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 fallout 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 ap-
proach 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) \( R = \{r_1, \ldots, r_m \} \) 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 \( n_j \) jobs \( J = \{J_1, \ldots, J_{n_j}\} \). 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 \( J_k = \{a_1, \ldots, a_i, a_{i+1}, \ldots, a_{n_k}\} \) 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. \( e_i = s_{i+1}, \ i = 1 \ldots n_k - 1 \). 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 \( S = \{s_1, s_2, \ldots, s_n, \} \) 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., \( C_{\text{max}} = \max_{1 \leq k < n_j} \{C_k\} \). An optimal solution \( S^* \) is a solution \( S \) with the minimum value of \( C_{\text{max}} \).

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 \( a_j \), \( a_i \leq r a_j \) or \( a_j \leq a_i \). Hence, in CSP terms, a set of decision variables \( a_{ijr} \) is defined for each pair of activities \( a_i \) and \( a_j \) requiring the same resource \( r \). Each decision variable \( a_{ijr} \) can take one of two values \( a_i \leq a_j \) or \( a_j \leq a_i \). As we will see in the next sections, the possible decision values for \( a_{ijr} \) can be represented as the following two temporal constraints: \( e_i - s_j \leq 0 \) (i.e. \( a_i \leq a_j \)) or \( e_j + s_i \leq 0 \) (i.e. \( a_j \leq a_i \)).

To support the search for a consistent assignment to the set of decision variables \( a_{ijr} \), 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 \( t_p \) 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 \((t_p, t_j)\) in \( E \) with label \( b \) represents the linear constraint \( t_j - t_p \leq b \). 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 \( \text{slack}(e_i, s_j) \) and \( \text{co-slack}(e_i, s_j) \) (complementary slack) play a central role in the definition of such new dominance conditions.

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

1. \( \text{slack}(e_i, s_j) = d(e_i, s_j) \) 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 \( a_j \).
2. \( \text{co-slack}(e_i, s_j) = -d(s_j, e_i) \) represents the minimum possible distance between \( a_i \) and \( a_j \); if \( \text{co-slack}(e_i, s_j) \geq 0 \), then there is no need to separate \( a_i \) and \( a_j \), as the separation constraint \( e_i - s_j \) 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. \( \text{slack}(e_i, s_j) < 0 \land \text{slack}(e_j, s_i) < 0 \)
2. \( \text{slack}(e_i, s_j) < 0 \land \text{co-slack}(e_j, s_i) < 0 \)
3. \( \text{slack}(e_i, s_j) \geq 0 \land \text{slack}(e_j, s_i) < 0 \land \text{co-slack}(e_i, s_j) < 0 \)
4. \( \text{slack}(e_i, s_j) \geq 0 \land \text{co-slack}(e_i, s_j) \geq 0 \)

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, 3, and 4, 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 \( a_j \leq a_i \) leaves \( G_d(V, E) \) consistent. It is worth noting that the presence of the condition \( \text{co-slack}(e_i, s_j) < 0 \) 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 \( e_j - s_i < 0 \). In other words, the \text{co-slack} condition avoids the imposition of unnecessary precedence constraints for trivially solved conflicts. Condition 3 works similarly, and implies that only the \( a_i \leq a_j \) 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.

\footnote{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\textsubscript{max})
1. S ← InitSolution(Problem, C\textsubscript{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 \textit{variable} and \textit{value} ordering heuristics for the decision variables \(a_{ijr}\) in all cases where no mandatory decisions can be deduced from the propagation phase. The basic idea is to repeatedly evaluate the decision variables \(a_{ijr}\) and select the one with the minimum heuristic evaluation. The selection of which variable to assign next is based on the \textit{most constrained first} (MCF) principle, and the selection of values follows the \textit{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 \textit{sequencing flexibility} between \(a_i\) and \(a_j\). The \textit{variable ordering heuristic} attempts to focus first on the conflict with the least amount of \textit{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

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

where \(S = \min \{\text{slack}(e_i, s_j), \text{slack}(e_j, s_i)\}^2\), is always selected for resolution.

In contrast to \textit{variable ordering}, the \textit{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 \textit{sequencing flexibility}. Specifically, \(a_i \leq a_j\) is selected if \(\text{slack}(e_i, s_j) > \text{slack}(e_j, s_i)\) and \(a_j \leq a_i\) is selected otherwise.

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 \textit{never-exceed} value \((C_{\text{max}})\) of the objective function of interest, used to impose an initial \textit{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 \textit{decision variables} \(a_{ijr}\) of the corresponding CSP problem), and decides their \textit{values} in terms of precedence ordering (i.e., \(a_i \leq a_j\) or \(a_j \leq a_i\), see previous section), on the basis of the response provided by the \textit{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 \text{Propagate}(S), which propagates the effects of posting a new solving decision (i.e., a constraint) in the graph \(G_d\). In particular, \text{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 \(o_{ijr}\) 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 \(S_{\text{est}} = \{s_i = -d(s_i, tp_0) : i = 1 \ldots n\}\)

The Optimization Metaheuristic

Figure 2 introduces the generic \textit{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 \(\gamma\) 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 \text{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. \text{RELAX}(S, γ)
5. \text{S} ← \text{PCP}(S, C_{\max}(S_{best}))
6. if $C_{\max}(S) < C_{\max}(S_{best})$ then
7. \text{S}_{best} ← \text{S}
8. counter ← 0
9. else
10. counter ← counter + 1
11. \text{return}(S_{best})
end

Figure 2: The IFS schema

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 retraction 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 $G = (A, J, X)$ 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 ($\gamma$ 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 $p = \text{pred}(a_i, r)$ 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.

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: \([la01-la05]\) \((10 \times 5)\), \([la06-la10]\) \((15 \times 5)\), \([la11-la15]\) \((20 \times 5)\), \([la16-la20]\) \((10 \times 10)\), \([la21-la25]\) \((15 \times 10)\), \([la26-la30]\) \((20 \times 10)\), \([la31-la35]\) \((30 \times 10)\), and finally \([la36-la40]\) \((15 \times 15)\).

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 (Lauriere 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 (Grobein and Klinkert 2009) and (Grobein, 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 \(\gamma\) retraction parameter value ranges from 0.1 to 0.8, while Table 2 exhibits the same pattern as \(\gamma\) 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 \(\{relaxation - flatter\}\) cycles (expressed in thousands) performed by our procedure to solve the subset instances for each \(\gamma\) value. The first value of the last row \(#\text{impr.}\) 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 \(\gamma\).

Table 1: Results with random selection procedure

<table>
<thead>
<tr>
<th>inst.</th>
<th>best</th>
<th>γ</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Av.C.</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
</tr>
<tr>
<td>10K.</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
</tr>
<tr>
<td>8K.</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
</tr>
<tr>
<td>3K.</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
</tr>
<tr>
<td>1K.</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
<td>134K</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>inst.</th>
<th>best</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>la01</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>la02</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
</tr>
<tr>
<td>la03</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>la04</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>la05</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>inst.</th>
<th>best</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>la01</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>la02</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
<td>119</td>
</tr>
<tr>
<td>la03</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>la04</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>la05</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
<td>120</td>
</tr>
</tbody>
</table>

Adding CP-OPT to the comparison

Table 3 compares the results obtained with the IFS and the CP-OPT procedures against the current best. In particular, the table lists the problem instances (inst. column) and for each instance 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 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 ~120 K cycles for the (10 x 5) instances down to an average of ~0.5 K cycles for the (30 x 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
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 Giraffe-Plus 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


