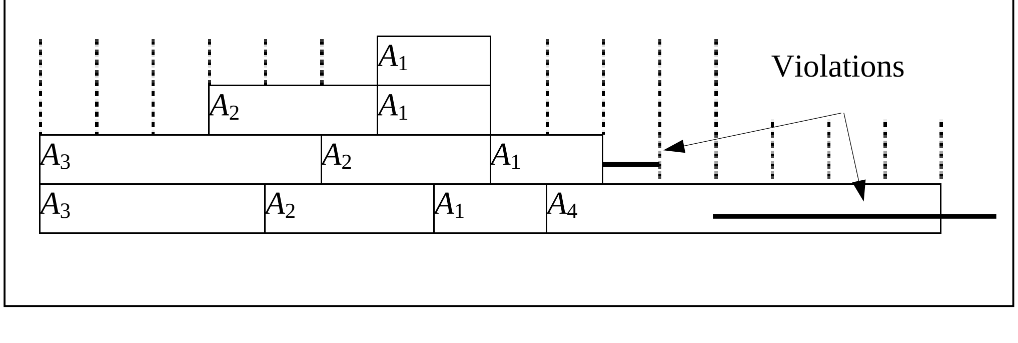
Université de Technologie de Compiègne, UMR CNRS 6599 HEUDIASYC

UNE ÉTUDE THÉORIQUE ET EXPÉRIMENTALE DE LA PROPAGATION DES CONTRAINTES DE RESSOURCES

A THEORETICAL AND EXPERIMENTAL STUDY OF RESOURCE CONSTRAINT PROPAGATION

Thèse soutenue le 30 octobre 1998, dans la spécialité Contrôle Des Systèmes, par Monsieur Philippe Baptiste, ingénieur civil des Mines de Nancy, pour l'obtention du grade de Docteur de l'UTC, devant le jury composé de :

Mademoiselle Marie-Claude Portmann	(Présidente)
Monsieur Peter Brucker	(Rapporteur)
Monsieur Marc Bui	
Monsieur Jacques Carlier	(Directeur de Thèse)
Monsieur Yves Caseau	
Monsieur Claude Le Pape	
Monsieur Wim Nuijten	
Monsieur Eric Pinson	(Rapporteur)
Monsieur Pierre Villon	



UNE ÉTUDE THÉORIQUE ET EXPÉRIMENTALE DE LA PROPAGATION DES CONTRAINTES DE RESSOURCES

A THEORETICAL AND EXPERIMENTAL STUDY OF RESOURCE CONSTRAINT PROPAGATION

Thèse soutenue le 30 octobre 1998 devant le jury composé de :

Mademoiselle Marie-Claude Portmann	(Présidente)
Monsieur Peter Brucker	(Rapporteur)
Monsieur Marc Bui	
Monsieur Jacques Carlier	(Directeur de Thèse)
Monsieur Yves Caseau	
Monsieur Claude Le Pape	
Monsieur Wim Nuijten	
Monsieur Eric Pinson	(Rapporteur)
Monsieur Pierre Villon	

Remerciements

Je tiens à remercier Monsieur Jacques Carlier, Professeur à l'Université de Technologie de Compiègne, qui m'a encadré tout au long de cette thèse et qui m'a fait partager ses brillantes intuitions. Qu'il soit aussi remercié pour sa gentillesse, sa disponibilité permanente et pour les nombreux encouragements qu'il m'a prodiguée.

Je remercie Monsieur Claude Le Pape, directeur du département de Recherche et de Développement de Bouygues-Telecom. Cette thèse est le fruit d'une collaboration de plus de cinq années avec lui. C'est à ses côtés que j'ai compris ce que rigueur et précision voulaient dire.

J'adresse tous mes remerciements à Monsieur Peter Brucker, Professeur à l'Université d'Osnabrück, ainsi qu'à Monsieur Eric Pinson, Professeur à l'Institut de Mathématiques Appliquées d'Angers, de l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de cette thèse.

Mademoiselle Marie-Claude Portmann, Professeur des Universités à l'Ecole des Mines de Nancy, m'a non seulement initié à la Recherche Opérationnelle et à la théorie de l'ordonnancement lorsque j'étais de ses élèves, mais elle m'a aussi prodigué de nombreux conseils pour bien débuter le troisième cycle universitaire dont cette thèse est l'accomplissement. Qu'elle en soit remerciée.

J'exprime ma gratitude à Monsieur Marc Bui et à Monsieur Pierre Villon, Professeurs à l'Université de Technologie de Compiègne, qui ont bien voulu être examinateurs.

Je tiens aussi à remercier Monsieur Yves Caseau, Directeur de la Direction des Technologies Nouvelles du groupe Bouygues et Professeur associé à l'ENS, qui m'a accueilli pendant deux ans au sein de son laboratoire. C'est grâce à lui que j'ai pu concilier avec bonheur recherche théorique et appliquée pendant cette thèse.

Merci aussi à Monsieur Wim Nuijten, responsable du développement de ILOG SCHEDULER, dont les thèmes de recherche ont fortement inspiré cette thèse.

Enfin, je tiens à remercier tous les membres de la Direction des Technologies Nouvelles du groupe Bouygues, Catherine Bernez, Tibor Kökény et Arnaud Linz, qui ont répondu avec calme et patience aux questions quotidiennes dont je les accablais. Un grand merci aussi à tous les membres du département de Génie Informatique de l'Université de Technologie de Compiègne et en particulier à, Emmanuel Néron.

Table of Contents

Table of Illustrations

Chapitre A. Introduction (en français)

Le but de cette thèse est de décrire et de comparer, tant d'un point de vue expérimental que théorique, de nouveaux algorithmes de propagation de contraintes de ressources en ordonnancement[1].

Dans une première partie (A.1) de ce chapitre introductif, nous donnons un aperçu rapide de la programmation par contraintes. Nous montrons alors (A.2) que l'efficacité de cette méthode est conditionnée à l'utilisation de méthodes déductives puissantes, *i.e.*, d'algorithmes de propagation de contraintes qui utilisent une formulation « globale » d'un ensemble de contraintes. Au cours de la troisième partie (A.3), nous proposons une classification simple (et un tant soit peu grossière) des problèmes d'ordonnancement que nous avons pu rencontrer. Cette classification nous permet d'identifier un ensemble de situations dans lesquelles il nous semble utile d'étudier et de développer de nouveaux algorithmes de propagation de contraintes de ressources. Nous illustrons la pertinence de ces contraintes de ressources à travers un ensemble de problèmes d'ordonnancement de la littérature. Nous terminons cette introduction en présentant un résumé des résultats obtenus et en annonçant le plan de ce mémoire.

[1] Les travaux présentés dans ce mémoire ont été réalisés pendant que l'auteur était ingénieur de recherche au sein du groupe Bouygues. Au long des deux années passées à la Direction des Technologies Nouvelles, l'auteur a pris part à plusieurs projets industriels incluant (1) la résolution d'un problème de gestion de projet partiellement préemptif (2) l'intégration dans le langage CLAIRE de plusieurs produits de programmation linéaire, (3) la résolution d'un problème de gestion de stocks [Baptiste *et al.*, 1998a] et (4) l'étude d'un problème de séquencement de véhicules généralisé [Régin et Puget, 1997]. Les deux dernières applications faisant partie du projet européen CHIC-2. Ces applications ne sont pas étudiées au long de ce mémoire. Certaines d'entre-elles sont cependant à la source des travaux de recherche présentés dans la suite.

A.1. Programmation par Contraintes

Le problème de satisfaction de contraintes (CSP) peut être énoncé d'une manière informelle comme suit : étant donné (1) un ensemble de variables, (2) pour chaque variable, un domaine (*i.e.*, un ensemble de valeurs possibles), et (3) un ensemble de contraintes entre ces variables, la question est de savoir s'il existe une affectation de valeurs pour chaque variable qui satisfasse toutes les contraintes. Les contraintes peuvent être représentées de manière implicite (*i.e.*, il est alors nécessaire d'effectuer un calcul pour vérifier que la contrainte est vérifiée pour une certaine instanciation des variables) ou de manière explicite (*i.e.*, les tuples de valeurs qui satisfont la contrainte sont enregistrés dans une base de données). D'une façon générale, la programmation par contraintes s'attaque à ce problème. Nous renvoyons le lecteur à, par exemple, [Prosser, 1993], [Esquirol *et al.*, 1995], [Caseau, 1996] pour une description plus poussée de la programmation par contraintes et de ses applications. L'un des intérêts majeurs de cette technique est que les contraintes sont utilisées dans un processus déductif, *i.e.*, la propagation, qui peut permettre de détecter rapidement une inconsistance ou de réduire les domaines des variables ; ce qui permet d'accélérer le traitement du problème.

Par exemple, si x et y sont des variables entières, sur lesquelles les contraintes $x < y$ et $x > 8$ sont imposées, la phase de propagation permet de déduire que la valeur de la variable y est au moins égale à 10. Si la contrainte $y \leq 9$ est ajoutée au système, une contradiction est immédiatement détectée. Sans cette phase de propagation, l'absence d'affectation faisable ne pourrait être prouvée qu'après une énumération plus ou moins longue.

De nombreuses techniques ont été proposées pour propager les contraintes. L'arc-consistance est une technique largement répandue.

Définition A-1.

Etant donnée une contrainte c sur n variables $x_1, \ldots, x_n$ et un domaine $d(x_i)$ pour chaque variable x_i, c est "arc-consistante" si et seulement si pour toute variable x_i et pour toute valeur val_i de $d(x_i)$, il existe des valeurs $val_1, \ldots, val_{i-1}, val_{i+1}, \ldots, val_n$ appartenant aux domaines $d(x_1), \ldots, d(x_{i-1}), d(x_{i+1}), \ldots, d(x_n)$ telles que la contrainte c soit vérifiée lorsque $\forall j \in \{1, ..., n\}, x_j = val_j$. □

Beaucoup de recherches ont été consacrées à des algorithmes de propagation capables de maintenir l'arc-consistance de toutes les contraintes d'un CSP binaire, *i.e.*, d'un CSP dans lequel les contraintes jouent sur deux variables ([Montanari, 1974], [Mackworth, 1977], [Mohr et Henderson, 1986], [Van Hentenryck *et al.*, 1992], [Bessière *at al.*, 1995]).

Dans le cas particulier d'un CSP dont les variables sont contraintes à prendre des valeurs numériques, les domaines sont parfois représentés sous la forme d'un intervalle $[lb(x), ub(x)]$. Cette représentation beaucoup plus compacte permet, d'un point de vue pratique, de manipuler un grand nombre de variables. La propagation de contraintes sur de telles variables se résume souvent à l'arc-B-consistance [Lhomme, 1993], c'est à dire à l'arc-consistance restreinte aux bornes des domaines. Il est aisé de rendre arc-B-consistantes certaines contraintes arithmétiques, comme des contraintes linéaires ([Lhomme, 1993]).

Définition A-2.
Etant donné une contrainte c sur n variables $x_1, \dots, x_n$ et un domaine $d(x_i) = [lb(x_i), ub(x_i)]$ pour chaque variable x_i, c est "arc-B-consistante" si et seulement si $\forall\, i$ et $\forall\, val_i \in \{lb(x_i), ub(x_i)\}$, il existe $val_1, \dots, val_{i-1}, val_{i+1}, \dots, val_n$ appartenant respectivement aux domaines $d(x_1), \dots, d(x_{i-1}), d(x_{i+1}), \dots, d(x_n)$ telles que la contrainte c soit vérifiée lorsque $\forall\, j \in \{1, ..., n\}, x_j = val_j$. ☐

Pour des raisons évidentes de complexité, la propagation des contraintes est généralement incomplète, toutes les conséquences des contraintes sur les domaines des variables n'étant pas calculables en un temps limité. Il est donc nécessaire de développer une recherche arborescente pour déterminer s'il existe ou non une affectation valide de valeurs aux variables. Les deux caractéristiques les plus importantes d'une telle recherche arborescente sont :

- Les heuristiques utilisées pour choisir la variable à instancier et pour déterminer la valeur du domaine de cette variable à essayer en premier (*e.g.*, choisir la variable dont le domaine est le plus petit et essayer de l'instancier à la valeur minimum de son domaine).
- La stratégie de retour arrière en cas d'échec, *i.e.*, lorsqu'il a été prouvé qu'aucune affectation faisable ne peut être dérivée de l'état courant du système. La plupart des outils de programmation par contraintes sont basés sur une recherche en profondeur d'abord : la dernière décision est remise en cause et l'alternative à cette décision est imposée. D'autres stratégies de retour arrière ont été proposées, comme le « backtrack » intelligent.

Le comportement général d'un système de contraintes peut se résumer à la figure A-1. Notons que le la définition du problème, la propagation des contraintes et la phase de prise de décision sont clairement séparés.

- En premier lieu, le problème est défini en termes de variables et de contraintes.
- Puis, les algorithmes de propagation de ces contraintes sont spécifiés. En pratique, l'utilisateur d'un système de contraintes peut soit utiliser des contraintes prédéfinies, par exemple des contraintes sur des entiers ou sur des ensembles, soit définir ses propres contraintes dont il pourra expliciter les méthodes de propagation.
- Enfin, le mécanisme de prise de décision, c'est-à-dire la façon dont l'arbre de recherche est construit, est défini. Il précise le type de décisions qui doivent être prises au fur et à mesure de la recherche (*e.g.*, instancier une variable à une valeur, ordonner une paire d'activités).

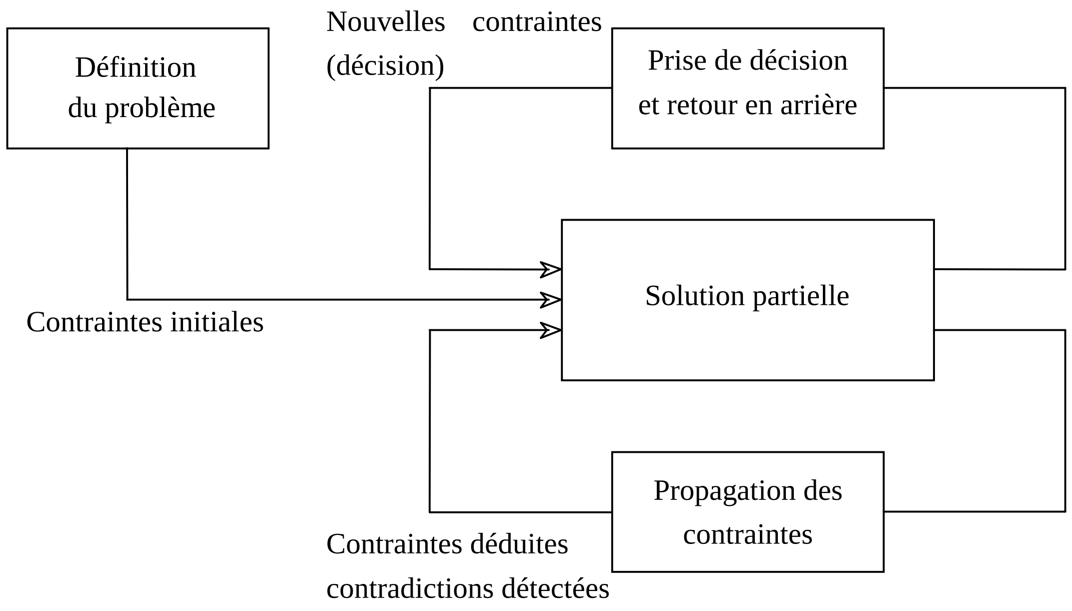

Définition du problème
Nouvelles contraintes (décision)
Prise de décision et retour en arrière
Solution partielle
Contraintes initiales
Propagation des contraintes
Contraintes déduites contradictions détectées

Figure A-1. Le comportement d'un système de programmation par contraintes

Le fait que la propagation des contraintes soit un mécanisme indépendant des autres parties du système est d'un grand intérêt en terme de réutilisation de code. En effet, les algorithmes de propagation de contraintes sont (ou devraient être) totalement génériques, et peuvent donc être réutilisés dans toutes les applications où la même contrainte est à nouveau présente. Tel n'est pas le cas des algorithmes de recherche qui sont difficilement réutilisables d'un problème à l'autre (ces algorithmes utilisent souvent des critères de dominance qui la plupart du temps ne sont plus vérifiés dès qu'une nouvelle contrainte est ajoutée au système).
Cette possibilité de réutiliser des algorithmes de propagation d'une application à l'autre est l'une des raisons de l'engouement des industriels pour des outils de programmation par contraintes, parfois au détriment d'autres techniques de résolution, comme la programmation linéaire qui, même si elle se montre extrêmement performante sur certains problèmes, nécessite souvent l'élaboration de modèles complexes. Parmi les systèmes de contraintes (commerciaux ou de domaine public), citons ILOG SOLVER [Puget, 1994], [Puget et Leconte, 1995], CHIP [Aggoun et Beldiceanu, 1993], [Beldiceanu et Contejean, 1994], ECLIPSE, et CLAIRE [Caseau et Laburthe, 1996b] accompagné d'ECLAIR [Laburthe *et al.*, 1998].

A.2. Recherche Opérationnelle et Programmation par Contraintes

L'utilisation d'algorithmes « dédiés » de propagation de contraintes permet d'accroître considérablement l'efficacité des systèmes de contraintes. De tels algorithmes sont capables de prendre en compte d'un point de vue global un ensemble de contraintes. Considérons par exemple la contrainte dite « tous-différents » qui contraint un ensemble de n variables à prendre des valeurs deux à deux distinctes. Un algorithme de propagation trivial consiste à décomposer cette contrainte en $n * (n - 1) / 2$ contraintes « locales » qui imposent pour toute paire de variables (x, y) que $x \neq y$. Propager cette contrainte se fait alors simplement par arc-consistance locale sur chacune des contraintes. [Régin, 1994] décrit un algorithme bien plus puissant qui garantit l'arc-consistance globale de la contrainte « tous-différents ». La contrainte est modélisée sous la forme d'un graphe biparti, où sont représentées d'un côté les variables, de l'autre l'union des domaines des variables. Une arête entre une variable et une valeur indiquant que cette valeur fait partie du domaine de la variable. La contrainte est évidemment consistante si et seulement si le couplage maximum du graphe est de cardinalité n. Régin utilise donc un algorithme de Recherche Opérationnelle pour assurer la consistance globale de la contrainte. Mais il étend aussi de façon originale ce mécanisme pour assurer, avec une complexité raisonnable, l'arc-consistance globale de la contrainte : pour chaque variable, les valeurs du domaine qui rendraient la contrainte insatisfiable sont retirées. Une autre contrainte globale célèbre est la contrainte de ressource qui impose à un ensemble d'activités de s'exécuter sur une machine. De nombreux travaux (*e.g.*, [Nuijten, 1994], [Caseau et Laburthe, 1995], [Baptiste et Le Pape, 1995b], [Colombani, 1996]) ont porté sur des algorithmes de propagation globaux pour cette contrainte. Tous reprennent les idées des travaux fondateurs de Carlier et Pinson sur le Job-Shop (*e.g.*, [Carlier et Pinson, 1989]).

L'intégration de tels algorithmes permet de bénéficier de l'efficacité de techniques de recherche opérationnelle dans le cadre très souple de la programmation par contraintes. En d'autres termes, nous disposons d'une part d'algorithmes très efficaces de Recherche Opérationnelle mais dont le spectre d'utilisation est parfois réduit, et d'autre part de techniques plus générales de propagation de contraintes dont le spectre est beaucoup plus large mais dont l'efficacité reste souvent à démontrer. Nous nous proposons de développer un ensemble d'algorithmes de Recherche Opérationnelle intégrables dans un système d'ordonnancement à base de contraintes.

A.3. L’Ordonnancement

Nous proposons une typologie rudimentaire des problèmes d’ordonnancement à contraintes de ressources. Fondée, en partie, sur les problèmes industriels que nous avons pu rencontrer au cours de ces dernières années, elle ne prétend pas être exhaustive. Au sens strict, un problème d’ordonnancement consiste à déterminer les dates d’exécutions d’activités qui utilisent une ou des quantités connues d’un ensemble donné de ressources dont les capacités sont limitées. Nous laissons donc de côté les problèmes d’affectation où le positionnement des activités est connu, et où l’on cherche à couvrir la demande en ressource de ces activités par une affectation adéquate de ressources aux activités.
Nous distinguons trois dimensions dans notre classification.

- Dans un problème d’ordonnancement **non-préemptif**, les activités sont exécutées sans interruption de leur date de début à leur date de fin. Au contraire, dans un problème **préemptif**, les activités peuvent être interrompues à tout instant pour laisser, par exemple, s’exécuter des activités plus urgentes.
- Dans un problème **disjonctif**, les ressources ne peuvent exécuter qu’une activité à la fois. Dans un problème **cumulatif**, une ressource peut exécuter plusieurs activités en parallèle.
- Dans la plupart des cas, les **contraintes de ressources** doivent être prises au sens **strict**, (*i.e.*, elles ne peuvent jamais être violées). Dans certains problèmes, lorsque la ressource est **surchargée**, les contraintes de ressources peuvent être prises dans un sens plus **large** : un nombre limité d’activités peuvent être sous-traitées, pour rendre la contrainte de ressource satisfiable. La ressource se caractérise alors par sa capacité totale et par le nombre d’activités qu’elle peut sous-traiter.

