

Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems

PHILIPPE BAPTISTE^{1,2} AND CLAUDE LE PAPE¹ baptiste@utc.fr, clepapeg@bouyguetelecom.fr

¹*Bouygues, Direction des Technologies Nouvelles, 1, av. E. Freyssinet, F-78061 Saint-Quentin-en-Yvelines*

²*UMR CNRS 6599 HEUDIASYC, Université de Technologie de Compiègne, F-60205 Compiègne*

Abstract. In recent years, constraint satisfaction techniques have been successfully applied to “disjunctive” scheduling problems, *i.e.*, scheduling problems where each resource can execute at most one activity at a time. Less significant and less generally applicable results have been obtained in the area of “cumulative” scheduling. Multiple constraint propagation algorithms have been developed for cumulative resources but they tend to be less uniformly effective than their disjunctive counterparts. Different problems in the cumulative scheduling class seem to have different characteristics that make them either easy or hard to solve with a given technique. The aim of this paper is to investigate one particular dimension along which problems differ. Within the cumulative scheduling class, we distinguish between “highly disjunctive” and “highly cumulative” problems: a problem is highly disjunctive when many pairs of activities cannot execute in parallel, *e.g.*, because many activities require more than half of the capacity of a resource; on the contrary, a problem is highly cumulative if many activities can effectively execute in parallel. New constraint propagation and problem decomposition techniques are introduced with this distinction in mind. This includes an $O(n^2)$ “edge-finding” algorithm for cumulative resources (where n is the number of activities requiring the same resource) and a problem decomposition scheme which applies well to highly disjunctive project scheduling problems. Experimental results confirm that the impact of these techniques varies from highly disjunctive to highly cumulative problems. In the end, we also propose a refined version of the “edge-finding” algorithm for cumulative resources which, despite its worst case complexity in $O(n^3)$, performs very well on highly cumulative instances.

Keywords: Resource-Constrained Project Scheduling, Cumulative Scheduling, Disjunctive Scheduling, Deduction Rules, Constraint Propagation.

1 Motivations

Many industrial scheduling problems are variants, extensions or restrictions of the “Resource-Constrained Project Scheduling Problem” (RCPSp). Given (1) a set of resources of given capacities, (2) a set of non-interruptible activities of given durations, (3) a network of precedence constraints between the activities, and (4) for each activity and each resource the amount of the resource required by the activity over its execution, the goal of the RCPSp is to find a schedule meeting all the constraints whose makespan (*i.e.*, the time at which all activities are finished) is minimal. The decision variant of the RCPSp, *i.e.*, the problem of determining if there is a schedule of makespan smaller than a given deadline, is NP-complete in the strong sense (Garey and Johnson, 1979).

In a constraint programming framework, two integer variables $\text{start}(A)$ and $\text{end}(A)$, representing the start time and the end time of A , can be associated with each activity A (we assume durations and resource capacities are integers). As usual, the earliest (respectively, latest) start and end times of A , EST_A and EET_A (LST_A and LET_A), are defined as the minimal (maximal) values in the domains of $\text{start}(A)$ and $\text{end}(A)$. See Figure 1. The goal is to minimize $\max_A(\text{end}(A))$ under the following constraints:

1. $0 \leq \text{start}(A)$, for every activity A ;
2. $\text{start}(A) + \text{duration}(A) = \text{end}(A)$, for every activity A ;
3. $\text{end}(A) \leq \text{start}(B)$, for every precedence constraint ($A \rightarrow B$);
4. $\sum_{A \text{ such that } \text{start}(A) \leq t < \text{end}(A)} \text{required-capacity}(A, R) \leq \text{capacity}(R)$, for every resource R and time t (cumulative constraint).

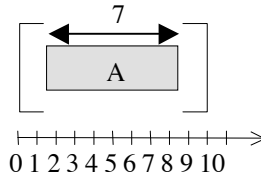


Figure 1. An activity A with earliest start time 0, latest end time 10 and duration 7. The earliest end time of the activity is 7 and its latest start time is 3.

Some extensions of the RCPSP include interruptible activities (if all activities are interruptible, the problem is called “preemptive”), resources whose capacity varies over time, consumable resources, and “elastic” activities. An activity is “elastic” when tradeoffs can be made between the duration and the amount of resources required by the activity, *e.g.*, when an activity can be performed either by 2 persons in 3 days, or by 3 persons in 2 days. In the most extreme case, the performance of the activity requires a given amount of “energy” (*e.g.*, 6 person-days) to be performed at any regular or irregular rate (*e.g.*, 4 persons on day 1 and 2 persons on day 2). In the following, we generally assume that the duration and the resource requirements of activities are fixed. When there is no ambiguity, we use p_A (resp. p_i) to denote the processing time of an activity A (resp. A_i), *i.e.*, p_A is the value of the variable $\text{duration}(A)$, and we use c_A (resp. c_i) to denote the capacity required by A (resp. A_i) for a given resource.

A common restriction of the RCPSP is encountered when all resources have capacity 1. In this case, any two activities A and B that require a common resource must be ordered: either A will execute before B , or B will execute before A . Such problems are called “disjunctive” scheduling problems. By extension, preemptive problems with resources of capacity 1 are called “disjunctive preemptive problems.” A substantial amount of work has been carried out on the application of constraint propagation techniques to disjunctive scheduling problems, *e.g.*, (Carlier and Pinson, 1990), (Nuijten, 1994), (Baptiste and Le Pape, 1995), (Caseau and Laburthe, 1995), (Colombani, 1996), with the result that constraint programming is now recognized as a method of choice for these problems.

Constraint programming algorithms have also been developed for “cumulative” scheduling problems, *i.e.*, problems like the RCPSP, such that several activities can use the same resource at the same time, *e.g.*, (Aggoun and Beldiceanu, 1993),

(Nuijten, 1994), (Caseau and Laburthe, 1996a). However, these algorithms tend to be less uniformly effective than their disjunctive counterparts. Different problems in the cumulative class seem to have different characteristics that make them either easy or hard to solve with a given technique.

The aim of this paper is to investigate one particular dimension along which problems differ. Within the cumulative scheduling class, we distinguish between *highly disjunctive* and *highly cumulative* problems: a scheduling problem is highly disjunctive when many pairs of activities cannot execute in parallel on the same resource; conversely, a scheduling problem is highly cumulative when many activities can execute in parallel on the same resource. To formalize this notion, we introduce the *disjunction ratio*, *i.e.*, the ratio between a lower bound of the number of pairs of activities which cannot execute in parallel and the overall number of pairs of distinct activities. A simple lower bound of the number of pairs of activities which cannot execute in parallel can be obtained by considering pairs $\{A, B\}$ such that either there is a chain of precedence constraints between A and B, or there is a resource constraint which is violated if A and B overlap in time. The disjunction ratio can be defined either globally (considering all the activities of a given problem instance) or for each resource R by limiting the pairs of activities to those that require at least one unit of R. The disjunction ratio of a disjunctive resource is equal to 1. The disjunctive ratio of a cumulative resource varies between 0 and 1, depending on the precedence constraints and on the amounts of capacity that are required to execute the activities. In particular, the ratio is equal to 0 when there is no precedence constraint and no activity requires more than half of the resource capacity.

