacturing firms tend to adopt lean manufacturing principles and design production lines with minimal buffering capacity in order to limit inventory costs (Mati, Rezg, and Xie 2001). Yet, one of the continuing obstacles to effective use of emerging flexible manufacturing system (FMS) technologies is the difficulty of keeping heterogeneous jobs moving continuously through constituent work stations in a way that maximizes throughput. The BJSSP formulation is also relevant in other application contexts. The core model of many railway scheduling problems, for example, is also similar to the blocking job-shop problem (Mascis and Pacciarelli 2002; Strotmann 2007).

Despite its practical relevance, the BJSSP has received relatively little attention in comparison to the traditional JSSP, and none from the Artificial Intelligence (AI) research community. The idea behind this paper is to perform an initial analysis of the efficacy of a known iterative improvement meta-heuristic approach from the AI area in solving the BJSSP and, in doing so, expose this problem as a target for future research. To this end, we adapt an existing iterative improvement scheduling algorithm based on Iterative Flattening Search (IFS) to the BJSSP and evaluate its performance on a reference BJSSP benchmark problem set. Given the strong results produced by IFS, we also perform some preliminary testing with the IBM ILOG CP Optimizer (CP-OPT), an off-the-shelf optimization tool that implements another variant of iterative improvement search. An initial comparative analysis is presented toward the objective of encouraging future experimentation with other AI planning and scheduling techniques on this reference BJSSP benchmark and more generally on this important class of problem.

IFS was first introduced in (Cesta, Oddi, and Smith 2000) as a scalable procedure for solving multi-capacity scheduling problems. Extensions to the original IFS procedure were made in two subsequent works (Michel and Van Hentenryck 2004; Godard, Laborie, and Nuitjen 2005) and more recently the works (Oddi et al. 2011; Pacino and Hentenryck 2011) have applied the same meta-heuristic approach to successfully solve very challenging instances of the Flexible Job Shop Scheduling problem described in (Mastrolilli and Gambardella 2000). The IFS variant that we propose here relies on a *core* constraint-based search procedure as the BJSSP solver. This procedure 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 BJSSP's features.

In solving the BJSSP, the iterative improvement approach generally, and the adapted IFS procedure specifically, are found to be quite effective. The IFS variant succeeds in significantly improving the current best known results (Groeflin and Klinkert 2009; Groeflin, Pham, and Burgy 2011) on the reference benchmark (improving 37 of 40 instances), and on one problem instance achieves the theoretical minimum. When coupled with the results produced by the CP-OPT variant, the entire set of benchmark instances is improved. Overall, the results substantially strengthen the state-of-the-art on this problem. As a side effect, the analysis also confirms the versatility of the IFS technology in both the problem representation and solving phases.

The paper is organized as follows. The following section defines the BJSSP problem and provides a brief survey of existing approaches in the literature. The core of the paper is contained in the next three sections, which respectively describe the basic CSP representation, our core constraint-based search procedure and the IFS optimization meta-heuristic. An experimental section then describes the performance of both the IFS and CP-OPT algorithms on the reference benchmark and summarizes important characteristics. Some conclusions end the paper.

The Scheduling Problem

The BJSSP entails synchronizing the use of a set of machines (or resources) $R = \{r_1, \dots, r_m\}$ to perform a set of n activities $A = \{a_1, \dots, a_n\}$ over time. The set of activities is partitioned into a set of nj jobs $\mathcal{J} = \{J_1, \dots, J_{nj}\}$. 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. Each activity a_i has a processing time p_i and requires the exclusive use of a *single resource* $r(a_i) = r_i \in R$ for its entire duration. No *preemption* is allowed.

The BJSSP differs from the classical Job Shop Scheduling Problem (JSSP) in that it assumes that there are no intermediate buffers for storing a job $J_k = \{a_1, \dots, a_i, a_{i+1}, \dots, a_{nk}\}$ as it moves from one machine to another. Each job has to *wait* on a given machine until it can be processed on the next machine. Hence, each activity $a_i \in J_k$ is a *blocking* activity and remains on the machine $r(a_i)$ until its successor activity a_{i+1} starts. Due to the above described blocking features characterizing the BJSSP, the two following additional constraints must hold for activities belonging to the same job J_k. Let the variables s_i and e_i represent the start and end time of $a_i \in J_k$:

1. $e_i = s_{i+1}$, $i = 1 \dots n_k - 1$. This synchronization constraint enforces the action that a job is handed over properly from a_i to the following a_{i+1} in J_k. Hence, the starting time s_{i+1} (i.e., the time when the job enters the machine $r(a_{i+1})$) and the end time e_i (i.e., the time when the job leaves the machine $r(a_i)$) must be equal. According to the usual BJSSP formulation, for each activity a_i we make the assumption that there is both an instantaneous *take-over* step, coinciding with s_i and an instantaneous *hand-over* step, coinciding with e_i.
2. $e_i - s_i \geq p_i$. The total holding time of the activity a_i on the machine r_i in the solution is greater or equal to the activity processing time p_i, as we have to consider an additional waiting time due to the blocking constraints.

