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 towards 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 24 out of 36 open instances, it furthermore proves to be very robust on the complete set of test instances. 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 framework (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 *capacity* that the activity requires from this resource. 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 required capacities 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 $\mathcal{A}$ of activities, a set $\mathcal{R}$ of resources, and a set $\mathcal{J}$ of jobs, each job j consisting of a sequence of activities $(a_1, ..., a_n)$. Each activity $a \in \mathcal{A}$ has a processing time $pt(a)$ and requires a capacity $c(a)$ from resource $r(a)$ to be executed. Each resource $r \in \mathcal{R}$ has a maximum available capacity $C(r)$ and a binary relation $\prec$ is given that decomposes $\mathcal{A}$ into chains, such that every chain corresponds to a job. A *schedule* is an assignment $s : \mathcal{A} \rightarrow \mathbb{Z}_0^+$ assigning a positive start time $s(a)$ to each activity $a \in \mathcal{A}$. A schedule s is *feasible* if it satisfies the *precedence constraints* between each pair of consecutive activities in a job:

$$\forall_{a,a' \in \mathcal{A} | a \prec a'} \; s(a) + pt(a) \leq s(a') \tag{1}$$

and the *resource capacity constraints* for each resource:

$$\forall_{r \in \mathcal{R}} \forall_{t \in \mathsf{Z}_0^+} \sum_{a \in \mathcal{A}_r | s(a) \leq t < s(a)+pt(a)} c(a) \leq C(r) \qquad (2)$$

where $\mathcal{A}_r = \{a \in \mathcal{A} \mid r(a) = r\}$. 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).

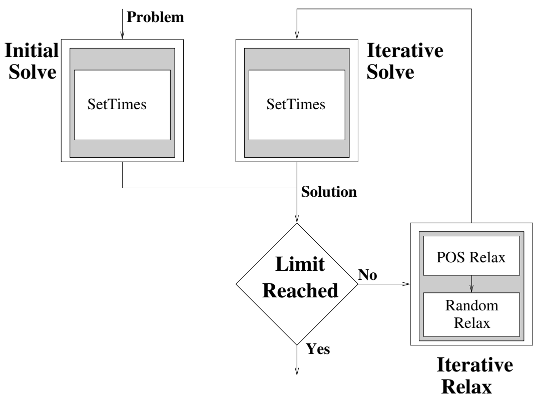

Figure 1: LNS Framework

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 RCPSP. They avoided 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.

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 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 to 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 *notselectable* 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 $a \in \mathcal{A}$ 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 $c(a, r)$ denote the capacity required by activity a on (sub-)resource r and initialize it to $c(a, r) = c(a)$.
2. If less than two activities are allocated to r, or if the sum of the two smallest required capacities 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 $C(r_{lower}) = \lceil C(r)/2 \rceil$ 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 $c(a, r)$, then activity a is allocated to both sub-resources: to sub-resource r_x with capacity $c(a, r_x) = slack_x(a)$ and to the other sub-resource with capacity $c(a, r) - slack_x(a)$.
   (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 capacity $c(a, r)$ 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 not 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.