Needless to say, the disjunction ratio is only one of a variety of indicators that could be associated with scheduling problem instances. For example, the *precedence ratio* (also known as *order strength* (Mastor, 1970), *flexibility ratio* (De Reyck and Herroelen, 1995), and *density* (De Reyck and Herroelen, 1995)), *i.e.*, the ratio between the number of pairs of activities which are ordered by precedence constraints and the overall number of pairs of distinct activities, is also important (a high precedence ratio makes the problem easier). Although some researchers, *e.g.*, (Kolisch *et al.*, 1995), have worked on such indicators, we believe much more work is necessary to discover which indicators are appropriate for designing, selecting, or adapting constraint programming techniques with respect to the characteristics of a given problem.

In the following, we explore the hypothesis that the disjunction ratio is an important indicator of which techniques shall be applied to a cumulative scheduling problem. Several new constraint propagation and problem decomposition techniques are introduced with this distinction in mind. This includes a simple quadratic “edge-finding” algorithm for cumulative resources, and a problem decomposition scheme dedicated to highly disjunctive project scheduling problems.

The paper is organized as follows: Section 2 presents our general approach to the resolution of the RCPSP; Section 3 presents the constraint propagation techniques we use; Section 4 introduces a heuristic algorithm for the generation of redundant disjunctive resource constraints; Section 5 presents dominance rules, which are used to dynamically decompose an instance of the RCPSP; Section 6 presents experimental results, which confirm that the techniques we use exhibit different behaviors on problems with different disjunction ratios. Consequently, in Section 7, we propose a refined version of the “edge-finding” algorithm for cumulative resources which performs very well on highly cumulative instances.

2 General Framework

The aim of this section is to present our general approach and establish a list (by no means exhaustive) of possible “ingredients” that can be incorporated in a constraint programming approach to the RCPSP. We limit the discussion to the standard RCPSP. However, some of the techniques we propose also apply to extensions of the RCPSP, such as problems with interruptible activities.

First, the RCPSP is an optimization problem. The goal is to determine a solution with minimal makespan and prove the optimality of the solution. We represent the makespan as an integer variable constrained to be greater than or equal to the end of any activity. Several strategies can be considered to minimize the value of that variable, *e.g.*, iterate on the possible values, either from the lower bound of its domain up to the upper bound (until one solution is found), or from the upper bound down to the lower bound (determining each time whether there still is a solution). In our experiments, a dichotomizing algorithm is used:

1. Compute an initial upper bound UB (the sum of the durations of all activities) and an initial lower bound LB (result of the propagation of the constraints) for the makespan.
2. Set $D = (LB + UB) / 2$.
3. Constrain the makespan to be lower than or equal to D. Solve the resulting constraint satisfaction problem, *i.e.*, determine a solution with makespan at most D or prove that no such solution exists. If a solution is found, set UB to the makespan of the solution; otherwise, set LB to $D + 1$.
4. Iterate steps 2 and 3 until $UB = LB$.

A branching procedure with constraint propagation at each node of the search tree is used to determine whether the problem with makespan at most D accepts a solution. As shown in the literature, there are many possible choices regarding the amount of constraint propagation that can be made at each node. Carlier and Latapie (1991), as well as Demeulemeester and Herroelen (1992), use simple bounding techniques compared to the more complex constraint propagation algorithms proposed by Nuijten (1994) or Caseau and Laburthe (1996a). Performing more constraint propagation serves two purposes: first, detect that a partial solution at a given node cannot be extended into a complete solution with makespan $\leq D$; second, reduce the domains of the start and end variables, thereby providing useful information on which variables are the most constrained. However, complex constraint propagation algorithms take time to execute,

so the cost of these algorithms may not always be balanced by the subsequent reduction of search. Section 3 introduces a new quadratic edge-finding algorithm for propagating cumulative resource constraints. Experimental results will show that it is worth using this algorithm when the disjunction ratio is low.

Artificially adding “redundant” constraints, *i.e.*, constraints that do not change the set of solutions, but propagate in a different way, is another method for improving the effectiveness of constraint propagation. For example, Carlier and Latapie (1991) and Carlier and Néron (1996) present branch-and-bound algorithms for the RCPSP which rely on the generation of redundant resource constraints. If S is a set of activities and m an integer value, and if for any subset s of S such that $|s| > m$, the activities of s cannot all overlap, then the following resource constraint can be added: “Each activity of S requires exactly one unit of a new (artificial) resource of capacity m ”. Several lower-bounding techniques have been developed for this resource constraint (Perregaard, 1995), (Carlier and Pinson, 1996). These techniques serve to update the minimal value of the makespan variable, but do not update the domains of the start and end time variables. Section 4 proposes the generation of artificial *disjunctive* resource constraints, for which standard disjunctive resource constraint propagation algorithms can be applied, resulting in a powerful update of earliest and latest start and end times. In particular, all the activities requiring more than half of the capacity of a given resource are known to be *in disjunction*, which allows more effective constraint propagation to take place.

Besides constraint propagation, a branching solution search procedure is also characterized by:

- *the types of decisions that are made at each node.* Most search procedures for the RCPSP chronologically build a schedule, from time 0 to time D . At a given time t , Demeulemeester and Herroelen (1992) schedule a subset of the available activities; other subsets are tried upon backtracking. The main interest of this strategy is that some resource can be maximally used at time t , prior to proceed to a time $t' > t$. However, there may be many subsets to try upon backtracking, especially if the problem is highly cumulative. Caseau and Laburthe (1996a) schedule a single activity and postpone it upon backtracking. The depth of the search tree increases, but each (smaller) decision is propagated prior to the making of the next decision. An example of a non-chronological scheduling strategy is given by Carlier and Latapie (1991). Their strategy is based on dichotomizing the domains of the start variables: at each node, the lower or the upper half of the domain of a chosen variable V is removed and the decision is propagated. This strategy may work well if there are good reasons for selecting the variable V , rather than another one (*e.g.*, when there is a clear resource bottleneck at a given time).
- *the heuristics that are used to select which possibilities to explore first.* When a chronological strategy is used, one can either try to “fill” the resources at time t (to avoid the insertion of resource idle time in the schedule) or select the most urgent activities among those that are available at time t . When a non-chronological strategy is used, the best is to focus first on identified bottlenecks.
- *the dominance rules that are applied to eliminate unpromising branches.* Several dominance rules have been developed for the RCPSP (see, for example, (Demeulemeester and Herroelen, 1992)). A dominance rule establishes that at least one optimal solution to the considered problem satisfies a given property.

Dominance rules enable a reduction of the search to a limited number of nodes, which satisfy the dominance properties. Section 5 proposes a new dominance rule that generalizes the “single incompatibility rule” of Demeulemeester and Herroelen. When it is applied, this rule leads to a decomposition of the remaining problem. As for constraint propagation, dynamically applying complex dominance rules at each node of the search tree may prove more costly than beneficial. Our generalization of the “single incompatibility rule” is worth using when the disjunctive ratio is high.

- *the backtracking strategy that is applied upon failure.* Most constraint programming tools rely on depth-first chronological backtracking. However, “intelligent” backtracking strategies can also be applied to the RCPSP. For example, the cut-set dominance rule of Demeulemeester and Herroelen (1992) can be seen as an intelligent backtracking strategy, which consists of memorizing search states to avoid doing the same work twice. When backtracking, the remaining sub-problem is saved. In the remainder of the search tree, the algorithm checks if the remaining sub-problem is not already proved unfeasible. The advantage of such techniques is that the identified impossible problem-solving situations are not encountered twice (or are immediately recognized as impossible). However, such techniques may require large amounts of memory to store the intermediate search results and, in some cases, significant time for their application.