A *solution* $S = \{s_1, s_2, \dots, s_n,\}$ is a set of assigned start times s_i that satisfy all of the above constraints. Let C_k be the completion time for the job J_k. The solution *makespan* is the greatest completion time of all jobs, i.e., $C_{max} = max_{1 \leq k \leq nj}\{C_k\}$. An *optimal* solution S^* is a solution S with the minimum value of C_{max}.

It should be underscored that in this work we consider the *swap* version of the BJSSP problem. The need to swap operations between machines is incurred if a set of blocking operations exists where each one is waiting for a machine occupied by another operation in the set. Thus, the only solution to this deadlock situation (caused by the blocking restriction) is that all operations of the set can swap to their next machine *simultaneously*, so that all corresponding successor operations can start at the same time. Note that in a BJSSP the last operations of all jobs are *non-blocking*. Moreover, swapping makes no sense for the last operations of jobs as they leave the system after their completion. It has been demonstrated that the swap BJSSP problem tackled in this work is *NP-complete* (Strotmann 2007).

In general, solving scheduling problems with blocking constraints is more difficult than solving the classical job shop. In fact, though each feasible partial JSSP solution always admits a feasible complete schedule, given a partial BJSSP feasible solution (with swaps allowed), the latter admits a feasible complete schedule only from the so-called *positional selections* (a special class of feasible partial schedules) (Mascis and Pacciarelli 2002). On the contrary, the same problem is *NP-complete* if swapping of operations is not allowed (Mascis and Pacciarelli 2002).

Existing Approaches

Job shop models with blocking constraints have been discussed by several authors. (Hall and Sriskandarajah 1996) gives a survey on machine scheduling problems with blocking and no-wait constraints (a no-wait constraint occurs when there exists a maximum temporal separation between the start times of two consecutive operations in a job). In (Mascis and Pacciarelli 2002) the authors analyze several types of job shop problems including the classical job shop, the Blocking Job Shop, with and without "swaps", (note that in this work we tackle the swap BJSSP version) and the no-wait job shop; the authors then formulate these problems by means of *alternative graphs*. They also propose three specialized dispatching heuristics for these job shop problems and present empirical results for a large set of benchmark instances. Recently, the work (Groeflin and Klinkert 2009) has introduced a *tabu search* strategy for solving an extended BJSSP problem including setup times, and successive work (Groeflin, Pham, and Burgy 2011) has proposed a further extension of the problem that also considers flexible machines. To the best of our knowledge, (Groeflin and Klinkert 2009) and (Groeflin, Pham, and Burgy 2011) also provide the best known results for the BJSSP benchmark used in this work, hence they provide the reference results for our empirical analysis.

A CSP Representation

There are different ways to model the problem as a *Constraint Satisfaction Problem* (CSP) (Montanari 1974). In this work we use an approach similar to (Mascis and Pacciarelli 2002), which formulates the problem as an optimization problem on a generalized *disjunctive graph* called an *alternative graph*. In particular, we focus on imposing *simple precedence constraints* between pairs of activities that require the same resource, so as to eliminate all possible resource usage conflicts.

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}, representing the beginning (reference) and the end (horizon) of the schedule, respectively. Each activity a_i is labelled with the resource r_i it requires. J is a set of directed edges (a_i, a_j) representing the precedence constraints among the activities (job precedences constraints), each one labelled with the processing times p_i of the edge's source activity a_i.

The set of undirected edges X represents the *disjunctive constraints* among the activities that require the same resource; in particular, there is an edge for each pair of activities a_i and a_j requiring the same resource r, labelled with 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 set of decision variables o_{ijr} is defined for each pair of activities a_i and a_j requiring the same resource r. Each decision variable o_{ijr} can take one of two values $a_i \preceq a_j$ or $a_j \preceq a_i$. As we will see in the next sections, the possible decision values for o_{ijr} can be represented as the following two temporal constraints: $e_i - s_j \leq 0$ (i.e. $a_i \preceq a_j$) or $e_j + s_j \leq 0$ (i.e. $a_j \preceq a_i$).

To support the search for a consistent assignment to the set of decision variables o_{ijr}, for any BJSSP we define the directed graph $G_d(V, E)$, called *distance graph*, which is an extended version of the graph $G(A_G, J, X)$. In $G_d(V, E)$, the set of nodes V represents *time points*, where tp_0 is the *origin* time point (the reference point of the problem), and 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, each edge $(tp_i, tp_j) \in E$ with label b represent the linear constraint $tp_j - tp_i \leq b$. For example, the constraint $e_i - s_i \geq p_i$ on the activity a_i is represented by the edge (e_i, s_i) with label $-p_i$. The graph $G_d(V, E)$ represents a *Simple Temporal Problem* (STP) and its consistency can be efficiently determined via *shortest path computations* (Dechter, Meiri, and Pearl 1991).

