Randomized Large Neighborhood Search for Cumulative Scheduling

Daniel Godard and Philippe Laborie and Wim Nuijten
ILOG S. A.
9, rue de Verdun - B. P. 85
94253 Gentilly Cedex, France
{dgodard,plaborie,wnuijten}@ilog.fr

Abstract

This paper presents a Large Neighborhood Search (LNS) approach based on constraint programming to solve cumulative scheduling problems. It extends earlier work on constraint-based randomized LNS for disjunctive scheduling as reported in (Nuijten & Le Pape 1998). A breakthrough development in generalizing that approach toward cumulative scheduling lies in the presented way of calculating a partial-order schedule from a fixed start time schedule. The approach is applied and tested on the Cumulative Job Shop Scheduling Problem (CJSSP). An empirical performance analysis is performed using a well-known set of benchmark instances. The described approach obtains the best known performance reported to date on the CJSSP. It not only finds better solutions than ever reported before for 33 out of 36 open instances, it also proves to be very robust on the complete set of test instances. Furthermore, among these 36 open instances, one is now closed. As the approach is generic, it can be applied to other types of scheduling problems, for example problems including resource types like reservoirs and state resources, and objectives like earliness/tardiness costs and resource allocation costs.

Introduction

Scheduling can be described as the process of allocating scarce resources to activities over time. Traditionally, the class of disjunctive scheduling problems, where each resource can execute at most one activity at a time, has received a lot of attention. In this paper we are concerned with the class of cumulative scheduling problems, where resources may execute several activities in parallel, provided the resource capacity is not exceeded.

Many practical scheduling problems are cumulative scheduling problems and in recent years the attention for cumulative scheduling problems in general (Baptiste, Le Pape, & Nuijten 1999) and the resolution of these problems by way of local search in particular has increased (Cesta, Oddi, & Smith 2000; Michel & Van Hentenryck 2003; Palpant, Artigues, & Michelon 2004). Our approach is an implementation of the Large Neighborhood Search frame-

work (Shaw 1998) which is based upon a process of continual relaxation and re-optimization. We apply this approach on a generalization of the Job Shop Scheduling Problem (French 1982) which we call the Cumulative Job Shop Scheduling Problem (CJSSP). Informally, this problem can be stated as follows. Given are a set of jobs and a set of resources. Each job consists of a set of activities that must be processed in a given order. Furthermore, for each activity is given an integer processing time, a resource by which it has to be processed, and an integer demand that represents the amount of resource required by the activity. Once an activity is started, it is processed without interruption. Resources may process several activities simultaneously. To this end, for each resource is given an integer capacity. A resource can simultaneously process only those sets of activities whose total demand do not exceed the capacity of the resource. A schedule assigns a start time to each activity. A feasible schedule is a schedule that meets the order in which the activities must be processed and in which the capacity of none of the resources is exceeded at any point in time. One is asked to find an optimal schedule, i.e., a feasible schedule that minimizes the makespan of the schedule, the makespan being defined as the maximum completion time of any of the activities. We remark that the CJSSP is the optimization variant of the Multiple Capacitated Job Shop Scheduling Problem of (Nuijten 1994; Nuijten & Aarts 1996).

The organization of the remainder of the paper is as follows. First we define the CJSSP after which we present the LNS approach we propose. We then present the computational results to finally discuss the conclusions of the paper and potential directions of future research.

The Cumulative Job Shop Scheduling Problem

The Cumulative Job Shop Scheduling Problem (CJSSP) can be formalized as follows. We are given a set $A$ of activities, a set $R$ of resources, and a set $J$ of jobs, each job $j$ consisting of a sequence of activities $(a_1, ..., a_n)$. Each activity $a \in A$ has a processing time $pt(a)$ and a demand $d(a)$ for resource $r(a)$ to be executed. Each resource $r \in R$ has a capacity $C(r)$ and a binary relation $\prec$ is given that decomposes $A$ into chains, such that every chain corresponds to a job. A schedule is an assignment $s: A \rightarrow \mathbb{Z}^+_0$ assigning a positive start time $s(a)$ to each activity $a \in A$. A schedule $s$...
is feasible if it satisfies the precedence constraints between each pair of consecutive activities in a job:
\[
\forall a, a' \in \mathcal{A} | a < a' \quad s(a) + pt(a) \leq s(a')
\]
(1)
and the resource capacity constraints for each resource:
\[
\forall r \in \mathcal{R} \quad \forall t \in \mathbb{Z}_+^+ \quad \sum_{a \in \mathcal{A}_r | s(a) \leq t < s(a) + pt(a)} d(a) \leq C(r)
\]
(2)
where \(\mathcal{A}_r = \{ a \in \mathcal{A} | r(a) = r \}\). One is asked to find an optimal schedule which is a feasible schedule that minimizes the makespan defined as \(\max_{a \in \mathcal{A}} s(a) + pt(a)\).