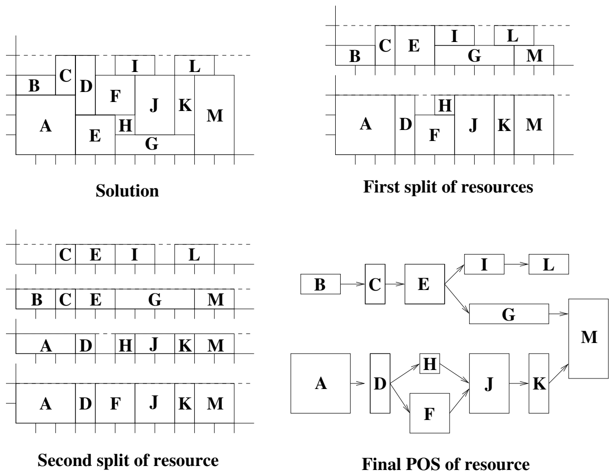

Figure 2: Transforming a fixed-time schedule into a *POS*

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 $O(n \log n)$. There are at most $\log C$ layers of resource splits. At layer i, there are at most 2^i sub-resources, each of which has approximatively $n/2^i$ activities. Allocating an activity a to one of the two sub-resources r_{lower} or r_{upper} can be done in a time proportional to the number of activities allocated to the sub-resource that start or end in the time interval $[s(a), s(a) + pt(a))$. If we assume this quantity to be a constant K, the average complexity for constructing layer i is thus equal to $2^i(Kn/2^i) = Kn$ and the cost for computing all layers is thus in $O(Kn \log C)$. This gives a rough estimate of the average complexity in $O(n \log n + Kn \log C))$. The algorithm described in (Policella *et al.* 2004) is very similar to ours but the authors did not give its complexity : a naive implementation of their algorithm leads to a complexity in $O(n \log n + nC))$ (the discrete resource of capacity C is virtually split into C unary resources), so an $O(C)$ factor compared to the $O(\log C)$ factor in our algorithm. A deeper comparison of the two approaches in terms of efficiency and flexibility is part of our future work.

LNS Relaxation

The basic idea of the LNS relaxation is to select $m \leq |\mathcal{A}|$ 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 $S = \{a_1, ..., a_m\}$ be the set of selected activities. 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 of the problem itself (the ones defined by $\prec$).
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 successors 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), (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.

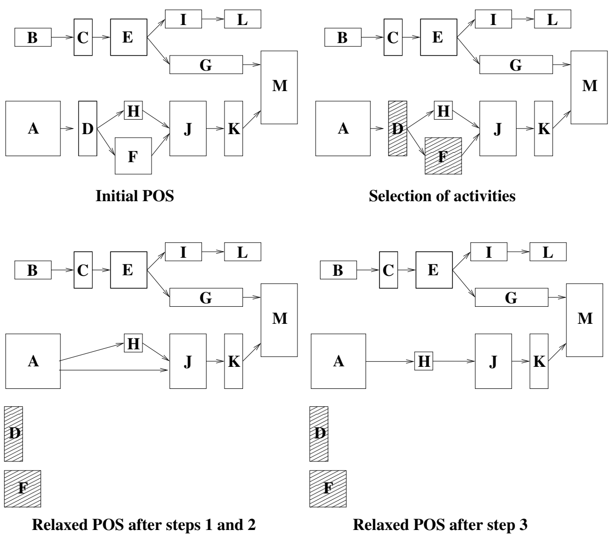

Figure 3: Computing a *Relaxed POS*

The temporal constraints of the obtained relaxed POS are then taken as new start point for the next iteration of the LNS. In our study, to build the set of selected activities S we choose them randomly with a probability α which is a parameter of the algorithm. It means that in average, $m = \alpha\, |\mathcal{A}|$.

This large neighborhood search has been implemented by using the LNS framework available in ILOG SCHEDULER 6.1.

Iterative Improvement

The iterative improvement procedure we use is described in Algorithm 1. The name of the algorithm - $STRand$ - stands for **S**et**T**imes + **Rand**om relaxation. We used the *SetTimes* procedure with three parameters:

- an integer parameter nb that gives a maximum allowed number of backtracks 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 nb is reached (or an optimal solution found), and
- an upper bound ub on the makespan.

The input parameters of $STRand$ are: the problem P to solve, the probability α to relax a given activity in the $RandomRelax$ procedure ($0 < \alpha \leq 1$), the improvement step β, the maximum number of backtracks nb, the flag fc, and a global time limit t.