Basic Constraint-based Search

The proposed procedure for solving instances of BJSSP 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.

Propagation Rules

The first way to exploit shortest path information is by introducing a set of *Dominance Conditions*, 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 $co\text{-}slack(e_i, s_j)$ (complementary slack) play a central role in the definition of such new dominance conditions.

Given two activities a_i, a_j and the shortest path distance distance $d(tp_i, tp_j)$ on the graph G_d, according to (Dechter, Meiri, and Pearl 1991), we have the following definitions:

– $slack(e_i, s_j) = d(e_i, s_j)$ represents the maximal distance between a_i and a_j (i.e., between the end-time e_i of a_i and the start-time s_j of a_j), and therefore provides a measure of the degree of *sequencing flexibility* between a_i and a_j [1]. If $slack(e_i, s_j) < 0$, then the ordering $a_i \preceq a_j$ is not feasible.

– $co\text{-}slack(e_i, s_j) = -d(s_j, e_i)$ represents the minimum possible distance between a_i and a_j; if $co\text{-}slack(e_i, s_j) \geq 0$, then there is no need to separate a_i and a_j, as the separation constraint $e_i \leq s_j$ is already satisfied.

For any pair of activities a_i and a_j that are competing for the same resource r, the dominance conditions describing the four possible cases of conflict are defined as follows:

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

Condition 1 represents an *unresolvable conflict*. There is no way to order a_i and a_j 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 $co\text{-}slack(e_j, s_i) < 0$ implies that the minimal distance between the end time e_j and the start time s_i is smaller than zero, and we still need to impose the constraint $e_j - s_i \leq 0$. In other words, the $co\text{-}slack$ condition avoids the imposition of unnecessary precedence constraints for trivially solved conflicts. Condition 3 works similarly, and implies 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*.

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

```
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 1: The PCP one-shot algorithm

Heuristic Analysis

The second way of exploiting shortest path information is by defining *variable* and *value* ordering heuristics for the decision variables o_{ijr} in all cases where no mandatory decisions can be deduced from the propagation phase. The basic idea is to repeatedly evaluate the decision variables o_{ijr} and select the one with the minimum heuristic evaluation. The selection of which variable to assign next is based on the *most constrained first* (MCF) principle, and the selection of values follows the *least constraining value* (LCV) heuristic, as explained below.

As previously stated, $slack(e_i, s_j)$ 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

$$VarEval(a_i, a_j) = \frac{1}{\sqrt{S}} min\{slack(e_i, s_j), slack(e_j, s_i)\}$$

where $S = \frac{min\{slack(e_i,s_j),slack(e_j,s_i)\}}{max\{slack(e_i,s_j),slack(e_j,s_i)\}}$ [2], is always selected for resolution.

In contrast 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 $slack(e_i, s_j) > slack(e_j, s_i)$ and $a_j \preceq a_i$ is selected otherwise.

[2] 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.

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 the previous section (A CSP Representation). The procedure also 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 all pairs (a_i, a_j) of activities that require the same resource (i.e., the *decision variables* o_{ijr} 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 previous section), on the basis of the response provided by the *dominance conditions*.

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 updates 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 PCP procedure shown in Figure 1, new constraints are added incrementally (one-by-one) to G_d, hence the complexity of this step is in the worst case $O(n^2)$. For the present analysis, we have adapted the incremental All Pair Shortest Path algorithm proposed in (Ausiello et al. 1991), as it guarantees that only the interested part of the temporal network is affected by each propagation.

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 decision variables o_{rij} are fixed and the imposed precedence constraints are satisfied), according to the following proposition: *A solution S is found when none of the four dominance conditions is verified on S* (Oddi and Smith 1997). At this point, each subset of activities A^r requiring the same resource r is totally ordered over time and the G_d graph represents a consistent Simple Temporal Problem. Moreover, as described in (Dechter, Meiri, and Pearl 1991), one possible solution to the problem is the so-called *earliest-time solution*, such that $S_{est} = \{S_i = -d(s_i, tp_0) : i = 1 \dots n\}$

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 non-improving iterations is reached. 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 explained in the following section. After initialization (Steps 1-2), a solution is repeatedly modified within the while loop (Steps 3-10) by applying the RELAX procedure (as explained below), and the PCP procedure shown in Figure 1 is used as flattening step. At each iteration, the RE-