### Randomized Large Neighborhood Search

Our approach is an implementation of the Large Neighborhood Search (LNS) framework (Shaw 1998) which is based upon a process of continual relaxation and re-optimization. It is a generalization of the randomized LNS approach for the Job Shop Scheduling Problem of (Nuijten & Le Pape 1998). The LNS framework is illustrated in Figure 1. A first solution is computed and iteratively improved. Each iteration consists of a relaxation step followed by a re-optimization of the relaxed solution. This process continues until some condition is satisfied (for instance a time limit or a number of non-improving iterations).

![Figure 1: LNS Framework](image)

One of the first approaches that used LNS was on the Job Shop Scheduling Problem (Applegate & Cook 1991). One of the challenges for applying LNS to cumulative scheduling (and to even more complex scheduling problems in general) is that most of the available algorithms to solve cumulative scheduling problems produce solutions with fixed start times. In this context, relaxing a solution by only unfreezing the start time of some activities in the schedule provides limited flexibility to re-optimize the relaxed solution. Previous work on cumulative job shop problems (Cesta, Oddi, & Smith 2000; Michel & Van Hentenryck 2003) avoids this issue by generating precedence constraints as decisions and producing a temporal constraint network which is more flexible thus more adequate for LNS than a completely instantiated schedule. (Palpant, Artigues, & Michelon 2004) also applied an LNS framework on resource-constrained project scheduling problems. They avoid the issue of lack of flexibility by composing a solution method to solve the sub-problem which does not necessarily decrease the makespan with a forward/backward heuristic (Li & Willis 1992) to left-shift frozen activities so as to take advantage of the more compact solution with relaxed activities.

In this paper, we investigate a slightly different approach where any algorithm can be used to produce a first solution and to iteratively re-optimize the current solution. The main advantage of this approach is its genericity, i.e., the type of search algorithm used to compute first solutions or for re-optimization is completely orthogonal to the type of LNS relaxation that is used. Consequently one can for instance apply this approach to problems involving minimization of earliness/tardiness costs or resource allocation costs.

To select the combination of search algorithms and LNS relaxations used in our approach, we first focused on a representative subset of the open instances of the CJSSP and made various experiments:

- For building a first solution, we compared various algorithms: some build solutions in chronological order, others are based on minimal critical sets, some are deterministic, others are based on random restart; for each algorithm, we tried different heuristics for selecting an activity or for selecting a pair of activities to order.
- We experimented the same algorithms for the search procedure used after each relaxation to re-optimize the current solution.
- We compared various neighborhoods: some based on selection of activities on the critical path, others based on selection of activities of a job or based on gliding time windows, we tried both deterministic and non-deterministic heuristics for selecting activities; we also compared various combinations of these neighborhoods.

Notice that the approach of (Cesta, Oddi, & Smith 2000) and (Michel & Van Hentenryck 2004) is included among all the combinations we experimented.

In this paper, we focus on the combination that offers the best trade-off between simplicity and quality. We used this combination to conduct an empirical performance analysis on the complete set of benchmarks.

The remaining part of this paper describes this combination and report the computational results. The algorithm we used to generate first solutions for the CJSSP is described in Section Finding a Solution. The completely instantiated solution generated by this algorithm is firstly relaxed to obtain a POS (see Section POS Relaxation), after which an LNS relaxation is applied on this schedule (see Section LNS Relaxation). The overall algorithm is described in Section Iterative Improvement and the used constraint propagation in Section Constraint Propagation.