Our overall research agenda is to look at all these aspects of the problem-solving strategy and determine (if at all possible) when to apply each technique. As a first step, we designed some of the constraint propagation techniques and dominance rules mentioned above with the intent of applying them either to highly disjunctive or to highly cumulative problems. For this reason, we decided to fix the types of decisions to be made at each node, the heuristics that are used to select which possibilities to explore first, and the backtracking strategy (depth-first chronological backtracking). Our search procedure slightly differs from the one proposed by Caseau and Laburthe (1996a):

1. Select an unscheduled activity A of minimal earliest start time. When several activities have the same earliest start time, select one of the most urgent, *i.e.*, one with minimal latest start time. Create a choice point.
2. Left branch: Schedule A from its earliest start time EST_A to its earliest end time EET_A (in other terms, set $start(A)$ to the smallest value in its domain). Propagate this decision. Apply the dominance rules. Goto step 1.
3. Right branch: If step 2 causes a backtrack, compute the set S of activities that could overlap the interval $[EST_A, EET_A]$ (according to current variable domains). Post a delaying constraint: “ A executes after at least one activity in S ”. Propagate this constraint. Apply the dominance rules. Goto step 1.
4. If both branches fail, provoke a backtrack to the preceding choice point (chronological backtracking).

This algorithm stops when all activities are scheduled (in step 1) or when all branches have been explored (no more preceding choice point in step 4).

Two points of flexibility remain in this procedure. The first concerns constraint propagation. As shown in Sections 3 and 4, several constraint propagation algorithms can be associated with each resource. One of these algorithms, based on a timetable mechanism, is systematically applied. It guarantees that, at the end of the propagation,

the earliest start time of each unscheduled activity is consistent with the start and end times of all the scheduled activities (*i.e.*, activities with bound start and end times). This, in turn, guarantees the correctness of the overall search procedure: adding the constraint “A executes after at least one activity in S” upon backtracking is correct, because if A could start before the end of all activities in S, then A could start at the earliest start time EST_A resulting from previous constraint propagation. The second point of flexibility concerns the dominance rules. Several dominance rules can be applied, which may lead to some decomposition of the problem (cf. Section 5).

3 Constraint Propagation Algorithms

Our implementation is based on *CLAIRE SCHEDULE* (Le Pape and Baptiste, 1997), a constraint-based library for preemptive and non-preemptive scheduling, itself implemented in *CLAIRE* (Caseau and Laburthe, 1996b), a high-level functional and object-oriented language. The aim of this section is to review the constraint propagation techniques we use in the context of the RCPSP.

The constraints of the RCPSP and the decisions made in the general framework of Section 2 are of the following types:

1. $0 \leq \text{start}(A)$, for every activity A;
2. $\text{start}(A) + \text{duration}(A) = \text{end}(A)$, for every activity A;
3. $\text{end}(A) \leq \text{makespan}$, for every activity A;
4. $\text{end}(A) \leq \text{start}(B)$, for every precedence constraint ($A \rightarrow B$);
5. $\sum_{A \text{ such that } \text{start}(A) \leq t < \text{end}(A)} \text{required-capacity}(A, R) \leq \text{capacity}(R)$, for every resource R and time t (cumulative constraint);
6. $\text{makespan} \leq D$;
7. $\text{start}(A) = s$, where A is an activity and s a value in its domain;
8. “A executes after at least one activity in S” *i.e.*, $\min_S(\text{end}(S)) \leq \text{start}(A)$, where A is an activity and S a set of activities.

Constraints 1, 2, 3, and 6, guarantee that each variable in the problem has a finite domain (since we use integer variables). The initial domain of each variable is set to $[0, D]$. As often in constraint programming, unary constraints (1, 6, 7) are propagated by reducing the domains of the corresponding variables.

Duration constraints (2) and precedence constraints (3, 4) are propagated using a standard arc-B-consistency algorithm (Lhomme, 1993).¹ In addition, a variant of Ford's algorithm due to Cesta and Oddi (1996) is used to detect any inconsistency between precedence and duration constraints, in time polynomial in the number of constraints (and independent of the domain sizes).

The constraint “A executes after at least one activity in S” (8) is propagated as follows: compute $\min_{B \in S} EET_B$, the minimal earliest end time of all activities in S, and update EST_A to $\max(EST_A, \min_{B \in S} EET_B)$. Moreover, when there is only one activity B in S that can end before LST_A , then LET_B is set to $\min(LET_B, LST_A)$.

For the resource constraints (5), *CLAIRE SCHEDULE* includes both a timetable mechanism and an edge-finding algorithm.

The timetable mechanism is an extension of the timetable mechanism of ILOG SCHEDULER (Le Pape, 1994), which supports both non-interruptible and interruptible activities, as well as activities requiring an amount of resource capacity that can vary over time (“elastic” activities). The timetable of a resource is a data structure that represents the amount of resource capacity that is and can be used over time. Propagation occurs both from activities to resources, and from resources to activities.

- From activities to resources

When an activity is known to execute at time t , this activity requires its resources at time t . The minimal value of the required capacities can consequently be used to update the resource timetables. Similarly, constraint propagation occurs when the minimal value of the required capacity changes: the contribution of the activity to the resource timetable increases with the required capacity.

- From resources to activities

The resource timetable is explored forward from the earliest start time of the activity. Information in the timetable is used to determine how much capacity $c(t)$ can be allocated to the activity at time t . The exploration stops when a time u is reached such that (a) $u + 1$ is greater than or equal to the earliest end time of the activity and (b) the sum of $c(t)$ from the earliest start time of the activity up to u equals or exceeds the minimal amount of energy required for the activity. The earliest end time of the activity is consequently updated (it becomes $u + 1$). A similar exploration is also performed backward from the latest end time of the activity. Needless to say, if the activity cannot start before (or end after) time t , or if it cannot be interrupted and cannot execute at time t , its earliest start time (if going forward) or its latest end time (if going backward) is updated, and the exploration restarts from the new value.

More details are available in (Le Pape and Baptiste, 1997). Globally, this constraint propagation algorithm guarantees that, at the end of the propagation, the earliest and the latest start and end times of each unscheduled activity are consistent with the start and end times of all the scheduled activities (*i.e.*, in the non-preemptive case, activities with bound start and end times).

The edge-finding algorithm² is an extension of classical *disjunctive* edge-finding bounding techniques (Carlier and Pinson, 1990), (Applegate and Cook, 1991), (Nuijten, 1994), (Baptiste and Le Pape, 1995), (Caseau and Laburthe, 1995). This extension supports both non-interruptible and interruptible activities. In the simplest case of non-preemptive scheduling and resources of capacity 1, edge-finding consists of determining whether an activity A can, cannot, or must, execute before (or after) a set of activities Ω which require the same resource. Two types of conclusions can then be drawn: new ordering relations (“edges” in the graph representing the possible orderings of activities) and new time-bounds, *i.e.*, strengthened earliest and latest start and end times of activities. Preemptive scheduling is more complex since activities can preempt one another. Then edge-finding consists of determining whether an activity A can, cannot, or must, start or end before (or after) a set of activities Ω . For a resource of capacity 1, the edge-finding algorithm of CLAIRE SCHEDULE computes for each activity A :

- when A is not interruptible: the earliest time at which A could start and the latest time at which A could end, if all the other activities were interruptible;

- when A is interruptible: the earliest time at which A could end and the latest time at which A could start, if all the other activities were interruptible.