```
IFS(S, MaxFail, γ)
begin
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)
end
```

Figure 2: The IFS schema

LAX step reintroduces the possibility of resource contention, and the PCP step is called again to restore resource feasibility. If 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.

Relaxation Procedure

The first keystone of the IFS cycle is the *relaxation step*, wherein a feasible solution is relaxed into a possibly resource infeasible but precedence feasible schedule, by retracting some number of scheduling decisions. In this phase we use a strategy similar to the one employed in (Godard, Laborie, and Nuitjen 2005) called *chain-based relaxation*. 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 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 that 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* procedure proceeds in two steps. First, a subset of activities a_i is randomly selected from the input solution S; the selection process is generally driven by a parameter $\gamma \in (0, 1)$ whose value is related to the probability that a given activity will be selected (γ is called the *relaxing factor*). Second, a procedure similar to CHAINING – used in (Policella et al. 2007) – is applied to the set of unselected activities. This operation is in turn accomplished in three steps: (1) all previously posted solving 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 related to the sequence $p \preceq a_i$ is posted (the dummy activity a_0 is the first activity of all chains). This last step is iterated until all the activities are linked by the correct 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 may generally recreate *conflicts* in resource usage.

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

Random selection According to this 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. The γ value represents the percentage of activities that will be relaxed from the current solution, and every activity retains the same probability to be selected for relaxation.

Slack-based selection As opposed to random selection where at each iteration every activity is potentially eligible for relaxation, the slack-based selection approach restricts the pool of relaxable activities to the subset containing those activities that are closer to the *critical path condition* (the activities on the critical path are those that determine the schedule's makespan). The rationale is that relaxing activities in the vicinity of the critical path should promote more efficient makespan reductions. For each activity a_i we define two values: 1) the *duration flexibility* $df_i = d(e_i, s_i) + d(s_i, e_i)$, representing the flexibility to extend the duration of activity a_i without changing the makespan; and 2) the waiting time $w_i = -d(s_i, e_i) - p_i$, representing the minimal additional time that activity a_i remains blocked on the requested machine $r(a_i)$ with respect to the processing time p_i. For BJSSP, we consider that both df_i and w_i play a role in determining an activity's *proximity* to the critical path condition, which is therefore assessed by combining these two values into a parameter called *duration slack* $ds_i = df_i + w_i$. An activity a_i is chosen for inclusion in the set of activities to be removed on a given IFS iteration with probability directly proportional to the γ parameter and inversely proportional to a_i's duration slack ds_i Note that for identical values of the relaxation parameter γ, the slack-based relaxation generally implies a smaller disruption to the solution S, as it operates on a smaller set of activities; those activities characterized by a large slack value will have a minimum probability to be selected.

Experimental Analysis

In this section, we present quantitative evidence of the effectiveness of the above described iterative improvement algorithms on a previously studied BJSSP benchmark. Our analysis proceeds in two steps. First, we perform a detailed comparison of the performance of various IFS configurations. Second we evaluate the performance of the IBM ILOG CP Optimizer (IBM Academic Initiative 2012), which makes use of a different iterative improvement search procedure, and add these results to the overall performance comparison. In both cases we observe a substantial improvement in performance relative to the current state-of-the-art, demonstrating both the versatility and the robustness of the iterative improvement approach.

Experimental Setup

We have performed extensive computational tests on a set of 40 Blocking Job Shop (BJSSP) benchmark instances obtained from the standard *la01-la40* JSSP testbed proposed by Lawrence (Lawrence 1984). These problems are directly loaded as BJSSP instances, by imposing the additional constraints as described in Section (The Scheduling Problem). The 40 instances are subdivided into the following 8 $(nJ \times nR)$ subsets, where nJ and nR represent the number of jobs and resources, respectively: *[la01-la05]* (10×5), *[la06-la10]* (15×5), *[la11-la15]* (20×5), *[la16-la20]* (10×10), *[la21-la25]* (15×10), *[la26-la30]* (20×10), *[la31-la35]* (30×10), and finally *[la36-la40]* (15×15).