### Finding a Solution

To find an initial solution to the CJSSP and to re-optimize relaxed solutions, we use the algorithm described in (Le Pape et al. 1994). This algorithm is available in ILOG SCHEDULER (ILOG 2005a) and is called SetTimes. It fixes the start
times of the activities in chronological order and can be summarized as follows.

1. Let $S$ be the set of selectable activities. Initialize $S$ to the complete set of activities of the schedule.

2. If all the activities have a fixed start time then exit: a solution has been found. Otherwise remove from $S$ the activities with fixed start time.

3. If the set $S$ is not empty:
   (a) Select an activity from $S$ which has the minimal earliest start time. Use the minimal latest end time to break ties.
   (b) Create a choice point (to allow backtracking) and fix the start time of the selected activity to its earliest start time. Goto step 2.

4. If the set $S$ is empty:
   (a) Backtrack to the most recent choice point.
   (b) Upon backtracking, mark the activity that was scheduled at the considered choice point as "not selectable" as long as its earliest start time has not changed. Goto step 2.

After each decision in step 3b, the earliest start times and latest end times of activities are maintained by constraint propagation. The status "not selectable" in step 4b is also maintained by constraint propagation.

This algorithm is clearly sound. It is complete in the case of job shop scheduling (simple temporal constraints with positive delays). On problems with more complex constraints and resource types, this algorithm can still be applied and usually leads to fairly good solutions for minimizing the makespan although it is in general incomplete.

**POS Relaxation**

As stated above, the solutions produced by the SetTimes algorithm have fixed start times. The problem with such a schedule in the context of LNS is its lack of flexibility. If part of the solution is relaxed whereas the rest of the solution remains frozen (fixed start times), there is limited room for re-optimization as there are limited possibilities to reschedule relaxed activities in between frozen activities.

In our approach, the fully instantiated solution is first relaxed into a partial-order schedule (POS). A POS is a graph $G(V, E)$ where the nodes $V$ are the activities of the scheduling problem and the edges are the temporal constraints between pairs of activities, such that any temporal solution to this graph is also a resource-feasible solution. More generally, a POS is a resource temporal network that satisfies the necessary truth criterion as defined in (Laborie 2003). A POS is by nature more flexible and thus more adequate for LNS.

Our POS-generation algorithm was developed independently from the one recently described in (Policella et al. 2004) in the context of dynamic and uncertain execution environments.

The main idea behind building a POS for a given resource is to split the resource into two smaller resources of half capacity, to split the activities between these two sub-resources, and to recursively call the POS-generation on each of the two sub-resources with their allocated activities. The leaves of the recursion are sub-resources for which all the allocated activities are in disjunction so that the POS consists of a chain of these activities. More precisely, the recursion to generate a POS $POS(r)$ for a resource $r$ works as follows.

1. Sort all activities $a \in A$ that are allocated to resource $r$ in chronological order of their start times $s(a)$. Let $l(r)$ be the list of chronological ordered activities. $C(r)$ denotes the capacity of (sub-)resource $r$. Let $d(a, r)$ denote the demand of activity $a$ for (sub-)resource $r$ and initialize it to $d(a, r) = d(a)$.

2. If less than two activities are allocated to $r$, or if the sum of the two smallest demands is greater than $C(r)$, then add the chain $l(r)$ to the POS: $POS(r) = POS(r) \cup l(r)$

3. Otherwise:
   (a) Create two sub-resources $r_{lower}$ with associated capacity $C(r_{lower}) = [C(r)/2]$ and $r_{upper}$ with associated capacity $C(r_{upper}) = C(r) - C(r_{lower})$.
   (b) For each activity $a \in l(r)$ traversed in chronological order of their start times $s(a)$ in the schedule, allocate $a$ to the sub-resource $r_{lower}$ or $r_{upper}$ with the largest capacity slack at time $s(a)$ given the activities already allocated to $r_{lower}$ and $r_{upper}$ and add the activity to $l(r_{lower})$ or $l(r_{upper})$ accordingly. Let $slack_x(a)$ (with $x \in \{lower, upper\}$) denote these slacks. If the largest slack $slack_x(a)$ is smaller than $d(a, r)$, then activity $a$ is allocated to both sub-resources: to sub-resource $r_x$ with demand $d(a, r_x) = slack_x(a)$ and to the other sub-resource with demand $d(a, r) - slack_x(a)$.
   (c) If the list $l(r_{lower})$ is not empty, then recursively goto step 2 with $r_{lower}$. Similarly, if the list $l(r_{upper})$ is not empty, then recursively goto to 2 with $r_{upper}$.