Dans le « meilleur » des cas, le problème consiste à déterminer un ordonnancement faisable, c’est à dire un ordonnancement qui respecte toutes les contraintes, mais le plus souvent, un critère doit être optimisé. Bien que le makespan, *i.e.*, la date de fin de l’ordonnancement, soit le critère le plus fréquemment utilisé (ce qui d’ailleurs ne correspond pas forcément à un critère d’une grande utilité concrète), d’autres critères peuvent être considérés. Citons par exemple le nombre d’activités exécutées dans un certain délai, le retard moyen ou pondéré, ou encore le pic d’utilisation d’une ressource.
La théorie de l’ordonnancement est un champ de recherches très large qui, tant d’un point de vue pratique qu’appliqué, a donné lieu à un nombre important de publications. Nous renvoyons le lecteur à [Baker, 1974], [Coffman, 1976], [French, 1982], [Carlier et Chrétienne, 1988], [GOThA, 1993], [Brucker, 1995] pour une introduction plus poussée à ce domaine.

Comment représenter cet ensemble de problèmes d'ordonnancement dans un système de contraintes ? Nous utilisons un modèle simple constitué de quatre entités : les activités, les ressources, les contraintes temporelles et les contraintes de ressources. Les activités sont liées entre elles par des contraintes temporelles. Activités et ressources sont liées entre elles par des contraintes de ressources.

A.3.1. Représentation des Activités et des Ressources

Dans le cas non-préemptif, deux variables *start*(A_i) et *end*(A_i) sont associées à chaque activité A_i. Elles représentent respectivement les dates de début et de fin de A_i. La plus petite valeur dans le domaine de *start*(A_i) est en fait la date de disponibilité r_i de l'activité, et la plus grande des valeurs dans le domaine de *end*(A_i) est la date d'échéance d_i de A_i. Nous appellerons lst_i la plus grande valeur dans le domaine de *start*(A_i), *i.e.*, la date de début au plus tard de A_i, et nous appellerons eet_i la plus petite valeur dans le domaine de *end*(A_i), *i.e.*, la date de fin au plus tôt de A_i. Le temps d'exécution de A_i est représenté par une autre variable *processingTime*(A_i) qui est contrainte à être égale à la différence entre *end*(A_i) et *start*(A_i). Le plus souvent, nous considérerons que la variable *processingTime*(A_i) est instanciée à une valeur p_i, (*i.e.*, les temps d'exécution sont connus et fixés).

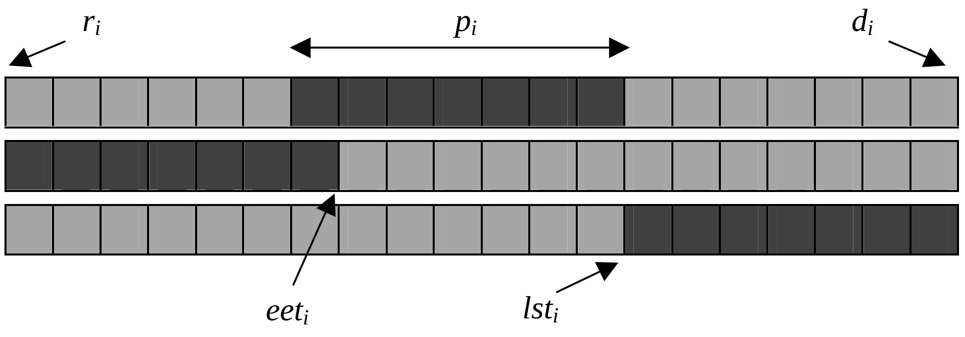


Figure A-2. La date de disponibilité, la date d'échéance, le temps d'exécution, la date de fin au plus tôt et la date de début au plus tard d'une activité (la couleur gris clair est utilisée pour représenter la fenêtre [r_i, d_i] de l'activité alors que le gris foncé représente la durée de l'activité).

Un problème d'ordonnancement préemptif est sensiblement plus complexe à représenter. Il est tout aussi possible d'associer une variable ensembliste (*i.e.*, une variable dont la valeur est un ensemble) *set*(A_i) à chaque activité A_i que d'utiliser des variables binaires $W(A_i, t)$ pour chaque activité A_i ; l'activité s'exécutant à l'instant t si et seulement si $W(A_i, t) = 1$. Sans tenir compte des détails d'implémentation, notons que

- $W(A_i, t)$ vaut 1 si et seulement si t appartient à $set(A_i)$
- $start(A_i) = \min_{t \in set(A_i)}(t)$ et $end(A_i) = \max_{t \in set(A_i)}(t + 1)$; ces variables étant indispensables pour connecter les activités par des contraintes temporelles, comme nous le verrons dans la suite. Notons que dans le cas non-préemptif, l'équation $set(A_i) = [start(A_i), end(A_i))$ est vérifiée. L'intervalle est fermé à gauche et ouvert à droite, ce qui permet de vérifier $|set(A_i)| = end(A_i) - start(A_i) = processingTime(A_i)$.

Nous représentons une ressource R par la variable $capacity(R)$ qui définit la capacité de la ressource, *e.g.*, le nombre de machines parallèles identiques disponibles dans l'atelier. Notons que si la capacité de la ressource varie au cours du temps, il suffit alors d'introduire pour chaque instant t, $capacity(R, t)$, la variable contrainte qui représente la capacité de la ressource R à l'instant t. Pour simplifier, nous noterons C_R la capacité maximale de la ressource ($C_R = ub(capacity(R))$). Dans la suite, l'indice R sera omis lorsqu'une seule ressource est considérée.

Un tel modèle permet de représenter un grand nombre de types de ressources. Cependant, pour prendre en compte le cas où une ressource est surchargée, nous introduisons une variable $reject(R)$ qui représente le nombre d'activités qui devraient s'exécuter sur la ressource, mais qui sont sous-traitées du fait de la surcharge.

A.3.2. Contraintes Temporelles et Contraintes de Ressources

Nous qualifions de contrainte temporelle une contrainte qui lie le début ou la fin de deux activités par une relation linéaire. Par exemple, une contrainte de précédence entre A_i et A_j se représente par l'équation linéaire $end(A_i) \leq start(A_j)$. De telles contraintes sont propagées en utilisant un algorithme d'arc-B-consistance [Lhomme, 1993]. De plus, une variante de l'algorithme de Ford proposée par [Cesta et Oddi, 1996] est utilisée pour détecter en temps polynomial en le nombre de contraintes toute inconsistance liée au réseau de contraintes de précédence (et aux contraintes de temps d'exécution).

Une contrainte de ressource représente le fait que les activités utilisent une certaine quantité de ressource tout au long de leur exécution. Etant données une activité A_i et une ressource R, nous noterons $capacity(A_i, R)$ la variable contrainte correspondant à la quantité de ressource R requise par l'activité A_i. $c_{i,R} = lb(capacity(A_i, R))$ est alors la quantité minimale de ressource utilisée pendant l'exécution de A_i. Une contrainte de ressource spécifie qu'à chaque instant t, la capacité de la ressource R est supérieure ou égale à la somme, sur toutes les activités, des capacités requises à l'instant t.

- Dans le cas non-préemptif, cette contrainte peut s'écrire

$$\forall R, \forall t \qquad \sum_{start(A_i)\leq t<end(A_i)} capacity(A_i, R) \leq capacity(R, t).$$

- Dans le cas préemptif,

$$\forall R, \forall t \qquad \sum_{start(A_i)\leq t<end(A_i)} W(A_i, t) * capacity(A_i, R) \leq capacity(R, t).$$

Examinons maintenant le cas où la machine est surchargée. Il est assez naturel d'introduire, pour chaque activité A_i et pour chaque ressource surchargée R sur laquelle elle peut s'exécuter, une variable $in(A_i, R)$ qui permet de déterminer si A_i est exécutée (*i.e.*, $in(A_i, R) = 1$) sur R ou si A_i est sous-traitée (*i.e.*, $in(A_i, R) = 0$). Lorsqu'une seule ressource est considérée, l'indice R sera omis. La contrainte de ressource impose alors que toutes les activités non sous-traitées vérifient une contrainte de ressource standard.

- Dans le cas non préemptif, cette contrainte peut s'écrire

$$\begin{cases} \forall R, \forall t, \qquad \sum\limits_{start(A_i)\leq t<end(A_i)} in(A_i, R) * capacity(A_i, R) \leq capacity(R, t) \\ \sum in(A_i, R) \geq n - reject(R) \end{cases}$$

- Dans le cas préemptif,

$$\begin{cases} \forall R, \forall t, \qquad \sum\limits_{start(A_i)\leq t<end(A_i)} in(A_i, R) * W(A_i, t) * capacity(A_i, R) \leq capacity(R, t) \\ \sum in(A_i, R) \geq n - reject(R) \end{cases}$$

Notons que ce modèle se généralise trivialement si un poids est associé à chaque activité et que la ressource ne peut sous-traiter qu'un poids total donné d'activités.

A.3.3. Des Problèmes Classiques d'Ordonnancement

Nous nous proposons de montrer la façon dont peuvent être représentés, au moyen du modèle proposé dans les paragraphes précédents, quatre problèmes classiques d'ordonnancement. Dans ces quatre cas, les problèmes de décisions associés sont NP-Complets au sens fort [Garey et Johnson, 1979].

Le problème du Job-Shop (JSSP)

Instance. Une instance de la variante de décision du Job-Shop est décrite par un ensemble de *m* machines, par un ensemble de *n* jobs et par une date d'échéance globale *D*. Chaque job J_i est constitué d'une liste L_i d'activités. Pour chaque activité A_i, un temps d'exécution p_i entier ainsi que la machine sur laquelle elle doit s'exécuter sont spécifiés.

Question. Existe-t-il un ordonnancement non-préemptif des activités, c'est-à-dire une date de démarrage pour chaque activité, tel que (1) chaque machine exécute au plus une activité à la fois, (2) les activités d'un même job sont exécutées dans l'ordre induit par la liste L_i et (3) les dates de fin des activités ne dépassent pas la date d'échéance *D* ?

Modèle. Chaque machine est représentée par une ressource disjonctive. Les activités sont non-interruptibles et utilisent la ressource correspondant à la machine qui leur est attribuée. Les contraintes de précédence induites par les jobs sont imposées sur les activités. Enfin, pour chaque activité A_i, les domaine initiaux des variables *start*(A_i) et *end*(A_i) sont fixés à [0, *D*].

Le problème du Job-Shop préemptif (PJSSP)

Instance. Mêmes données que celles du JSSP.

Question. Existe-t-il un ordonnancement préemptif des activités, c'est-à-dire, pour chaque activité, un ensemble d'intervalles de temps dont la durée totale est la durée de l'activité, tel que (1) chaque machine exécute au plus une activité à la fois, (2) les activités d'un même job sont exécutées dans l'ordre induit par la liste L_i et (3) les dates de fin des activités ne dépassent pas la date d'échéance *D* ?

Modèle. Chaque machine est représentée par une ressource disjonctive. Les activités sont interruptibles et utilisent la ressource correspondant à la machine qui leur est attribuée. Pour chaque activité, les domaine initiaux des variables *start*(A_i) et *end*(A_i) sont fixés à [0, *D*].

Alors que pour la plupart des problèmes d'ordonnancement, la relaxation préemptive est plus « facile » que le problème d'origine, le PJSSP est « plus difficile » que le JSSP. En effet, il a été démontré que lorsque le nombre de machines est fixé à 2 et que le nombre de jobs est fixé à 3, le PJSSP est NP-difficile alors que le problème non-préemptif est fortement polynomial ([Brucker *et al.*, 1999]).

Le problème de gestion de projet à contraintes de ressources (RCPSP)

Instance. Une instance de la variante de décision du RCPSP est décrite par (1) un ensemble de ressources de capacités données, (2) un ensemble d'activités non-interruptibles de durées données, (3) un graphe orienté sans cycle représentant les contraintes de précédence entre les activités, (4) un entier par activité et par ressource représentant la quantité de ressource utilisée par l'activité tout au long de son exécution, et enfin (5) par une date d'échéance globale D.

Question. Existe-t-il un ordonnancement, *i.e.*, un ensemble de dates de démarrage des activités, qui permet de satisfaire à la fois les contraintes de précédence et les contraintes de ressources, et dont la durée totale est inférieure ou égale à D ?

Modèle. Les activités de l'instance sont représentées par des activités non-interruptibles, chaque ressource est représentée par une ressource cumulative dont la capacité est fixée (*i.e.*, la variable *capacity*(R) est instanciée). Des contraintes de ressources sont imposées entre activités et ressources (*capacity*(A_i, R) est instanciée). Des contraintes temporelles sont imposées conformément au graphe de précédence. Enfin, pour chaque activité A_i, les domaine initiaux des variables *start*(A_i) et *end*(A_i) sont fixés à $[0, D]$.

Minimiser le nombre de jobs en retard sur une machine ($1|r_j|\Sigma U_j$)

Instance. Une instance de la variante de décision de ce problème est constituée d'un ensemble de n jobs (chaque job étant décrit par une date de disponibilité r_i, une date d'échéance d_i et un temps d'exécution p_i avec $r_i + p_i \leq d_i$) et d'un entier N.

Question. Existe-t-il un ordonnancement des jobs, *i.e.*, un ensemble de dates de démarrage, tel que (1) un job au plus s'exécute à chaque instant, (2) chaque job débute après sa date de disponibilité, (3) moins de N jobs finissent après leur date d'échéance ?

Modèle. Chaque job est représenté par une activité. Dates de disponibilités, dates d'échéances et durées sont imposées aux variables *start*, *end* et *processingTime*. Une ressource disjonctive R, dont la surcharge est autorisée, est utilisée pour modéliser la machine. Les activités A_i telles que *in*(A_i, R) = 1 sont à l'heure. Les autres sont en retard et peuvent être ordonnancées arbitrairement tard. Enfin, le domaine de la variable *reject*(R) est fixé à $[0, N]$.

A.4. Résumé des Résultats et Plan de la Thèse

Le modèle que nous avons présenté est très général. Pour être totalement exhaustif, il nous faudrait étudier huit types de contraintes de ressources (préemptif *vs*. Non-préemptif, disjonctif *vs*. cumulatif et contrainte de ressource stricte *vs*. contrainte de ressource surchargée). En pratique, nous ne nous sommes pour l'instant intéressé en détail qu'aux cas détaillés ci-dessous.

- **Contraintes de ressources disjonctives sans préemption**. Suite aux travaux de [Nuijten, 1994], la plupart des systèmes de programmation par contraintes ont maintenant intégré des variantes des travaux de Carlier et Pinson sur le problème à une machine. L'idée sous-jacente de ces travaux est de comparer les caractéristiques temporelles d'une activité par rapport à un ensemble d'activités. Il est alors possible de déduire qu'une activité A_i peut, ne peut pas, ou doit s'exécuter après un ensemble d'autres activités S ; ceci se traduisant par une réduction des fenêtres de temps des activités. Plusieurs algorithmes ont été proposés pour effectuer toutes les déductions possibles du type « A_i doit s'exécuter après S ». En particulier, [Carlier et Pinson, 1994] décrit un algorithme dont la complexité théorique n'est que $O(n \log(n))$. Les règles permettant de prouver qu'une activité ne peut pas s'exécuter après un ensemble de tâches ont été beaucoup moins étudiées. Nous proposons le premier algorithme capable d'effectuer toutes les déductions possibles à partir de ces règles. La complexité de cet algorithme est $O(n^2)$.
- **Contraintes de ressources disjonctives dans le cas préemptif**. A notre connaissance, de telles contraintes de ressources n'ont jamais été étudiées en tant que telles. Nous proposons plusieurs algorithmes de propagation de cette contrainte : le premier est basé sur un emploi du temps de la ressource, le second sur une formulation disjonctive du problème, le troisième sur un problème de flot dans un réseau de transport et le dernier sur une extension au cas préemptif des algorithmes d'ajustement de Carlier et Pinson.
- **Contraintes de ressources cumulatives**. Contrairement au cas disjonctif, la communauté de Recherche Opérationnelle a peu étudié les méthodes déductives pour ce genre de problème, mis à part évidemment des calculs de borne inférieure pour certains cas particuliers de cette contrainte de ressource. De nombreux travaux ont été menés au sein de la communauté de programmation par contraintes pour tenter de généraliser les résultats obtenus dans le cas disjonctif (*e.g.*, [Aggoun and Beldiceanu, 1993], [Nuijten, 1994], [Caseau et Laburthe, 1996a]). Les résultats sont moins satisfaisants que dans le cas disjonctif. Nous abordons l'étude de cette contrainte de

ressource en introduisant un problème de décision, le « Cumulative Scheduling Problem » (CuSP), dont nous étudions deux relaxations — une relaxation dite totalement élastique qui permet de se ramener à un problème à une machine, et une relaxation partiellement élastique. Nous étudions aussi un ensemble de conditions nécessaires à l'existence d'un ordonnancement faisable basées sur une approche énergétique [Lopez *et al.*, 1992]. Dans tous les cas, nous proposons des algorithmes permettant de vérifier des conditions nécessaires d'existence et d'ajuster les fenêtres temporelles des tâches. Nous comparons ces résultats aux bornes inférieures de la littérature pour le problème à m-machines. Nous montrons que la relaxation partiellement élastique est équivalente à une borne inférieure connue sous le nom de la subset bound [Perregaard, 1995], elle-même équivalente à la borne obtenue par une relaxation pseudo-préemptive proposée par [Carlier et Pinson, 1996]. Nous montrons aussi que le raisonnement énergétique domine strictement toutes les autres techniques précédemment citées. Notons enfin que certains des résultats énoncés sont valables dans le cas cumulatif préemptif.

- **Les contraintes de ressources disjonctives et surchargées**. Dans le cadre de la programmation par contraintes, de telles contraintes de ressources n'ont pas été étudiées. Dans la communauté de Recherche Opérationnelle, un grand nombre de travaux ont été effectués sur le problème de la minimisation du nombre de « jobs » en retard sur une machine. En particulier, de nombreux cas particulier ont été traités. Nous montrons d'ailleurs que deux d'entre eux, jusqu'alors ouverts, sont polynomiaux : deux algorithmes fortement polynomiaux décrits en annexe permettent de minimiser dans les cas préemptif et non-préemptif le nombre pondéré de jobs en retard lorsque les durées des jobs sont égales.
  Nous proposons plusieurs techniques pour la contrainte de ressource surchargée. Nous étudions la relaxation préemptive de cette contrainte de ressource et nous proposons un algorithme en $O(n^4)$ qui permet de calculer l'optimum préemptif. Nous améliorons ainsi l'algorithme de [Lawler, 1990]. Nous montrons aussi qu'en utilisant une relaxation encore plus forte, une borne de moindre qualité peut être obtenue (à moindre coût). De plus, cette relaxation nous permet de proposer un algorithme capable de déduire que certaines activités doivent obligatoirement être sous-traitées alors que d'autres doivent être obligatoirement exécutées sur la ressource.

Le Chapitre B est consacré à l'étude de ces différentes contraintes et aux algorithmes de propagation associés. Dans le but d'évaluer d'un point de vue expérimental leur efficacité, nous décrivons dans le Chapitre C des méthodes arborescentes avec propagation de contraintes pour résoudre les problèmes classiques d'ordonnancement que nous avons évoqués précédemment. Le Job-Shop préemptif, le problème de gestion de projet à contraintes de ressources, ainsi que le problème de la minimisation du nombre de jobs en retard sur une machine ($1|r_j|\Sigma U_j$) sont étudiés. En sus de la propagation, nous utilisons un certain nombre de critères de dominance qui, bien exploités, permettent de réduire considérablement l'espace de recherche. Des résultats expérimentaux sont décrits pour chaque problème. Ils nous permettent non seulement d'évaluer l'efficacité relative des algorithmes de propagation du chapitre B, mais aussi de nous comparer, pour chaque problème, aux meilleures procédures de séparation et d'évaluation connues.

- A notre connaissance, aucune méthode exacte n'a été proposée pour résoudre le problème du Job-Shop preemptif. Nous nous contentons donc de comparer les différentes méthodes que nous proposons. Notre schéma de branchement est chronologique et nous appliquons un critère de dominance qui impose que les ordonnancements des machines soient du type de ceux de Jackson. Ce schéma de branchement, associé aux ajustements qui étendent les travaux de Carlier et Pinson au cas preemptif, est relativement efficace puisque toutes les instances de la littérature de taille 10*10 sont résolues.
- Nous proposons plusieurs variantes du même schéma de branchement pour le RCPSP. Une caractérisation simple des instances nous permet de déterminer si celles-ci sont « fortement disjonctives » ou « fortement cumulatives ». Nous montrons alors que les différents schémas de branchement que nous utilisons sont plus ou moins efficaces suivant le type d'instance. Il est clair que sur des instances fortement disjonctives, notre méthode n'est pas aussi efficace que les procédures utilisant les résultats de [Demeulemeester et Herroelen, 1995]. Nous montrons cependant que sur des instances fortement cumulatives, nous obtenons des résultats extrêmement encourageants. Résultats confirmés d'ailleurs sur le problème du flow-shop hybride [Néron *et al.*, 1998].
- Nous proposons pour finir une procédure arborescente pour minimiser le nombre de jobs en retard sur une machine. Le schéma de branchement est extrêmement simple puisqu'il consiste à choisir un job et à le mettre à l'heure ou en retard. A chaque nœud de l'arborescence, nous vérifions que les jobs à l'heure peuvent être ordonnancés (ce sous problème est NP-difficile mais est extrêmement bien résolu par la méthode de [Carlier, 1982]). L'efficacité de cette procédure provient de la propagation des contraintes mais aussi de l'utilisation d'un certain nombre de critères de dominance. Les précédentes méthodes exactes ([Dauzère-Pérès, 1995]) pour résoudre ce problème étaient limitées à une dizaine de jobs. Notre procédure résout 90 % des instances à 100 jobs. De plus nos résultats se comparent très favorablement à la dernière procédure de [Dauzère-Pérès et Sevaux, 1998b].