As stated earlier, in order to broaden the analysis of the performance of existing constraint-based approaches we performed further experiments on the same benchmarks using the IBM ILOG CP Optimizer (CP-OPT). CP-OPT implements the approach described in (Laborie and Godard 2007) called Self-Adapting Large Neighborhood Search (SA-LNS). Similarly to IFS, SA-LNS is a randomized iterative improvement procedure based on the cyclic application of a relaxation step followed by a re-optimization step of the relaxed solution. In SA-LNS, both steps may generally vary between any iteration, according to a *learning algorithm* which performs the self adaptation.

To keep the experiments conditions as equal as possible to those of previously published results, the time limit for each run was set to 1800 sec. The IFS algorithm variants were implemented in Java and run on a AMD Phenom II X4 Quad 3.5 Ghz under Linux Ubuntu 10.4.1. The CP-OPT Optimizer was instead run on the same machine under Windows 7.

Comparing IFS results with current bests

Table 1 and Table 2 show the performance of the IFS solving procedure using each of the two selection strategies explained above, *Random Relaxation* and the *Slack-based* Relaxation, respectively. In Tables 1 and 2, the *inst.* column lists all the benchmark instances according to the following criteria: instances in bold are those that have been improved with respect to the current best, while the bold underlined instances represent improvements with respect to their counterparts in the other table, in case both solutions improve the current best. The most recent known results available in literature to the best of our knowledge are shown in the *best* column of both tables. These numeric values have been obtained by intersecting the best results presented in (Groeflin and Klinkert 2009) and (Groeflin, Pham, and Burgy 2011), as they represent the most recent published results for BJSSP instances. The remaining 8 columns of Table 1 present the best result obtained for each instance as the γ retraction parameter value ranges from 0.1 to 0.8, while Table 2 exhibits the same pattern as γ ranges between 0.2 and 0.9. For each instance, bold values represent improved solutions with respect to current bests (*relative* improvements), while bold underlined values represent the best values obtained out of all runs (*absolute* improvements). Values marked with an asterisk correspond to theoretical optima found by (Mascis and Pacciarelli 2002), specifically, the la19 instance, with $\gamma = 0.8$ (see Table 1). For all instances, the best out of 2 different runs was chosen. In both tables, all $(nJ \times nR)$ activity subsets have been interleaved with specific rows (*Av.C.*) presenting the average number of {*relaxation - flatten*} cycles (expressed in thousands) performed by our procedure to solve the subset instances for each γ value. The first value of the last row (*# impr.*) shows the total number of absolute improvements with respect to the current bests out of all runs, while the remaining values represent such improvements for each individual value of γ.

Table 1: Results with random selection procedure

inst.	***best***	γ							
		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
la01	820	907	859	865	850	857	**818**	**818**	**793**
la02	793	854	848	858	815	826	793	793	793
la03	740	787	773	770	**721**	**715**	740	740	**715**
la04	764	810	802	770	**761**	**756**	766	**756**	**743**
la05	666	776	761	689	680	671	**664**	680	**664**
10x5	Av.C.	143K	124K	109K	95K	83K	79K	85K	98K
la06	1180	1234	**1165**	1182	1196	**1151**	**1087**	**1121**	**1177**
la07	1084	1188	1130	1114	1102	1087	**1046**	**1070**	1127
la08	1125	1212	1152	1157	**1105**	**1097**	**1107**	**1087**	**1100**
la09	1223	1294	1308	1240	1241	1251	1235	1226	**1212**
la10	1203	1287	1224	1243	1232	**1110**	**1127**	**1167**	**1179**
15x5	Av.C.	47K	39K	35K	30K	26K	26K	27K	28K
la11	1584	1726	**1516**	1588	**1498**	**1566**	**1536**	1635	1672
la12	1391	1554	1512	**1370**	**1358**	**1290**	**1272**	1463	1429
la13	1541	1673	1614	**1523**	**1465**	**1479**	**1482**	1638	1657
la14	1620	1766	1649	**1590**	**1556**	**1610**	**1594**	1665	1729
la15	1630	1779	1682	**1576**	**1527**	**1564**	**1586**	1692	1789
20x5	Av.C.	19K	16K	14K	12K	10K	11K	10K	10K
la16	1142	1278	1165	**1134**	1170	1168	**1106**	**1086**	1151
la17	977	1184	1130	1025	997	1076	**930**	995	986
la18	1078	1182	1214	1154	**1040**	1132	1081	1082	**1049**
la19	1093	1145	1193	1127	1176	1127	1104	**1053**	**1043***
la20	1154	1228	1170	1229	1173	1161	1142	**1074**	**1099**
10x10	Av.C.	29K	25K	22K	21K	19	20K	19K	18
la21	1545	1742	1731	1678	1587	**1530**	1688	1724	1808