In step 2, the recursion reaches a leaf when the sub-resource $r$ is disjunctive, that is, no pair of activities can overlap in time without over-consuming the sub-resource.

In step 3b, activities are considered in chronological order of the start time and, for two activities starting at the same date, the one with largest demand $d(a, r)$ is selected. In case both sub-resources have the same slack for an activity $a$ ($slack_{lower}(a) = slack_{upper}(a)$), ties are broken randomly.

The global POS is then made of the union of temporal constraints of the problem itself and the POS $POS(r)$ of each resource $r$ as generated by the above given algorithm.

Figure 2 illustrates this algorithm on a resource with capacity 5. The resource is first split in two parts, a lower half with capacity 3 and an upper half with capacity 2. Activities are then assigned to each parts as shown in the upper-right part of the figure. As both sub-resources are still disjunctive, they are split again as shown in the lower-left part of the figure. The POS generated for this resource is depicted in the lower-right part.

Let $n$ denote the number of activities on the resource and $C$ the capacity of the resource. The initial sort of activities by increasing start times can be performed in $O(n \log n)$. 
LNS Relaxation

The basic idea of the LNS relaxation is to select \( m \leq |A| \) activities and to relax those activities in the POS constructed in the previous section so as to leave room for improvement in the next iteration. Let \( S = \{a_1, \ldots, a_m\} \) be the set of selected activities and \( P \) be the set of temporal constraints of the problem itself (the ones defined by \( \prec \)). The relaxed POS is obtained by applying the following algorithm:

1. For each selected activity \( a \in S \), remove from the POS all temporal constraints regarding \( a \) except the temporal constraints belonging to \( P \).
2. For all removed temporal constraints \((a', a)\), if \( a' \notin S \) and \( a \in S \) then add new temporal constraints between \( a' \) and the first not selected successor of \( a \) in the original temporal graph.
3. Optionally remove redundant temporal constraints that may have been introduced in the previous step by applying a topological sort of the temporal graph.

This algorithm is illustrated in Figure 3. The upper-left drawing displays the original POS. We assume this temporal graph contains no temporal constraints belonging to \( P \). Two activities are selected (upper-right drawing): activity D and activity F. At steps 1 and 2, temporal constraints (A,D), (D,H), (D,F), and (F,J) are removed and two temporal constraints are added (A, H) and (A,J). The redundant temporal constraint (A,J) is removed in step 3.

Iterative Improvement

The iterative improvement procedure we use is described in Algorithm 1. The name of the algorithm - STRAND - stands for SetTimes + Random relaxation. We used the SetTimes procedure with three parameters:

- parameter \( \gamma \) that specifies the maximum allowed number of backtracks (expressed in percentage of the number of activities) for one execution of the procedure,
- a flag \( fc \in \{first, cont\} \) that tells whether the search must stop at the first solution found or continue search trying to minimize the makespan until the maximum number of backtracks \( \gamma \) is reached (or an optimal solution found), and
- an upper bound \( ub \) on the makespan.

The parameter \( \gamma \) allows to control the SetTimes procedure from a simple descent (\( \gamma = 0 \) no backtracks allowed) to a complete search tree (\( \gamma = \infty \)).
The input parameters of STRand are: the problem $P$ to solve, the probability $\alpha$ to relax a given activity in the RandomRelax procedure ($0 < \alpha \leq 1$), the improvement step $\beta$, the maximum number of backtracks $\gamma$, the flag $fc$, and a global time limit $t$.