Nous présentons nos conclusions au cours du chapitre D et nous évoquons quelques directions de recherche qui nous semblent prometteuses dans un avenir proche.

Chapter A. Introduction

The aim of this thesis is to describe and to evaluate, both from a theoretical and experimental point of view, new resource constraint propagation algorithms for several classes of scheduling problems[2].

A brief overview of constraint programming is provided in the first section of this introductory chapter (Section A.1). We then show that a key reason for the efficiency of this technique is the use of powerful deductive methods, *i.e.*, of global propagation algorithms that are applicable on a whole set of constraints (Section A.2). We provide in Section A.3 a simple, and somewhat naive, classification of scheduling problems. It allows us do identify some of the scheduling areas where it could be worth to study a (new) global resource constraint. The relevance of these resource constraints is illustrated through a set of scheduling problems from the literature. Finally, an outline of the thesis and of our research results is provided in Section A.4.

[2] This thesis has been done while the author was working as an engineer at the "Direction des Technologies Nouvelles" of Bouygues. Along the two years spent in this department, the author has taken a part in the development of several industrial projects. This includes (1) the resolution of a partially preemptive project scheduling problem (2) the integration into the Claire programming language of LP solvers, (3) the resolution of a complex stock management problem [Baptiste *et al.*, 1998a] and, (4) in the context of the CHIC-2 ESPRIT project, the study of a generalization of the Car-Sequencing Problem [Régin and Puget, 1997]. These applications are not studied throughout this thesis. However, they have motivated a large amount of the research presented in the following.

A.1. *Constraint Programming*

Constraint programming is concerned with solving instances of the Constraint Satisfaction Problem (CSP). Informally speaking, an instance of the CSP is described by a set of variables, a set of possible values (domain) for each variable, and a set of constraints between the variables. The question is whether there exists an assignment of values to variables, so that all the constraints are satisfied. Constraints are stated either implicitly (*e.g.*, an arithmetic formula) or explicitly (each constraint is a set of tuples of values that satisfy the constraint). For an overview of constraint programming and of its applications, see for instance [Prosser, 1993], [Esquirol *et al.*, 1995], [Caseau, 1996]. The interest of this technique lies in using constraints to reduce the computational effort needed to solve combinatorial problems. Constraints are used not only to test the validity of a solution, as in conventional programming languages, but also in a constructive mode to deduce new constraints and rapidly detect inconsistencies.

For example, from $x < y$ and $x > 8$, we deduce, if x and y denote integers, that the value of y is at least 10. If later we add the constraint $y \leq 9$, a contradiction can be immediately detected. Without propagation, the "$y \leq 9$" test could not be performed before the instantiation of y and thus no contradiction would be detected at this stage of the problem-solving process.

Several techniques have been developed to propagate constraints. Among these techniques, let us mention arc-consistency.

Definition A-1.

Given a constraint c over n variables $x_1, \ldots, x_n$ and a domain $d(x_i)$ for each variable x_i, c is "arc-consistent" if and only if for any variable x_i and any value val_i in $d(x_i)$, there exist values $val_1, \ldots, val_{i-1}, val_{i+1}, \ldots, val_n$ in $d(x_1), \ldots, d(x_{i-1}), d(x_{i+1}), \ldots, d(x_n)$ for the variables $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ such that the constraint c is consistent (*i.e.*, holds when $\forall j \in \{1, \ldots, n\}, x_j = val_j$). □

A huge amount of work has been carried on constraint propagation algorithms that maintain arc-consistency on the constraints of a binary CSP, *i.e.*, of a CSP whose constraints link at most two variables ([Montanari, 1974], [Mackworth, 1977], [Mohr and Henderson, 1986], [Van Hentenryck *et al.*, 1992], [Bessière *at al.*, 1995]).

Numeric CSPs are special cases of the CSP where the variables are constrained to take numeric values. The domain of a variable x can then be represented by an interval $[lb(x), ub(x)]$. This compact representation is often used to tackle real life problems for which maintaining explicitly the set of values that can be taken by each variable throughout the search tree may not be reasonable. A usual way to propagate constraints on such variables is to achieve arc-B-consistency [Lhomme, 1993], *i.e.*, arc-consistency restricted to the Bounds of the domains. Arc-B-consistency can be easily achieved on some arithmetic constraints such as linear constraints [Lhomme, 1993].

Definition A-2.
Given a constraint c over n variables $x_1, \ldots, x_n$ and a domain $d(x_i) = [lb(x_i), ub(x_i)]$ for each variable x_i, c is "arc-B-consistent" if and only if $\forall\ i$ and $\forall\ val_i \in \{lb(x_i), ub(x_i)\}$, there exist values $val_1, \ldots, val_{i-1}, val_{i+1}, \ldots, val_n$ in $d(x_1), \ldots, d(x_{i-1}), d(x_{i+1}), \ldots, d(x_n)$ for the variables $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ such that the constraint c is consistent (*i.e.*, holds when $\forall\ j \in \{1, \ldots, n\}, x_j = val_j$). □

For complexity reasons, constraint propagation is usually incomplete. This means that some but not all the consequences of constraints are deduced. In particular, constraint propagation cannot detect all inconsistencies. Consequently, tree search algorithms must be implemented to determine if the CSP instance is consistent or not. To precisely define the search tree, one has to specify both the heuristic selection and the backtracking strategies.

- Most of the generic search strategies use dynamic criteria to choose both the variable x to instantiate and the value val to which x is to be instantiated (*e.g.*, select the unbound variable with the smallest domain and bound it to the minimum value of its domain).
- The backtracking strategy states how the system shall behave when a contradiction is detected, *i.e.*, when it is proven that there is no feasible assignment of values to variables given the original data of the CSP and given the heuristic choices that have been made. Most constraint programming tools rely on depth-first chronological backtracking: The last decision is undone and the alternative constraint is imposed. More complex backtracking strategies have also been proposed (*e.g.*, intelligent backtracking).

The overall behavior of a constraint-based system is depicted on Figure A-1. This figure underlines the fact that problem definition, constraint propagation and decision making are clearly separated.

- First, the problem is defined in terms of variables and of constraints.
- Then, constraint propagation algorithms are specified. In practice the user of a constraint programming tool can use some pre-defined constraints (*e.g.*, constraints on integers, constraints on sets, scheduling constraints) for which the corresponding propagation algorithms have been pre-implemented.
- Finally, the decision-making process, *i.e.*, the way the search tree is built, is specified. It states how new constraints are added to the system (*e.g.*, instantiating a variable to a value, ordering a pair of activities).

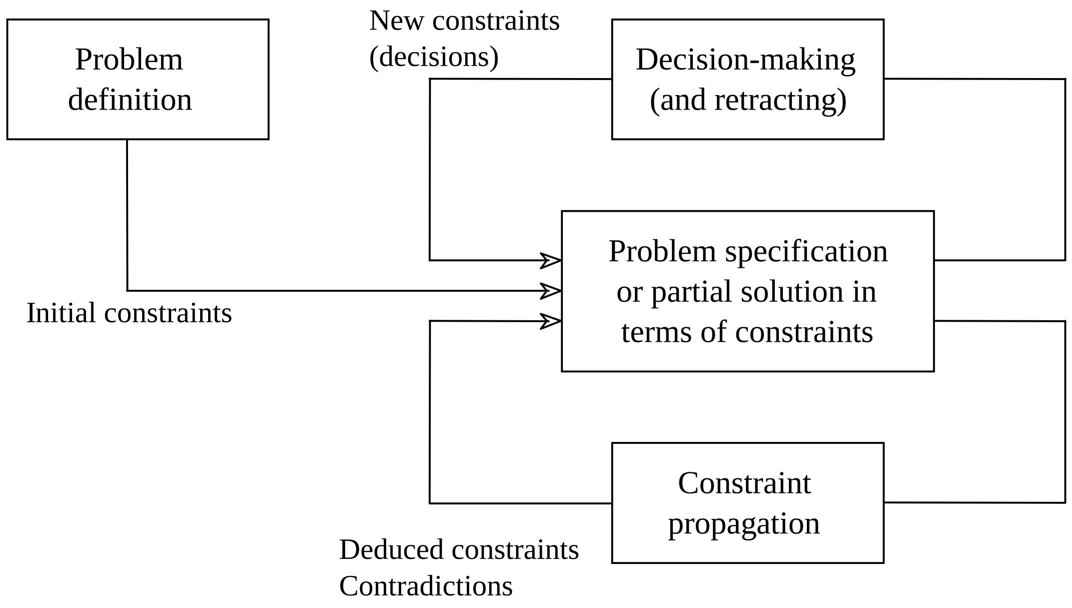

Problem definition
New constraints (decisions)
Decision-making (and retracting)
Problem specification or partial solution in terms of constraints
Initial constraints
Constraint propagation
Deduced constraints
Contradictions

Figure A-1. The behavior of a constraint programming system

The separation between constraint propagation and the other parts of the system is a key feature of constraint programming. It impacts a lot on the reusability of the constraint propagation algorithms in the several applications where similar constraints apply. This explains the success of commercial and public domain constraint programming packages such as ILOG SOLVER [Puget, 1994], [Puget and Leconte, 1995], CHIP [Aggoun and Beldiceanu, 1993], [Beldiceanu and Contejean, 1994], ECLIPSE, CLAIRE [Caseau and Laburthe, 1996b] and ECLAIR [Laburthe *et al.*, 1998]

A.2. Incorporating Efficient O.R. Algorithms in Constraint-Based Systems

It appeared in the past few years that the use of specific constraint propagation algorithms can drastically enhance the efficiency of constraint-based systems. Such algorithms are able to take into account a set of constraints from a "global" point of view, and can propagate them very efficiently. Let us consider for instance the so-called "all-different" constraint. It constrains a set of n variables to take pairwise distinct values. Such a constraint can be obviously propagated by maintaining arc-consistency on $n * (n - 1) / 2$ "local" constraints that state for any pair of variables $\{x, y\}$ that $x \neq y$. [Régin, 1994] describes an algorithm to achieve the global consistency of the "all-different" constraint. The constraint is modeled by a bi-partite graph. One of the sets is the set of the variables while the other one is the union of the domains. An edge between a variable and a value states that the given value is in the domain of the given variable. The constraint is consistent if and only if there is a matching whose cardinality is n. Régin uses an Operations Research algorithm to ensure the global consistency of the constraint. On top of that, an extension is proposed to achieve, with a reasonable algorithmic cost, the global arc-consistency of the constraint. The domains of the variables are filtered to remove the values that would make the constraint inconsistent. Another famous global constraint is the resource constraint that states that a set of activities has to execute on a single machine. [Nuijten, 1994], [Caseau and Laburthe, 1995], [Baptiste and Le Pape, 1995b], [Colombani, 1996] describe several constraint propagation algorithms for this constraint. All of them are based upon the work of Carlier and Pinson for the Job-Shop problem ([Carlier and Pinson, 1989]).

These algorithms, integrated in a constraint-based tool, allow any user to benefit from the efficiency of operations research techniques in a flexible framework (*e.g.*, [Baptiste *et al.*, 1995a]). Stated another way, on the one hand operations research offers efficient algorithms to solve problems that however might not be well suited to be used in practice, and on the other hand "classical" constraint propagation offers algorithms that are more generally applicable, but that might suffer from somewhat poor performance. Naturally, we want the best of both worlds, *i.e.*, we want efficient algorithms that we can apply to a wide range of problems.

A.3. Scheduling

Following the idea developed above, we tried to identify some of the (deterministic) scheduling areas where it could be worth to study in details global resource constraints. Partially based upon the scheduling problems we encountered in the industry, we introduce a simple (and necessarily incomplete) typology of scheduling. In pure scheduling problems (*e.g.*, job-shop scheduling), the capacity of each resource is defined over a number of time intervals and the problem consists of positioning resource-demanding activities over time, without ever exceeding the available capacity. In the following, we do not consider problems where a resource allocation dimension occurs. Three broad families are distinguished.

- In **non-preemptive** scheduling, activities cannot be interrupted. Each activity must execute without interruption from its start time to its end time. In **preemptive** scheduling, activities can be interrupted at any time, *e.g.*, to let some other activities execute.
- In **disjunctive** scheduling, each resource can execute at most one activity at a time. In **cumulative** scheduling, a resource can run several activities in parallel, provided that the resource capacity is not exceeded.
- Most often, resource constraints must be taken in the **strict** sense, *i.e.*, they can never be violated. For some problems, when the resource is overloaded, resource constraints can be taken in a broader sense: A limited number of activities can be **sub-contracted** to make the resource constraint consistent. The resource is then characterized by its overall capacity and by the number of activities it can sub-contract.

On top of that, several optimization criteria can be considered. The problem sometimes lies in finding a feasible schedule but most often a criteria has to be optimized. Although the minimization of the makespan, *i.e.*, the finishing time of the schedule, is commonly used, other criteria are sometimes of great practical interest (*e.g.*, the number of activities performed with given delays, the maximal or average tardiness or earliness, the peak or average resource utilization). Over the years, the theory and application of scheduling has grown into an important field of research, and an extensive body of literature exists on the subject. For more elaborate introductions to the theory of scheduling, we refer to [Baker, 1974], [Coffman, 1976], [French, 1982], [Carlier and Chrétienne, 1988], [GOThA, 1993] and [Brucker, 1995].

To represent this set of scheduling problems, we use a simple model based upon four entities: Activities, resources, temporal constraints and resource constraints.

A.3.1. *Representation of Activities and Resources*

A non-preemptive scheduling problem can be encoded efficiently as a constraint satisfaction problem: two variables, *start*(A_i) and *end*(A_i), are associated with each activity A_i; they represent the start time and the end time of A_i. The smallest values in the domains of *start*(A_i) and *end*(A_i) are called the release date and the earliest end time of A_i (r_i and eet_i). Similarly, the greatest values in the domains of *start*(A_i) and *end*(A_i) are called the latest start time and the deadline of A_i (lst_i and d_i). The processing time of the activity is an additional variable *processingTime*(A_i), that is constrained to be lower than or equal to the difference between the end and the start times of the activity (most often, processing time is known and bound to a value p_i).

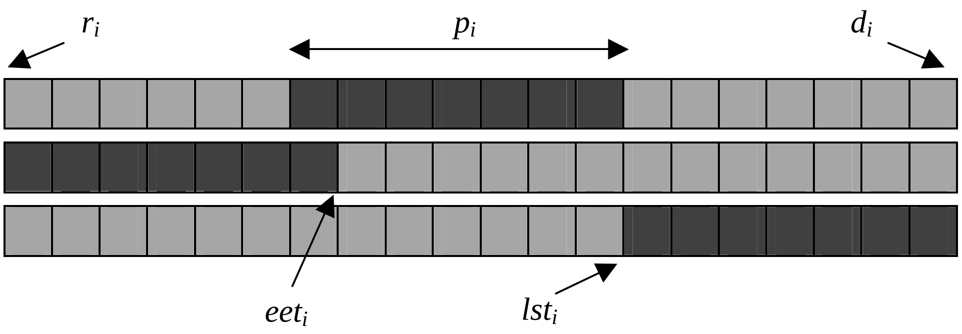


Figure A-2. The release date, the deadline, the processing time, the earliest end time and the latest start time of an activity (light gray is used to depict the time-window [r_i, d_i] of an activity and dark gray is used to represent the processing time of the activity).

A preemptive scheduling problem is more difficult to represent. One can either associate a set variable (*i.e.*, a variable the value of which will be a set) *set*(A_i) with each activity A_i, or define a 0-1 variable $W(A_i, t)$ for each activity A_i and time t; *set*(A_i) represents the set of times at which A_i executes, while $W(A_i, t)$ assumes value 1 if and only if A_i executes at time t. Ignoring implementation details, let us note that:

- the value of $W(A_i, t)$ is 1 if and only if t belongs to *set*(A_i).
- assuming time is discretized, *start*(A_i) and *end*(A_i) can be defined, in the preemptive case, by $start(A_i) = \min_{t \in set(A_i)}(t)$ and $end(A) = \max_{t \in set(A_i)}(t + 1)$; such variables are often needed to connect activities together by temporal constraints. Notice that in the non-preemptive case, $set(A_i) = [start(A_i), end(A_i))$, with the interval $[start(A_i), end(A_i))$ closed on the left and open on the right so that $|set(A_i)| = end(A_i) - start(A_i) = processingTime(A_i)$.

In the following, *capacity*(R) denotes the constrained variable used to represent the capacity of the resource R, the number of parallel identical machines that are available in R. To model resources with variable profile, we can also introduce *capacity*(R, t), the constrained variable that represents the capacity of the resource R available at time t. We note C_R the maximum capacity available, *i.e.*, $C_R = ub(capacity(R))$.
This model allows to represent a large variety of resource types. However, to handle the case of overloaded resources, we introduce another variable *reject*(R) that represents the number of activities among the n activities requiring the resource that can be sub-contracted. Most often, a single resource will be considered at a time. Hence, to simplify notations, the reference to R will be omitted.

A.3.2. Temporal and Resources-Constraints

Temporal relations between activities can be expressed by linear constraints between the start and end variables of activities. For instance, a precedence between two activities A_i, A_j is modeled by the linear constraint $end(A_i) \leq start(A_j)$. Such constraints can be easily propagated using a standard arc-B-consistency algorithm [Lhomme, 1993]. In addition, a variant of Ford's algorithm proposed in [Cesta and Oddi, 1996] is used to detect any inconsistency between precedence and processing time constraints, in time polynomial in the number of constraints (and independent of the domain sizes).
Resource constraints represent the fact that activities use some amount of resource throughout their execution. Given an activity A_i and a resource R, *capacity*(A_i, R) is the constrained variable that represents the amount of resource R required by activity A_i. $c_{i,R}$ is the minimal amount of the capacity of the resource required by the activity, *i.e.*, $c_{i,R} = lb(capacity(A_i, R))$. A resource constraint states that at any time t, the resource capacity is never exceeded by the sum of the resource requirements.

- In the non-preemptive case, this leads to

$$\forall R, \forall t \qquad \sum_{start(A_i) \leq t < end(A_i)} capacity(A_i, R) \leq capacity(R, t).$$

- In the preemptive case, this leads to

$$\forall R, \forall t \qquad \sum_{start(A_i) \leq t < end(A_i)} W(A_i, t) * capacity(A_i, R) \leq capacity(R, t).$$

Consider now the situation where the resource is overloaded. It is fairly natural to introduce an extra binary constrained variable $in(A_i, R)$ that states whether A_i is

- performed on the resource R, *i.e.*, $in(A_i, R) = 1$
- or subcontracted, *i.e.*, $in(A_i, R) = 0$.

When a single resource is considered, the index R will be omitted. The resource constraint simply states that on-time activities must satisfy a usual resource constraint.

- In the non-preemptive case, this leads to

$$\begin{cases} \forall R, \forall t, \quad \sum\limits_{start(A_i) \le t < end(A_i)} in(A_i, R) * capacity(A_i, R) \le capacity(R, t) \\ \sum in(A_i, R) \ge n - reject(R) \end{cases}$$

- In the preemptive case, this leads to

$$\begin{cases} \forall R, \forall t, \quad \sum\limits_{start(A_i) \le t < end(A_i)} in(A_i, R) * W(A_i, t) * capacity(A_i, R) \le capacity(R, t) \\ \sum in(A_i, R) \ge n - reject(R) \end{cases}$$

A.3.3. Modeling some Classical Scheduling Problems

We examine four well-known scheduling problems and we show how they can be represented within the model described above. The decision variant of each of these problems is NP-Complete in the strong sense [Garey and Johnson, 1979].

The Job-Shop Scheduling Problem (JSSP)

Instance. An instance of the decision variant of the Job-Shop Scheduling Problem is described by (1) a number m of machines, (2) a set of n jobs and (3) an overall deadline D. Each job J_l consists of a list L_l of activities. Each activity is given an integer processing time and a machine on which it has to be processed.

Question. The problem is to find a non-preemptive schedule, *i.e.*, an assignment of start times to activities such that (1) each machine executes one activity at a time (2) activities of the same job J_l are processed in the order induced by the list L_l and (3) all activities end before time D.

Model. Each machine of the Job-Shop Scheduling Problem is modeled by a non-preemptive disjunctive resource. Precedence constraints are imposed between activities of the same jobs. The overall deadline D is imposed to all activities.

The preemptive Job-Shop Scheduling Problem (PJSSP)

Instance. Same data as those for the JSSP.

Question. The problem is to find a preemptive schedule, *i.e.*, a set of execution times for each activity such that (1) each machine executes one activity at a time (2) activities of the same job J_l are processed in the order induced by the list L_l and (3) all activities end before time *D*.

Model. Each machine of the Job-Shop Scheduling Problem is modeled by a preemptive disjunctive resource. Precedence constraints are imposed between activities of the same jobs. The overall deadline *D* is imposed to all activities.

Surprisingly, the preemptive version of the job-shop is “harder” than the non-preemptive one. As shown in [Brucker *et al.*, 1999], if the number of machines is fixed and equals 2 and if the number of jobs is fixed and equals to 3, the preemptive problem is NP-hard while the corresponding non-preemptive one can be solved in polynomial time.

The Resource-Constrained Project Scheduling Problem (RCPSP)

Instance. An instance of the decision variant of the RCPSP consists of (1) a set of resources of given capacities, (2) a set of non-interruptible activities of given processing times, (3) an acyclic network of precedence constraints between the activities, (4) for each activity and each resource the amount of the resource required by the activity over its execution and (5) an overall deadline *D*.

Question. The problem is to find a start time assignment that satisfies the precedence and resource capacity constraints, and whose makespan (*i.e.*, the time at which all activities are completed) is at most *D*.

Model. Each resource of the instance is modeled by a cumulative resource. Following the structure of the network, precedence constraints are imposed between activities. The overall deadline *D* is imposed to all activities.

