Just-In-Time Scheduling with Constraint Programming

Anonymous and **friends**
Some affiliation
Some place
Some Country

Abstract

This paper considers Just-In-Time Job-Shop Scheduling, in which each activity has an earliness and a tardiness cost with respect to a due date. It proposes a constraint programming approach, which includes a novel filtering algorithm and dedicated heuristics. The filtering algorithm uses a machine relaxation to produce a lower bound that can be obtained by solving a Just-In-Time Pert problem. It also includes pruning rules which update the variable bounds and detect precedence constraints. The paper presents experimental results which demonstrate the effectiveness of the approach over a wide range of benchmarks.

Introduction

Scheduling problems may feature a variety of objective functions. Minimizing makespan and the sum of weighted tardiness are probably the most commonly used and they aim at scheduling activities early. Just-In-Time Scheduling is a class of problems which has been gaining in importance and whose goal is not to schedule activities as soon as possible but rather at the right moment. The simplest objective capturing this high-level goal consists in having linear earliness and tardiness costs with respect to a fixed due date for each activity or each job.

This paper studies the Just-In-Time Job-Shop Problem (JITJSP) proposed in (Baptiste, Flamini, and Sourd 2008) and whose definition is the following. Let N be the number of jobs and M be the number of machines. Each job is composed of a sequence of M activities. Each activity A is described by the following information:

- dur(A) : The execution time of activity A.
- m(A) : The machine required by activity A.
- d(A) : The due date of activity A.
- e(A) : The earliness unit cost of activity A.
- t(A) : The tardiness unit cost of activity A.

The constraints impose that the activities of a job must be executed in the given order and that two activities requiring the same machine cannot execute at the same time. The objective is to minimize the sum of the earliness and tardiness costs of all the activities. More formally, if $C(A)$ is the completion time of activity A, the earliness cost of activity A is defined as

$$E(A) = max(0, e(A) * (d(A) - C(A)))$$

and the tardiness cost as

$$T(A) = max(0, t(A) * (C(A) - d(A))).$$

The objective is to minimize the sum of $E(A) + T(A)$ for all activities. We assume that $e(A)$ and $t(A)$ are positive for all activities.

Compared to other Just-In-Time Job-Shop problems, this problem requires the costs to be defined on all activities instead of on the last activity of each job only. As discussed in (Baptiste, Flamini, and Sourd 2008), it is more realistic to have earliness costs (e.g., storage cost) not only for the finished products but also for all the partial products.

A version of the problem with costs on the last activities only was studied in several papers (Beck and Refalo 2003; Danna and Perron 2003), as well as in Just-In-Time Resource Constrained Project Scheduling Problems (JITRCPSP) (Vanhoucke, Demeulemeester, and Herroelen 2001). However, to the best of our knowledge, the JITJSP has only been tackled in (Baptiste, Flamini, and Sourd 2008), in which the authors propose several lower bounds and use a local search procedure to produce upper bounds in order to assess the value of their Lower Bounds.

This paper proposes a constraint-programming (CP) approach to the JITJSP. Its main focus is on the design of a global constraint for the earliness and tardiness costs but it also presents heuristics to guide the search, as well as a Large Neighborhood Search (LNS) to scale to larger instances. The paper starts by presenting the general search algorithm. It then presents the filtering algorithm and some of its theoretical results before introducing some hybrid search strategies. Finally, the paper presents the experimental results and concludes.

Branch-and-Bound for JITJSP

This section describes the CP model and the Branch-and-Bound (BB) strategy to solve the JITJSP. The basic CP model is given in Figure 1. It is specified in COMET, an object-oriented optimization programming language (Van

```
1  range Activities = 1..N*M;
2  range Machines = 1..M;
3  range Jobs = 1..N;
4
5  Scheduler<CP> cp(0,infinity);
6  UnaryResource<CP> r[Machines](cp);
7  Activity<CP> a[i in Activities](cp,dur[i]);
8  var<CP>{int}[] C = all(i in acts)a[i].end();
9  var<CP>{int} cost(cp,0..infinity);
10 var<CP>{int} E[Activities](cp,0..infinity);
11 var<CP>{int} T[Activities](cp,0..infinity);
12
13 minimize<cp>
14   cost
15 subject to {
16   forall(j in Jobs)
17     forall(i in job[j].low()..job[j].up()-1)
18       a[job[j][i]].precedes(a[job[j][i+1]]);
19   forall(i in Activities){
20     a[i].requires(r[m[i]]);
21     cp.post(E[i]==max(0,e[i]*(d[i]-C[i])));
22     cp.post(T[i]==max(0,t[i]*(C[i]-d[i])));
23   }
24   cp.post(cost==sum(i in Activities)(E[i]+T[i]));
25 }
```

Figure 1: The Comet Model for the JITJSP.

```
1  Input: S : JITJSP instance
2  Input-Output: UB : Global Variable (Upper Bound)
3  Output: Best found solution
4  solve(S, UB){
5    propagate();
6    solve a machine relaxation of S;
7    Let LB be the value of the solution;
8    if (LB>=UB) fail;
9    if (the solution is machine-feasible){
10     UB := LB;
11     Save current solution;
12   }else{
13     find 2 conflicting activities A and B;
14     try{
15       S.add(A precedes B);
16       solve(S, UB);
17     }or{
18       S.add(B precedes A);
19       solve(S, UB);
20     }
21   }
22 }
```

Figure 2: Pseudo-code of the Branch-and-Bound

Hentenryck and Michel 2005) which features a module to solve scheduling problems with Constraint Programming.