Algorithm 1 Iterative Improvement Algorithm

1: procedure STRAND($P, \alpha, \beta, \gamma, fc, t$)
2:     $R := P$
3:     $m^* := \infty$ \text{▷} $m^*$: Best makespan so far
4:     $ub := \infty$ \text{▷} $ub$: Upper bound for SetTimes
5:     while $time < t$ do
6:         $s := \text{SetTimes}(R, \gamma, fc, ub)$
7:         if $(\text{makespan}(s) < \text{best})$ then
8:             $s^* := s$ \text{▷} $s^*$: Best schedule so far
9:             $m^* := \text{makespan}(s)$
10:            $ub := (1 + \beta)m^*$
11:        end if
12:     $R := \text{POSRelax}(P, s)$
13:     $R := \text{RandomRelax}(R, P, \alpha)$
14: end while
15: return $s^*$
16: end procedure

Constraint Propagation

When re-optimizing a relaxed solution $R$ at line 6, the number of precedence relations may be large as $R$ contains a subset of the temporal constraints generated when converting the fixed-time schedule to a POS. To efficiently propagate these temporal constraints, we have implemented two algorithms:

- A topological sort on the direct acyclic graph of temporal constraint is used to compute the initial time bounds of the activities. The complexity of this algorithms is $O(n + p)$ where $n$ is the number of activities and $p$ the number of temporal constraints. This algorithm is run only once at each iteration.

- The algorithm described in (Michel & Van Hentenryck 2003) to incrementally maintain the longest paths in direct acyclic graphs is used to incrementally compute the time bounds of activities. The complexity of this algorithm is in $O(||\delta|| \log ||\delta||)$ where $||\delta||$ is a measure of the change in the graph since the previous propagation. This algorithm is activated after each decision taken during the search procedure.

This temporal propagation was implemented as a global constraint in the CP framework of ILOG Solver (ILOG 2005b) and ILOG Scheduler (ILOG 2005a). The capacity of resources is propagated using the timetable constraint of ILOG Scheduler (Le Pape 1994).

Benchmarks

The benchmarks we used are the standard benchmarks from (Nuijten 1994). These benchmarks are derived from job shop scheduling problems by introducing a certain number of duplicates for each job and increasing the capacity of the resources accordingly. The benchmarks are classified in 5 groups:

- Set A : Lawrence LA01-LA10 duplicated and triplicated.
- Set B : Lawrence LA11-LA20 duplicated and triplicated.
- Set C : Lawrence LA21-LA30 duplicated and triplicated.
- Set D : Lawrence LA31-LA40 duplicated and triplicated.
- Set MT : MT06, MT10, and MT20 duplicated and triplicated.

Because of the way these instances are constructed, upper bounds on the optimal makespan can be derived from the corresponding job shop scheduling instances. Incidentally, the results of (Michel & Van Hentenryck 2004; Cesta, Oddi, & Smith 2000; Nuijten & Aarts 1996) have shown that these upper bounds are not so easy to find as the algorithms used are not aware of the underlying structure of the instances. Notice also that the instances by construction contain equivalent solutions. In our study, as in previous ones, no constraint has been used to break those symmetries.

In terms of size, with these different sets we have quite a large spectrum ranging from 50 to 900 activities and from 5 to 15 resources. Sets A, B, and MT consist of small to medium size instances whereas sets C and D consist of medium to large size instances.

Preparing Time Equivalent Tests

To conduct fair comparisons both between our approach and previously reported approaches as well as between different parameter settings for our approach we aim to do time equivalent tests. To do so we determined maximum running times for each instance in the following way. First, we applied a search procedure that is comparable in complexity to the ones used by (Cesta, Oddi, & Smith 2000; Michel & Van Hentenryck 2004) by allowing no backtracks during search ($\gamma = 0$), i.e., the search procedure $\text{SetTimes}$ either returns a solution obtained without any backtracking or stops as soon as a backtrack occurs. To allow a better comparison to (Michel & Van Hentenryck 2004) we use the same stop criterion they used, i.e., the algorithm is stopped if for a certain number of iterations the solution was not improved. We set this maximum number of so called $stable$ iterations to 1000. The other parameters are as follows: $\alpha = 0.2, and \beta = 0$. For each instance we then do 10 runs and take the average CPU time as maximum running time that we will use throughout our experiments for that instance.

The machine used for the computational study is a Pentium 4 at 2.8 Ghz. Average results reported are always computed over 10 runs unless specified otherwise.

Results Summary

We found the best performance for our approach when choosing the following values for the different parameters:
\(\alpha = 0.2, \beta = 0, \gamma = 0.15,\) and \(fc = cont.\) Running time for each instance was limited to the maximum running time computed in the previous section.