Minimizing the number of late jobs on a single machine ($1|r_j|\Sigma U_j$)

Instance. An instance of the decision-variant of this problem consists of a set of n jobs described by a release date r_i, a due-date[3] d_i and a processing time p_i ($r_i + p_i \leq d_i$) and an integer N.

Question. The problem is to find an assignment of start times to jobs such that (1) jobs do not overlap in time, (2) each job starts after its release date and (3) the number of jobs that end after their due-date is lower than or equal to N.

Model. Each job is modeled by an activity. Release dates, due dates and processing times are imposed. A non-preemptive disjunctive overloaded resource R is used to model the problem. Activities A_i such that $in(A_i, R) = 1$ are on-time, the other ones are late. The upper-bound of the constrained variable $reject(R)$ is set to N.

[3] The term deadline d_i is used when an activity has to execute before d_i (otherwise the schedule is not feasible). When activities can execute after d_i (in such a case these activities are late), the term due-date is more appropriate than deadline. To keep the same name "deadline" for d_i throughout the document, we make a slight misuse of language.

A.4. Summary of Results and Outline of the Thesis

The model that has been provided in the previous section is very general. An exhaustive study of all type of resource constraints would lead us to consider 8 types of resource constraints (preemptive *vs*. non-preemptive, disjunctive *vs*. cumulative, strict *vs*. overloaded). Up to now, we have only considered the following cases:

- **Disjunctive resource constraint in the non-preemptive case**. Following the work of [Nuijten, 1994], most of the constraint-based scheduling tools have integrated variants of the adjustments techniques of Carlier and Pinson, initially developed for the Job-Shop Scheduling Problem. The basic consists of deducing that some activities from a given set Ω must, can, or cannot, execute first (or last) in Ω. Such deductions lead to new ordering relations and new time-bounds, *i.e.*, strengthened release dates and deadlines of activities. Several algorithms have been proposed to compute all the possible adjustments due to deductions like "activity A_i must be the first (last) one to execute among activities in Ω". The deductive rules that allow to prove that an activity cannot be the first (last) one to execute have been less studied. We provide the first algorithm that is able to perform all the possible adjustments corresponding to this rule. It runs in $O(n^2)$.
- **Disjunctive resource constraint in the preemptive case**. To our knowledge, such constraints have never been studied. We propose several propagation algorithms. The first one is based upon a time-tabling technique, the second one on a disjunctive formulation of the problem, the third on a flow formulation of the problem and finally, we show that the adjustments of Carlier and Pinson can be generalized to the preemptive case.
- **Cumulative resource constraint**. In comparison with the disjunctive case, few work has been carried in the Operations Research community on deductive techniques for cumulative problems (except for the computation of lower-bound on some special cases). The constraint programming community has paid more attention to this constraint. Several attempts have been made to generalize the results obtained in the disjunctive case (*e.g.,* [Aggoun and Beldiceanu, 1993], [Nuijten, 1994], [Caseau and Laburthe, 1996a]). We tackle the cumulative resource constraint through a particular decision problem, namely the Cumulative Scheduling Problem (CuSP). We study two relaxations of this problem: A fully elastic relaxation, which can be seen as a preemptive one-machine problem, and a partially elastic relaxation. We also study a

set of necessary conditions based upon an energetic formulation [Lopez *et al.*, 1992]. Each time, we propose some algorithms that are able to verify that some necessary conditions of existence hold. On top of that we propose new algorithms to adjust release dates and deadlines. We compare our results to some lower-bounds of the literature for the m-machines problem, a special case of the CuSP. In particular, we show that the partially elastic relaxation is equivalent to the subset bound [Perregaard, 1995], itself as good as the lower bound obtained by Carlier and Pinson through a pseudo-preemptive relaxation of the m-machine Problem [Carlier and Pinson, 1996]. We also show that energetic reasoning strictly dominates all the other techniques cited above.

- **Disjunctive and overloaded resource constraint**. As far as we know, such constraints have never been studied in the constraint programming community. A large amount of work has been carried, in the Operations Research field, on the minimization of the number of late jobs on a single machine. Several particular case are known to be solvable in polynomial time. In the appendices, we describe strongly polynomial algorithms for two open problems that consist in minimizing in the preemptive and in the non-preemptive case the weighted number of late jobs with release dates and deadlines.

  We propose several techniques to propagate the overloaded resource constraint. First, we study its preemptive relaxation. We propose an $O(n^4)$ dynamic programming algorithm for this problem; which improves the time and space complexities of a previous algorithm of [Lawler, 1990]. We also propose a weaker relaxation that gives a (weaker) bound which can be computed in a quadratic amount of time. On top of that, this relaxation is the basis of an adjustment scheme that is able to deduce that some activities have to be performed on the resource while some others have to be sub-contracted.

Chapter B is dedicated to the study of these different constraint propagation algorithms. To evaluate from an experimental point of view the efficiency of these algorithms, we describe in Chapter C some branching schemes with constraint propagation to solve some classical scheduling problems. The Preemptive Job-Shop Scheduling Problem, the Resource Constrained Project Scheduling Problem and the problem of the minimization of the number of late jobs are studied. On top of the propagation, we use several dominance properties that allow to drastically reduce the search space. Experimental results are provided for each problem and allow us to compare our approach to some well known exact approaches of the literature.

- As far as we know, no exact approach has been proposed to solve the Preemptive Job-Shop Scheduling Problem. We use a chronological branching scheme and we apply a dominance property that imposes that the premptive schedules of each machine “look like” Jackson Preemptive Schedules. This branching scheme, combined with the adjustments that extend the work of Carlier and Pinson to the preemptive case, proves to be efficient since all the 10*10 benchmark instances from the literature are solved.

- We propose several variants of the same branching scheme for the RCPSP. We show that, these variants perform more or less well, depending on the type of instances. On highly disjunctive instances (*i.e.*, on instances that have a strong disjunctive dimension), our procedure does not perform as well as other branch and bound procedures based upon the dominance rule of [Demeulemeester and Herroelen, 1995]. However, on highly cumulative instances (*i.e.*, on instances for which the disjunctive dimension is not very important), we obtain very promising results, that have been confirmed on the multi-processor Flow-Shop, a special case of the RCPSP [Néron *et al*., 1998].
- Finally, we propose a branch and bound procedure to minimize the number of late jobs on a single machine. Our branching scheme simply consists of deciding whether a given job is late or on-time. At each node, we check that there is a feasible schedule of the jobs that have to be on-time (this is an NP-hard problem, however, it is very well solved thanks to a variant of the procedure of [Carlier, 1982]). On top of that, we use a strong dominance property. Previous exact approaches ([Dauzère-Pérès, 1995]) relying on MIP formulation could not consider instances with more than 10 jobs because of the size of the MIP. Our procedure is able to solve 90% of the 100-jobs instances in less than one hour. Moreover, our results compare very well to the very recent branch and bound of [Dauzère-Pérès and Sevaux, 1998b].

We draw some conclusions in Chapter D and we give some promising research directions.

Chapter B. Propagation of Resource Constraints

The propagation of resource constraints is a purely deductive process that allows to deduce inconsistencies and to tighten the characteristics of activities and resources. In the simplest case, the release dates and the deadlines of activities are updated. When preemption is allowed, modifications of earliest end times and latest start times also apply. When resources are overloaded, the propagation process does not only aim at adjusting the time-windows of activities but also at deducing automatically that some activities have to be processed on the resource or have to be subcontracted.

In this chapter, we study four resource constraints that correspond to the disjunctive non-preemptive case (Section B.1), the disjunctive preemptive case (Section B.2), the cumulative case (Section B.3) and the disjunctive overloaded case (Section B.4). In the last two cases, some remarks concerning the preemptive variant of the resource constraint will be provided. Each time, several propagation algorithms are described and compared from a theoretical point of view.

B.1. The Non-Preemptive Disjunctive Case

In the following sections, we study several methods to propagate the non-preemptive disjunctive resource constraint. A set of n non-interruptible activities $\{A_1, ..., A_n\}$ require the same resource of capacity 1. This resource constraint is an exact transposition of the decision variant of the One-Machine Problem [Garey and Johnson, 1979]: Is there a feasible schedule, *i.e.*, a start-time assignment such that activities are scheduled between their release dates and their deadlines and such that they do not overlap in time? Given an instance of this problem, our aim is (1) to detect some cases in which we can prove that there is no feasible schedule and (2) to adjust release dates and deadlines of the activities. The adjustments consist of removing from the domain of each activity A_i values t for which we can prove that there is no feasible schedule on which A_i starts at t.

First we consider the simple Time-Table mechanism, widely used in constraint based scheduling tools, that allows to propagate the resource constraint in an incremental fashion. We then consider the disjunctive constraint that compares the temporal characteristics of pairs of activities. In the third part, we describe the edge-finding propagation technique, which has been shown to be extremely efficient for solving disjunctive problems like the Job-Shop problem. In the last section of this chapter, we present a mechanism that extends the basic edge-finding mechanism and that allows to make some additional deductions. For this mechanism, known as Not-First / Not-Last, we propose a quadratic algorithm that overcomes, in terms of complexity, the previous known algorithmic results.

B.1.1. Time-Table Constraint

A simple mechanism to propagate resource constraints in the non-preemptive case relies on an explicit data structure called "timetable" to maintain information about resource utilization and resource availability over time. Resource constraints are propagated in two directions: from resources to activities, to update activity time bounds (release dates and deadlines) according to the availability of resources; and from activities to resources, to update the timetables according to the time bounds of activities. Although several variants exist [Le Pape, 1988], [Fox, 1990], [Le Pape, 1995], [Smith, 1994], [Caseau and Laburthe, 1996a], [Lock, 1996], the propagation mainly consists of maintaining arc-B-consistency [Lhomme, 1993] on the formula:

$$\sum_i [W(A_i, t) * capacity(A_i)] \leq capacity(t)$$

where *capacity*(A_i) denotes the capacity of the resource required by activity A_i, *capacity*(t) denotes the capacity available at time t, and $W(A_i, t)$ is an implicit 0-1 variable representing the Boolean value $W(A_i, t) = (start(A_i) \leq t) \wedge (t < end(A_i))$.

Before Propagation	r_i	d_i	p_i
A_1	0	3	2
A_2	0	4	2
Propagation 1	r_i	d_i	p_i
A_1	0	3	2
A_2	2	4	2
Propagation 2	r_i	d_i	p_i
A_1	0	2	2
A_2	2	4	2

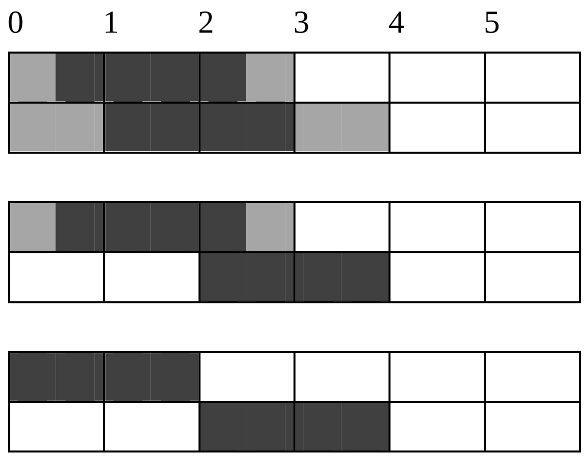


Figure B-1. Propagation of the timetable constraint.

Example.
Figure B-1 displays two activities A_1 and A_2 which require the same resource of capacity 1. The latest start time (d_1 - p_1 = 1) of A_1 is smaller than its earliest end time ($r_1 + p_1 = 2$). Hence, it is guaranteed that A_1 will execute between 1 and 2. Over this period, $W(A_1, t)$ is set to 1 and the corresponding resource amount is no longer available for A_2. Since A_2 cannot be interrupted and cannot be finished before 1, the release date of A_2 is updated to 2 (propagation 1). Then, $W(A_2, t)$ is set to 1 over the interval [2, 4), which results in a new propagation step, where the deadline of A_1 is set to 2 (propagation 2).

B.1.2. Disjunctive Constraint Propagation

In non-preemptive disjunctive scheduling, two activities A_i and A_j which require a common resource R cannot overlap in time: either A_i precedes A_j or A_j precedes A_i. If n activities $A_1 \ldots A_n$ require R, the resource constraint can be implemented as $n*(n - 1) / 2$ (explicit or implicit) disjunctive constraints. As for timetable constraints, variants exist in the literature [Erschler, 1976], [Carlier, 1984], [Esquirol, 1987], [Le Pape, 1988], [Smith and Cheng, 1993], [Varnier *et al.*, 1993], [Baptiste and Le Pape, 1995a], but in most cases the propagation consists of maintaining arc-B-consistency on the formula

$$(end(A_i) \leq start(A_j)) \vee (end(A_j) \leq start(A_i)).$$

Enforcing arc-B-Consistency on this formula is done as follows: Whenever the smallest possible value of *end*(A_i) (earliest end time of A_i) exceeds the greatest possible value of *start*(A_j) (latest start time of A_j), A_i cannot precede A_j; hence A_j must precede A_i; the time-bounds of A_i and A_j are consequently updated with respect to the new temporal constraint *end*(A_j) $\leq$ *start*(A_i). Similarly, when the earliest possible end time of A_j exceeds the latest possible start time of A_i, A_j cannot precede A_i. When neither of the two activities can precede the other, a contradiction is detected.

Disjunctive constraints provide more precise time bounds than the corresponding timetable constraints. Indeed, if an activity A_j is known to execute at some time t between the release date r_i and the earliest end time $r_i + p_i$ of A_i, then the first disjunct of the above formula is false and thus, A_j must precede A_i and the propagation of the disjunctive constraint implies *start*(A_i) $\geq r_j + p_j > t$.

The following example shows that, in some cases, disjunctive constraints propagate more than time-table constraints.

Before Propagation	r_i	d_i	p_i
A_1	0	4	2
A_2	1	5	2
Propagation	r_i	d_i	p_i
A_1	0	3	2
A_2	2	5	2

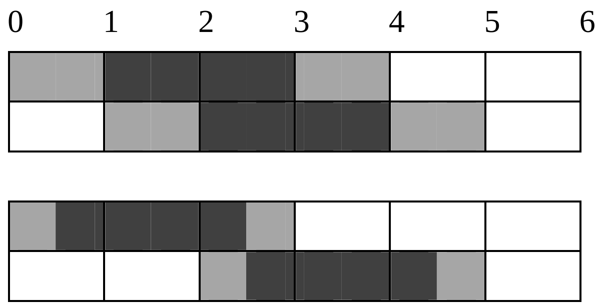


Figure B-2. Propagation of the disjunctive constraint (non-preemptive case)

Example.

Figure B-2 displays two activities A_1 and A_2 which require the same resource of capacity 1. The earliest end time of each activity does not exceed its latest start time, so the timetable constraint cannot deduce anything. On the contrary, the propagation of the disjunctive constraint imposes *end*(A_1) $\leq$ *start*(A_2) which, in turn, results in updating both d_1 and r_2.

B.1.3. Edge-finding

The term “edge-finding” is often used in non-preemptive disjunctive scheduling [Applegate and Cook, 1991]. It denotes both a “branching” and a “bounding” technique. The branching technique consists of ordering activities that require the same resource. At each node, a set of activities Ω is selected and, for each activity A_i in Ω, a new branch is created where A_i is constrained to execute first (or last) among the activities in Ω. The bounding technique consists of deducing that some activities from a given set Ω must, can, or cannot, execute first (or last) in Ω. Such deductions lead to new ordering relations ("edges" in the graph representing the possible orderings of activities) and new time-bounds, *i.e.*, strengthened release dates and deadlines of activities.

In the following, let r_Ω denote the smallest of the release dates of the activities in Ω, let d_Ω be the greatest of the deadlines of the activities in Ω, and let p_Ω be the sum of the minimal processing times of the activities in Ω. Let A_i « A_j (A_i » A_j) mean that A_i executes before (after) A_j and A_i « Ω (A_i » Ω) mean that A_i executes before (after) all the activities in Ω. Once again, variants exist [Pinson, 1988], [Carlier and Pinson, 1990], [Carlier and Pinson, 1994], [Caseau and Laburthe, 1994], [Nuijten, 1994], [Brucker and Thiele, 1996], [Lévy, 1996], [Martin and Shmoys, 1996], [Péridy, 1996] but the following rules capture the "essence" of the edge-finding bounding technique:

$$[\forall\Omega, \forall A_i \notin \Omega, d_{\Omega\cup\{Ai\}} - r_\Omega < p_\Omega + p_i] \Rightarrow A_i \ll \Omega$$

$$[\forall\Omega, \forall A_i \notin \Omega, d_\Omega - r_{\Omega\cup\{Ai\}} < p_\Omega + p_i] \Rightarrow A_i \gg \Omega$$

$$A_i \ll \Omega \Rightarrow [end(A_i) \le \min_{\Omega'\subseteq\Omega} (d_{\Omega'} - p_{\Omega'})]$$

$$A_i \gg \Omega \Rightarrow [start(A_i) \ge \max_{\Omega'\subseteq\Omega} (r_{\Omega'} + p_{\Omega'})]$$

If n activities require the resource, there are a priori $O(n * 2^n)$ pairs (A, Ω) to consider. An algorithm that performs all the time-bound adjustments in $O(n^2)$ is presented in [Carlier and Pinson, 1990]. It consists of a "primal" algorithm to update release dates and a "dual" algorithm to update deadlines. The primal algorithm runs as follows:

- Compute "Jackson's preemptive schedule" (JPS) for the resource under consideration. JPS is the preemptive schedule obtained by applying the following priority rule: whenever the resource is free and one activity is available, schedule the activity A_i for which d_i is the smallest. If an activity A_j becomes available while A_i is in process, stop A_i and start A_j if d_j is strictly smaller than d_i; otherwise continue A_i.
- For each activity A_i, compute the set Ψ of the activities which are not finished at $t = r_i$ on JPS. Let p_j* be the residual processing time on the JPS of the activity A_j at time t. Take the activities of Ψ in decreasing order of due dates and select the first activity A_k such that:

$$r_i + p_i + \sum_{A_j \in \Psi - \{A_i\} \mid d_j \le d_k} (p_j*) > d_k$$

If such an activity A_k exists, then post the following constraints:

$$A_i \gg \{A_j \in \Psi - \{A_i\} \mid d_j \le d_k\}$$

$$start(A_i) \ge \max_{A_j \in \Psi - \{A_i\} \mid d_j \le d_k} (JPS(A_j))$$

where $JPS(A_j)$ is the completion time of activity A_j in JPS.

Example.

Figure B-3 presents the JPS of a resource of capacity 1 required by 3 activities. On this example, the edge-finding propagation algorithm deduces *start*(A_i) ≥ 8, when the timetable and the disjunctive constraint propagation algorithms deduce nothing.

	r_i	d_i	p_i
A_1	0	17	6
A_2	1	11	4
A_3	1	11	3

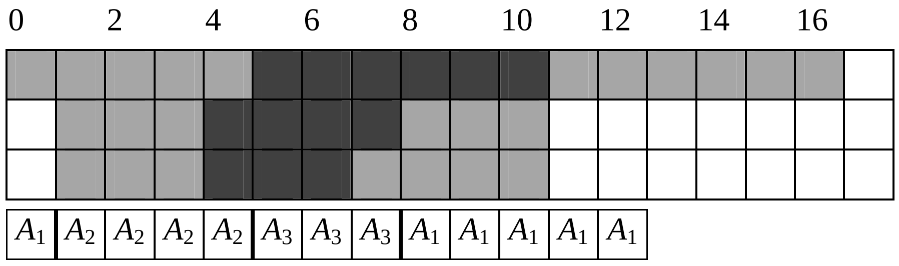

0
2
4
6
8
10
12
14
16
Schedule
A_1 A_2 A_2 A_2 A_2 A_3 A_3 A_3 A_1 A_1 A_1 A_1 A_1

Figure B-3. The JPS of 3 activities.

[Nuijten, 1994] and [Martin and Shmoys, 1996] present variants of this algorithm, which also run in $O(n^2)$, but do not require the computation of Jackson's preemptive schedule. [Carlier and Pinson, 1994] presents another variant, which runs in $O(n*log(n))$ but requires much more complex data structures. [Caseau and Laburthe, 1994] presents another variant, based on the explicit definition of "task intervals." This variant runs in $O(n^3)$ in the worst case, but works in an incremental fashion and allows the performance of additional deductions (*cf.*, Section B.1.4). Finally, [Brucker and Thiele, 1996] proposes several extensions to take setup times into account. [Baptiste, 1995] and [Martin and Shmoys, 1996] establish an interesting property of the edge-finding technique: considering only the resource constraint and the current time bounds of activities, the algorithm computes the smallest release date at which each activity A_i could start if all the other activities were interruptible.

As shown in [Lévy, 1996], the edge-finding algorithms above may perform different deductions from the more standard disjunctive constraint propagation algorithms. Examples given in [Lévy, 1996] show that each of the two techniques (edge-finding and disjunctive constraint propagation) performs some deductions that the other technique does not perform. Examples given in [Baptiste, 1995] show that the same result applies to the edge-finding rules and the energetic reasoning rules of [Erschler *et al.*, 1991]. In practice, an edge-finding algorithm is often coupled with a disjunctive constraint propagation algorithm to allow a maximal amount of constraint propagation to take place.