The model contains the input data presented in the first section. In addition, the matrix job contains, for each job, the ordered indexes of its activities. For instance, $job[4][3]$ is the index of the third activity of the fourth job. The declaration of the data is not shown in Figure 1. The decision variables are the completion dates of the activities and $C(A)$ denotes the completion date of activity A in the remainder of the paper. In the model, array C declared in line 8 contains the completion dates of the activities. The arrays E and T contain the auxiliary variables for the earliness and tardiness of each activity and the variable $cost$ is the total cost to be minimized (lines 13-14). Those auxiliary variables are linked to the decision variables by the constraints in lines 21, 22, and 24. The other constraints of the problem are the precedences into the jobs (lines 16–18) and the machine requirements (line 20).

Each unary resource (which represents a machine) supports the traditional disjunctive scheduling, edge-finding (Carlier and Pinson 1989; 1994), and not-first-not-last (Carlier and Pinson 1990; Vilim 2004) algorithms. An almost identical model may be used to solve efficiently the classical Job-Shop Problem with makespan minimization. However, the above model is inefficient for the JITJSP, as the sum constraint in line 24 does not propagate information from the cost variable back to the decision variables (i.e., the completion time of the activities). In order to overcome this shortcoming, this paper introduces a global constraint to perform deductions based on the cost variable and the current state of the schedule.

The search procedure is given by the pseudo-code in Figure 2. This procedure is recursive and each call corresponds to a node of the search tree. After constraint propagation, a relaxation of the problem is solved. This relaxation consists in removing the machines of the problem, resulting in a PERT problem with convex cost functions that can be solved in polynomial time (Chrétienne and Sourd 2003). The value of this relaxation gives a valid Lower Bound (LB) for the original problem. If this LB is larger or equal to the current Upper Bound (UB), it is useless to explore the subtree rooted at the current node. In contrast, if the LB is smaller than the UB and the solution to the relaxation satisfies all the machine constraints (meaning that no two tasks that require the same machine overlap in time), the search has found a new best solution and it updates the UB. Finally, if the solution of the relaxation is not feasible in the original problem, this means that at least two activities requiring the same machine overlap in time. This constraint violation can be repaired by adding a precedence constraint between the two conflicting activities. As there are two possibilities to order the two activities, it is necessary to branch and explore the two situations recursively (which is represented in Figure 2 by the non-deterministic instruction "try{}or{}" (Van Hentenryck and Michel 2005)). Note that adding a precedence can only increase the total cost.

A Global Constraint for Earliness/Tardiness

This section introduces a global constraint to reduce the search space using both the UB and the machine relaxation of the problem. The global constraint updates the domains

of the variables, detects implied precedence relations, and provides heuristic information for branching.

The machine relaxation (removing the machines and the associated requirements) is a PERT problem with convex cost functions. This problem can be solved in $\mathcal{O}(n \max\{n, m\})$ with the algorithm of (Chrétienne and Sourd 2003) for linear earliness and tardiness costs. In this paper, the algorithmic complexities are expressed in function of the number of activities (n) and the number of precedences between activities (m). Sourd and Chrétienne's algorithm can also be generalized naturally to the case of convex piecewise linear cost functions. This generalization is important in this paper to accommodate the release dates and deadlines of the activities. As noted in (Hendel and Sourd 2007), it suffices to add almost vertical segments to the cost function of an activity at its release date and deadline to model these constraints.

As mentioned, from the optimal solution of the relaxation, it is possible to perform several deductions such as bounds reduction and precedence detection. The bound reductions take the form of unary constraints on the decision variables $C(A)$, while the precedences are binary constraints on pairs of decision variables. We now review these two forms of propagation and the heuristic information provided by the global constraint.

Bound Reduction

The solution of the machine relaxation (i.e., the PERT problem) gives a fixed completion time for each activity A which is denoted by $C^*(A)$. Both kinds of deduction are based on what happens to the cost if the optimal solution of the relaxation is perturbed by the displacement of an activity A earlier or later than $C^*(A)$. The modification of the cost as a function of the completion time of an activity is a convex piecewise linear function whose minimum coincides with the optimal solution of the machine relaxation. Let Δ_A denote the function for activity A giving the increase in cost with respect to the optimal solution of the PERT problem. Δ_A is a convex piecewise linear function of the completion time of an activity and its minimum is $\Delta_A(C^*(A)) = 0$.