Table 1 reports average deviation (in percentage) from the upper bounds (UB) reported in (Nuijten & Aarts 1996) and used in the evaluation in (Cesta, Oddi, & Smith 2000) and in (Michel & Van Hentenryck 2004).

As can be observed, our results are on average significantly better than in (Cesta, Oddi, & Smith 2000)) and in (Michel & Van Hentenryck 2004). Table 1 summarizes prior results used in the evaluation in (Cesta, Oddi, & Smith 2000) and algorithms \(IFlatIRelax 4(20)\) (relax probability of 20% and a number of relaxations of 4 as in (Michel & Van Hentenryck 2004)) with respectively 1000, 5000 and 10000 iterations. The column \(STRand 1000\) reports the results obtained when computing average running times.

<table>
<thead>
<tr>
<th>Set</th>
<th>(IFlat5)</th>
<th>(IFlatRelax 4(20))</th>
<th>(ST Rand)</th>
<th>(ST Rand)</th>
<th>(ST Rand)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7.76</td>
<td>1.63</td>
<td>0.90</td>
<td>0.14</td>
<td>-0.13</td>
</tr>
<tr>
<td>B</td>
<td>7.10</td>
<td>1.04</td>
<td>0.47</td>
<td>0.24</td>
<td>-0.59</td>
</tr>
<tr>
<td>C</td>
<td>13.03</td>
<td>-</td>
<td>4.2</td>
<td>1.18</td>
<td>-0.16</td>
</tr>
<tr>
<td>D</td>
<td>11.92</td>
<td>2.4</td>
<td>1.11</td>
<td>0.22</td>
<td></td>
</tr>
<tr>
<td>MT</td>
<td>4.76</td>
<td>3.41</td>
<td>3.12</td>
<td>1.03</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>2.13</td>
<td>0.59</td>
<td>-0.23</td>
</tr>
</tbody>
</table>

Table 1: Results summary of \(IFlat5, IFlatRelax\) and \(ST Rand\)

On average, \(ST Rand(\alpha = 0.2, \beta = 0, \gamma = 0.15, fc = cont)\) is within -1.10% and 0.55% of UB. Furthermore, we have obtained new upper bounds on 33 instances out of 86.

For detailed results, see Section Detailed Results below.

**Impact of the Relaxation Probability**

To study the influence of the different parameters we did a series of tests. We firstly studied the influence of varying the relaxation probability \(\alpha\). Table 2 reports average deviation in percentage from UB when \(\alpha\) is varying between 0.1 and 0.3. Running time for each instance is limited to the maximum running time computed previously. The other parameters are as follows: \(\beta = 0, \gamma = 0.15,\) and \(fc = cont.\)

<table>
<thead>
<tr>
<th>Set</th>
<th>0.1</th>
<th>0.15</th>
<th>0.2</th>
<th>0.25</th>
<th>0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.09</td>
<td>-0.09</td>
<td>-0.13</td>
<td>-0.12</td>
<td>-0.13</td>
</tr>
<tr>
<td>B</td>
<td>-0.57</td>
<td>-0.87</td>
<td>-1.10</td>
<td>-1.26</td>
<td>-1.25</td>
</tr>
<tr>
<td>C</td>
<td>-0.27</td>
<td>-0.41</td>
<td>-0.16</td>
<td>0.32</td>
<td>0.58</td>
</tr>
<tr>
<td>D</td>
<td>0.52</td>
<td>0.25</td>
<td>0.22</td>
<td>0.33</td>
<td>0.58</td>
</tr>
<tr>
<td>MT</td>
<td>1.15</td>
<td>0.76</td>
<td>0.55</td>
<td>0.39</td>
<td>0.43</td>
</tr>
<tr>
<td>Total</td>
<td>0.03</td>
<td>-0.21</td>
<td>-0.23</td>
<td>-0.14</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 2: Impact of the Relaxation Probability \(\alpha\) on \(ST Rand(\alpha, \beta = 0, \gamma = 0.15, fc = cont)\)

Overall, the best results are obtained with \(\alpha = 0.2\) although the quality obtained with \(\alpha = 0.15\) is very closed. It gives the best results on 2 sets while \(\alpha = 0.15\) works better for set C and \(\alpha = 0.25\) works better for set B and set MT.