This algorithm is detailed in (Le Pape and Baptiste, 1996). It requires quadratic time and linear space. Compared to the timetable mechanism, it is time-consuming. However, this is usually balanced by a drastic reduction of the search space.

To deal with highly cumulative problems, we developed an extension of this algorithm to the cumulative case. This extension supports both non-interruptible and interruptible activities. It relies on a very simple idea which consists of reducing the cumulative resource to a resource of capacity 1 thanks to the following transformation.

TRANSFORMATION: Consider a set $\{A_1, \dots, A_n\}$ of n activities requiring a resource R of capacity C and let $EST_i, LST_i, EET_i, LET_i, p_i, c_i$ be respectively the earliest start time, latest start time, earliest end time, latest end time, duration, and required capacity (i.e., required amount of resource R) of A_i . Let R' be a resource of capacity 1 and for each activity A_i , let A_i' be an interruptible activity with $EST_i' = C * EST_i$, $LET_i' = C * LET_i$, $p_i' = p_i * c_i$ and $c_i' = 1$.

PROPOSITION 1: If there exists a schedule of A_1, \dots, A_n on R then there exists a schedule of A_1', \dots, A_n' on R' .

Proof : Let $S(A_i, t)$ be the number of units of A_i executed at t on R ($S(A_i, t) = 0$ or $S(A_i, t) = c_i$). A schedule of A_1', \dots, A_n' on R' can be built as follows: for each activity A_i (taken in any order) and each time t , schedule $S(A_i, t)$ units of A_i' on R' as early as possible after time $C * t$. For any activity A_i and time t , the number of units of A_i executed at t on R is equal to the number of units of A_i' executed between $C * t$ and $C * (t + 1)$ on R' since this algorithm consists of cutting the schedule of A_1, \dots, A_n into slices of one unit and of rescheduling these slices on R' . Consequently, for any activity A_i' , exactly $p_i * c_i$ units of A_i' are scheduled between $C * EST_i$ and $C * LET_i$ and thus the release dates and the due dates are met. \square

The same transformation can be used to update time-bounds (earliest and latest start and end times). This leads to a simple edge-finding algorithm for the cumulative case:

1. Apply the transformation;
2. Update the earliest and latest start and end times of A_1', \dots, A_n' thanks to the disjunctive edge-finding algorithm;
3. Update the four time-bounds of each A_i : $EST_i = \lfloor EST_i' / C \rfloor$, $LST_i = \lfloor LST_i' / C \rfloor$, $EET_i = \lceil EET_i' / C \rceil$ and $LET_i = \lceil LET_i' / C \rceil$.

This algorithm runs in a quadratic number of steps since the transformation (step 1) is a linear operation, the mixed edge-finding algorithm (step 2) runs in $O(n^2)$, and step 3 is linear. It can compute less precise time-bounds than some other cumulative edge-finding algorithms (Nuijten, 1994), (Caseau and Laburthe, 1996a). However, it has two advantages: (1) it can be applied to interruptible activities and to any sort of elastic activities; (2) it corresponds to a precise relaxation of the resource constraint. Indeed, it is shown in (Baptiste *et al.*, 1998) that when all activities are “fully elastic” (i.e., can be assigned any amount of capacity from 0 to C at any point in time), this algorithm computes the best possible time-bounds, i.e., if only the resource constraint is considered, each of the obtained earliest and latest start and end times can be reached by

some “fully elastic” schedule. For this reason, we call this algorithm the “fully elastic edge-finding algorithm”.

As usual in a constraint propagation scheme, the algorithm may have to be applied several times during the propagation of the same decisions. For example, let A, B, C, D be four non-interruptible activities requiring a resource of capacity 2. The tables below provide the temporal characteristics of the cumulative problem and of the corresponding preemptive disjunctive problem.

	EST	LET	p	c
A	0	30	10	1
B	1	10	4	1
C	1	10	4	1
D	1	10	4	1

	EST	LET	p
A'	0	60	10
B'	2	20	4
C'	2	20	4
D'	2	20	4

It is easy to see that A' cannot end before time 22. Indeed, if A' ends before 22 then either B', C' or D' must end after 20; which is impossible. The earliest end time of A can then be set to $\lceil 22 / 2 \rceil = 11$. This wakes up the duration constraint: since p_A is 10, the earliest start time of A is set to 1. Let us apply again the same algorithm:

	EST	LET	p	c
A	1	30	10	1
B	1	10	4	1
C	1	10	4	1
D	1	10	4	1

	EST	LET	p
A'	2	60	10
B'	2	20	4
C'	2	20	4
D'	2	20	4

We can now verify that activity A' cannot end before time 24; thus, the earliest end time of A can be set to 12, and its earliest start time to 2.

4 Redundant Disjunctive Resource Constraints

Since some project scheduling problems are highly disjunctive, we consider the generation of redundant disjunctive resource constraints as a means to strengthen constraint propagation (see also (Brucker *et al.*, 1997)). The basic idea is simple: if a set S of activities is such that no two activities in S can execute in parallel, a new (artificial) resource of capacity 1 can be created, and all the activities in S can be constrained to require the new resource. The disjunctive edge-finding constraint propagation algorithm can then be applied to the new resource, in order to guarantee a better update of the earliest and latest start and end times of these activities.

To detect the relevant sets S, we use an incompatibility graph $G = (X, E)$ where X is a set of vertices corresponding to the activities of the RCPSP and E is a set of edges (A, B), such that $(A, B) \in E$ if and only if A and B are not compatible (*i.e.*, cannot execute in parallel). We distinguish three subsets E_{cap} , E_{prec} , and E_{time} of E. These subsets denote respectively the incompatibilities due to resource capacity constraints, to precedence constraints, and to time-bounds.

- $(A, B) \in E_{\text{cap}}$ if and only if there is a resource R such that the sum of the capacities required by A and B on R is greater than the overall capacity of R .
- $(A, B) \in E_{\text{prec}}$ if and only if there is a precedence constraint between A and B (the transitive closure³ of the precedence graph is computed for this purpose).
- $(A, B) \in E_{\text{time}}$ if and only if either $LET_A \leq EST_B$ or $LET_B \leq EST_A$.

Any clique⁴ of the incompatibility graph is a candidate disjunctive resource constraint. We remark that two activities A and B with $(A, B) \in E_{\text{prec}}$ or $(A, B) \in E_{\text{time}}$ are already ordered. Nevertheless, including A and B in the same clique can be useful, *e.g.*, when another activity C with $(A, C) \in E_{\text{cap}}$ and $(B, C) \in E_{\text{cap}}$ belongs to the clique. Indeed, “ A , B and C are in disjunction” is stronger than “ A and C are in disjunction and B and C are in disjunction”.

However, since the edge-finding constraint propagation algorithm is costly in terms of CPU time, very few redundant disjunctive constraints can be generated. Hence, we have to heuristically select some of these cliques (otherwise, the cost of the disjunctive resource constraint propagation would be too high to be compensated by the subsequent reduction of search). Since the problem of finding a maximal clique (*i.e.*, a clique of maximal size) is NP-hard (Garey and Johnson, 1979), we use a simple heuristic which increases step by step the current clique C : among the activities which are incompatible with all activities of the current clique, we select one of maximal duration. Our hope is that the resulting constraint will be tight since several activities with large processing times will require the same disjunctive resource.