The first type of pruning is the bound reduction of the completion time variable of each activity. This pruning is enforced by the constraint

$$\Delta_A(C(A)) < (UB - LB)$$

and, since Δ_A is a convex function, this constraint may directly update the bounds of the variable $C(A)$. Figure 3 shows an example of a Δ_A function with the new inferred bounds denoted by $mC(A)$ and $MC(A)$ in the figure.

Precedence Detection

The second type of pruning consists in detecting precedences that must hold between two tasks in conflict in the original problem (i.e., two tasks requiring the same machine and overlapping in time in the optimal solution of the PERT relaxation). If two activities A and B are in conflict, either A must precede B or B must precede A. If A is forced to precede B, A and B cannot stay at the

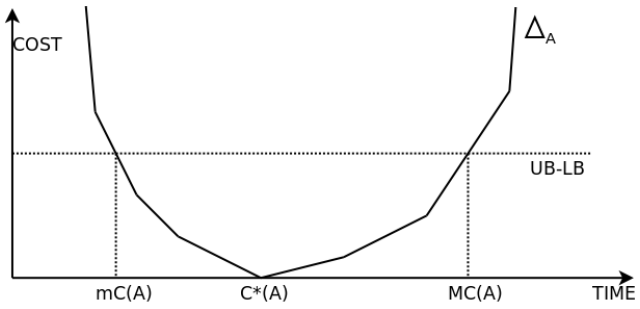

Figure 3: Illustration of the Delta Function and the Bound Reduction.

minimum of their respective Δ_A (or Δ_B) functions. They must move and their optimal positions minimize the function $\Delta_A(x) + \Delta_B(x + dur(B))$ over x, where x represents the value given to variable $C(A)$. Let us call $Inc(A, B)$ the minimum of this function, i.e.,

$$Inc(A, B) = \min_x(\Delta_A(x) + \Delta_B(x + dur(B))).$$

$Inc(A, B)$ represents the minimum increase of the total cost when A is forced to precede B and $C^*(B) - C^*(A) < dur(B)$. This last condition is true whenever A and B are in conflict. As $C^*(B) - C^*(A) < dur(B)$ holds and the Δ functions are convex, it follows that the increase is minimized when there is no free time between A and B, i.e. if A ends at x and B ends at $x + dur(B)$. The filtering can be written as:

- if $Inc(A, B) > (UB - LB)$, then post (B precedes A).
- if $Inc(B, A) > (UB - LB)$, then post (A precedes B).

If $Inc(A, B)$ is larger than the allowed increase, then the opposite precedence can be posted. As the sum of two convex functions is also a convex function, it is easy to compute $Inc(A, B)$ and $Inc(B, A)$ and check whether some precedence constraint must be posted.

Branching Heuristics

In addition to the above pruning, the information computed for the filtering can be used to guide the search heuristically. As indicated in Figure 2, the branching consists in adding precedences between two conflicting activities. The first-fail principle commands to choose two activities to detect failures earlier in the search tree. In the present problem, this suggests choosing a pair of activities that improves the lower bound the most.

More precisely, the search strategy adopted in the algorithm resolves all the conflicts of one machine before going onto another machine. It chooses the machine with the largest sum of minimum increase ($Inc(A, B)$) for all its conflicts. Among the activities requiring the chosen machine, the heuristic chooses to branch on the two conflicting activities maximizing the minimal increase in cost when they are ordered (as computed for the filtering), i.e.,

$$\underset{A\,and\,B}{\text{argmax}}\{\max(Inc(A, B), Inc(B, A))\}$$

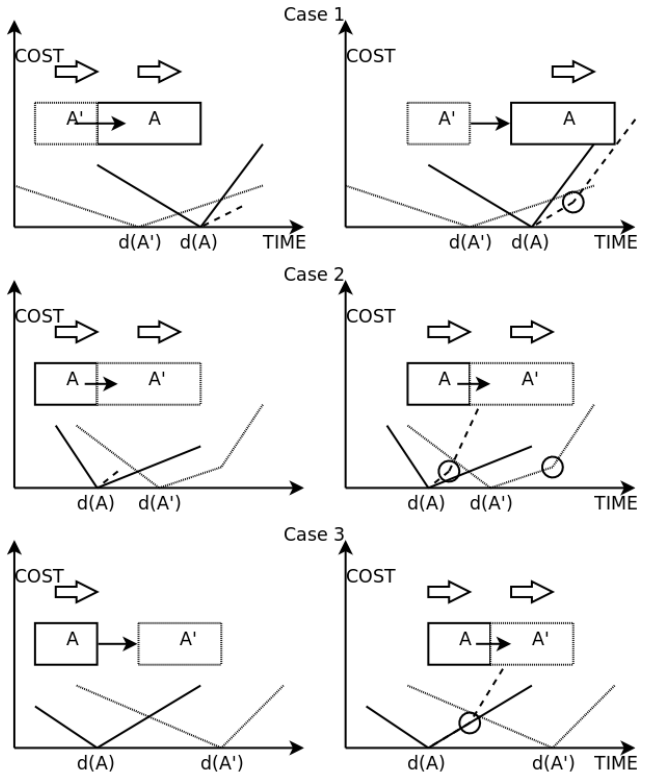

Figure 4: Illustration of the three cases in the proof of Theorem 1, before (left) and after (right) a breakpoint. The simple arrows depict precedences and the large arrows show the activities that are moving. Below the activities are the corresponding individual cost functions and the right of the Δ_A cost function (dashed).