B.1.4. *Not-First, Not-Last*[4]

The algorithm presented in the preceding section mostly focuses on determining whether an activity A_i **must** execute before (or after) a set of activities Ω requiring the same resource. A natural complement consists of determining whether A_i **can** execute before (or after) Ω. In the non-preemptive disjunctive case, this leads to the following rules [Pinson, 1988], [Carlier and Pinson, 1990], [Caseau and Laburthe, 1994]:

$$\forall\Omega, \forall A_i \notin \Omega, d_\Omega - r_i < p_\Omega + p_i \Rightarrow \neg(A_i \ll \Omega)$$
$$\forall\Omega, \forall A_i \notin \Omega, d_i - r_\Omega < p_\Omega + p_i \Rightarrow \neg(A_i \gg \Omega)$$
$$\neg(A_i \ll \Omega) \Rightarrow start(A_i) \geq \min_{A_j \in \Omega} (r_j + p_j)$$
$$\neg(A_i \gg \Omega) \Rightarrow end(A_i) \leq \max_{A_j \in \Omega} (d_j - p_j)$$

The problem which consists of performing all the time-bound adjustments corresponding to the first and third rules can be called the “not-first” problem, since it consists of updating the release date of every activity A_i which cannot be first to execute in a set $\Omega \cup \{A_i\}$. Similarly, the problem which consists of performing all the time-bound adjustments corresponding to the second and fourth rules can be called the “not-last” problem (it consists of updating the deadline of every activity A_i which cannot be last to execute in a set $\Omega \cup \{A_i\}$).
Most researchers who have been working on edge-finding techniques have considered the “not-first” and “not-last” rules above [Pinson, 1988], [Carlier and Pinson, 1990], [Caseau and Laburthe, 1994], [Nuijten, 1994], [Baptiste and Le Pape, 1995b], [Lévy, 1996], but in the absence of low-polynomial algorithms for solving the complete “not-first” problem, the rules had to be applied in an incomplete way, allowing only **some** but not all of the possible time-bound adjustments. In this section, we present an $O(n^2)$ time and $O(n)$ space algorithm to solve the “not-first” problem. The “not-last” problem is solved in a symmetric fashion. To our knowledge, this is the first reported algorithm to perform all the deductions allowed by the rules above in quadratic time.

Let us first introduce some assumptions and notations. We assume that the relation $r_i + p_i \leq d_i$ holds for every activity A_i. Otherwise, the scheduling problem clearly allows no solution and the constraint propagation process can stop. We also assume that the activities A_1, ..., A_n which require the resource under consideration are sorted in non-decreasing order of deadlines (this can be done in $O(n * \log(n))$ time). Hence, $i \leq j$ implies $d_i \leq d_j$. For a given j and a given k, $\Omega(j, k)$ denotes the set of indices m in $\{1 ... k\}$ such that

[4] Most of the results presented in this section come from [Baptiste and Le Pape, 1996b].

$r_j + p_j \leq r_m + p_m$ and $\Omega(i, j, k)$ denotes $\Omega(j, k) - \{i\}$. Hence, if i does not belong to $\Omega(j, k)$, $\Omega(i, j, k)$ is equal to $\Omega(j, k)$. Let $S_{j,k} = p_{\Omega(j,k)}$ if $j \leq k$ and $S_{j,k} = -\infty$ otherwise. Let $\delta_{j,k} = \min_{l \leq k} (d_l - S_{j,l})$.

Proposition B-1.
For a given j, the values $\delta_{j,1}, ..., \delta_{j,n}$ can be computed in $O(n)$ time.

Proof.
Indeed, the values of $S_{j,k}$ and $\delta_{j,k}$ can be computed in constant time from the values of $S_{j,k-1}$ and $\delta_{j,k-1}$. One just has to test whether k verifies $r_j + p_j \leq r_k + p_k$ or not. □

Proposition B-2.
If the “not-first” rules applied to activity A_i and set Ω allow to update the release date of A_i to $r_j + p_j$ then there exists an index $k \geq j$ such that the “not-first” rules applied to activity A_i and set $\Omega(i, j, k)$ allow to update the release date of A_i to $r_j + p_j$.

Proof.
Let k be the maximal index of the activities in Ω. Ω is included in $\Omega(i, j, k)$ and d_Ω is equal to $d_{\Omega(i,j,k)}$. Hence the rules can be applied to A_i and $\Omega(i, j, k)$ and provide the conclusion that A_i cannot start before $r_j + p_j$ since every m in $\Omega(i, j, k)$ satisfies $r_j + p_j \leq r_m + p_m$. □

Proposition B-3.
Let i and j be such that $r_i + p_i < r_j + p_j$. In this case, the “not-first” rules allow to update the release date of A_i to $r_j + p_j$ if and only if $r_i + p_i > \delta_{j,n}$.

Proof.
Necessary condition.

Let us assume that the rules allow to update the release date of A_i to $r_j + p_j$. According to Proposition B-2, there exists $k \geq j$ such that $\Omega(i, j, k)$ is not empty and $d_k - r_i < p_{\Omega(i,j,k)} + p_i$. Since $r_i + p_i < r_j + p_j$ implies that i does not belong to $\Omega(j, k)$, this implies $r_i + p_i > d_k - S_{j,k} \geq \delta_{j,n}$.

Sufficient condition.

Let us assume that $r_i + p_i > \delta_{j,n}$. $\delta_{j,n}$ is finite, so there exists an index $k \geq j$ such that $\delta_{j,n} = d_k - S_{j,k}$. Since i does not belong to $\Omega(j, k)$, we have $d_k - r_i < p_{\Omega(i,j,k)} + p_i$. So, the rules allow to update the release date of A_i to the value $r_j + p_j$. □

Proposition B-4.
Let i and j be such that $r_i + p_i \geq r_j + p_j$. In this case, the "not-first" rules allow to update the release date of A_i to $r_j + p_j$ if and only if either $r_i + p_i > \delta_{j,\, i\text{-}1}$ or $r_i > \delta_{j,\, n}$.

Proof.
Necessary condition.

Let us assume that the rules allow to update the release date of A_i to $r_j + p_j$. According to Proposition B-2, there exists $k \geq j$ such that $\Omega(i, j, k)$ is not empty and $d_k - r_i < p_{\Omega(i, j, k)} + p_i$. Two cases, $k < i$ and $i < k$, can be distinguished. If $k < i$, i does not belong to $\Omega(j, k)$. This implies that $r_i + p_i > d_k - S_{j,\, k} \geq \delta_{j,\, i\text{-}1}$. On the contrary, if $i < k$, i belongs to $\Omega(j, k)$. Then $p_{\Omega(j, k)} = p_{\Omega(i, j, k)} + p_i$ and $r_i > d_k – S_{j,k} \geq \delta_{j,n}$.

Sufficient condition.

If $r_i + p_i > \delta_{j,\, i\text{-}1}$, $\delta_{j,\, i\text{-}1}$ is finite, so there exists an index k, not greater than i, such that $\delta_{j,\, i\text{-}1} = d_k - S_{j,\, k}$. Since i does not belong to $\Omega(j, k)$, we have $d_k - r_i < p_{\Omega(i, j, k)} + p_i$. So, the rules allow to update the release date of A_i to $r_j + p_j$. Let us now assume that $r_i + p_i \leq \delta_{j,\, i\text{-}1}$ and $r_i > \delta_{j,\, n}$. Then there exists an index $k \geq j$ such that $\delta_{j,\, n} = d_k – S_{j,\, k}$. Note that $k \geq i$ (otherwise, $\delta_{j,\, n} = \delta_{j,\, i\text{-}1} < r_i$ contradicts $r_i + p_i \leq \delta_{j,\, i\text{-}1}$). Consequently, i belongs to $\Omega(j, k)$. In addition, $\Omega(j, k)$ is not reduced to $\{i\}$, otherwise we would have $r_i > \delta_{j,\, n} = d_k – S_{j,\, k} = d_k – p_i \geq d_i – p_i$ which contradicts the initial assumption that $r_i + p_i \leq d_i$ for all i. Hence, $\Omega(i,j,k) \neq \emptyset$ satisfies $d_k – r_i < p_{\Omega(i,j,k)} + p_i$. So, the rules allow to update the release date of A_i to $r_j + p_j$. □

We now introduce Algorithm B-1 that performs the same time-bound adjustments as the "not-first" rules.

Algorithm B-1.

```
For $i = 1$ To $i = n$
  $r_i' = r_i$
End For
For $j = 1$ To $j = n$
  compute $delta_{j,1}, ..., delta_{j,n}$
  For $i = 1$ To $i = n$
    If $r_i + p_i < r_j + p_j$ Then
      If $r_i + p_i > delta_{j,n}$ Then
        $r_i' = max(r_i', r_j + p_j)$
      End If
    Else
      If $r_i + p_i > delta_{j,i-1}$ or $r_i > delta_{j,n}$ Then
        $r_i' = max(r_i', r_j + p_j)$
      End If
    End If
  End For
End For
For $i = 1$ To $i = n$
  $r_i = r_i'$
End For
```

Proposition B-5.
Algorithm B-1 performs the same time-bound adjustments as the "not-first" rules. It runs in $O(n^2)$ time and $O(n)$ space.

Proof.
Propositions B-3 and B-4 imply that the algorithm performs exactly the deductions implied by the rules. Thanks to the introduction of the variables `ri'`, one does not need to resort activities inside the loops. The algorithm runs in $O(n^2)$ steps since for each j in the outer loop, $O(n)$ steps are required to compute $\delta_{j,1} \dots \delta_{j,n}$ and for each i in the inner loop, $O(1)$ steps are required to perform the relevant tests. In addition, the algorithm requires a linear amount of memory space since only the values $\delta_{j,1} \dots \delta_{j,n}$ for a given j are required. □

Let us note that when the processing times of activities are fixed, the "not-first" and "not-last" rules subsume the disjunctive constraint propagation technique mentioned in Section B.2.2. Hence, no disjunctive constraint propagation algorithm is needed when the "not-first" algorithm above and its dual "not-last" algorithm are applied.

B.2. The Preemptive Disjunctive Case, the Mixed Case[5]

In the following sections, we study several methods to propagate the preemptive disjunctive resource constraint. A set of n interruptible activities $\{A_1, ..., A_n\}$ require the same resource of capacity 1. This resource constraint can be seen as the decision variant of the preemptive One-Machine Problem: Is there a preemptive feasible schedule, *i.e.*, an assignment of execution times such that activities are scheduled between their release dates and their deadlines, and such that they do not overlap in time? Like for the non-preemptive case, our aim is to detect some cases in which we can prove that there is no feasible schedule, and to tighten the temporal characteristics of each activity.
It is well-known that the preemptive One-Machine Problem can be solved in polynomial time by Jackson's algorithm. However, we consider this problem as a relaxation of another more complex problem (*e.g.*, the preemptive Job-Shop Scheduling Problem) and it is of great interest to study some algorithms that are able to tighten the temporal characteristics of activities.
We study several methods to propagate the preemptive one-machine resource constraint. First, we show that both the Time-Table mechanism and the disjunctive constraint can be extended. We then propose a resource-constraint based on network-flows. In the last section, we present a mechanism that extends the edge-finding mechanism. In particular, we show that this last resource constraint propagation scheme is able to handle the mixed case, *i.e.*, the case in which both interruptible and non-interruptible activities are mixed.

B.2.1. Time-Table Constraint

At the first glance, it seems that the main principle of the timetable mechanism directly applies to both the preemptive and the mixed case. However, an important difference appears in the relation between the five variables $W(A_i, t)$, $set(A_i)$, $start(A_i)$, $end(A_i)$, and $processingTime(A_i)$. The earliest start time r_i can easily be set to "the first time t at which $W(A_i, t)$ can be 1." Similarly, the deadline d_i can easily be set to 1 + "the last time t at which $W(A_i, t)$ can be 1." However, the earliest end time eet_i must be computed so that there possibly exist $processingTime(A_i)$ time points in $set(A_i) \cap [r_i, eet_i)$, and the latest start time lst_i must be computed so that there possibly exist $processingTime(A_i)$ time points in

[5] Most of the results presented in this section come from [Baptiste, 1995], [Le Pape and Baptiste, 1996] and [Le Pape and Baptiste, 1998a].

set(A_i) ∩ [lst_i, d_i). These additional propagation steps make the overall propagation process far more complex.

In the reverse direction, it is important to notice that $W(A_i, t)$ cannot be set to 1 as soon as $lst_i \leq t < eet_i$. The only situation in which $W(A_i, t)$ can be deduced to be 1 is when no more than *processingTime*(A_i) time points can possibly belong to *set*(A_i). This is unlikely to occur before decisions (choices in a search tree) are made to instantiate *set*(A_i). Therefore, constraint propagation cannot prune much.

Before Prop.	r_i	eet_i	lst_i	d_i	p_i
A_1 (non-int.)	0	2	1	3	2
A_2 (int.)	0	2	2	4	2
Prop. 1	r_i	eet_i	lst_i	d_i	p_i
A_1 (non-int.)	0	2	1	3	2
A_2 (int.)	0	3	2	4	2

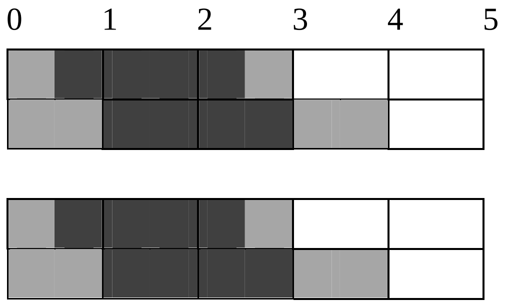


Figure B-4. Propagation of the time-table constraint (mixed case)

Example.

Given the data of Figure B-1, the timetable mechanism cannot deduce anything if both activities can be interrupted. Figure B-4 shows what happens when only A_2 can be interrupted. As in Figure B-1, it is guaranteed that A_1 will execute between $lst_1 = 1$ and $eet_1 = 2$. Over this period, the corresponding resource amount is no longer available for A_2. The earliest end time of A_2 is then set to 3. Then the propagation process stops since there is no time point at which A_2 is guaranteed to execute.

Both costly and in most cases ineffective, the timetable mechanism appears far from satisfactorily applicable to preemptive problems. Several researchers incorporated *some* possibilities of preemption in their constraint-based algorithms or applications (*e*.g., interrupt a machining operation in favor of a planned machine maintenance but not in favor of another machining operation, interrupt an activity at most once or twice [Zweben *et al.*, 1993], [Smith, 1994], [Le Pape, 1996], [Pegman *et al.*, 1997]). Few did attack the general problem of preemptive and mixed scheduling. [Demeulemeester, 1992] presents a branch and bound algorithm for a particular preemptive cumulative scheduling problem: the preemptive Resource-Constrained Project Scheduling Problem. This algorithm relies on memorizing states rather than on constraint propagation to prune the search space. [Baptiste, 1994] reports on a tentative implementation, in ILOG SOLVER [Puget, 1994], [Puget and Leconte, 1995], of preemptive time-table constraints. The reported results confirm that even on simple problems the propagation process is rather slow. Let us note, however, that the timetable mechanism can easily be generalized to other types of resources, such as resources of capacity $m > 1$, or resources which must be in a specific state for an activity to execute [Le Pape, 1994]. Such is not the case for the techniques described in the following sections.

B.2.2. Disjunctive Constraint Propagation

In the preemptive disjunctive case, the fact that activities A_i and A_j cannot overlap is most naturally represented by two alternative formulas:

$$set(A_i) \cap set(A_j) = \emptyset \text{ or } \forall t, [W(A_i, t) = 0 \text{ or } W(A_j, t) = 0].$$

When one of these formulas is adopted, the preemptive disjunctive constraints and the corresponding preemptive timetable constraints deduce the same time bounds. However, a simple rewriting of the non-preemptive disjunctive constraint

$$\begin{aligned} & [start(A_i) + processingTime(A_i) \leq end(A_j) - processingTime(A_j)] \\ \text{or } & [start(A_j) + processingTime(A_j) \leq end(A_i) - processingTime(A_i)] \end{aligned}$$

suggests an additional preemptive disjunctive constraint:

$$\begin{aligned} & [start(A_i) + processingTime(A_i) + processingTime(A_j) \leq end(A_i)] \\ \text{or } & [start(A_i) + processingTime(A_i) + processingTime(A_j) \leq end(A_j)] \\ \text{or } & [start(A_j) + processingTime(A_i) + processingTime(A_j) \leq end(A_i)] \\ \text{or } & [start(A_j) + processingTime(A_i) + processingTime(A_j) \leq end(A_j)] \end{aligned}$$

which can serve as a complement to $set(A_i) \cap set(A_j) = \emptyset$. Arc-B-Consistency is achieved on this additional constraint. Note that in the mixed case, the first (fourth) disjunct can be removed from the disjunction if A_i (respectively, A_j) cannot be interrupted.

Before Prop.	r_i	eet_i	lst_i	d_i	p_i
A_1 (int.)	0	4	2	6	4
A_2 (int.)	2	3	3	4	1
Prop. 1	r_i	eet_i	lst_i	d_i	p_i
A_1 (int.)	0	5	1	6	4
A_2 (int.)	2	3	3	4	1

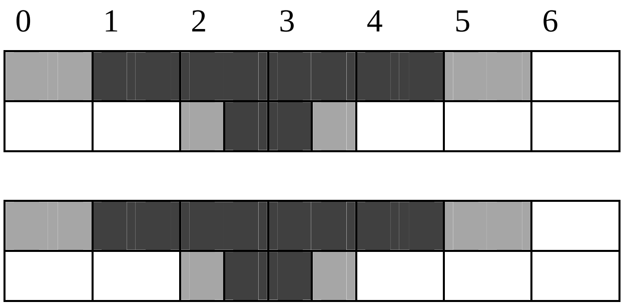


Figure B-5. Propagation of the disjunctive constraint (preemptive case)

Example.

In the example of Figure B-5, the propagation of the redundant constraint provides $start(A_1) \leq 1$ and $end(A_1) \geq 5$.

B.2.3. Network-Flow based Constraints

[Régin, 1994] describes an algorithm, based on matching theory, to achieve the global consistency of the “all-different” constraint. This constraint is defined on a set of variables and constrains these variables to assume pairwise distinct values. Régin’s algorithm maintains arc-consistency on the n-ary “all-different” constraint, which is shown to be more powerful than achieving arc-consistency for the $n*(n - 1) / 2$ corresponding binary “different” constraints.

Basically, Régin’s algorithm consists of building a bipartite graph $G(X, Y, E)$ where X is a set of vertices corresponding to the variables of the “all-different” constraint, Y is a set of vertices corresponding to the possible values of these variables, and E is a set of edges (x, y), $x \in X$, $y \in Y$, such that $(x, y) \in E$ if and only if y is a possible value for x. As a result, the “all-different” constraint is satisfiable if and only if there exists a 0-1 function f on E such that:

$$\forall x \in X, \Sigma_{(x, y) \in E} f(x, y) = 1$$
$$\forall y \in Y, \Sigma_{(x, y) \in E} f(x, y) \leq 1$$

In addition, a given value y_j is a possible value for a given variable x_i if and only if there exists a 0-1 function f_{ij} such that:

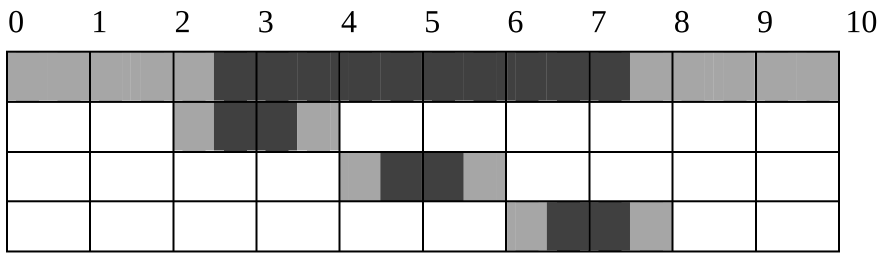
$$\forall x \in X, \Sigma_{(x, y) \in E} f_{ij}(x, y) = 1$$
$$\forall y \in Y, \Sigma_{(x, y) \in E} f_{ij}(x, y) \leq 1$$
$$f_{ij}(x_i, y_j) = 1$$

The problem of finding such a function (flow) f or f_{ij} can be solved in polynomial time. In addition, the current value of f can be used to generate f_{ij} at low cost, and to compute the new value of f when the domain of a variable changes. See [Régin, 1994], [Régin, 1995], [Régin, 1996] for details and extensions.