Algorithm 1 Iterative Improvement Algorithm

1: **procedure** STRAND(P,α,β,nb,fc,t)
2: $\quad R := P$
3: $\quad m^* := \infty$ ▷ m^*: Best makespan so far
4: $\quad ub := \infty$ ▷ ub: Upper bound for SetTimes
5: $\quad$ **while** $time < t$ **do**
6: $\quad\quad s := \mathrm{SetTimes}(R, nb, fc, ub)$
7: $\quad\quad$ **if** $(makespan(s) < best)$ **then**
8: $\quad\quad\quad s^* := s$ ▷ s^*: Best schedule so far
9: $\quad\quad\quad m^* := makespan(s)$
10: $\quad\quad\quad ub := (1 + \beta)m^*$
11: $\quad\quad$ **end if**
12: $\quad\quad R := \mathrm{POSRelax}(P, s)$
13: $\quad\quad R := \mathrm{RandomRelax}(R, P, \alpha)$
14: $\quad$ **end while**
15: $\quad$ **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/SCHEDULER. The capacity of resources is propagated using the *timetable* constraint of ILOG SCHEDULER (Le Pape 1994).

Computational Results

In this section, we report the computational results of the algorithm described in the previous section.

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. Incidently, 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 aproaches 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, i.e., the search procedure *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. For each instance we then do 5 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 2.0 Ghz. Average results reported are always computed over 5 runs unless specified otherwise[1].

Results Summary

We found the best performance for our approach when choosing the following values for the different parameters: $\alpha = 0.2$, $\beta = 0$, $nb = 100$, 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 both in terms of quality as robustness than the ones published in (Cesta, Oddi, & Smith 2000) and (Michel & Van Hentenryck 2004). Table 1 summarizes prior results of algorithm $IFlat_5$ (iterative flattening procedure with 5 random restarts as in (Cesta, Oddi, & Smith 2000)) and algorithm *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.

Set	$IFlat_5$	IFlatIRelax 4(20)			STRand	STRand
		1000	5000	10000	1000	0.2/0/100/cont
A	7.76	1.63	1.07	0.90	0.52	**-0.12**
B	7.10	1.04	0.47	0.24	-0.63	**-0.91**
C	13.03	-	4.2	-	1.58	**0.33**
D	11.92	-	2.4	-	1.05	**0.42**
MT	-	4.76	3.41	3.12	1.50	**0.47**
Total			2.13		0.69	**-0.03**

Table 1: Results summary of $IFlat_5$, $IFlatIRelax$ and $STRand$

On average, $STRand$ is within -0.91% and 0.47% of *UB*. Furthermore, we have obtained new upper bounds on 24 intances out of 86. For detailed results, see Section *Detailed Results* below.

[1]*Note that average results in this paper are computed over 5 runs using different seeds whereas result reported in (Michel & Van Hentenryck 2004) were obtained using 100 runs or more. In the final version of the paper, we will provide results using more runs. This could in particular provide better upper bounds.*

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 α. Table 2 reports average deviation in percentage from *UB* when α 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: $\beta = 0$, $nb = 100$, and $fc = first$. [2]

Set	0.1	0.2	0.3
A	0.32	**-0.01**	0.27
B	0.16	-0.86	**-1.09**
C	**0.30**	0.59	1.52
D	0.89	**0.58**	0.98
MT	1.59	**0.97**	1.02
Total	0.50	**0.14**	0.46

Table 2: Impact of the Relaxation Probability α on $STRand(\alpha, \beta = 0, nb = 100, fc = first)$

Overall the best results are obtained with $\alpha = 0.2$. It gives the best results on 3 sets although $\alpha = 0.3$ works better for set B and $\alpha = 0.1$ for set C.

Impact of the Improvement Step

Table 3 reports the average deviation in percentage from *UB* when the improvement step β is varying from $-\epsilon$ (enforce to find a solution strictly better than the best solution so far) to ∞ (no enforcement at all). Running time for each instance is limited to the maximum running time computed previously. The other parameters are as follows: ($\alpha = 0.2$, $nb = 100$, and $fc = first$.

Set	$-\epsilon$	0	0.01	∞
A	0.25	**-0.01**	1.29	1.85
B	-0.41	**-0.86**	1.90	2.62
C	0.87	**0.59**	7.28	8.08
D	0.95	**0.58**	4.47	4.81
MT	1.62	**0.97**	4.51	6.48
Total	0.50	**0.14**	3.79	4.49

Table 3: Impact of the Improvement Step β on $STRand(\alpha = 0.2, \beta, nb = 100, fc = first)$

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. We plan to study dynamic adaption schemes of α as used in (Nuijten & Le Pape 1998) to escape from these local minima, which in turn may change the influence of β.