In our first experiments, we built one disjunctive resource per cumulative resource plus one more “global” disjunctive resource. For each cumulative resource, we arbitrarily put in the clique all the activities requiring more than half of the resource and the clique was completed thanks to the heuristic described above. The extra disjunctive resource was fully generated according to the heuristic rule. A careful examination of the generated problems showed that many activities were added in the cliques because of precedence and time-bound constraints. It is far more interesting to generate disjunctive problems where most of the activities are incompatible because of resources. To achieve this, the generation heuristic has been split in two different procedures: (1) build a maximal clique C_{cap} of (X, E_{cap}) ; (2) extend C_{cap} to a maximal clique C of G .

The overall complexity of building the cliques is $O(n^3 + n^2m)$ where m is the number of resources in the problem instance. Indeed, building the transitive closure of the precedence graph requires $O(n^3)$, building the incompatibility graph from the transitive closure of the precedence graph requires either $O(n^2m)$ or $O(n^3 + n^2m)$ (depending on the encoding of the graph) and building each clique requires $O(n^2)$ (with the help of a counter for each node, counting the number of neighbors of the node already in the clique).

Example 1: Let A, B, C, D be four activities requiring respectively 3, 2, 1 and 1 units of a resource of capacity 4. These activities last respectively 3, 6, 3 and 8 units of time. Moreover, there are 4 precedence constraints ($A \rightarrow C$), ($A \rightarrow D$), ($B \rightarrow C$), and ($B \rightarrow D$). Let us build the incompatibility graph of this instance (see Figure 2). First, we add the edges corresponding to the precedence constraints (dotted lines). Then we consider each pair of activities and add an edge (solid line) between the corresponding vertices if and only if the sum of the resource requirements of both

activities exceed 4. In this example, there are two maximal cliques: $\{A, B, C\}$ and $\{A, B, D\}$. Our algorithm starts with $\{A\}$ and successively adds B (based on C_{cap}) and D (which is longer than C) to the clique.

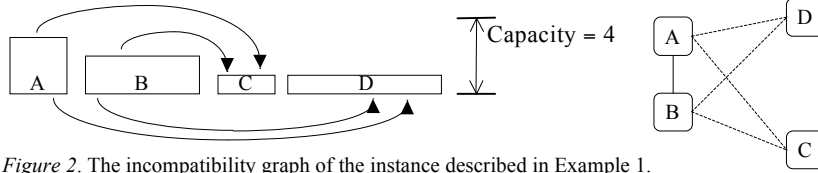


Figure 2. The incompatibility graph of the instance described in Example 1.

5 Dominance Rules

Our search procedure incorporates several dominance rules. Each of them consists of expressing additional constraints which do not impact the existence of a solution schedule (if there exists a schedule satisfying all constraints posted so far, then at least one such schedule also satisfies the additional constraints).

Immediate scheduling rule

Let A be an unscheduled activity of minimal earliest end time. Let O be the set of activities which can be “partially” scheduled in the interval $[EST_A, EET_A]$, i.e., $O = \{X \mid LET_X > EST_A \text{ and } EET_A > EST_X\}$.

PROPOSITION 2: *If all activities in O can be scheduled in parallel, i.e., on each resource, the amount required to execute all activities in O is lower than or equal to the resource capacity, then A can be scheduled at EST_A .*

Proof: Suppose that there is a schedule S that satisfies all the constraints posted so far. Let us examine S . All predecessors of A have been scheduled before EST_A since the earliest end time of A is minimal. Moreover, there is enough space on each resource to schedule A at EST_A since all activities in O can execute in parallel. S can thus be modified by bringing A back to EST_A . \square

Each application of this rule requires $O(nm)$ where n is the number of activities and m the number of resources.

Single incompatibility rule (Demeulemeester and Herroelen, 1992)

Let t_{min} be the minimal earliest start time among the earliest start times of unscheduled activities.

PROPOSITION 3: *If no activity is in progress at time t_{min} and if there is an activity A , available at t_{min} , such that A cannot be scheduled together with any other unscheduled activity at any time instant without violating precedence or resource constraints, then activity A can be scheduled at time t_{min} .*

Proof: see (Demeulemeester and Herroelen, 1992). \square

Each application of this rule requires $O(n^2)$, using the incompatibility graph of the previous section.

Incompatible set decomposition rule

We propose an extension of the single incompatibility rule based on a directed compatibility graph $\Gamma = (X, U)$, built from the undirected incompatibility graph $G = (X, E)$ of Section 4. Let t_{\min} and t_{\max} be respectively the minimal earliest start time and the maximal latest end time among unscheduled activities. X is restricted to the set of vertices corresponding to the activities A such that $t_{\min} < EET_A$ and $LST_A < t_{\max}$. U is the set of directed arcs such that $(A, B) \in U$ if and only if either $(A, B) \notin E$ (i.e., A and B are not incompatible as defined in Section 4) or A must precede B in the transitive closure of the precedence network. Let X_1, X_2, \dots, X_m be the strongly connected components of Γ , i.e., $\{X_1, X_2, \dots, X_m\}$ is a partition of X such that any two activities A and B belong to the same X_i if and only if there is a directed path from A to B and from B to A . Let γ be the quotient graph of Γ (the “strongly connected” relation is an equivalence relation; the quotient graph is obtained by identifying all the nodes in the same equivalence class). γ is a directed acyclic graph and thus the strongly connected components can be totally ordered. We suppose without any loss of generality that this total order is X_1, X_2, \dots, X_m .

PROPOSITION 4: *If there exists a schedule satisfying all the constraints posted so far, then one such schedule of minimal makespan is such that for all i in $[1, m-1]$, all activities in X_i end before all activities in X_{i+1} start.*

Proof: We only prove that all activities in X_1 can be scheduled before all activities in $X - X_1$. The remaining part of the proof can be achieved by induction. Suppose that there exists a schedule satisfying all constraints posted so far. Let S be such a schedule, such that the first time point t_A at which an activity A of X_1 is scheduled after an activity of $X - X_1$ is minimal. Let $t_{A'}$ be the first time after t_A such that no activity of X_1 is scheduled at $t_{A'}$. Let t_B be the minimal start time among start times of activities in $X - X_1$ in S . Let us modify S into S' by exchanging the schedule blocks $[t_B, t_A]$ and $[t_A, t_{A'}]$ (cf. Figure 3). The schedule S' satisfies precedence constraints, otherwise X_1 would not be the first strongly connected component. The resource constraints are also satisfied. Moreover, the activities are not interrupted since at times t_B, t_A and $t_{A'}$, there is no activity in progress on S (otherwise two activities in different components would be compatible, which contradicts our hypothesis). Thus, schedule S' is a solution and contradicts the hypothesis that t_A exists and is minimal. \square

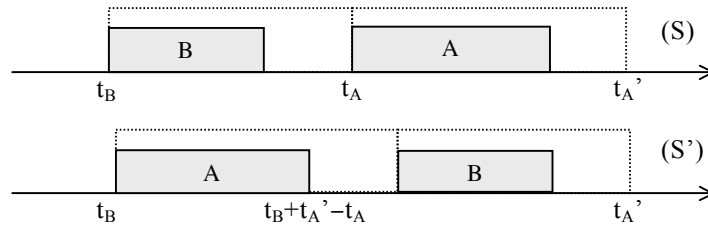


Figure 3. The relative positions of A and B

Ordering the subsets X_1, \dots, X_m is interesting for two reasons. First, additional precedence constraints can be added; which results in stronger constraint propagation. Second, the problem can be decomposed into m optimization problems. Indeed, since

subsets X_1, \dots, X_m are ordered, it is sufficient to find the optimal solutions to the RCPSP restricted to each X_i .