**Impact of the Improvement Step**

Table 3 reports the average deviation in percentage from UB when the improvement step \(\beta\) is varying from \(-\epsilon\) (enforce to find a solution strictly better than the best solution so far) to \(\infty\) (no enforcement at all). Running time for each instance is limited to the maximum running time computed previously.

<table>
<thead>
<tr>
<th>Set</th>
<th>(-\epsilon)</th>
<th>0</th>
<th>0.01</th>
<th>(\infty)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.23</td>
<td>-0.13</td>
<td>0.16</td>
<td>1.15</td>
</tr>
<tr>
<td>B</td>
<td>-0.37</td>
<td>-1.10</td>
<td>-0.53</td>
<td>0.77</td>
</tr>
<tr>
<td>C</td>
<td>0.76</td>
<td>-0.16</td>
<td>3.54</td>
<td>5.50</td>
</tr>
<tr>
<td>D</td>
<td>0.92</td>
<td>0.22</td>
<td>2.31</td>
<td>3.21</td>
</tr>
<tr>
<td>MT</td>
<td>1.58</td>
<td>0.55</td>
<td>1.03</td>
<td>3.90</td>
</tr>
<tr>
<td>Total</td>
<td>0.47</td>
<td>-0.23</td>
<td>1.35</td>
<td>2.74</td>
</tr>
</tbody>
</table>

Table 3: Impact of the Improvement Step \(\beta\) on \(ST Rand(\alpha = 0.2, \beta, \gamma = 0.15, fc = cont)\)

For positive improvement steps, increasing the value leads to increasing the average deviation and thus decreasing the quality. The best quality is obtained with a null improvement step, that is when the search procedure is enforced to find a solution better or equal to the previous one. Results obtained when applying \(-\epsilon\) as improvement step are not as good. With \(-\epsilon\) we observe that the first iterations decrease the cost effectively, but then the search is often trapped in a local optimum.

**Impact of the Maximum Number of Backtracks**

Table 4 reports the average deviation in percentage from UB when the maximum number of allowed backtracks \(\gamma\) is varying from 0% to 20% with the number of activities. Running time for each instance is limited to the maximum running time computed previously. The other parameters are as follows: \(\alpha = 0.2, \beta = 0,\) and \(fc = cont.\)

<table>
<thead>
<tr>
<th>Set</th>
<th>0</th>
<th>0.05</th>
<th>0.10</th>
<th>0.15</th>
<th>0.20</th>
<th>0.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.42</td>
<td>-0.16</td>
<td>-0.15</td>
<td>-0.13</td>
<td>-0.16</td>
<td>-0.13</td>
</tr>
<tr>
<td>B</td>
<td>-0.67</td>
<td>-1.06</td>
<td>-1.07</td>
<td>-1.10</td>
<td>-1.09</td>
<td>-1.06</td>
</tr>
<tr>
<td>C</td>
<td>0.87</td>
<td>-0.14</td>
<td>-0.17</td>
<td>-0.16</td>
<td>-0.06</td>
<td>-0.01</td>
</tr>
<tr>
<td>D</td>
<td>0.91</td>
<td>0.24</td>
<td>0.25</td>
<td>0.22</td>
<td>0.18</td>
<td>0.28</td>
</tr>
<tr>
<td>MT</td>
<td>1.21</td>
<td>0.38</td>
<td>0.64</td>
<td>0.55</td>
<td>0.52</td>
<td>0.66</td>
</tr>
<tr>
<td>Total</td>
<td>0.44</td>
<td>-0.23</td>
<td>-0.22</td>
<td>-0.23</td>
<td>-0.22</td>
<td>-0.17</td>
</tr>
</tbody>
</table>

Table 4: Impact of the Number of Backtracks \(\gamma\) on \(ST Rand(\alpha = 0.2, \beta = 0, \gamma, fc = cont)\)

Overall, for \(\gamma\) varying between 5% and 20%, there is no much difference in quality. Increasing \(\gamma\) further than 20% leads to increasing the average deviation and thus decreasing the quality.