Impact of the Maximum Number of Backtracks

Table 4 reports the average deviation in percentage from *UB* when the maximum number of allowed backtracks nb is varying from 0 to 500. 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 $fc = first$.

Set	0	100	500
A	0.52	**-0.01**	0.21
B	-0.66	**-0.86**	-0.75
C	1.45	**0.59**	0.89
D	1.03	**0.58**	0.66
MT	1.47	**0.97**	1.12
Total	0.65	**0.14**	0.32

Table 4: Impact of the Number of Backtracks nb on $STRand(\alpha = 0.2, \beta = 0, nb, fc = first)$

The best quality is obtained with a maximum number of backtracks of 100.

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 $nb = 100$.

Set	$first$	$cont$
A	-0.01	**-0.12**
B	-0.86	**-0.91**
C	0.59	**0.33**
D	0.58	**0.42**
MT	0.97	**0.47**
Total	0.14	**-0.03**

Table 5: Impact of the Search Method on $STRand(\alpha = 0.2, \beta = 0, nb = 100, fc = first/cont)$

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 again on all sets of instances.

[2] *Note that the comparisons that follow were done using $fc = first$, this while $fc = cont$ gives better results. The latter is a result we found only at the end of our study, and as properly performing the comparisons takes considerable time, we could not yet include the comparisons using $fc = cont$. For the final version of the paper we will perform comparisons using $fc = cont$, as we want to compare with the best approach we found and the impact of for instance the maximum number of backtracks nb may be different for $fc = cont$ than for $fc = first$.*

Detailed Results

Tables 6, 7, 8, 9 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 deviation, the average deviation and the average running time obtained when performing 1000 stable iterations. The following two columns report the best and average deviation obtained with $STRand$ when using the following parameters: $\alpha = 0.2$, $\beta = 0$, $nb = 100$, and $fc = cont$. Running time for each instance is limited to the maximum running time computed previously. Best and average deviation are computed over 5 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.

Instance	LB/UB	MvH	Avg Running Times			0.2/0/100/cont		Best
			best	avg	time	best	avg	
la01d	666	-	666	666	1	666	666	666
la01t	666	-	666	666	1	666	666	666
la02d	655	-	655	660	32	655	655	655
la02t	655	-	658	662	53	655	655	655
la03d	593/597	-	596	602	25	598	603	**595**
la03t	590/597	-	595	601	46	594	599	**594**
la04d	572/590	577	611	613	16	576	580	**576**
la04t	570/590	584	592	609	40	575	577	**575**
la05d	593	-	593	593	1	593	593	593
la05t	593	-	593	593	1	593	593	593
la06d	926	-	926	926	1	926	926	926
la06t	926	-	926	926	1	926	926	926
la07d	890	-	890	890	3	890	890	890
la07t	890	-	890	890	43	890	890	890
la08d	863	-	863	863	3	863	863	863
la08t	863	-	863	863	29	863	863	863
la09d	951	-	951	951	1	951	951	951
la09t	951	-	951	951	1	951	951	951
la10d	958	-	958	958	1	958	958	958
la10t	958	-	958	958	1	958	958	958

Table 6: Detailed Results on Set A

On set A, 4 new upper bounds upon 4 open instances have been found. Two of them improve results reported in (Michel & Van Hentenryck 2004) (LA04D and LA04T).

On set B, 6 new upper bounds upon 10 open instances have been found. All these 6 new upper bounds improve results reported in (Michel & Van Hentenryck 2004) (LA16D, LA17D, LA18D, LA18T, LA19D, and LA19T). For the 4 other open instances, (Michel & Van Hentenryck 2004) have the best results.

On set C, 6 new upper bounds upon 10 open instances have been found (LA21D, LA21T, LA24D, LA24T, LA25D, and LA25T), 3 of them improve results reported in (Michel & Van Hentenryck 2004).

On set D, again 6 new upper bounds upon 10 open instances have been found (LA36D, LA36T, LA38T, LA39D, LA40D, and LA40T), one of them improves a result reported in (Michel & Van Hentenryck 2004).