The overall algorithm which implements this incompatible set dominance rule runs in $O(n^2)$ since there are potentially $O(n)$ vertices in X and thus, building the set U requires at most a quadratic number of steps (we assume the transitive closure of the initial precedence graph has been computed once and for all). Moreover, searching for the strongly connected components of Γ can be done in $O(|U|)$, using the depth first algorithm of Tarjan (Gondran and Minoux, 1984).

Example 2: Let A, B, C, D, E, F be 6 activities requiring respectively 2, 3, 1, 2, 1 and 2 units of a resource of capacity 4 (cf. Figure 4). Let us suppose that the following precedence constraints apply: $(A \rightarrow D)$, $(A \rightarrow E)$, $(B \rightarrow E)$, $(C \rightarrow D)$, $(C \rightarrow E)$, and $(E \rightarrow F)$. To simplify the example, we do not consider the time-bounds of activities and thus, the values of the durations are not necessary for the example.

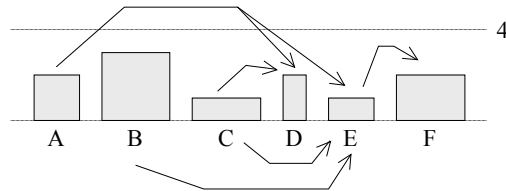


Figure 4. A simple instance of the RCPSP

The transitive closure of the precedence network consists of adding arcs $(A \rightarrow F)$, $(B \rightarrow F)$ and $(C \rightarrow F)$. The pairs of activities which are incompatible because of resource constraints are (A, B) , (B, D) , and (B, F) . Consequently, the pairs of activities which are not incompatible are (A, C) , (B, C) , (D, E) , (D, F) ; which corresponds to the graph depicted on Figure 5. There are two strongly connected components $\{A, B, C\}$ and $\{D, E, F\}$. Our dominance rule states that there exists an optimal solution in which $\{A, B, C\}$ is scheduled before $\{D, E, F\}$.

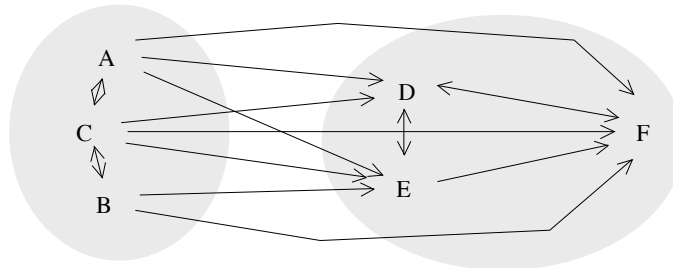


Figure 5. The directed graph associated to activities A, B, C, D, E and F.

6 Computational Results

The following tables provide the results obtained on different sets of benchmarks with four different versions of our search procedure: with or without Fully Elastic cumulative

edge-finding (“FE” or “NO” in column E.F.), with or without incompatible set decomposition (“YES” or “NO” in column Dec.). All versions of the algorithm use precedence constraint propagation, resource constraint propagation based on timetables, edge-finding on redundant disjunctive resource constraints, the immediate scheduling rule, the single incompatibility rule, and their symmetric counterparts. In each of the tables, column "Solved" denotes the number of instances solved to optimality (optimality proof included) within a limit of 4000 backtracks. Column "BT" provides the average number of backtracks over those problems that have been solved by all algorithms. Column "CPU" provides the corresponding average CPU time, in seconds on a Pentium PC running at 133 MHz. Table 1 provides the results obtained on the highly disjunctive Patterson problem set (problems with 14 to 51 activities) (Patterson, 1984). These results compare well to other constraint programming approaches. For example, Caseau and Laburthe (1996a) solve the overall Patterson set in an average of 1000 backtracks and 3.5 seconds. Our algorithm requires approximately the same CPU time, but a much smaller number of backtracks. Using the cumulative edge-finder and the incompatible set decomposition rule on this set decreases the average number of backtracks needed to solve the problem to optimality. However, the cost of these techniques is such that the overall CPU time increases.

We also applied the four algorithms to the 480 instances of Kolisch, Sprecher, and Drexel (1995) (KSD, 30 activities each). These instances are interesting because they are classified according to various indicators, including the “resource strength,” *i.e.*, the resource capacity, normalized so that the “strength” is 0 when for each resource R , $\text{capacity}(R) = \max_A(\text{required-capacity}(A, R))$, and the “strength” is 1 when scheduling each activity at its earliest start time (ignoring resource constraints) results in a schedule that satisfies resource constraints as well. Table 2 provides the results for the 120 instances of resource strength (RS) 0.2, Table 3 provides the results for the 120 instances of resource strength 0.7, and Table 4 provides the overall results. Clearly, the decomposition rule is very useful for the highly disjunctive problems. Considering the overall set, the decomposition rule allows the resolution of 14 additional instances, 13 of which are in the most highly disjunctive set. Unfortunately, the instances of resource strength 0.7 are “easy”, so for this subset the interest of the more complex techniques does not appear.

Table 6 provides the average precedence ratio (cf. Section 1), disjunctive ratio, and resource strength, and their standard deviations on the different problem sets. It clearly appears that even KSD instances with high resource strength have large disjunction ratios (0.53) due to large precedence ratios. For experimental purposes, this led us to generate a new series of 40 highly cumulative problems (the BL set). More precisely, we generated 80 instances with 3 resources, and either 20 or 25 activities, and we kept the 40 most difficult of these instances. Each activity requires the 3 resources, with a required capacity randomly chosen between 0 and 60% of the resource capacity. 15 precedence constraints were randomly generated for problems with 20 activities; 45 precedence constraints were generated for problems with 25 activities. This simple parameter setting allowed us to generate problems with average precedence and disjunctive ratios of 0.33, with a standard deviation of 0.07, smaller than the standard deviation observed on the classical benchmarks from the literature, and a relatively low resource strength (0.34 on average). Table 5 provides the results. It clearly shows that

the cumulative edge-finder is a crucial technique for solving these instances. However, one may wonder whether the versions with no cumulative edge-finding could “catch up” if given more CPU time. To evaluate that, we ran the BL instances again with a limit of 20000 backtracks. This led the versions with no cumulative edge-finding to solve only 4 additional instances in an average of 8173 backtracks and 146.7 seconds. With cumulative edge-finding, these 4 instances are solved in an average of 994 backtracks and 23.8 seconds.

Globally, these results show that highly disjunctive and highly cumulative problems require different types of constraint propagation and problem decomposition techniques.