\(\text{computing average running times.}\)
\(\text{Running time for each instance is limited to the maximum running time computed previously.}\)
\(\text{The column } \text{ST Rand 1000} \text{ reports the results obtained when}\)
\(\text{computing average running times.}\)
\(\text{Table 1 reports average deviation (in percentage) from the upper bounds (UB) reported in (Nuijten & Aarts 1996) and}\)
\(\text{used in the evaluation in (Cesta, Oddi, & Smith 2000) and in (Michel & Van Hentenryck 2004). Table 1 summarizes prior results}\)
\(\text{used in the evaluation in (Cesta, Oddi, & Smith 2000) and algorithms } IFlatIRelax 4(20) \text{ (relax probability of 20% and a number of}\)
\(\text{relaxations of 4 as in (Michel & Van Hentenryck 2004)) with respectively 1000, 5000 and 10000 iterations. The column } \text{ST Rand 1000} \text{ reports the results obtained when}\)
\(\text{computing average running times.}\)
\(\text{Table 1: Results summary of } IFlat5, IFlatRelax \text{ and } ST Rand \)
Impact of the Search Method

Table 5 compares the average deviation in percentage from UB when continuing search at each iteration until all allowed backtracks are exhausted with the one obtained when stopping search at the first solution found. Running time for each instance is limited to the maximum running time computed previously. The other parameters are as follows: \( \alpha = 0.2, \beta = 0, \) and \( \gamma = 0.15. \)

Table 5: Impact of the Search Method on STRand(\( \alpha = 0.2, \beta = 0, \gamma = 0.15, fc = \text{first/cont} \))

When continuing search obviously more time is taken at each iteration: the search returns the best solution found using the limited amount of backtracks instead of returning the first solution found. As the time is limited, this implies less iterations. As can be observed continuing search leads to improved quality on all sets of instances.

Detailed Results

Tables 6, 7, 8, and 10 display in the second column the lower bounds reported in (Nuijten & Aarts 1996) including some recent improvements described in (Laborie 2005) followed by the upper bounds found by (Nuijten & Aarts 1996) or derived from the corresponding job shop scheduling instance. If these two values are equal, only one number is given. The third column reports the upper bounds found by (Michel & Van Hentenryck 2004). The following three columns report the best upper bounds, the average upper bounds and the average running time obtained when performing 1000 stable iterations. The following two columns report the best upper bounds and the average upper bounds obtained with STRand when using the following parameters: \( \alpha = 0.2, \beta = 0, \gamma = 0.15, \) and \( fc = \text{cont}. \) Running time for each instance is limited to the maximum running time computed previously. Best and average upper bounds are computed over 10 runs. The last column reports the best overall upper bound obtained with STRand during this study. Results are in bold when a new upper bound is reported. Note that this last column indeed reports the best upper bounds found during all the tests we did during this computational study, not just the best of the two parameter settings reported on in this section.

On set A, new upper bounds for all 4 open instances have been found. All results reported in (Michel & Van Hentenryck 2004) are improved (la04d and la04t). Instance la03d is now closed.

On set B, new upper bounds have been found for all 10 open instances. All improve results reported in (Michel & Van Hentenryck 2004).

Table 6: Detailed Results on Set A

Table 7: Detailed Results on Set B

Table 8: Detailed Results on Set C
In this paper we described a generic approach to solve cumulative scheduling problems based on randomized large neighborhood search. We presented a way to calculate a partial-order schedule from a fixed start time schedule, which is a crucial step in the generalization to cumulative scheduling from earlier randomized LNS work for disjunctive scheduling. We applied and tested the approach on the Cumulative Job Shop Scheduling Problem. The empirical performance analysis we performed using a well-known set of benchmark instances showed that our approach obtains the best known performance reported to date on the CJSPP. New upper bounds have been found on 33 out of 36 open instances and among them one instance is now closed. Furthermore, our approach proves to be very robust on the complete set of test instances. As for future work, we plan to study how our approach competes on other types of problems such as the Resource Constrained Project Scheduling Problem. Another direction we plan to study is how this approach can be extended, on one hand to other types of resources such as state resources and discrete reservoirs, and on the other hand to other types of objectives such as earliness/tardiness costs and resource allocation costs. Flexibility of the generated POS is clearly a key factor for the approach. We plan to compare the flexibility of our POS generation procedure with the one of (Policella et al. 2004) and work on improving this flexibility by exploiting interactions between resources.

## References