Instance	LB/UB	MvH	Avg Running Times			0.2/0/100/cont		Best
			best	avg	time	best	avg	
la11d	1222	-	1222	1222	1	1222	1222	1222
la11t	1222	-	1222	1222	1	1222	1222	1222
la12d	1039	-	1039	1039	3	1039	1039	1039
la12t	1039	-	1039	1039	47	1039	1039	1039
la13d	1150	-	1150	1150	2	1150	1150	1150
la13t	1150	-	1150	1150	23	1150	1150	1150
la14d	1292	-	1292	1292	1	1292	1292	1292
la14t	1292	-	1292	1292	1	1292	1292	1292
la15d	1207	-	1207	1207	13	1207	1207	1207
la15t	1207	-	1207	1207	70	1207	1207	1207
la16d	892/935	929	933	941	56	928	936	**927**
la16t	887/935	927	939	942	168	928	936	928
la17d	754/765	756	763	763	53	757	761	**755**
la17t	753/765	761	764	767	84	764	765	762
la18d	803/844	818	814	829	78	828	837	**814**
la18t	783/844	813	816	822	125	813	815	**809**
la19d	756/840	803	819	826	60	803	812	**802**
la19t	740/840	801	803	808	174	799	803	**793**
la20d	849/902	864	877	891	79	872	883	871
la20t	842/902	863	872	874	136	870	872	869

Table 7: Detailed Results on Set B

Instance	LB/ UB	MvH	Avg Running Times			0.2/0/100/cont		Best
			best	avg	time	best	avg	
la21d	1017/1046	-	1041	1048	262	1039	1040	**1035**
la21t	1012/1046	-	1068	1078	540	1040	1044	**1036**
la22d	913/927	-	932	941	113	928	939	928
la22t	913/927	-	933	938	401	929	937	929
la23d	1032	-	1032	1032	36	1032	1032	1032
la23t	1032	-	1032	1033	226	1032	1032	1032
la24d	885/935	932	927	935	194	905	911	**905**
la24t	884/935	929	926	932	661	901	910	**901**
la25d	907/977	-	960	970	217	958	966	**956**
la25t	903/977	965	962	969	545	949	959	**947**
la26d	1218	-	1218	1218	378	1218	1218	1218
la26t	1218	-	1218	1218	996	1218	1218	1218
la27d	1235	-	1251	1257	638	1244	1247	1242
la27t	1235	-	1266	1270	1062	1248	1254	1239
la28d	1216	-	1245	1262	380	1226	1238	1226
la28t	1216	-	1259	1271	710	1239	1256	1224
la29d	1117/1130	-	1168	1186	519	1147	1165	1144
la29t	1116/1130	-	1203	1212	1408	1146	1158	1144
la30d	1355	-	1355	1355	194	1355	1355	1355
la30t	1355	-	1355	1355	467	1355	1355	1355

Table 8: Detailed Results on Set C

Instance	LB/UB	MvH	Avg Running Times			0.2/0/100/cont		Best
			best	avg	time	best	avg	
la31d	1784	-	1784	1784	66	1784	1784	1784
la31t	1784	-	1784	1784	452	1784	1784	1784
la32d	1850	-	1850	1850	1	1850	1850	1850
la32t	1850	-	1850	1850	4	1850	1850	1850
la33d	1719	-	1719	1719	73	1719	1719	1719
la33t	1719	-	1719	1719	611	1719	1719	1719
la34d	1721	-	1721	1721	277	1721	1721	1721
la34t	1721	-	1721	1721	871	1721	1721	1721
la35d	1888	-	1888	1888	343	1888	1888	1888
la35t	1888	-	1888	1888	1042	1888	1888	1888
la36d	1229/1268	-	1262	1273	421	1255	1262	**1255**
la36t	1227/1268	-	1261	1265	834	1256	1258	**1253**
la37d	1378/1397	-	1439	1449	381	1423	1437	1423
la37t	1370/1397	-	1446	1455	1112	1431	1441	1431
la38d	1092/1196	1185	1213	1231	446	1207	1213	1189
la38t	1087/1196	1195	1203	1229	1407	1178	1189	**1178**
la39d	1221/1233	-	1248	1250	454	1229	1239	**1229**
la39t	1221/1233	-	1240	1255	809	1236	1246	1236
la40d	1180/1222	-	1228	1240	470	1227	1231	**1216**
la40t	1176/1222	-	1236	1249	1145	1221	1228	**1221**