la22	1458	1653	1523	1468	**1455**	1494	1485	1593	1651
la23	1570	1851	1695	1626	**1531**	1597	**1553**	1728	1795
la24	1546	1740	1584	1675	**1503**	**1517**	1631	1675	1696
la25	1499	1618	1552	1670	1545	**1437**	1562	1660	1666
15x10	Av.C.	8K	8K	6K	6K	5K	5K	5K	5K
la26	2125	2230	2249	2265	**2109**	2320	2410	2434	2437
la27	2175	2385	2355	2267	**2172**	2427	2661	2642	2667
la28	2071	2287	2211	**2027**	2162	2449	2456	2529	2571
la29	1990	2379	**1988**	2047	2108	2100	2296	2301	2427
la30	2097	2266	2218	2162	**2095**	2146	2437	2442	2482
20x10	Av.C.	3K	3K	3K	2K	2K	2K	2K	2K
la31	3137	3422	3175	3213	3745	3848	3913	3876	3933
la32	3316	-	3336	3673	3963	4057	4127	4158	4157
la33	3061	3315	3147	3252	3521	3710	3816	3800	3960
la34	3146	3273	3267	3479	3526	3827	3904	3919	3924
la35	3171	-	**3148**	3654	3718	3881	3882	3871	3882
30x10	Av.C.	0.6K	0.7K	0.6K	0.6K	0.5K	0.5K	0.4K	0.5K
la36	1919	2155	2096	**1793**	1973	2241	2097	2322	2223
la37	2029	2165	2037	2167	**2004**	2034	2270	2445	2386
la38	1828	2091	1931	**1775**	1852	1839	2070	2090	2059
la39	1882	2108	2074	1914	**1783**	**1828**	1884	2064	2110
la40	1925	2207	**1914**	**1831**	2036	**1884**	2125	2068	2246
15x15	Av.C.	3K	2K	2K	2K	2K	2K	2K	1K
# impr.	**35**	**0**	**2**	**4**	**13**	**4**	**5**	**4**	**6**

The results in Tables 1 and 2 show that both the *Random* and the *Slack-based* relaxation procedure exhibit remarkably good performance in terms of number of absolute improvements (35 and 34, respectively, on a total of 40 instances), despite the fact that the Slack-based approach allows a fewer number of solving cycles within the allotted time, due to the more complex selection process. This circumstance is even more remarkable once we highlight that the quality of the improved solutions obtained with the slack-based relaxation is often higher than the quality obtained with the random counterpart; it is enough to count the number of the bold underlined instances in both tables

Table 2: Results with the slack-based selection procedure

inst.	*best*	γ							
		0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
la01	820	829	**794**	884	**793**	**793**	**818**	**818**	**793**
la02	793	858	814	837	793	793	814	793	793
la03	740	798	761	754	740	**715**	740	**715**	**715**
la04	764	770	**756**	**756**	776	**743**	**743**	**743**	**743**
la05	666	723	704	693	**664**	679	671	**664**	**664**
10x5	Av.C.	126K	112K	97K	90K	84	80K	85K	90K
la06	1180	**1170**	**1158**	**1148**	**1112**	**1076**	**1064**	**1123**	**1138**
la07	1084	1180	1099	1088	**1081**	**1079**	**1038**	**1075**	**1063**
la08	1125	1205	1172	1132	1135	**1062**	**1087**	**1113**	1150
la09	1223	1304	1290	1283	1257	1228	**1185**	**1205**	1238
la10	1203	**1174**	**1197**	**1141**	**1158**	**1181**	**1119**	**1159**	**1146**
15x5	Av.C.	40K	35K	32K	29K	26K	25K	27K	27K
la11	1584	1670	**1582**	**1559**	**1501**	**1466**	1604	1621	1664
la12	1391	**1383**	**1367**	**1345**	**1321**	**1296**	**1309**	**1295**	1452
la13	1541	1583	**1480**	**1515**	**1471**	**1498**	**1520**	**1514**	1623
la14	1620	**1610**	**1596**	**1577**	**1567**	**1575**	**1548**	1628	1696
la15	1630	1702	**1612**	**1629**	**1547**	**1606**	**1551**	**1566**	1753
20x5	Av.C.	17K	14K	12K	11K	11K	10K	10K	10K
la16	1142	1173	1186	1208	**1086**	**1108**	**1119**	**1084**	**1086**
la17	977	1095	1063	1036	1000	**974**	**951**	**930**	**967**
la18	1078	1141	**1038**	**1040**	1120	1090	**1026**	**1026**	**1026**
la19	1093	1207	1122	1109	**1077**	**1082**	**1076**	**1077**	**1068**
la20	1154	1211	1165	1156	1166	**1122**	**1141**	**1087**	**1094**
10x10	Av.C.	24K	23K	21K	19K	19K	19K	19K	19K
la21	1545	1762	1689	1618	**1521**	1572	1696	1771	1712
la22	1458	1541	1504	1486	1490	**1425**	1546	1551	1635
la23	1570	1639	1595	**1554**	**1538**	**1538**	1681	1736	1694
la24	1546	1690	1547	**1538**	**1498**	**1544**	**1518**	1548	1682
la25	1499	**1495**	1547	1527	**1424**	1557	1501	1574	1688
15x10	Av.C.	7K	7K	6K	6K	6K	5K	5K	5K
la26	2125	2180	**2045**	**2117**	2179	2292	2395	2420	2437
la27	2175	2233	2176	**2104**	**2172**	2427	2661	2642	2667
la28	2071	2287	2211	2104	2132	2352	2500	2476	2649
la29	1990	2049	2004	2010	**1963**	2163	2305	2300	2389
la30	2097	2109	2156	2135	2125	2419	2460	2492	2485
20x10	Av.C.	3K	3K	2K	2K	2K	2K	2K	2K
la31	3137	**3078**	3271	3500	3771	3899	3888	3863	3941
la32	3316	3428	3827	4045	3852	4158	4064	4157	4152
la33	3061	3372	3213	3436	3741	3717	3896	3949	3970
la34	3146	3328	**3125**	3752	3796	3917	3935	3929	3933
la35	3171	3243	3274	3631	3818	3822	3875	-	-
30x10	Av.C.	0.6K	0.6K	0.6K	0.6K	0.5K	0.5K	0.4K	0.5K
la36	1919	**1916**	1938	1939	**1891**	2010	2004	2283	2199
la37	2029	2172	2055	**1984**	**1983**	**2004**	2179	2227	2396
la38	1828	**1798**	1894	1849	**1708**	1854	2088	1995	2121
la39	1882	1918	**1872**	**1806**	**1848**	**1862**	2136	2141	2161
la40	1925	**1917**	**1777**	**1849**	**1831**	1992	2108	2048	2195
15x15	Av.C.	3K	2K	2K	2K	2K	2K	2K	2K