To guide the search towards good solutions (and improve the upper bound), the branch with the smallest increase in cost is visited first

$$\text{argmin}\{Inc(A, B), Inc(B, A)\}.$$

Slope Computations for the Cost Functions

This section presents the algorithmic and theoretical results behind the computation of the Δ_A functions. It first proves that the cost functions are convex and piecewise linear. It then proposes an approximation of the Δ_A function and shows that it is sound with respect to pruning. Finally, the section describes how to compute these functions.

The Shape of the Slope

Theorem 1 *Starting from the optimal solution of a machine relaxation, the function Δ_A, the evolution of the cost as a function of the completion time of activity A, is convex piecewise linear.*

Proof (Sketch) From the optimal solution, moving a task to the right or to the left can only increase the cost, as the current position is at the minimum. Let $LS(A)$ and $RS(A)$ be the unit increase cost (or slope) directly to the left and to the right of $C^*(A)$ respectively. In general, slopes of the Δ_A functions are denoted by an uppercase S, while the slopes of the individual cost function of the activities (i.e., their joint earliest/tardiness cost functions with additional segments for capturing the release and deadline constraints) are denoted by a lowercase s. For brevity, we only consider the right part of the Δ_A function as the left part is similar. When activity A is pushed to the right, some other activities must move to satisfy the precedence constraints or should move to reduce the increase in cost. At any point in time, the slope S of the function Δ_A is the sum of all the individual slopes s of the currently moving activities. Some slope s may be negative but their overall sum is positive on the right of $C^*(A)$. Directly on the right of $C^*(A)$, the cost is increased by $RS(A)$. However, the cost will increase further subsequently when breakpoints of other slopes are reached. The breakpoints are of several kinds.

1. An activity A' (or a group of activities) reaches an optimum of its individual cost function and is not forced to move by precedence constraints. This activity is left at its optimum and the slope S of Δ_A is increased by the opposite of the slope s at the left of the optimum of the individual function of A'. This is a positive increment as the slope s on the left of the optimum is negative by definition. This is illustrated in the first part of Figure 4 which considers the move of activity A. Before reaching $d(A')$ (left of the figure), A' moves with A. Passed $d(A')$, A continues alone and A' is left at its optimum (right of the figure).
2. A moving activity A' reaches a breakpoint of its own individual cost function (and it does not respect the conditions of case 1): The slope S of Δ_A is increased by the difference between the slopes s on the left and on the right of the breakpoint. This increment is positive as the individual cost functions are convex. This is shown in the middle part of Figure 4.[1]
3. The block of moving activities reaches a non-moving activity A'. The slope S of Δ_A is increased by $RS(A')$ which is positive, as this activity was optimally scheduled. This case is illustrated in the third part of Figure 4.

Each successive breakpoint falls in one of those three categories. The increase of the slope S of Δ_A at every breakpoint is thus positive making the whole function convex. The slope S is only modified at breakpoints and, between two breakpoints, the function follows linear segments, making the whole function piecewise linear. ■

Approximating the Δ_A Function

The exact outline of the Δ_A function for each activity can be computed with a variation of the PERT algorithm of (Chrétienne and Sourd 2003). This would lead to a time complexity of $\mathcal{O}(n^2 \max\{n, m\})$. For this reason, our implementation uses a lower approximation $\widetilde{\Delta_A}$ of the Δ_A function that only considers the initial slopes ($RS(A)$ and

[1]Note that the individual functions may have more than 2 segments due to the release and deadline constraints.

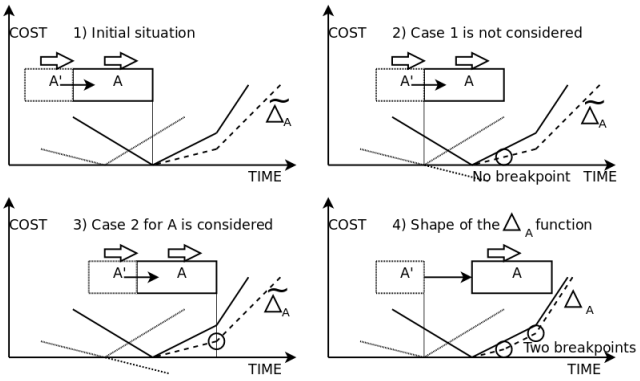

Figure 5: Illustration of the relaxation of the cost function. Parts 1-3 show the relaxed function $\widetilde{\Delta_A}$ while Part 4 (lower right) presents Δ_A.