Notice that when all activities have unitary processing times, Régin’s algorithm can be directly applied. In the preemptive case, this can be generalized to activities of arbitrary processing times by seeing each activity A_i as *processingTime*(A_i) sub-activities of unitary processing times 1. Then, each sub-activity has to pick a value (the time at which the sub-activity executes) and the values of the sub-activities that require a given resource have to be pairwise distinct. However, under this naive formulation, both the number of variables and the number of values would be too high (dependent on the sum of the processing times of the activities) for practical use. This led us to another formulation where the nodes x in X correspond to activities, and the nodes y in Y correspond to a partition of the time horizon in n disjoint intervals $I_1 = [s_1, e_1) \dots I_n = [s_n, e_n)$ such that $[s_1, e_n)$ represents the complete time horizon, $e_i = s_{i+1}$ $(1 \le i < n)$, and $\{s_1, ..., s_n, e_n\}$ includes all the time points at which the information available about $W(A, t)$ changes (Figure B-6 illustrates this formulation on a small example). In particular, $\{s_1, ..., s_n, e_n\}$ includes all the earliest start times and latest end times of activities, but it can also include bounds of intervals over which $W(A, t)$ is constrained to be true or false (in this sense, the flow model is more general than preemptive edge-finding described in B.2.4, but it does not generalize to the mixed case). E is defined as the set of pairs (x, y) such that activity x can execute during interval y. The maximal capacity $c_{max}(x, y)$ of edge (x, y) is set to $length(y)$, and the minimal capacity $c_{min}(x, y)$ of edge (x, y) is set to $length(y)$ if x is constrained to execute over y and to 0 otherwise. As a result, the preemptive resource constraint is satisfiable if and only if there exists a function f on E such that:

$$\forall x \in X, \Sigma_{(x, y) \in E} f(x, y) = processingTime(x)$$
$$\forall y \in Y, \Sigma_{(x, y) \in E} f(x, y) \le length(y)$$
$$\forall e \in E, c_{min}(e) \le f(e) \le c_{max}(e)$$

	r_i	d_i	p_i
A_1	0	10	5
A_2	2	4	1
A_3	4	6	1
A_4	6	8	1



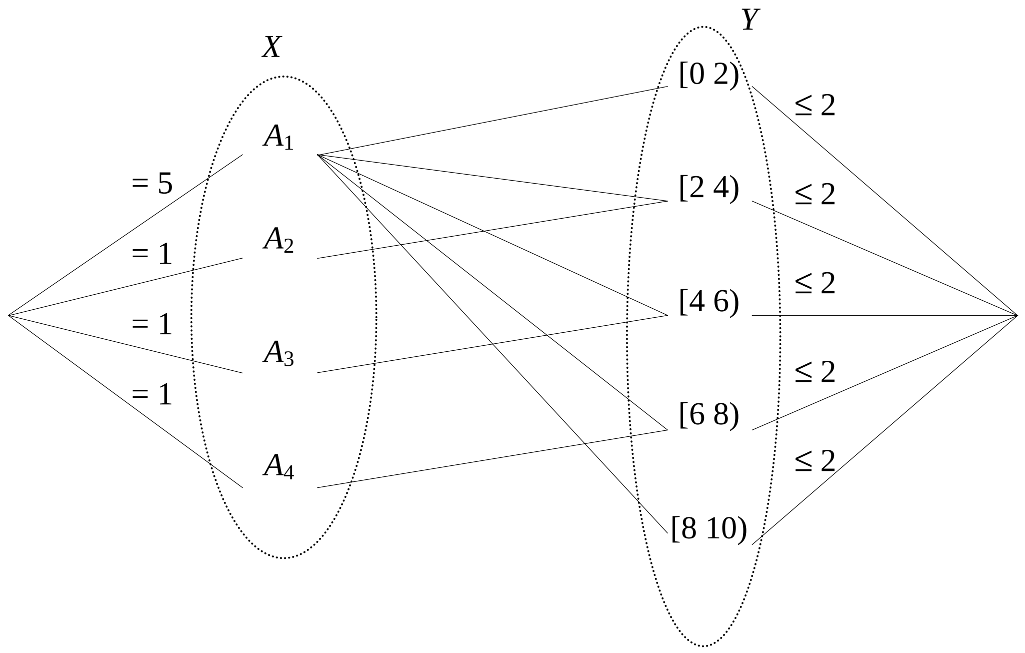


Figure B-6. A particular network flow. Arrows have been omitted, they all go from left to right.

Similar models are commonly used in Operations Research. For example, [Federgruen and Groenevelt, 1986] use a more general model to solve particular polynomial scheduling problems with multiple parallel resources operating at different speeds. Following [Régin, 1994], what we propose below is to use network flow techniques, not only to find solutions to polynomial sub-problems, but also to update the domains of the variables.
[Baptiste, 1995] provides two algorithms for the search of a compatible flow f (SCF). The first algorithm uses Herz's algorithm, as described in [Gondran and Minoux, 1995], to construct the compatible flow, starting from $f(x, y) = 0$ for all x and all y. It runs in $O(|X| * |Y| * \Sigma_{x \in X}\, processingTime(x))$. The second algorithm builds a variant of Jackson's preemptive schedule which respects the intervals during which activities are required to execute. This can be done in $O(|Y| * log(|Y|))$. This schedule is then used as an initial (possibly incompatible) flow, repaired by Herz's algorithm in $O(|X| * |Y| * F)$, where F denotes the sum, over the activities, of the sizes of the intervals included in $[r_i, d_i]$ during which the activity A_i is not allowed to execute (for reasons that are not directly related to the use of the resource by other activities).
To reduce variable domains, the most natural generalization of Régin's algorithm consists of varying $c_{min}(e)$ and $c_{max}(e)$ for each edge e in turn. The following algorithm updates the minimal flow $c_{min}(x, y)$ that can pass through an edge (x, y). The maximal flow $c_{max}(x, y)$ is obtained in a similar fashion.

```
• Set $u = c_{min}(x, y)$ and $v = c_{max}(x, y)$.
• While $(u \neq v)$
•     Set $w = \lfloor (u + v) / 2 \rfloor$
•     Search for a compatible flow $f$ with $f(x, y) \leq w$.
•     If such a flow $f$ exists, set $v = w$, otherwise set $u = w + 1$.
• Set $c_{min}(x, y) = u$.
```

It is proven in [Baptiste, 1995] that this adjustment of edge capacities (AEC) can be done for all edges (x, y) in $O(|X|^2 * |Y| * H)$, where H denotes the overall time horizon $e_n - s_1$. This complexity is reached by systematically reusing the previous flow as a start point when computing the flow f with the new constraint $f(x, y) \leq w$.
Then the following rules can be applied:

$$c_{max}(x, y) = 0 \quad \Rightarrow \quad \forall t \in y,\ W(x, t) = 0$$
$$c_{min}(x, y) = length(y) \quad \Rightarrow \quad \forall t \in y,\ W(x, t) = 1$$
$$c_{min}(x, [s_i\ e_i]) \neq 0 \quad \Rightarrow \quad [start(x) \leq e_i - c_{min}(x, [s_i\ e_i])]$$
$$c_{min}(x, [s_i\ e_i]) \neq 0 \quad \Rightarrow \quad [end(x) \geq s_i + c_{min}(x, [s_i\ e_i])]$$

However, SCF and AEC are not sufficient to determine the best possible time bounds for activities. Let us consider, for example, the four activities A_1, A_2, A_3, A_4 defined on Figure B-6. In this case, $c_{min}(A_1, I)$ remains equal to 0 for all I; yet A_1 cannot start after 3 and cannot end before 7. However, the flow model can be used to compute the best possible earliest end times. First, given x and the intervals $y_1 \dots y_n$ (sorted in reverse chronological order) to which x is connected, one can find the maximal integer k such that there exists a compatible flow f with $f(x, y_i) = 0$ for $1 \leq i < k$. Then, one can compute the minimal flow $f_{min}(x, y_k)$ through (x, y_k), under the constraints $f(x, y_i) = 0$ for $1 \leq i < k$. Under these conditions, $end(x) \geq s_k + f_{min}(x, [s_k, e_k])$ provides the best possible earliest end time for x. It is shown in [Baptiste, 1995] that this global update of time bounds (GUTB) can be done for all activities x in $O(|X|^2 * |Y| * H)$. As for AEC, this complexity is reached by systematically reusing the previous flow as a start point for computing the new flow when an additional capacity constraint is added.
Let us remark that the incrementality of Herz's algorithm is a key factor for both the worst-case and the practical complexity of SCF, AEC and GUTB. Of course, strongly polynomial algorithms (with complexity independent of the schedule duration) could also be used for the search of a compatible flow [Gondran and Minoux, 1995].

B.2.4. Edge-Finding

The edge-finding algorithm detailed in Section B.1.3 can be extended to take into account the preemptive case and also the mixed case (*i.e.*, the case where interruptible and non-interruptible activities are mixed). As mentioned in Section B.1.3., [Baptiste, 1995] and [Martin and Shmoys, 1996] have established an interesting property of the non-preemptive edge-finding technique. Considering only the resource constraint and the current time bounds of activities, the algorithm computes the earliest start time at which each activity A_i could start if all the other activities were interruptible. This suggests a logical extension of the technique to preemptive and mixed cases: for each activity A_i requiring the resource, if A_i is not interruptible, the non-preemptive edge-finding bound applies; if A_i is interruptible then, considering only the resource constraint and the current time bounds, it would be nice to determine the earliest start and end times between which A_i could execute if all the activities were interruptible.
Let us define $\rangle\rangle$ so that $A_i \rangle\rangle \Omega$ means "A_i ends after all activities in Ω " and substitute $\rangle\rangle$ for » in the rules of the primal algorithm.

$$\forall \Omega, \forall A_i \notin \Omega, [d_\Omega - r_{\Omega \cup \{Ai\}} < p_\Omega + p_i] \Rightarrow A_i \rangle\rangle \Omega$$
$$A_i \rangle\rangle \Omega \Rightarrow [start(A_i) \geq \max_{\Omega' \subseteq \Omega} (r_{\Omega'} + p_{\Omega'})]$$

When A_i cannot be interrupted, these two rules remain valid (even if other activities can be interrupted) and the adjustment of r_i is the same as in the non-preemptive case. When A_i can be interrupted, the first rule is still valid but the second is not. However, the second rule can be replaced by a weaker one:

$$A_i \rangle\rangle \Omega \Rightarrow [end(A_i) \geq \max_{\Omega' \subseteq \Omega} (r_{\Omega' \cup \{Ai\}} + p_{\Omega' \cup \{Ai\}})]$$

This leads to a more general primal edge-finding algorithm:

- Compute “Jackson’s preemptive schedule” for the resource under consideration.
- For each activity A_i, compute the set Ψ of the activities which are not finished at $t = r_i$ on JPS. Let p_j* be the residual processing time on the JPS of the activity A_j at time t. Take the activities of Ψ in decreasing order of deadlines and select the first activity A_k such that:

  $$r_i + p_i + \sum_{A_j \in \Psi - \{A_i\} \mid d_j \leq d_k} (p_j^*) > d_k$$

  If such an activity A_k exists, then post the following constraints:
  - $A_k \rangle\rangle \{A_j \in \Psi - \{A_i\} \mid d_j \leq d_k\}$
  - $start(A_i) \geq \max_{A_j \in \Psi - \{A_i\} \mid d_j \leq d_k} (JPS(A_j))$ if A_i cannot be interrupted
  - $end(A_i) \geq r_i + p_i + \sum_{A_j \in \Psi - \{A_i\} \mid d_j \leq d_k} (p_j^*)$ if A_i can be interrupted

Example.
In the example of Figure B-3, the algorithm above deduces $start(A_1) \geq 8$ if A_1 cannot be interrupted. It deduces $end(A_1) \geq 13$ if A_1 can be interrupted.
It is proven in [Baptiste, 1995] that considering only the resource constraint and the current time bounds of activities, this algorithm computes,

- when A_i is not interruptible: the earliest time at which A_i could start if all the other activities were interruptible.
- when A_i is interruptible: the earliest time at which A_i could end if all the other activities were interruptible.

Nuijten’s edge-finding algorithm can be modified in a similar fashion. The following algorithm B-2 is equivalent to the algorithm sketched above. We assume that activities are sorted in increasing order of release dates.

Algorithm B-2.

```
1  For k = 1 To k = n
2    P = 0, C = $-\infty$, H = $-\infty$,
3    For i = n Down To i = 1
4      If $d_i \le d_k$ Then
5        P = P + $p_i$
6        C = max(C, $r_i$ + P)
7        If C > $d_k$ Then
8          there is no feasible schedule, exit
9        End If
10     End If
11     $C_i$ = C
12   End For
13   For i = 1 to i = n
14     If $d_i \le d_k$ Then
15       H = max(H, $r_i + p_i$)
16       P = P - $p_i$
17     Else
18       If $r_i$ + P + $p_i$ > $d_k$ Then
19         If $A_i$ can be interrupted Then
20           $eet_i$ = max($eet_i$, $r_i$ + P + $p_i$)
21         Else
22           $r_i$ = max($r_i$, $C_i$)
23         End If
24       End If
25       If H + $p_i$ > $d_k$ Then
26         If $A_i$ can be interrupted Then
27           $eet_i$ = max($eet_i$, H + $p_i$)
28         Else
29           $r_i$ = max($r_i$, C)
30         End If
31       End If
32     End If
33   End For
34 End For
```

Release dates and earliest end times are adjusted inside the inner loop (lines 20, 27 and 22, 29). Actually, the adjusted values should be stored and applied at the end to avoid the resorting of the activities (*cf.*, Algorithm B.1.). The proof that this algorithm is equivalent to the JPS-based algorithm follows the proof of Nuijten's algorithm in [Nuijten, 1994].

First, if A_i cannot be interrupted, the new algorithm makes the same conclusions as Nuijten's algorithm, so the proof in [Nuijten, 1994] applies to the new algorithm. Let us now assume that A_i can be interrupted. It is proven in [Baptiste, 1995] that the earliest time at which A_i could end if all the other activities could be interrupted is equal to the maximal value of $r_{\Omega\cup\{Ai\}} + p_{\Omega\cup\{Ai\}}$ for Ω triggering the edge-finding rules. The earliest end times computed by the new algorithm are, when they are used, equal to $r_{\Omega\cup\{Ai\}} + p_{\Omega\cup\{Ai\}}$ for such Ω. To prove that the best possible bound is reached, consider the two cases distinguished in [Nuijten, 1994]: if all activities A_u in Ω are such that $i < u$ then, either Ω or a superset of Ω is detected by the first test ($r_i + P + p_i > d_k$); if some activity A_u of Ω is such that $u < i$ then, either Ω or a superset of Ω is detected by the second test ($H + p_i > d_k$). In both cases, a bound greater than or equal to $r_{\Omega\cup\{A\}} + p_{\Omega\cup\{A\}}$ is found.
This algorithm can be further improved:

- When A_i can be interrupted and *set*(A_i) is known to contain a series of time intervals $I_1 \dots I_m$, A_i can be replaced by $(m + 1)$ activities $A_i^1, \dots, A_i^m, A_i'$, with each A_i^l forced to execute over I_l and A_i' with the same release date and deadline as A_i and a processing time equal to $(p_i - \Sigma_{1 \leq l \leq m}\ length(I_l))$; where $length(I_l)$ denotes the length of the interval I_l.
- When A_i can be interrupted and either $(r_i + P = d_k)$ or $(H = d_k)$ in the course of the algorithm, it is certain that A_i cannot start before d_k. Hence, the algorithm can also be used to update the release date of interruptible activities.

Remark.
When activities have fixed processing times, the computation and the use of C_i and C to compute $\max_{\Omega'\subseteq\Omega}(r_{\Omega'} + p_{\Omega'})$ serves only to avoid repeated iterations of the algorithm. Indeed, suppose a purely preemptive edge-finding algorithm is used and suppose A_i is not interruptible. The purely preemptive edge-finding algorithm uses the following rules:

$$\forall\Omega, \forall A \notin \Omega, [d_\Omega - r_{\Omega\cup\{Ai\}} < p_\Omega + p_i] \Rightarrow A_i \rangle\rangle \Omega$$

$$A_i \rangle\rangle \Omega \Rightarrow [end(A_i) \geq \max_{\Omega'\subseteq\Omega}(r_{\Omega'\cup\{Ai\}} + p_{\Omega'\cup\{Ai\}})]$$

When constraint propagation stops, the earliest end time of A_i is set to a value eet_i such that if all activities were interruptible, there would be a schedule S of the resource such that (1) A_i does not start before r_i and (2) A_i ends at eet_i. If the processing time of A_i is fixed, the propagation of the constraint $start(A_i) + processingTime(A_i) = end(A_i)$ guarantees that when constraint propagation stops $r_i + p_i = eet_i$. Consequently, A_i is not interrupted in S, which implies that the non-preemptive edge-finding algorithm cannot find a better bound for r_i.

B.3. The Cumulative Case[6]

Our aim is to extend the results obtained on the One-Machine Problem to the cumulative case. The Cumulative Scheduling Problem (CuSP) is the framework on which we are going to work. An instance of the CuSP consists of (1) one resource with a given capacity C and (2) a set of n activities $\{A_1, ..., A_n\}$, together with a release date r_i, a deadline d_i, a processing time p_i, and a resource capacity requirement c_i for each activity A_i. We assume that all data are integers and that $\forall i, r_i + p_i \leq d_i$ and $c_i \leq C$. The problem is to decide whether there exists a feasible schedule, *i.e.*, a start time assignment that satisfies all timing constraints and the resource constraint. The CuSP obviously belongs to NP. It is an extension of the decision variant of both the One-Machine Problem ($C = 1$, $c_i = 1$) and the m-Machine Problem ($C = m$, $c_i = 1$). Thus it is NP-complete in the strong sense [Garey and Johnson, 1979].

The CuSP can be seen as a relaxation of the decision variant of the well-known Resource-Constrained Project Scheduling Problem (RCPSP) [Garey and Johnson, 1979]: Given an instance of the RCPSP, release dates and deadlines of activities can be derived from the network of precedence constraints and from the overall deadline (for example using Ford's algorithm [Gondran and Minoux, 1995]). The CuSP is then the relaxation in which precedence constraints are relaxed and where a single resource is considered at a time.

As mentioned in the previous sections, a large amount of work has been carried out on the One-Machine Problem. Similarly, lower bounds have been developed for the optimization variant of the m-Machine Problem (*e.g.*, [Carlier and Pinson, 1996], [Perregaard, 1995]). Obviously, these lower bounds can be seen as necessary conditions of existence for the decision variant of the m-Machine Problem. As far as we know, no specific algorithm for adjusting release dates and deadlines has been proposed. On the CuSP itself, little work has been done. Constraint propagation algorithms have been developed to adjust time-bounds of activities (*e.g.*, [Caseau and Laburthe, 1996a], [Lopez *et al.*, 1992], [Nuijten, 1994], [Nuijten and Aarts, 1996]), but they tend to be less uniformly effective than the algorithms available for the One-Machine Problem.

[6] Most of the results presented in this section come from [Baptiste and Le Pape, 1997b] and [Baptiste *et. al.*, 1998b].

In the following sections, we study three necessary conditions of existence of a feasible schedule for the CuSP.

- The first necessary condition is based on the resolution of the Fully Elastic CuSP, a relaxation of the CuSP (*cf.* Figure B-7). An instance of the Fully Elastic CuSP is described by the same data as an instance of the CuSP. The problem is to decide whether there exists a feasible fully elastic schedule, *i.e.*, an integer function $fes(t, i)$ representing the number of units of the resource assigned to A_i over the interval $[t, t + 1)$, such that:

$$\forall i, \forall t \notin [r_i, d_i), fes(t, i) = 0$$
$$\forall i, \textstyle\sum_t fes(t, i) = p_i * c_i$$
$$\forall t, \textstyle\sum_i fes(t, i) \leq C$$

- The second necessary condition is based on the resolution of the Partially Elastic CuSP, a tighter relaxation of the CuSP (*cf.* Figure B-7). An instance of the Partially Elastic CuSP is described by the same data as an instance of the CuSP. The problem is to decide whether there exists a feasible partially elastic schedule, *i.e.*, an integer function $pes(t, i)$ such that:

$$\forall i, \forall t \notin [r_i, d_i), pes(t, i) = 0$$
$$\forall i, \textstyle\sum_t pes(t, i) = p_i * c_i$$
$$\forall t, \textstyle\sum_i pes(t, i) \leq C$$
$$\forall i, \forall t \in [r_i, d_i), \Sigma_{x<t}\, pes(x, i) \leq c_i * (t - r_i)$$
$$\forall i, \forall t \in [r_i, d_i), \Sigma_{t\leq x}\, pes(x, i) \leq c_i * (d_i - t)$$

- The third necessary condition, called the "left-shift / right-shift" necessary condition, is not based on a well-identified relaxation of the CuSP but on energetic reasoning as defined in [Erschler *et al.*, 1991], [Lopez *et al.*, 1992].

For each of these necessary conditions, we propose a polynomial algorithm, running in $O(n * log(n))$ for the fully elastic condition, and in $O(n^2)$ for the partially elastic and the left-shift / right-shift conditions. In the particular case of the m-Machine Problem, these necessary conditions can be theoretically compared with other results from the literature. The subset bound [Perregaard, 1995] (seen as a necessary condition) is equivalent to the partially elastic relaxation and the left-shift / right-shift necessary condition is strictly stronger than the partially elastic relaxation.

We also propose three time-bound adjustment schemes for the CuSP.

- The first one is based on the fully elastic relaxation. An $O(n^2)$ algorithm is described.
- The second one is based on the partially elastic relaxation. An $O(n^2 * \log(|\{c_i\}|))$ algorithm is described (where $|\{c_i\}|$ is the number of distinct resource capacity requirements).

- The third one is based on the left-shift / right-shift necessary condition. An $O(n^3)$ algorithm is described.

	r_i	d_i	p_i	c_i
A_1	0	10	8	2

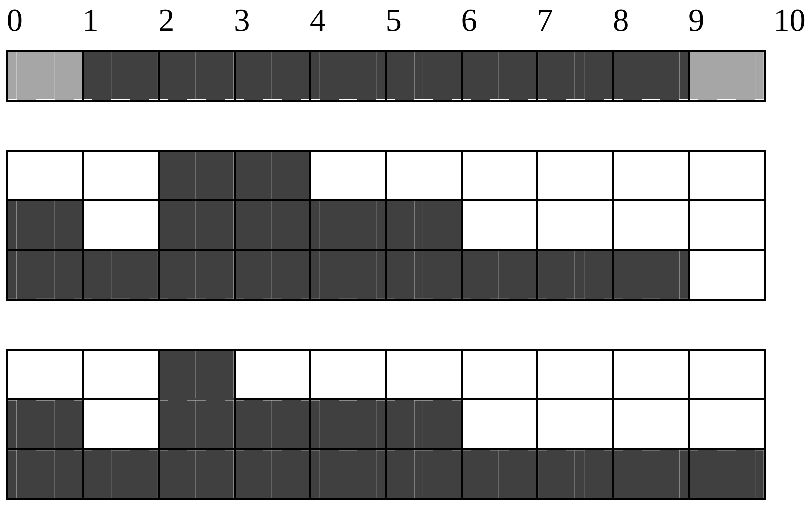


Figure B-7. Consider a resource of capacity 3 and an activity with release date 0, deadline 10, processing time 8 and resource requirement 2. Both Gantt charts correspond to feasible fully elastic schedules. The first one is not a feasible partially elastic schedule. Indeed, 9 units of the resource are used in [0, 4), which is more than 2∗(4-0). The second one is a feasible partially elastic schedule.