# impr.	**34**	**1**	**3**	**2**	**22**	**7**	**7**	**7**	**6**

to confirm that the slack-based relaxations outperforms the random relaxations on 20 improved solutions, while the opposite is true in 11 cases only. This difference in efficacy is further confirmed by the higher number of relative improvements obtained with the slack-based approach, i.e., 129 against 80 (these last figures are not explicitly shown in the tables).

Regarding the relation between solution quality and γ values, the following can be observed. Regardless of the chosen selection procedure, both approaches tend to share the same behavior: as the γ value increases, the effectiveness of the solving procedure seems to increase until reaching a maximum, before showing the opposite trend. Indeed, we observe that the highest number of absolute improvements in both tables is obtained with $\gamma = 0.4$ and $\gamma = 0.5$ in the random and slack-based case, respectively.

The obtained results also seem to convey that there exists a relation between the most effective γ value and the size of the instance. In particular, it can be observed that smaller instances generally require a greater relaxation factor, while bigger instances are best solved by means of small relaxations. This circumstance is verified in both tables; the instances belonging to the *[la1-la20]* are best solved within the $[0.5, 0.9]\gamma$ range, while the best solutions to the *[la21-la40]* instances are found within the $[0.2, 0.4]$ range.

Table 3: Comparison with the results obtained with CP-OPT

inst.	*best*	*cp*	*ifs*	*inst.*	*best*	*cp*	*ifs*	*inst.*	*best*	*cp*	*ifs*
la01	820	**793**	**793**	*la15*	1630	1571	**1527**	*la29*	1990	**1898**	1963
la02	793	815	**793**	*la16*	1142	1150	**1084**	*la30*	2097	2147	**2095**
la03	740	790	**715**	*la17*	977	996	**930**	*la31*	3137	**2921**	3078
la04	764	784	**743**	*la18*	1078	1135	**1026**	*la32*	3316	**3237**	3336
la05	666	**664**	**664**	*la19*	1093	1108	**1043**	*la33*	3061	**2844**	3147
la06	1180	1131	**1064**	*la20*	1154	1119	**1074**	*la34*	3146	**2848**	3125
la07	1084	1106	**1038**	*la21*	1545	1579	**1521**	*la35*	3171	**2923**	3148
la08	1125	1129	**1062**	*la22*	1458	**1379**	1425	*la36*	1919	1952	**1793**
la09	1223	1267	**1185**	*la23*	1570	**1497**	1531	*la37*	2029	**1952**	1983
la10	1203	1168	**1110**	*la24*	1546	1523	**1498**	*la38*	1828	1880	**1708**
la11	1584	1520	**1466**	*la25*	1499	1561	**1424**	*la39*	1882	1813	**1783**
la12	1391	1308	**1272**	*la26*	2125	**2035**	2045	*la40*	1925	1928	**1777**
la13	1541	1528	**1465**	*la27*	2175	2155	**2104**				
la14	1620	**1506**	1548	*la28*	2071	2062	**2027**				

Adding CP-OPT to the comparison

Table 3 compares the results obtained with the IFS and the CP-OPT procedures against the current bests. In particular, the table lists all the problem instances (***inst.*** column) and for each instance it compares the current best (***best*** column) with the CP-OPT bests (***cp*** column) and with the IFS bests (***ifs*** column). The CP-OPT results are obtained running the solver once for every instance, using the default search parameters in (IBM Academic Initiative 2012). In the table, bold underscored figures represent the absolute best values obtained between IFS and CP-OPT.

If we compare the previous results with those related to the CP-OPT solver, we notice that the latter performs exceptionally well, given that the CP-OPT results are compared against the merged best results obtained in Tables 1 and 2, and that all CP-OPT runs have been performed only once. In this regard, it should be noted how CP-OPT succeeds in improving 25 of the 40 total instances with respect to the current bests, in its first attempt. These results indicate the effectiveness of the iterative improvement solving approach to the BJSSP class of problems tackled in this work, regardless whether it is implemented through IFS or SA-LNS.

As a last observation, it should be noted how the performance of IFS is affected by the size of the problem in terms of average number of solving cycles. In Table 1, for example, we pass from an average of $\approx 120K$ cycles for the (10×5) instances down to an average of $\approx 0.5K$ cycles for the (30×10) instances. This same effect is observable in Table 2. Indeed, this aspect represents the most important limitation of IFS, even more in comparison to the performance of CP-OPT. In the IFS case, if reasoning over an explicit representation of time on the one hand provides a basis for very efficient search space cuts by means of propagation, it can on the other hand become a bottleneck as the problem size increases. This limitation is confirmed by the lower quality results obtained for the *[la31-la35]* subset, where IFS random (slack-based) selection is able to improve only 1 (resp. 2) of 5 solutions. In contrast, the problem instances with the highest concentration of CP-OPT improvements with respect to IFS are the larger sized problems, which highlights the higher efficiency of the SA-LNS procedure implemented in