$LS(A)$) and a subset of the breakpoints. Considering only a subset of the breakpoints gives a convex piecewise linear function that is a lower bound of Δ_A. This lower approximation is used instead of Δ_A in the implementation.

The breakpoints used in the lower approximation belong to the second category given in the proof of Theorem 1. For such breakpoints, it is relatively easy to compute the individual breakpoints of each activity and the associated increases in slope. The activities considered for $\widetilde{\Delta_A}$ are the successors of A in the transitive closure of the precedence graph. The breakpoints for all $\widetilde{\Delta_A}$ functions can then be computed in one pass over the precedence graph.

Figure 5 illustrates the functions Δ_A and $\widetilde{\Delta_A}$ on a small example with two activities. Part 1 of the figure shows the initial state at $C^*(A)$. Part 2 shows that breakpoints of the first category are not considered for $\widetilde{\Delta_A}$, as if the individual cost function of A' was a linear decreasing function. Part 3 shows a considered breakpoint of the individual cost function of A. Finally, part 4 presents the shape of Δ_A when all breakpoints are considered.

Computing $RS(A)$

It remains to show how to compute $RS(A)$ and $LS(A)$ for every activity A. For brevity, we consider the $RS(A)$ case only. Its computation can be performed in several ways. The first possibility is to reuse the algorithm for PERT scheduling with the additional constraint that A cannot finish earlier than $C^*(A)+1$. The difference between the optimal total cost of the variation and the optimal cost of the base version gives $RS(A)$. This mechanism has the disadvantage to run the whole PERT algorithm for every activity giving a time complexity of $\mathcal{O}(n^2 \max\{n,m\})$. A better way is to use an adaptation of the PERT algorithm. Starting from the optimal solution for the machine relaxation, it consists in adding a new fictional activity A' that has only A as successor. The due date of this task is $C^*(A) - dur(A) + 1$ and its earliness cost is set to an arbitrarily large value $e(A') = M$. The idea is to perform the first steps of the PERT algorithm

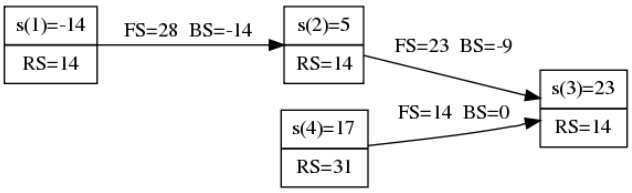

Figure 6: Illustration of the computation of the RS(A)

until the step where it is necessary to move the block of A. At this point, the slope of the block containing A and A' is equal to $RS(A)+M$. The time complexity is bounded by $\mathcal{O}(n^2 \max\{n,m\})$. However, in practice, there are very few steps of the algorithm to perform and it is easy to remove A' from the schedule in order to be ready to compute the next RS.

Faster Computation of $RS(A)$

A faster and more elegant way to compute the $RS(A)$ exists for a special case that appears often in practice. Call equality arcs the arcs (A,B) of the precedence graph satisfying $C^*(B) - C^*(A) = dur(B)$, that is the arcs between two activities that are directly chaining up. The equality graph is the restriction of the precedence graph to the equality arcs. Clearly, only the activities that are part of the connected component of A in the equality graph may impact the value of $RS(A)$. In the special case where the equality graph consists of trees (there is no cycle in the underlying undirected graph), the following recurrence relations allow to compute the $RS(A)$ for all activities efficiently:

- $RS(A) = s(A) + \sum_{(A,B)} FS(A,B) + \sum_{(B,A)} BS(A,B)$
- $FS(A,B) = s(B) + \sum_{(B,C)} FS(B,C) + \sum_{(C,B)\neq(A,B)} BS(B,C)$
- $BS(A,B) = \min(0, s(B) + \sum_{(B,C)\neq(B,A)} FS(B,C) + \sum_{(C,B)} BS(B,C))$

where $s(A)$ is the slope of the individual cost function of activity A directly to the right of $C^*(A)$. The summations are performed over every in- or out-arcs of an activity in the equality graph. As the equality graph is a collection of trees, each arc separates a connected component into two disconnect parts. $FS(A,B)$ (forward slope) is the slope induced by the part containing B when A is moved. The same is true for $BS(A,B)$ (backward slope) except that it may be zero as the part containing B is not forced to move when A moves if its slope is positive. The base of the recurrence relations happen when the activity B of $FS(A,B)$ (resp. $BS(A,B)$) is incident only to the edge (A,B) (resp. (B,A)). In such a case, $FS(A,B) = s(B)$ (resp. $BS(A,B) = \min(0, s(B))$). Another special case is when an activity A is not incident to any edge. In this case, $RS(A) = s(A)$.There are two values to compute for each arc and one for each node, that is there are $\mathcal{O}(n+m)$ values to compute in total.