Table 1. Patterson (110 instances of average disjunctive ratio 0.67)

E.F.	Dec.	Solved	BT	CPU
NO	NO	110	77	2.68
NO	YES	110	71	3.75
FE	NO	110	63	3.67
FE	YES	110	58	4.65

Table 3. KSD RS 0.7 (120 instances of average disjunctive ratio 0.53)

E.F.	Dec.	Solved	BT	CPU
NO	NO	119	101	4.85
NO	YES	119	101	7.33
FE	NO	119	100	7.96
FE	YES	119	100	10.64

Table 5. BL (40 instances of average disjunctive ratio 0.33)

E.F.	Dec.	Solved	BT	CPU
NO	NO	4	1241	29.5
NO	YES	4	1241	47.0
FE	NO	28	407	13.9
FE	YES	28	407	20.1

Table 2. KSD RS 0.2 (120 instances of average disjunctive ratio 0.65)

E.F.	Dec.	Solved	BT	CPU
NO	NO	51	369	12.52
NO	YES	64	253	11.70
FE	NO	51	366	17.79
FE	YES	64	251	14.82

Table 4. KSD ALL (480 instances of average disjunctive ratio 0.56)

E.F.	Dec.	Solved	BT	CPU
NO	NO	388	121	5.19
NO	YES	402	105	7.03
FE	NO	389	119	7.88
FE	YES	403	104	9.56

Table 6. Average ratios and standard deviations for different problem sets

	Precedence ratio		Disjunction ratio		Resource strength	
	Average	Std	Average	Std	Average	Std
Patterson	0.64	0.10	0.67	0.11	0.50	0.21
KSD RS 0.2	0.52	0.09	0.65	0.11	0.20	0.02
KSD RS 0.5	0.52	0.09	0.53	0.09	0.52	0.03
KSD RS 0.7	0.52	0.08	0.53	0.08	0.70	0.03
KSD RS 1.0	0.52	0.09	0.52	0.09	1.00	0.00
BL	0.33	0.07	0.33	0.07	0.34	0.09

7 Further Work on Highly Cumulative Project Scheduling Problems

Because there are some highly cumulative industrial problems that are difficult to solve, we decided to pursue our efforts on highly cumulative instances. In this section, we will present the first results we obtained by designing and using a more complex but more powerful deduction scheme (not yet available as part of CLAIRE SCHEDULE).

The “left-shift/right-shift” scheme is based on energetic reasoning (Lopez *et al.*, 1992). The basic idea is to compare over a given time interval, the amount of resource required by activities to the amount of resource available. More precisely, given an activity A_i and a time interval $[t_1 t_2]$, $W_{Sh}(A_i, t_1, t_2)$, the “left-shift/right-shift” required energy consumption of A_i over $[t_1 t_2]$, is c_i times the minimum of the three following durations:

- $t_2 - t_1$, the length of the interval;
- $p_i^+(t_1) = \max(0, p_i - \max(0, t_1 - EST_i))$, the number of time units during which A_i executes after time t_1 if A_i is left-shifted, *i.e.*, scheduled as soon as possible;
- $p_i^-(t_2) = \max(0, p_i - \max(0, LET_i - t_2))$, the number of time units during which A_i executes before time t_2 if A_i is right-shifted, *i.e.*, scheduled as late as possible.

This leads to $W_{Sh}(A_i, t_1, t_2) = c_i * \min(t_2 - t_1, p_i^+(t_1), p_i^-(t_2))$ (see Figure 6 for an example).

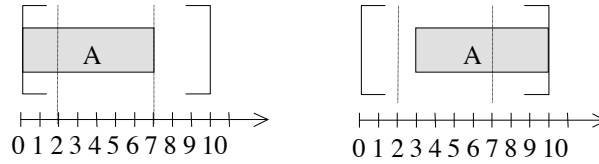


Figure 6. The required energy consumption of an activity A ($EST_i = 0, LET_i = 7$ and $c = 2$) over $[t_1 t_2] = [2 7]$. At least 4 time units of A have to be executed in $[2 7]$; which corresponds to the formula $W_{Sh}(A, 2, 7) = 2 * \min(5, 5, 4) = 8$.

We can now define the left-shift/right-shift overall required energy consumption $W_{Sh}(t_1, t_2)$ over an interval $[t_1 t_2]$ as the sum over all activities A_i of $W_{Sh}(A_i, t_1, t_2)$. Obviously, $W_{Sh}(t_1, t_2)$ must be lower than or equal to $C * (t_2 - t_1)$. An important issue is to characterize the time points t_1 and t_2 over which such a test must be performed. It is shown in (Baptiste *et al.*, 1998) that there are $O(n^2)$ relevant intervals $[t_1 t_2]$: if, for each of these $O(n^2)$ intervals, the required energy is lower than or equal to the available energy, then this is also true for any other interval. However, the overall set of $O(n^2)$ intervals is not the Cartesian product of $O(n)$ time points t_1 with $O(n)$ time points t_2 . Even though the relevant tests can still be made in $O(n^2)$ (Baptiste *et al.*, 1998), we decided to implement a simple algorithm which performs the analysis on the Cartesian product of $\{EST_i, 1 \leq i \leq n\} \cup \{LST_i, 1 \leq i \leq n\}$ and $\{LET_i, 1 \leq i \leq n\} \cup \{EET_i, 1 \leq i \leq n\}$.

The values of W_{Sh} can also be used to adjust time-bounds. Given an activity A_i and a time interval $[t_1 t_2]$ with $t_2 < d_i$, we examine whether A_i can end before t_2 .

PROPOSITION 5:

If $\exists t_1$ such that $t_1 < t_2$ and $W_{Sh}(t_1, t_2) - W_{Sh}(A_i, t_1, t_2) + c_i * p_i^+(t_1) > C * (t_2 - t_1)$ then a valid lower bound of the end time of A_i is:

$$t_2 + (W_{Sh}(t_1, t_2) - W_{Sh}(A_i, t_1, t_2) + c_i * p_i^+(t_1) - C * (t_2 - t_1)) / c_i$$

Proof: The rationale for this adjustment is that the numerator is the number of energy units of A_i which have to be shifted after time t_2 . We can divide this number of units by

the amount of resource required by A_i to obtain a lower bound of the duration required to execute these units. \square

It can easily be shown that this proposition, applied to all intervals $[t_1 t_2]$ with t_1 in $\{\text{EST}_i, 1 \leq i \leq n\} \cup \{\text{LST}_i, 1 \leq i \leq n\}$ and t_2 in $\{\text{LET}_i, 1 \leq i \leq n\} \cup \{\text{EET}_i, 1 \leq i \leq n\}$, results in tighter adjustments of the earliest and latest start and end times than the fully elastic edge-finding algorithm described in Section 3. However, the best algorithm we found so far for performing these adjustments runs in $O(n^3)$. This algorithm is very simple. Indeed, there are $O(n^2)$ intervals of interest and n activities which can be adjusted. Given an interval and an activity, the adjustment procedure runs in $O(1)$. Hence, the complexity of the algorithm is $O(n^3)$.

The “left-shift/right-shift” adjustments have been tested on the 40 instances of the BL set. The time allotted to solve each instance was limited to 30 minutes on a Pentium PC running at 200 MHz. Three versions of the algorithm corresponding to the use of a cumulative edge-finder (NO, FE or LSRS (standing for left-shift/right-shift)) were tested. Thirty instances are solved by the most simple version (NO). Nine additional instances are solved when using the fully elastic edge-finder. The LSRS edge-finder is able to solve the 40 instances. In particular, the instance that could not be solved by the fully elastic edge-finder is solved in less than 3 seconds with the LSRS edge-finder.