CP-OPT when the size of the problem increases beyond a certain threshold. Yet, despite the IFS limitation highlighted above, it can be observed that the overall solution quality remains acceptable at least for one γ value, which indicates that IFS is characterized by a rather good converging speed.

Conclusions and Future Work

In this paper we have proposed the use of iterative improvement search to solve the Blocking Job Shop Scheduling Problem (BJSSP). The BJSSP represents a relatively unstudied problem of practical real-world importance, and one goal of this paper has been to raise awareness of this problem class within the AI research community.

Two different iterative improvement algorithms have been evaluated: (1) Iterative Flattening Search (IFS) (adapted from (Oddi et al. 2011)), and (2) Self-Adapting, Large Neighborhood Search (SA-LNS) ((Laborie and Godard 2007)) as it is implemented in the IBM ILOG CP Optimizer. Experimental results on a reference BJSSP benchmark problems demonstrated the general efficacy of both algorithms. Both variants were found to produce very good results with respect to the currently published best known results. Overall, new best solutions were found for the entire benchmark problem set, and in one case the known theoretical optimum value was achieved. With respect to the two individual approaches, SA-LNS was found to exhibit better scaling properties while IFS achieved better results on problem instances of medium and smaller size. Our hypothesis is that this behavior is due to the more specific heuristic employed in IFS in comparison to the general search strategy provided in the default implementation of the SA-LNS algorithm. Given the generality and versatility of both procedures, we believe that one immediate extension of the current work can be towards the JSSP with Limited Capacity Buffers ((Brucker et al. 2006)).

Acknowledgments

CNR authors are partially supported by EU under the GiraffPlus project (Contract FP7.288173), and MIUR under the PRIN project 20089M932N (funds 2008). Stephen F. Smith is funded in part by DARPA Contract W911NF-11-2-0009 and the CMU Robotics Institute.

References

Ausiello, G.; Italiano, G. F.; Marchetti-Spaccamela, A.; and Nanni, U. 1991. Incremental algorithms for minimal length paths. *J. Algorithms* 12(4):615–638.

Brucker, P.; Heitmann, S.; Hurink, J. L.; and Nieberg, T. 2006. Job-shop scheduling with limited capacity buffers. *OR Spectrum* 28(2):151–176.

Cesta, A.; Oddi, A.; and Smith, S. F. 2000. Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems. In *Proceedings of AAAI 2000*, 742–747.

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 *Proceedings of ICAPS-05*, 81–89.

Groeflin, H., and Klinkert, A. 2009. A new neighborhood and tabu search for the blocking job shop. *Discrete Applied Mathematics* 157(17):3643–3655.

Groeflin, H.; Pham, D.; and Burgy, R. 2011. The flexible blocking job shop with transfer and set-up times. *Journal of Combinatorial Optimization* 22:121–144.

Hall, N. G., and Sriskandarajah, C. 1996. A survey of machine scheduling problems with blocking and no-wait in process. *Operations Research* 44(3):510–525.

IBM Academic Initiative. 2012. IBM ILOG CPLEX Optimization Studio v12.4.

Laborie, P., and Godard, D. 2007. Self-Adapting Large Neighborhood Search: Application to single-mode scheduling problems. In *Proceedings of 3rd Multidisciplinary International Scheduling Conference: Theory and Applications (MISTA 2007). Paris (France), Aug. 2007*, 276–284.

Lawrence, S. 1984. Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement). Technical report, Graduate School of Industrial Administration, Carnegie Mellon University.

Mascis, A., and Pacciarelli, D. 2002. Job-shop scheduling with blocking and no-wait constraints. *European Journal of Operational Research* 143(3):498 – 517.

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

Mati, Y.; Rezg, N.; and Xie, X. 2001. Scheduling problem of job-shop with blocking: A taboo search approach. In *MIC 2001 - 4th Metaheuristics International Conference, Portugal*, 643–648.

Michel, L., and Van Hentenryck, P. 2004. Iterative Relaxations for Iterative Flattening in Cumulative Scheduling. In *Proceedings of ICAPS-04*, 200–208.

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

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.; Rasconi, R.; Cesta, A.; and Smith, S. F. 2011. Iterative flattening search for the flexible job shop scheduling problem. In *IJCAI*, 1991–1996.

Pacino, D., and Hentenryck, P. V. 2011. Large neighborhood search and adaptive randomized decompositions for flexible jobshop scheduling. In *IJCAI*, 1997–2002.

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

Strotmann, C. 2007. *Railway scheduling problems and their decomposition*. Ph.D. Dissertation, Fachbereich Mathematik/Informatik der Universität Osnabrück.