B.3.1. Necessary Conditions for the Existence of a Feasible Schedule

B.3.1.1. A Necessary and Sufficient Condition of existence for the Fully Elastic CuSP

We exhibit in this section a strong link between the Fully Elastic CuSP and the decision variant of the Preemptive One-Machine Problem. An instance of the Preemptive One-Machine Problem is defined by a set of n activities $\{A_1, ..., A_n\}$, together with a release date r_i, a deadline d_i and a processing time p_i for each activity A_i. The problem is to decide whether there exists a feasible preemptive one-machine schedule of the given activities.

Transformation F.
For any instance I of the Fully Elastic CuSP, let $F(I)$ be the instance of the Preemptive One-Machine Problem defined by n activities $A_1', ..., A_n'$ with $\forall i$, $r_i' = C * r_i$, $d_i' = C * d_i$, $p_i' = p_i * c_i$.

Proposition B-6
For any instance I of the Fully Elastic CuSP, there exists a feasible fully elastic schedule of I if and only if there exists a feasible preemptive schedule of $F(I)$.

Proof.
Let C be the capacity of the resource R of the instance I. Let R' be the resource of the instance $F(I)$. We first prove that if there is a feasible fully elastic schedule of I, then there is a feasible preemptive schedule of $F(I)$. Let $fes(t, i)$ be the number of units of A_i executed at t. We build a schedule of A_1', ..., A_n' on R' as follows. For each time t and each activity A_i, schedule $fes(t, i)$ units of A_i' on R' as early as possible after time $C * t$. It is obvious that at any time t, for any activity A_i, 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, ..., A_n into slices of one time unit and rescheduling these slices on R'. Consequently, for any activity A_i', exactly $p_i * c_i$ units of A_i' are scheduled between $C * r_i$ and $C * d_i$ and thus the release dates as well as deadlines are met. A symmetric demonstration would prove that if there is a feasible preemptive schedule of $F(I)$ then there is a feasible fully elastic schedule of I. □

Consider now Jackson's Preemptive Schedule. JPS is feasible if and only if there exists a feasible preemptive schedule. Moreover, JPS can be built in $O(n * \log(n))$ steps (see [Carlier, 1984] for details). Consequently, thanks to Proposition B-6, we have an $O(n * log(n))$ algorithm to solve the Fully Elastic CuSP. In the following, Jackson's Fully Elastic Schedule (JFES) denotes the fully elastic schedule obtained (1) by applying JPS on the transformed instance and (2) by rescheduling slices as described in the proof of Proposition B-6.

B.3.1.2. A Necessary and Sufficient Condition of existence for the Partially Elastic CuSP

The Partially Elastic CuSP is slightly more complex. We first introduce a pseudo-polynomial algorithm to solve this problem. We then present the concept of required energy consumption, which enables us to show that the Partially Elastic CuSP is equivalent to another problem for which we can provide a quadratic algorithm. In the following, "I" denotes an instance of the Partially Elastic CuSP. Let us first introduce a new transformation.

Transformation G.
Consider the instance $G(I)$ of the Fully Elastic CuSP defined by replacing every activity A_i by p_i activities $A_i^1, \dots, A_i^{p_i}$, each having a resource requirement $c_i^j = c_i$, a release date $r_i^j = r_i + j - 1$, a deadline $d_i^j = d_i - (p_i - j)$ and a processing time p_i^j of 1 (the resource capacity of $G(I)$ is C as for I).

B.3.1.2.1. Jackson's Partially Elastic Schedule

Jackson's Partially Elastic Schedule (JPES) is the schedule built by scheduling each activity A_i at the time points at which the activities A_i^j are scheduled on JFES of $G(I)$. Given the definition of G, it is easy to verify that if JFES is a feasible fully elastic schedule of $G(I)$ then JPES is a feasible partially elastic schedule of I.

Proposition B-7
There exists a feasible partially elastic schedule if and only if JPES is a feasible partially elastic schedule.

Proof (sketch).
Consider a feasible partially elastic schedule S of an instance I. It is then possible to build a feasible fully elastic schedule of $G(I)$ obtained from S by a similar transformation as G (*i.e.*, for any activity A_i, schedule A_i^1 at the same place as the "first c_i units" of A_i on S, iterate ...). Since there is a feasible fully elastic schedule of $G(I)$, JFES is also a feasible fully elastic schedule of $G(I)$ (Proposition B-6). Thus, JPES is a feasible partially elastic schedule of I. ☐
Since transformation G is done in $O(\Sigma\, p_i)$ and since the fully elastic problem $G(I)$ can be solved in $O(\Sigma\, p_i * \log(\Sigma\, p_i))$, Proposition B-7 leads to an $O(\Sigma\, p_i * \log(\Sigma\, p_i))$ algorithm to test the existence of a feasible partially elastic schedule.

B.3.1.2.2. Energetic Reasoning

We adapt the notion of "required energy consumption" defined in [Lopez, 1991] and [Lopez *et al.*, 1992] to partially elastic activities. The required energy consumption $W_{PE}(A_i, t_1, t_2)$ of an activity over an interval $[t_1, t_2]$ is defined as follows (*cf.* Figure B-8).

$$W_{PE}(A_i, t_1, t_2) = c_i * \max(0, p_i - \max(0, t_1 - r_i) - \max(0, d_i - t_2))$$

To get an intuitive picture of the formula, notice that $\max(0, t_1 - r_i)$ is an upper bound of the number of time units during which A_i can execute before time t_1 and $\max(0, d_i - t_2)$ is an upper bound of the number of time units during which A_i can execute after time t_2. Consequently, $\max(0, p_i - \max(0, t_1 - r_i) - \max(0, d_i - t_2))$ is a lower bound of the number of time units during which A_i executes in $[t_1, t_2]$.

	r_i	d_i	p_i	c_i
A_1	0	10	7	2

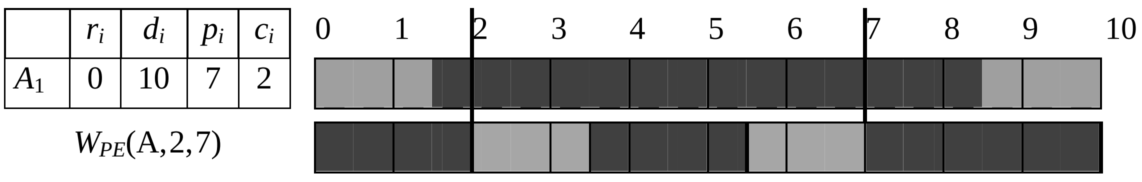


Figure B-8. The required energy consumption of A_1 over [2,7]. A_1 must execute during at least 2 time units in [2, 7]; i.e., $W_{PE}(A,2,7)=2*(7-(2-0)-(10-7))=4$

We now define the overall required energy consumption $W_{PE}(t_1, t_2)$ over $[t_1, t_2]$ as the sum over all activities A_i of $W_{PE}(A_i, t_1, t_2)$. Note that for $t_1 = t_2$, $W_{PE}(t_1, t_2)$ is defined and, under the assumption $r_i + p_i \le d_i$, is equal to 0.

Proposition B-8.
There is a feasible partially elastic schedule of I if and only if for any non-empty interval $[t_1, t_2]$, $W_{PE}(t_1, t_2) \le C * (t_2 - t_1)$.

Proof.
The fact that $W_{PE}(t_1, t_2) \le C * (t_2 - t_1)$ is a necessary condition is obvious. Suppose now that there is no feasible partially elastic schedule of I. Then there is no feasible preemptive schedule of $F(G(I))$. Consequently, there is a set of activities S of $F(G(I))$ such that between the minimum release date of activities in S and the maximum deadline of activities in S there is not enough "space" to schedule all activities in S [Carlier, 1984]. This leads to

$$\min_{A_i^j \in S} r_i^j + \sum_{A_i^j \in S} p_i^j > \max_{A_i^j \in S} d_i^j \Rightarrow \sum_{\left\{\substack{r_i^j \ge C*t_1 \\ d_i^j \le C*t_2}\right.} p_i^j > C*(t_2 - t_1)$$

where $C * t_1$ is the minimum release date in S and $C * t_2$ the maximum deadline in S (recall that release dates and deadlines of activities in S are multiple of C). Then the equation becomes:

$$\sum_i \sum_{\substack{j \text{ in } [1, p_i] \text{ such that:} \\ C*(r_i + j - 1) \ge C*t_1 \\ C*(d_i - p_i + j) \le C*t_2}} c_i > C*(t_2 - t_1)$$

For each i, let us now count the values of j in $[1, p_i]$ such that (1) $C * (r_i + j - 1) \ge C * t_1$ and (2) $C * (d_i - p_i + j) \le C * t_2$, *i.e.*, the number of integers in $[max(1, t_1 + 1 - r_i), min(p_i, t_2 + p_i - d_i)]$. This number is equal to:

$$\begin{aligned} &\max(0, 1 + \min(p_i, t_2 + p_i - d_i) - \max(1, t_1 + 1 - r_i)) \\ &= \max(0, p_i + \min(0, t_2 - d_i) - \max(0, t_1 - r_i)) \\ &= \max(0, p_i - \max(0, d_i - t_2) - \max(0, t_1 - r_i)). \end{aligned}$$

Therefore, the previous equation becomes $\sum_i W_{PE}(A_i, t_1, t_2) > C*(t_2 - t_1)$. □

B.3.1.2.3. A Quadratic Algorithm

We propose a quadratic algorithm to determine whether there exists a feasible partially elastic schedule for a given instance of the Partially Elastic CuSP. This algorithm is derived from the algorithm used in [Perregaard, 1995] to compute the subset bound of the m-Machine Problem. It consists of computing the overall required energy consumption over each interval $[r_i, d_k]$ and to test whether this energy exceeds the energy provided by the resource over this interval. We prove that such tests guarantee the existence of a feasible partially elastic schedule. To achieve this proof, we study the slack function $S_{PE}(t_1, t_2) = C * (t_2 - t_1) - W_{PE}(t_1, t_2)$.

Proposition B-9.
Let t_1, t_2 be two integer values such that $t_1 < t_2$.

- If t_1 is not a release date, then
  either $S_{PE}(t_1 + 1, t_2) < S_{PE}(t_1, t_2)$ or $S_{PE}(t_1 - 1, t_2) \leq S_{PE}(t_1, t_2)$.
- If t_2 is not a deadline, then
  either $S_{PE}(t_1, t_2 - 1) < S_{PE}(t_1, t_2)$ or $S_{PE}(t_1, t_2 + 1) \leq S_{PE}(t_1, t_2)$.

Proof.
The two items of the proposition being symmetric, we only prove the first item. Suppose that $S_{PE}(t_1, t_2) \leq S_{PE}(t_1 + 1, t_2)$ and $S_{PE}(t_1, t_2) < S_{PE}(t_1 - 1, t_2)$. Let us then define the sets $\Psi = \{i \mid p_i - \max(0, t_1 - r_i) - \max(0, d_i - t_2) > 0\}$ and $\Phi = \{i \mid r_i \leq t_1\}$.
The equation $S_{PE}(t_1, t_2) \leq S_{PE}(t_1 + 1, t_2)$ can be rewritten

$$-C + \sum_{i\in\Psi} c_i * (-max(0, t_1 - r_i) + max(0, t_1 + 1 - r_i)) \geq 0.$$

Since $\forall i \notin \Phi$, $max(0, t_1 - r_i) = 0$ and $\max(0, t_1 + 1 - r_i) = 0$, the previous equation becomes:

$$\sum_{i\in\Psi\cap\Phi} c_i * (-\max(0, t_1 - r_i) + \max(0, t_1 + 1 - r_i)) \geq C \Rightarrow \sum_{i\in\Psi\cap\Phi} c_i \geq C.$$

The equation $S_{PE}(t_1, t_2) < S_{PE}(t_1 - 1, t_2)$ can be rewritten as follows:

$$\sum_{i} c_i * \max(0, p_i - \max(0, t_1 - 1 - r_i) - \max(0, d_i - t_2))$$

$$- \sum_{i} c_i * \max(0, p_i - \max(0, t_1 - r_i) - \max(0, d_i - t_2)) < C$$

$$\Rightarrow \sum_{i\in\Psi} c_i *(-\max(0, t_1 - 1 - r_i) + \max(0, t_1 - r_i))$$

$$+ \sum_{i\notin\Psi} c_i * \max(0, p_i - \max(0,t_1 - 1 - r_i) - \max(0, d_i - t_2)) < C$$

$$\Rightarrow \sum_{i\in\Psi} c_i *(-\max(0, t_1 - 1 - r_i) + \max(0, t_1 - r_i)) < C$$

Consider now two cases.

- If $i \in \Phi$ then $t_1 - r_i \geq 0$. Moreover, $t_1 - r_i - 1 \geq 0$ since t_1 is not a release date.
- If $i \notin \Phi$ then $t_1 - r_i < 0$ and $t_1 - r_i - 1 < 0$.

Previous equation then becomes $\sum_{i \in \Psi \cap \Phi} c_i < C$, which contradicts $\sum_{i \in \Psi \cap \Phi} c_i \geq C$. □

Proposition B-10.

$[\forall r_j, \forall d_k > r_j, S_{PE}(r_j, d_k) \geq 0] \Leftrightarrow [\forall t_1, \forall t_2 > t_1, S_{PE}(t_1, t_2) \geq 0]$

$\Leftrightarrow$ [There exists a feasible partially elastic schedule]

Proof.

Note that if $t_1 < \min_i(r_i)$, the slack strictly increases when t_1 decreases, and if $t_2 > \max_i(d_i)$, the slack strictly increases when t_2 increases. Hence, the slack function assumes a minimal value over an interval $[t_1, t_2]$ with $\min_i(r_i) \leq t_1 \leq t_2 \leq \max_i(d_i)$. We can assume that both t_1 and t_2 are integers (if t_1 is not, the function $t \rightarrow S_{PE}(t, t_2)$ is linear between $\lfloor t_1 \rfloor$ and $\lceil t_1 \rceil$; thus either $S_{PE}(\lfloor t_1 \rfloor, t_2) \leq S_{PE}(t_1, t_2)$ or $S_{PE}(\lceil t_1 \rceil, t_2) \leq S_{PE}(t_1, t_2)$). Among the pairs of integer values (t_1, t_2) which realize the minimum of the slack, let (u_1, u_2) be the pair such that u_1 is minimal and u_2 is maximal (given u_1).

We can suppose that $S_{PE}(u_1, u_2) < 0$ (otherwise the proposition holds). Consequently, $u_1 < u_2$ and thus, according to Proposition B-9, either u_1 is a release date or $S_{PE}(u_1 + 1, u_2) < S_{PE}(u_1, u_2)$ or $S_{PE}(u_1 - 1, u_2) \leq S_{PE}(u_1, u_2)$. Since $S_{PE}(u_1, u_2)$ is minimal, the previous inequalities lead to $S_{PE}(u_1 - 1, u_2) = S_{PE}(u_1, u_2)$; which contradicts our hypothesis on u_1. Consequently, u_1 is a release date. A symmetric demonstration proves that u_2 is a deadline. □

This proposition is of great interest since it allows us to restrict the computation of W_{PE} to intervals $[t_1, t_2]$ where t_1 is a release date and t_2 is a deadline. Before describing the algorithm, we introduce the notation $p_i^+(t_1)$ which denotes the minimal number of time units during which A_i must execute after t_1, *i.e.*,

$$p_i^+(t_1) = \max(0, p_i - \max(0, t_1 - r_i)).$$

Algorithm B-3 computes $W_{PE}(t_1, t_2)$ over all relevant intervals. The basic underlying idea is that, for a given t_1, the values of t_2 at which the slope of the $t \rightarrow W_{PE}(t_1, t)$ function changes are either of the form d_i or of the form $d_i - p_i^+(t_1)$. The procedure iterates on the relevant values of t_1 and t_2. Each time t_2 is modified, $W_{PE}(t_1, t_2)$ is computed, as well as the new slope (just after t_2) of the $t \rightarrow W_{PE}(t_1, t)$ function. Each time t_1 is modified, the set of activities with relevant $d_i - p_i^+(t_1)$ is incrementally recomputed and resorted.

Algorithm B-3.

```
procedure update(DP, old_t1, t1)
move = $\emptyset$, no_move = $\emptyset$     // initialize two empty lists
for act in DP
  if ($p^+_{act}$(t1) > 0) then
    if ($p^+_{act}$(t1) = $p^+_{act}$(old_t1)) then add act to no_move
    else add act to the list move
    end if
  end if
end for
DP = merge(move, no_move)

procedure energies
DD = activities sorted in increasing order of $d_{act}$
DP = activities sorted in increasing order of $d_{act}$ - $p_{act}$
old_t1 = $\min_i(r_i)$
for t1 in the set of release dates (sorted in inc. order)
  update(DP, old_t1, t1)
  old_t1 = t1, iDD = 0, iDP = 0
  W = 0, old_t2 = t1, slope = $\Sigma$ act $W_{PE}$(act, t1, t1 + 1)
  while (iDP < length(DP) or iDD < n)
    if (iDD < n) then t2_DD = $d_{DD[iDD + 1]}$
    else t2_DD = $\infty$
    end if
    if (iDP < length(DP)) then t2_DP = $d_{DP[iDP+1]} - p^+_{DP[iDP+1]}$(t1)
    else t2_DP = $\infty$
    end if
    t2 = min(t2_DD, t2_DP)
    if (t2 = t2_DP) then iDP = iDP + 1, act = DP[iDP]
    else iDD = iDD + 1, act = DD[iDD]
    end if
    if (t1 < t2) then
      W = W + slope * (t2 - old_t2)
      WPE(t1, t2) = W   // energy over [t1, t2]
      old_t2 = t2
      slope = slope + $W_{PE}$(act, t1, t2 + 1)
             - 2*$W_{PE}$(act, t1, t2) + $W_{PE}$(act, t1, t2 - 1)
    end if
  end while
end for
```

Let us detail the procedure `energies`.

- Lines 13 and 14 initialize `DD` and `DP`. `DD` is the array of activities sorted in increasing order of deadlines and `DP` is the array of activities sorted in increasing order of $d_i - p_i$.
- The main loop (line 16) consists in an iteration over all release dates t_1. Notice that old_t_1 allows to keep the previous value of t_1.
- The procedure $update(DP, old\_t_1, t_1)$ reorders the array `DP` in increasing order of $d_i - p_i^+(t_1)$. This procedure will be described later on.
- Before starting the inner loop, a variable `slope` is initialized (line 19). It corresponds to the slope of the function $t \to W_{PE}(t_1, t)$ immediately after the time point `old_t2`. `old_t2` and `slope` are initialized line 19 and updated lines 34 and 35.
- The inner loop on t_2 (lines 20-38) consists in iterating on both arrays `DD` and `DP` at the same time. Because both arrays are sorted, some simple operations (line 21 to 27) determine the next value of t_2. Notice that t_2 can take at most $2 * n$ values and that t_2 takes all the values which correspond to a deadline. The indices i_{DD} and i_{DP} correspond to the current position in arrays `DD` and `DP` respectively.
- Lines 28 to 30 enable to increase one of the indices and to determine the activity `act` which has induced the current iteration.
- To understand lines 31 to 36, consider the following rewriting of $W_{PE}(A_i, t_1, t_2)$.

$$W_{PE}(A_i, t_1, t_2) = \begin{cases} 0 & \text{If } t_2 \le d_i - p_i^+(t_1) \\ c_i * (t_2 - d_i + p_i^+(t_1)) & \text{If } d_i - p_i^+(t_1) < t_2 \le d_i \\ c_i * p_i^+(t_1) & \text{If } d_i < t_2 \end{cases}$$

  Between two consecutive values of t_2 in the inner loop, the function W_{PE} is linear. The required energy consumption between t_1 and old_t_2 is `W`, as computed at the end of the previous iteration. In addition, the slope of $t \to W_{PE}(t_1, t)$ between old_t_2 and t_2 is `slope`. So, the required energy consumption between t_1 and t_2 is $W + slope * (t_2 - old\_t_2)$. Then, `slope` is updated to take into account the non-linearity of the required energy consumption of activity A_i (`act` in the pseudo code) at time t_2. Notice that the algorithm may execute several times lines 31-36 for the same values of t_1 and t_2 (*e.g.*, if $d_i = d_j$ for some i and j). In such a case, the slope is modified several times, with respect to all the activities inducing a non-linearity at time t_2.

Let us now detail the procedure `update`. This procedure reorders the array `DP` in increasing order of $d_i - p_i^+(t_1)$. This is done in linear time. We rely on the fact that when we move from old_t_1 to t_1, three cases can occur.

- Either $p_i^+(t_1)$ is null and then the required energy consumption of A_i in $[t_1, t_2]$ is null; and A_i can be removed;
- Or $p_i^+(t_1) = p_i^+(old\_t_1)$ (line 5);

- Or $p_i^+(t_1) = p_i^+(old\_t_1) - (t_1 - old\_t_1)$ (line 6).