Table 7 provides the average number of backtracks and the average CPU time over the 30 instances that are solved by all algorithms (columns BT (30) and CPU (30)). We also provide the average values for the 39 instances that are solved by both FE and LSRS.

Table 7. Experimental results on the BL set.

E.F.	Solved	BT (30)	CPU (30)	BT (39)	CPU (39)
NO	30	115457	249.6	---	---
FE	39	5929	33.7	19501	90.0
LSRS	40	2211	30.3	3634	39.4

These results clearly indicate that the use of efficient constraint propagation algorithms is crucial for solving highly cumulative project scheduling problems. Concerning highly disjunctive instances, our best results are, in CPU time, not as good as those of Demeulemeester and Herroelen (1992). As already mentioned, their algorithm relies a lot on the cut-set dominance rule, which we have deliberately decided not to use in our current study, due to its high memory cost (our current program requires no more than 500K for 51 activities). A limited use of this rule, together with constraint propagation, is the subject of a further study.

Notes

1. Given a constraint c over n variables $v_1 \dots v_n$ and a domain D_i for each variable v_i , c is “arc-consistent” if and only if for any variable v_i and any value val_i in the domain of v_i , there exist values $val_1 \dots val_{i-1} val_{i+1} \dots val_n$ in $D_1 \dots D_{i-1} D_{i+1} \dots D_n$ such that $c(val_1 \dots val_n)$ holds. Arc-B-consistency, where B stands for bounds, guarantees only that $val_1 \dots val_{i-1} val_{i+1} \dots val_n$ exist for val_i equal to either the smallest or the greatest value in D_i . In CLAIRESCHEDULE,

domains of start and end time variables are represented as intervals, so in general not more than arc-B-consistency is achieved. Needless to say, arc-B-consistency is not even achieved for resource constraints, since the underlying decision problem is NP-complete.

2. The term “edge-finding” refers to the fact that, in the non-preemptive disjunctive case, the technique consists of deducing that some activities must precede other activities, thereby resulting in orienting edges in a graph representing the possible orderings of activities. This term seems to apply less well in the preemptive and in the cumulative case, since some activities can overlap on the same resource. However, it appears that the propagation rules applied in the preemptive and in the cumulative case deduce orderings of the start and end times of activities, which can be seen as an extension of the ordering of activities.
3. Given a directed graph $G = (X, E)$, $G' = (X, E')$ is the transitive closure of G if and only if $\forall x \in X, \forall y \in X, (x, y) \in E'$ if and only if there exists a directed path from x to y in E .
4. Given a graph $G = (X, E)$, $C \subseteq X$ is a clique of G if and only if $\forall x \in C, \forall y \in C, (x, y) \in E$.

References

- A. Aggoun and N. Beldiceanu (1993). Extending CHIP in Order to Solve Complex Scheduling and Placement Problems. In *Mathematical and Computer Modelling* 17:57-73.
- D. Applegate and W. Cook (1991). A Computational Study of the Job-Shop Scheduling Problem. In *ORSA Journal on Computing* 3(2):149-156.
- Ph. Baptiste and C. Le Pape (1995). A Theoretical and Experimental Comparison of Constraint Propagation Techniques for Disjunctive Scheduling. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*.
- Ph. Baptiste, C. Le Pape, and W. P. M. Nuijten (1998). Satisfiability Tests and Time-Bound Adjustments for Cumulative Scheduling Problems. *Research Report 98-97, Université de Technologie de Compiègne* (<http://www.hds.utc.fr/~baptiste>).
- P. Brucker, S. Knust, A. Schoo, and O. Thiele (1997). A Branch and Bound Algorithm for the Resource-Constrained Project Scheduling Problem. *Working Paper, University of Osnabrück*.
- J. Carlier and B. Latapie (1991). Une méthode arborescente pour résoudre les problèmes cumulatifs. In *RAIRO Recherche opérationnelle / Operations Research* 25(3):311-340.
- J. Carlier and E. Néron (1996). A New Branch-and-Bound Method for Solving the Resource-Constrained Project Scheduling Problem. In *Proceedings of the International Workshop on Production Planning and Control*.
- J. Carlier and E. Pinson (1990). A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem. In *Annals of Operations Research* 26:269-287.
- J. Carlier and E. Pinson (1996). Jackson's Pseudo-Preemptive Schedule for the Pm/ri,qi/Cmax Scheduling Problem. *Technical Report, Université de Technologie de Compiègne*.
- Y. Caseau and F. Laburthe (1995). Disjunctive Scheduling with Task Intervals. *Technical Report, Ecole Normale Supérieure*.
- Y. Caseau and F. Laburthe (1996a). Cumulative Scheduling with Task Intervals. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*.
- Y. Caseau and F. Laburthe (1996b). CLAIRE: A Parametric Tool to Generate C++ Code for Problem Solving. *Working Paper, Bouygues, Direction Scientifique*.
- A. Cesta and A. Oddi (1996). Gaining Efficiency and Flexibility in the Simple Temporal Problem. In *Proceedings of the 3rd International Workshop on Temporal Representation and Reasoning*.

- Y. Colombani (1996). Constraint Programming: An Efficient and Practical Approach to Solving the Job-Shop Problem. In *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming*, 149-163, Springer-Verlag.
- E. Demeulemeester and W. Herroelen (1992). A Branch-and-Bound Procedure for the Multiple Resource-Constrained Project Scheduling Problem. In *Management Science* 38(12):1803-1818.
- B. De Reyck and W. Herroelen (1995). Assembly Line Balancing by Resource-Constrained Project Scheduling Techniques: A Critical Appraisal. *Technical Report, Katholieke Universiteit Leuven*.
- M. R. Garey and D. S. Johnson (1979). *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.
- M. Gondran and M. Minoux (1984). *Graphs and Algorithms*. John Wiley and Sons.
- R. Kolisch, A. Sprecher, and A. Drexel (1995). Characterization and Generation of a General Class of Resource-Constrained Project Scheduling Problems. In *Management Science* 41(10):1693-1703.
- C. Le Pape (1994). Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems. In *Intelligent Systems Engineering* 3(2):55-66.
- C. Le Pape and Ph. Baptiste (1996). Constraint Propagation Techniques for Disjunctive Scheduling: The Preemptive Case. In *Proceedings of the 12th European Conference on Artificial Intelligence*.
- C. Le Pape and Ph. Baptiste (1997). A Constraint Programming Library for Preemptive and Non-Preemptive Scheduling. In *Proceedings of the 3rd International Conference on the Practical Application of Constraint Technology*.
- O. Lhomme (1993). Consistency Techniques for Numeric CSPs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*.
- P. Lopez, J. Erschler and P. Esquirol (1992). Ordonnancement de tâches sous contraintes : une approche énergétique. In *RAIRO Automatique, Productique, Informatique Industrielle* 26:453-481.
- A. A. Mastor (1970). An Experimental Investigation and Comparative Evaluation of Production Line Balancing Techniques. In *Management Science* 16(11):728-746.
- W. P. M. Nuijten (1994). *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach*. PhD Thesis, Eindhoven University of Technology.
- J. H. Patterson (1984). A Comparison of Exact Approaches for Solving the Multiple Constrained Resource Project Scheduling Problem. In *Management Science* 30(7):854-867.
- M. Perregaard (1995). *Branch and Bound Methods for the Multi-Processor Job Shop and Flow Shop Scheduling Problem*. MSc Thesis, University of Copenhagen.