Figure 6 presents a little example with 4 activities. The $s(A)$ and $RS(A)$ are noted in the respective activities, while the $FS(A, B)$ and $BS(A, B)$ are noted on the arcs. The arc $(4, 3)$ illustrates the case where activity 4 does not move when any of the three others move, as it would incur an additional cost of 17 ($BS(3, 4) = 0$). On the contrary, activities 1 and 2 are moved whenever activity 3 is displaced because it reduces the cost ($BS(2, 1)$ and $BS(3, 2)$ are negative).

The same relations exist for $LS(A)$. In the experiments, the faster computation mechanism is used whenever it is possible. The variation of the PERT algorithm is only used in the cases where the equality graph is not composed of trees.

Additional Heuristics

As JITJSPs are extremely hard problems, we embedded two additional mechanisms into our search procedure: a simple local search to post-optimize each solution and a Large Neighborhood Search (Shaw 1998; Danna and Perron 2003).

Simple Local Search

The local search starts from a feasible solution and try to improve it greedily by swapping the order of two tasks that execute successively on the same machine and undo the move if it does not improve the value of the solution. This is repeated until a local optimum is obtained. Each time the branch-and-bound finds a new solution, the local search is run from the current solution trying to improve it. The value of the local optimum is then used as new UB. This makes it possible to obtain good upper bounds early. It is important to note that the addition of this local search preserves the completeness of the Branch-and-Bound search.

Large Neighborhood Search

In addition to the above complete search procedure, we also implemented an incomplete Large Neighborhood Search (LNS). LNS is a local search whose moves explore the solutions of a subproblem using CP. For the JITJSP, our LNS consists of the following steps:

1. Let $n = 20$;
2. Choose $n/10$ machines.
3. Relax the current solution by removing the precedences between activities executed on the chosen machines.
4. Solve the problem with the B&B search limited to 1000 fails.
5. Update the current solution if a better solution has been found.
6. Increase n if the search was complete, decrease it otherwise (with a minimum of 20).
7. Go to step 2 unless the running time is exhausted.

The choice of the machines to relax is performed randomly according to a distribution that reflects the costs incurred by the activities executed on each machine.

Instance	LB	BF&S	CP	CP+ls	LNS
10x2-e-t-1	434	**453**	461.96	461.96	522.9
10x2-e-t-2	*448.32*	458	**448.32**	**448.32**	484.86
10x5-e-t-1	660	826	935.78	783.43	**764.8**
10x5-e-t-2	612	848	**779.40**	**779.40**	808.64
10x10-e-t-1	1126	1439	1622.26	**1339.64**	1527.28
10x10-e-t-2	1535	2006	1930.65	1930.65	**1902.3**
10x2-e-l-1	*224.84*	225	**224.84**	**224.84**	225.81
10x2-e-l-2	313	324	**319.37**	**319.37**	347.65
10x5-e-l-1	1263	1905	1995.50	1877.93	**1823.85**
10x5-e-l-2	878	1010	1851.56	1155.89	**999.14**
10x10-e-l-1	331	**376**	620.29	403.87	381.88
10x10-e-l-2	246	260	325.92	274.31	**256.78**
10x2-t-t-1	*179.46*	195	**179.46**	**179.46**	193.44
10x2-t-t-2	143	**147**	173.67	164.38	164.38
10x5-t-t-1	361	405	444.64	407.40	**398.37**
10x5-t-t-2	461	708	722.76	707.81	**639.16**
10x10-t-t-1	574	855	928.98	806.74	**773.26**
10x10-t-t-2	666	**800**	1094.71	879.50	830.39
10x2-t-l-1	416	**416**	**416.44**	**416.44**	**416.44**
10x2-t-l-2	137	138	148.31	**137.94**	147.00
10x5-t-l-1	168	188	243.06	199.91	**182.64**
10x5-t-l-2	355	572	733.69	**513.91**	542.29
10x10-t-l-1	356	409	476.82	402.27	**387.05**
10x10-t-l-2	138	152	152.66	151.97	**144.94**

Table 1: Results for instances with 10 jobs

Experimental Validation

The aim of the experiments is to demonstrate the effectiveness of the global constraint and the influence of the two additional mechanisms to help solving the JITJSP. The benchmarks are those introduced in (Baptiste, Flamini, and Sourd 2008) and include 72 instances ranging from 20 to 200 activities. The instances are distributed following four criteria: tightness of the due dates, repartition of the costs, number of jobs, and number of machines. The due dates are either tight or loose. If they are tight, the distance between the due dates of two successive activities of a job are equal to the duration of the second activity. If they are loose, some free time is allocated between two due dates. The earliness and tardiness unit costs are either taken randomly in $[0.1, 1]$ (equal scheme) or in $[0.1, 0.3]$ for the earliness cost and in $[0.1, 1]$ for the tardiness cost (tard scheme). The number of jobs is 10, 15, or 20, and the number of machines is 2, 5, or 10. Only 1 instance of this benchmark was previously closed (optimum known and proved), namely the 10x2-t-l-1 instance (10 jobs, 2 machines, tight due dates, loose cost scheme, number 1).