Table 9: Detailed Results on Set D

Instance	LB/UB	MvH	Avg Running Times			0.2/0/100/cont		Best
			best	avg	time	best	avg	
mt06d	55	-	56	56	8	55	55	55
mt06t	55	-	56	56	17	55	55	55
mt10d	837/930	913	904	920	94	900	909	**900**
mt10t	828/930	912	895	905	216	887	890	**883**
mt20d	1165	1186	1212	1222	113	1185	1211	1185
mt20t	1165	1205	1200	1215	213	1197	1215	1184

Table 10: Detailed Results on Set MT

On set MT, new upper bounds have been found for two instances (MT10D and MT10T).

On MT20D and MT20T, $STRand$ also improves the upper bounds ever reported but still does not reach the upper bound derived from jobshop scheduling.

Overall, we improved 24 upper bounds upon 36 open instances. For 6 other instances (LA27D, LA27T, LA28D, LA28T, MT20D, MT20T) we found solutions with a makespan that was not yet reported before, although we did not yet find a schedule with the makespan that can be derived from the corresponding job shop scheduling instances. On all other instances, the optimal solution has been systematically obtained.

Conclusion

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 CJSSP. New upper bounds have been found on 24 out of 36 open instances. 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

Applegate, D., and Cook, W. 1991. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing* 3(2):149–156.

Baptiste, P.; Le Pape, C.; and Nuijten, W. 1999. Satisfiability tests and time-bound adjustments for cumulative scheduling problems. *Annals of Operations Research* 92:305–333.

Cesta, A.; Oddi, A.; and Smith, S. 2000. Iterative flattening: A scalable method for solving multi-capacity scheduling problems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-00)*.

French, S. 1982. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*. Wiley & Sons.

Laborie, P. 2003. Resource Temporal Networks: Definition and Complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*.

Laborie, P. 2005. New results on the resource-constrained project scheduling problem based on a pure constraint programming approach. In *Submitted paper*.

Le Pape, C.; Couronne, P.; Vergamini, D.; and Gosselin, V. 1994. Time-versus-capacity compromises in project scheduling. In *Proceedings of the Thirteenth Workshop of the U.K. Planning Special Interest Group*.

Le Pape, C. 1994. Implementation of resource constraints in ilog schedule: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering* 3(2):55–66.

Li, K., and Willis, R. 1992. An iterative scheduling technique for resource-constrained project scheduling. *European Journal of Operational Research* 56:370–379.

Michel, L., and Van Hentenryck, P. 2003. Maintaining longest paths incrementally. In *Proceedings of the International Conference on Constraint Programming (CP-03)*.

Michel, L., and Van Hentenryck, P. 2004. Iterative relaxations for iterative flattening in cumulative scheduling. In *Proceedings of the 14Th International Conference on Automated Planning & Scheduling (ICAPS-04)*.

Nuijten, W., and Le Pape, C. 1998. Constraint-based job shop scheduling with ILOG SCHEDULER. *Journal of Heuristics* 3:271–286.

Nuijten, W. 1994. *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach*. Ph.D. Dissertation, Eindhoven University of Technology.

Nuijten, W., and Aarts, E. 1996. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operational Research* 90(2):269–284.

Palpant, M.; Artigues, C.; and Michelon, P. 2004. LSSPER: Solving the resource-constrained project scheduling problem with large neighbourhood search. *Annals of Operations Research* 31(1):237–257.

Policella, N.; Cesta, A.; Oddi, A.; and Smith, S. 2004. Generating robust schedules through temporal flexibility. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 04)*.

Shaw, P. 1998. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceeding CP98*.