We ran three versions of our search on the whole collection of benchmarks. The first one, denoted *CP*, is the pure CP approach with the novel global constraint. *CP+ls* adds the local search described in the previous section and *LNS* adds the local search and the Large Neighborhood Search of the previous section. Every run is allocated 600 seconds (10 minutes) and is performed on one core of a Intel Core 2 Quad at 2.40GHz with 4MB of memory. The entire algorithm is implemented in Comet.

Instance	LB	BF&S	CP	CP+ls	LNS
15x2-e-t-1	3316	**3559**	4269.09	3641.19	3641.19
15x2-e-t-2	1449	1579	1578.20	**1534.12**	**1534.12**
15x5-e-t-1	1052	1663	1604.52	1538.09	**1504.04**
15x5-e-t-2	1992	**2989**	3042.32	2993.50	3096.60
15x10-e-t-1	4389	8381	9870.99	9089.61	**8189.70**
15x10-e-t-2	3539	7039	10072.79	5665.38	**5536.07**
15x2-e-l-1	1032	**1142**	1453.60	1249.68	1249.68
15x2-e-l-2	490	**520**	550.86	524.10	560.15
15x5-e-l-1	2763	4408	5011.53	3757.93	**3745.96**
15x5-e-l-2	2818	4023	5449.25	3418.87	**3397.42**
15x10-e-l-1	758	1109	1747.86	1083.02	**1033.06**
15x10-e-l-2	1242	2256	3703.48	1937.27	**1792.67**
15x2-t-t-1	786	913	1214.39	**835.52**	**835.52**
15x2-t-t-2	886	956	1100.32	**947.17**	**947.17**
15x5-t-t-1	1014	1538	1567.86	**1530.96**	1597.90
15x5-t-t-2	626	843	959.25	785.36	**775.01**
15x10-t-t-1	649	972	1458.38	**921.67**	923.88
15x10-t-t-2	955	**1656**	2341.59	1663.05	1693.04
15x2-t-l-1	650	730	869.72	**666.37**	**666.37**
15x2-t-l-2	278	**310**	370.98	336.48	336.48
15x5-t-l-1	1098	1723	3802.18	1528.36	**1478.97**
15x5-t-l-2	314	**374**	585.72	409.60	401.65
15x10-t-l-1	258	312	564.65	342.49	**300.11**
15x10-t-l-2	476	855	1378.26	**658.90**	717.90

Table 2: Results for instances with 15 jobs

Instance	LB	BF&S	CP	CP+ls	LNS
20x2-e-t-1	1901	**2008**	2148.81	2115.58	2115.58
20x2-e-t-2	912	**1014**	1400.20	1104.20	1124.47
20x5-e-t-1	2506	**3090**	4085.52	3349.28	3349.28
20x5-e-t-2	5817	**7537**	10226.19	8112.00	7883.50
20x10-e-t-1	6708	**12951**	21405.91	14537.12	14004.85
20x10-e-t-2	5705	9435	13363.65	8603.76	**8535.88**
20x2-e-l-1	2546	**2708**	3410.65	2789.07	2789.07
20x2-e-l-2	3013	**3318**	3760.64	3386.88	3386.88
20x5-e-l-1	6697	9697	15069.25	**9481.56**	**9481.56**
20x5-e-l-2	6017	**8152**	13138.46	8835.72	8835.72
20x10-e-l-1	3538	6732	10773.33	6206.30	**6101.67**
20x10-e-l-2	1344	2516	4797.62	2006.67	**1963.05**
20x2-t-t-1	1515	1913	2395.97	**1892.22**	**1892.22**
20x2-t-t-2	1375	**1594**	2121.18	1704.26	1744.25
20x5-t-t-1	3244	4147	6420.98	**4067.73**	**4067.73**
20x5-t-t-2	1633	**1916**	3497.69	2040.70	2040.70
20x10-t-t-1	3003	5968	9871.52	5172.14	**5125.88**
20x10-t-t-2	2740	**3788**	6229.24	3992.48	3938.51
20x2-t-l-1	1194	**1271**	1545.92	1409.73	1409.73
20x2-t-l-2	735	**857**	1170.59	907.60	907.60
20x5-t-l-1	2524	**3377**	5091.80	4015.62	4644.44
20x5-t-l-2	3060	5014	6946.88	**4539.36**	**4539.36**
20x10-t-l-1	2462	**6237**	20511.34	7462.39	7287.00
20x10-t-l-2	1226	1830	2776.79	1741.44	**1727.88**

Table 3: Results for instances with 20 jobs

Size	BF&S	CP	CP+ls	LNS
10x2	**2.80**	4.78	3.02	8.78
15x2	10.11	29.60	**10.06**	10.98
20x2	**12.31**	40.81	18.08	18.73
10x5	33.54	58.66	32.06	**27.08**
15x5	46.59	90.08	37.48	**37.31**
20x5	**34.49**	102.88	39.91	42.53
10x10	21.48	45.06	21.74	**19.05**
15x10	67.69	152.07	56.67	**51.30**
20x10	84.43	253.81	81.91	**78.53**

Table 4: Average gap by size (in %). Bold values are the Best on Their Benchmarks

Results

The results are presented in Tables 1–3. The column *LB* gives the best-known lower bound. These values are taken from (Baptiste, Flamini, and Sourd 2008), except those we improved which are shown in italic. The column *BF&S* shows the upper bounds from (Baptiste, Flamini, and Sourd 2008) and serves as reference for our results. The last three columns show the cost of the best solutions found by each version of our algorithm. The bold values are the best ones for each instance. On the whole, *CP* improves over *BF&S* on 9 instances. *CP+ls* and *LNS* provide significant benefits and improve over *BF&S* on 40 instances each. In addition, *CP+ls* was able to prove optimality on 5 instances thus closing 4 new instances. These new instances with their respective total costs are:

- 10x2-e-t-2 : 448.32
- 10x2-e-l-1 : 224.84
- 10x2-t-t-1 : 179.46
- 10x2-t-l-2 : 137.94

To give a better idea of the performance of our algorithms, we summarized the results in Table 4. In this table, we give the gap averaged by size. The gap is defined as $(UB - LB)/LB$ where LB is the best-known lower bound of an instance and UB is the total cost computed by each algorithm for the same instance. This table indicates that, for the smallest instances (with 2 machines), *BF&S* gives better solutions. However, as size increases, *CP+ls* and *LNS* give better results on average. The pure *CP* search is not effective due to the lack of a good upper bound early in the search which would allow to prune large parts of the search tree. Adding good upper bounds, thanks to the local search, greatly improved the pure *CP* approach. Finally, LNS improves the larger instances but is useless on instances with 2 machines. Indeed, the idea of the LNS is to relax at least 2 machines, which is the complete problem in 2-machines instances. For such instances, LNS only serves as a restart and cannot drive the search as it is the case for larger problems.

Conclusion and Future Work

This paper introduced a Constraint Programming approach for the Just-In-Time Job-Shop Problem, a Job-Shop variant

where the total earliness and tardiness costs must be minimized. This approach relies on efficient filtering algorithms and intelligent search heuristics. In particular, the paper presented a global filtering algorithm based on a machine relaxation and an analysis of the cost evolution under modification of the completion time of activities. The filtering algorithm reduces the bounds of the decision variables, detects precedences between activities, and provides heuristic information for branching in order to improve the lower bound earlier in the search tree. The CP algorithm was then enhanced by a simple local search performed on each solution to get good upper bounds early. Finally, a large neighborhood search was implemented to find high-quality solutions quickly.

Experimental results show that constraint programming with the local search at the leaves and/or large neighborhood search produces good results. Among the 72 benchmark instances introduced in (Baptiste, Flamini, and Sourd 2008), the algorithm closed 4 open instances and improved the best-known solutions on 46 instances. Additional experiments have shown that the PERT algorithm adapted from (Chrétienne and Sourd 2003) is the bottleneck of the search. Future work will be devoted to the development of an incremental version of a convex PERT algorithm and to an extension of the approach to other kinds of Just-In-Time Scheduling Problems that feature cumulative and state resources. In particular, it will focus on the adaptation of the global constraint for these problems.

References

Baptiste, P.; Flamini, M.; and Sourd, F. 2008. Lagrangian bounds for just-in-time job-shop scheduling. *Comput. Oper. Res.* 35(3):906–915.

Beck, J. C., and Refalo, P. 2003. A hybrid approach to scheduling with earliness and tardiness costs. *Annals OR* 118(1-4):49–71.

Carlier, J., and Pinson, E. 1989. An algorithm for solving the job-shop problem. *Management Science* 35(2):164–176.

Carlier, J., and Pinson, E. 1990. A practical use of jackson's preemptive schedule for solving the job-shop problem. *Ann. Oper. Res.* 26:269–287.

Carlier, J., and Pinson, E. 1994. Adjustment of heads and tails for the job-shop problem. *European J. Oper. Res.* 78:146–161.

Chrétienne, P., and Sourd, F. 2003. Pert scheduling with convex cost functions. *Theor. Comput. Sci.* 292(1):145–164.

Danna, E., and Perron, L. 2003. Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs. In *CP*, 817–821.

Hendel, Y., and Sourd, F. 2007. An improved earliness-tardiness timing algorithm. *Computers & OR* 34(10):2931–2938.

Shaw, P. 1998. Using constraint programming and local search methods to solve vehicle routing problems. *CP* 417431.

Van Hentenryck, P., and Michel, L. 2005. *Constraint-Based Local Search*. The MIT Press.

Vanhoucke, M.; Demeulemeester, E.; and Herroelen, W. 2001. An exact procedure for the resource-constrained weighted earliness-tardiness project scheduling problem. *Annals of Operations Research* 102:179–196.

Vilim, P. 2004. $\mathcal{O}(n \log n)$ filtering algorithms for unary resource constraint. *CPAIOR* 335–347.