Activities are taken in the initial order of `DP` and are stored in either the list `no_move` (second item) or in the list `move` (third item). Notice that `no_move` is sorted in increasing order of $d_i - p_i^+(old\_t_1) = d_i - p_i^+(t_1)$. Moreover, `move` is sorted in increasing order of $d_i - p_i^+(\text{old\_}t_1)$ but `move` is also sorted in increasing order of $d_i - p_i^+(t_1)$ since the difference between $p_i^+(t_1)$ and $p_i^+(\text{old\_}t_1)$ is constant for all activities in `move`. This means that we only have to merge `move` and `no_move` to obtain the reordered array.

The overall algorithm runs in $O(n^2)$ since (1) the initial sort can be done in $O(n * \log(n))$, (2) the procedure `update` is basically a merging procedure which runs in $O(n)$, (3) the initial value of `slope` for a given t_1 is computed in $O(n)$, and (4) the inner and outer loops of the algorithm both consist in $O(n)$ iterations.

B.3.1.3. A "Left-Shift / Right-Shift" Necessary Condition of existence for the CuSP

The required energy consumption as defined in Section B.3.1.2.2 is still valid if we consider that activities can be interrupted. In fact, [Erschler *et al.*, 1991] and [Lopez *et al.*, 1992] propose a sharper definition of the required energy consumption that takes into account the fact that activities cannot be interrupted. 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;
- $p_i^+(t_1) = \max(0, p_i - \max(0, t_1 - r_i))$, 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;
- $p_i^-(t_2) = \max(0, p_i - \max(0, d_i - t_2))$, 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 $W_{Sh}(A_i, t_1, t_2) = c_i * \min(t_2 - t_1, p_i^+(t_1), p_i^-(t_2))$ (*cf.* Figure B-9 for an example).

	r_i	d_i	p_i	c_i
A_1	0	10	7	2

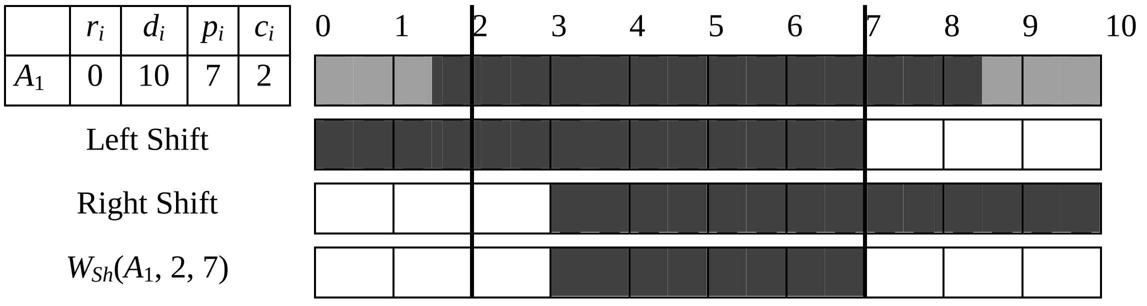


Figure B-9. The required energy consumption of A_1 over [2, 7]. At least 4 time units of A have to be executed in [2, 7]; i.e., $W_{Sh}(A,2,7)=2*\min(5,5,4)=8$.

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 $W_{Sh}(A_i, t_1, t_2)$. We can also define the left-shift / right-shift slack over $[t_1, t_2]$: $S_{Sh}(t_1, t_2) = C * (t_2 - t_1) - W_{Sh}(t_1, t_2)$. It is obvious that if there is a feasible schedule of an instance of the CuSP then $\forall\ t_1$, $\forall\ t_2 \geq t_1$, $S_{Sh}(t_1, t_2) \geq 0$.

B.3.1.3.1. Characterization of relevant and irrelevant intervals

In Section B.3.2 we showed that for the partially elastic relaxation, it was sufficient to calculate the slack only for those intervals $[t_1, t_2]$ that are in the Cartesian product of the set of release dates and of the set of deadlines. In this section we show that in the left-shift / right-shift case, a larger number of intervals must be considered. On top of that, we provide a precise characterization of set of intervals for which the slack needs to be calculated to guarantee that no interval with negative slack exists.

Proposition B-11.
Let us define the sets O_1, O_2 and $O(t)$.

$O_1 = \{r_i, 1 \leq i \leq n\} \cup \{d_i - p_i, 1 \leq i \leq n\} \cup \{r_i + p_i, 1 \leq i \leq n\}$

$O_2 = \{d_i, 1 \leq i \leq n\} \cup \{r_i + p_i, 1 \leq i \leq n\} \cup \{d_i - p_i, 1 \leq i \leq n\}$

$O(t) = \{r_i + d_i - t, 1 \leq i \leq n\}$

We claim that:

$$\begin{aligned}
\forall\ t_1, \forall\ t_2 \geq t_1\ S_{Sh}(t_1, t_2) \geq 0 \quad \Leftrightarrow \quad & \forall\ s \in O_1, \forall\ e \in O_2, e \geq s\ , S_{Sh}(s, e) \geq 0 \\
\text{and} \quad & \forall\ s \in O_1, \forall\ e \in O(s), e \geq s, S_{Sh}(s, e) \geq 0 \\
\text{and} \quad & \forall\ e \in O_2, \forall\ s \in O(e), e \geq s, S_{Sh}(s, e) \geq 0
\end{aligned}$$

To prove Proposition B-11, we first need to prove some technical properties of W_{Sh} (Propositions B-12, B-13 and B-14). In the following, we consider that W_{Sh} is defined on $\Re^2$ and equal to 0 when $t_2 \leq t_1$.

Proposition B-12.
Let A_i be an activity and $(t_1, t_2) \in \Re^2$ with $t_1 < t_2$. If $t_1 \notin \{r_i, d_i - p_i, r_i + p_i\}$ and if $t_2 \notin \{d_i, d_i - p_i, r_i + p_i\}$, then $\Phi(h) = W_{Sh}(A_i, t_1 + h, t_2 - h)$ is linear around 0.

Proof.
$\Phi(h)$ can be rewritten $\Phi(h) = c_i * \max(0, \min(t_2 - t_1 - 2*h, p_i, r_i + p_i - t_1 - h, t_2 - d_i + p_i - h))$. Each of the terms $0, t_2 - t_1 - 2*h, p_i, r_i + p_i - t_1 - h, t_2 - d_i + p_i - h$ is linear in h and if for $h = 0$, one term only realizes $\Phi(0)$, we can be sure that a small perturbation of h will have a linear effect. Assume two terms are equal and realize $\Phi(0)$. Since there are five terms, this leads us to distinguish ten cases.

1. $\Phi(0) = 0 = t_2 - t_1$,
2. $\Phi(0) = 0 = p_i$,
3. $\Phi(0) = 0 = r_i + p_i - t_1$,
4. $\Phi(0) = 0 = t_2 - d_i + p_i$,
5. $\Phi(0) = t_2 - t_1 = p_i$,
6. $\Phi(0) = t_2 - t_1 = r_i + p_i - t_1$,
7. $\Phi(0) = t_2 - t_1 = t_2 - d_i + p_i$,
8. $\Phi(0) = p_i = r_i + p_i - t_1$,
9. $\Phi(0) = p_i = t_2 - d_i + p_i$,
10. $\Phi(0) = r_i + p_i - t_1 = t_2 - d_i + p_i$.

According to our hypotheses, all cases are impossible except (5) and (10).

- We claim that case (5) cannot occur. Since $t_2 - t_1 = p_i$ and since this value is equal to $\Phi(0)$, we have $p_i < r_i + p_i - t_1$ and $p_i < t_2 - d_i + p_i$ (equality cannot occur because of our hypotheses). Thus, $t_2 - t_1 = p_i > d_i - r_i$; which contradicts $r_i + p_i \leq d_i$.
- If (10) holds, then $r_i + p_i - t_1 - h = t_2 - d_i + p_i - h$. We can moreover suppose that these two terms are the only ones to realize $\Phi(0)$ (otherwise one of the previous cases would occur). Around 0, $\Phi(h)$ can be rewritten $c_i * (r_i + p_i - t_1 - h)$; which is linear. □

Proposition B-13.

Let A_i be an activity and $(t_1,t_2) \in \Re^2$ such that $t_1 < t_2$ and $t_2 \notin \{d_i, d_i - p_i, r_i + p_i, r_i + d_i - t_1\}$, then $\Theta(h) = W_{Sh}(A_i, t_1, t_2 - h)$ is linear around 0.

Proof.

$\Theta(h)$ can be rewritten $\Theta(h) = c_i * \max(0, \min(t_2 - t_1 - h, p_i, r_i + p_i - t_1, t_2 - d_i + p_i - h))$. Each of the terms is linear in h and if for $h = 0$, one term only realizes $\Theta(h)$, we can be sure that a small perturbation of h will have a linear effect. Two terms are equal and realize $\Theta(0)$ if either

1. $\Theta(0) = 0 = t_2 - t_1$ or
2. $\Theta(0) = 0 = p_i$ or
3. $\Theta(0) = 0 = r_i + p_i - t_1$ or
4. $\Theta(0) = 0 = t_2 - d_i + p_i$ or
5. $\Theta(0) = t_2 - t_1 = p_i$ or
6. $\Theta(0) = t_2 - t_1 = r_i + p_i - t_1$ or
7. $\Theta(0) = t_2 - t_1 = t_2 - d_i + p_i$ or
8. $\Theta(0) = p_i = r_i + p_i - t_1$ or
9. $\Theta(0) = p_i = t_2 - d_i + p_i$ or
10. $\Theta(0) = r_i + p_i - t_1 = t_2 - d_i + p_i$.

According to our hypotheses, all cases are impossible except (3), (5), (7), (8).

- If $t_1 = r_i + p_i$ (3) then $\forall h$, $\Theta(h) = 0$.
- Case (5) cannot occur otherwise we would have $p_i \le r_i + p_i - t_1$ and $p_i < t_2 - d_i + p_i$; which contradicts $r_i + p_i \le d_i$.
- If $t_2 - t_1 = t_2 - d_i + p_i$ (7) then $\forall h$, $\Theta(h) = c_i * \max(0, \min(t_2 - t_1 - h, p_i, r_i + p_i - t_1))$. We can moreover suppose that $t_2 - t_1$ is the only term in the new expression to realize $\Theta(0)$ (otherwise case (1), (5) or (6) would occur) and thus $\Theta(h)=c_i*(t_2-t_1-h)$ around 0.
- If $p_i=r_i+p_i-t_1$ realizes $\Theta(0)$ (8) then $\Theta(h)=c_i*\max(0,\min(t_2-r_i-h,p_i,t_2-d_i+p_i-h))$. Since $\Theta(0)=c_i*p_i>0$, around 0 we must have $\Theta(h)=c_i*\min(t_2-r_i-h,p_i,t_2-d_i+p_i-h)$. Moreover, since $r_i \le d_i - p_i$, around 0 we have $\Theta(h)=c_i*\min(p_i, t_2-d_i+p_i-h)$. Finally, we know that $t_2 \ne d_i$ thus, $p_i < t_2-d_i+p_i$. Consequently, around 0, $\Theta(h)=c_i*p_i$. ☐

Proposition B-14.

Let $(t_1, t_2) \in \Re^2$ such that $t_1 < t_2$.

- If $t_1 \notin O_1$ and $t_2 \notin O_2$, $h \rightarrow S_{Sh}(t_1 + h, t_2 - h)$ is linear around 0.
- If $t_2 \notin O_2 \cup O(t_1)$, $h \rightarrow S_{Sh}(t_1, t_2 - h)$ is linear around 0.
- If $t_1 \notin O_1 \cup O(t_2)$, $h \rightarrow S_{Sh}(t_1 + h, t_2)$ is linear around 0.

Proof (sketch).

We prove the first item. Since $t_1 \notin O_1$ and $t_2 \notin O_2, \forall i, h \rightarrow W_{Sh}(A_i, t_1+h, t_2-h)$ is linear around 0 (Proposition B-12). Thus, $h \rightarrow S_{Sh}(t_1+h, t_2-h)$ is linear around 0. The same proof applies for other items (Proposition B-13 and its symmetric counterpart are used). ☐

Proof of Proposition B-11.

The implication from left to right is obvious. Suppose now that the right hand side of the equivalence holds and that there exists an interval $[t_1, t_2]$ for which the slack is strictly negative. As in the partially elastic case, we remark that when t_1 is smaller than $r_{min} = \min(r_i)$, the slack strictly increases when t_1 decreases. Similarly, when t_2 is greater than $d_{max} = \max(d_i)$, the slack strictly increases when t_2 increases. Since the slack function is also continuous, it assumes a minimal value over an interval $[t_1, t_2]$ with $r_{min} \le t_1 \le t_2 \le d_{max}$. Let us consequently select a pair (t_1, t_2) at which the slack is minimal. In case several pairs (t_1, t_2) minimize the slack, an interval with maximal length is chosen. Since this slack is strictly negative, we must have $t_1 < t_2$.

Case 1: If $t_1 \notin O_1$ and $t_2 \notin O_2$, then according to Proposition B-14, the function $\varphi(h)=S_{Sh}(t_1+h, t_2-h)$ is linear around 0. Since (t_1, t_2) is a global minimum of the slack, $\varphi(h)$ is constant around 0, which contradicts the fact that the length of $[t_1, t_2]$ is maximal.

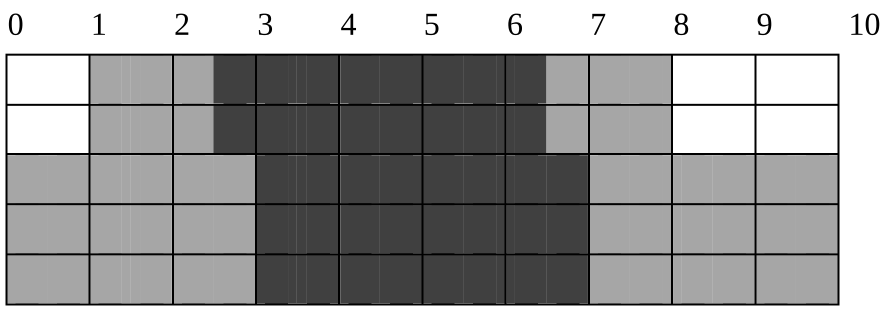
Case 2: If $t_1 \in O_1$ then $t_2 \notin O_2 \cup O(t_1)$, otherwise the slack is non-negative. According to Proposition B-14, $\theta(h) = S_{Sh}(t_1, t_2 - h)$ is linear around 0. Since (t_1, t_2) is a global minimum of the slack, θ(h) is constant around 0, which contradicts the fact that the length of [t_1, t_2] is maximal.

<u>*Case 3:*</u> If $t_2 \in O_2$ then $t_1 \notin O_1 \cup O(t_2)$, otherwise the slack is non-negative. According to Proposition B-14, $\theta(h) = S_{Sh}(t_1 + h, t_2)$ is linear around 0. Since (t_1, t_2) is a global minimum of the slack, θ(h) is constant around 0, which contradicts the fact that the length of [t_1, t_2] is maximal.

The combination of cases 1, 2 and 3 leads to a contradiction. ☐

Proposition B-11 provides a characterization of interesting intervals over which the slack must be computed to ensure it is always non-negative over any interval. This characterization is weaker than the one proposed for the partially elastic case where the interesting time intervals [t_1, t_2] are in the Cartesian product of the set of the release dates and of the set of deadlines. However, there are still only $O(n^2)$ relevant pairs (t_1, t_2) to consider. Some of these pairs belong to the Cartesian product $O_1 * O_2$. The example of Figure B-10 proves that some pairs do not.

	R_i	d_i	P_i	c_i
A_1	1	8	4	1
A_2	1	8	4	1
A_3	0	10	4	1
A_4	0	10	4	1
A_5	0	10	4	1



$t_2 \setminus t_1$	0	1	2
8	16 – 4 * 2 - 2 * 3 = 2	14 - 4 * 2 - 2 * 3 = 0	12 - 3 * 2 - 2 * 3 = 0
9	18 – 4 * 2 - 3 * 3 = 1	16 - 4 * 2 - 3 * 3 = -1	14 - 3 * 2 - 2 * 3 = 2
10	20 – 4 * 2 - 4 * 3 = 0	18 - 4 * 2 - 3 * 3 = 1	16 - 3 * 2 - 2 * 3 = 4

Figure B-10. Some interesting time intervals are outside the Cartesian product $O_1 * O_2$. In this example, (resource of capacity 2 and 5 activities A_1, A_2, A_3, A_4, A_5), the pair (1, 9) corresponds to the minimal slack and does not belong to {0,1,4,5,6}{4,5,6,8,10}. In this interval, the slack is negative, which proves that there is no feasible schedule. Notice that neither the fully elastic nor the partially elastic relaxation can trigger a contradiction.*

B.3.1.3.2 A Quadratic Algorithm

We propose an $O(n^2)$ algorithm to compute the required energy consumption W_{Sh} over all interesting pairs of time points. Actually, the algorithm first computes all the relevant values taken by W_{Sh} over time intervals $[t_1, t_2]$ with $t_1 \in O_1$, and then computes all the relevant values taken by W_{Sh} over time intervals $[t_1, t_2]$ with $t_2 \in O_2$. The characterization obtained in the previous section ensures that all the interesting time intervals are examined in at least one of these steps. For symmetry reasons, we will only describe the first computation. It is achieved by the same type of technique than in the partially elastic case. An outer loop iterates on all values $t_1 \in O_1$ sorted in increasing order. Then, we consider the function $t \to W_{Sh}(A_i, t_1, t)$. This function is linear on the intervals delimited by the values d_i, $r_i + p_i$, $d_i - p_i$ and $r_i + d_i - t_1$. We rely on this property to incrementally maintain the slope of the function $t \to W_{Sh}(A_i, t_1, t)$.

Algorithm B-4.

```
DD = activities sorted in increasing order of $d_{act}$
RP = activities sorted in increasing order of $r_{act} + p_{act}$
DP = activities sorted in increasing order of $d_{act} - p_{act}$
RD = activities sorted in increasing order of $r_{act} + d_{act}$
for t1 in the set O1 (sorted in increasing order)
  iDD = 0, iRP = 0, iDP = 0, iRD = 0
  W = 0, old_t2 = t1, slope = $\Sigma$act $W_{Sh}$(act, t1, t1 + 1)
  while (iDD < n or iRP < n or iDP < n or iRD < n)
    if (iDD < n) then
      t2_DD = $d_{DD[iDD + 1]}$
    else
      t2_DD = $\infty$ end if
    if (iRP < n) then
      t2_RP = $r_{RP[iRP + 1]} + p_{RP[iRP + 1]}$
    else
      t2_RP = $\infty$ end if
    if (iDP < n) then
      t2_DP = $d_{DP[iDP + 1]} - p_{DP[iDP + 1]}$
    else
      t2_DP = $\infty$
    end if
    if (iRD < n) then
      t2_RD = $r_{RD[iRD + 1]} + d_{RD[iRD + 1]}$ - t1
    else
      t2_RD = $\infty$
    end if
    t2 = min(t2_DD, t2_RP, t2_DP, t2_RD)
    if (t2 = t2_DD) then iDD = iDD + 1, act = DD[iDD]
    else if (t2=t2_iRP) then iRP = iRP + 1, act = RP[iRP]
    else if (t2=t2_iDP) then iDP = iDP + 1, act = DP[iDP]
    else if (t2=t2_iRD) then iRD = iRD + 1, act = RD[iRD]
    end if
    if (t1 < t2) then
      W = W + slope * (t2 - old_t2)
      WSh(t1, t2) = W // energy over [t1, t2] is computed
      old_t2 = t2
      slope = slope + $W_{Sh}$(act, t1, t2 + 1)
              - 2*$W_{Sh}$(act, t1, t2) + $W_{Sh}$(act, t1, t2 - 1)
    end if
  end while
end for
```

There are some few differences with the partially elastic case.

- The computation of `t2` is slightly more complex since there are more interesting values to consider.
- One does not need to reorder any list: when `t1` increases, none of the values d_i, $r_i + p_i$, $d_i - p_i$ and $r_i + d_i - t_1$ changes except the last one; which corresponds to the list RD. Since RD is initially sorted in increasing order of $r_i + d_i$, it is also sorted in increasing order of $r_i + d_i - t_1$.
- In line 37, one shall in fact be careful not to update the slope more than once for the same tuple (t_1, t_2, act). This can be done easily by marking the activity act with t_2. We have not included this marking in the pseudo-code to keep it simple.

B.3.1.4. Synthesis of Theoretical Results

Figure B-11 summarizes the theoretical results related to the conditions described in Sections B.3.1.1, B.3.1.2 and B.3.1.3. The most satisfactory results are obtained for the Fully Elastic CuSP and for the Partially Elastic CuSP since the related problems are polynomially solvable by either a reduction to a One-Machine Preemptive Problem or by some simple and intuitive energetic computation. Less strong results are obtained for the CuSP. No polynomial sufficient condition for the existence of a feasible schedule is proposed (which seems reasonable since the CuSP is NP-complete). As for the Partially Elastic CuSP, the left-shift / right-shift necessary condition relies on energetic reasoning, but there are more time intervals to consider in practice and the structure of these intervals is far more complex and poorly intuitive.

Needless to say, all relaxations can be used as part of the resolution of a non-preemptive cumulative problem. However, it is easy to see that the necessary condition based on the fully elastic relaxation is subsumed by the one based on the partially elastic relaxation, which in turn is subsumed by the left-shift / right-shift necessary condition.

	Fully Elastic	Partially Elastic	Left-Shift / Right-Shift
Characterization	Necessary and sufficient	Necessary and sufficient	Necessary
Complexity	$O(n * log(n))$	$O(n^2)$	$O(n^2)$
Method	One-Machine reduction	Slack computation	Slack computation
Intervals [t_1, t_2]	-	$O(n^2)$ in a Cartesian product	$O(n^2)$

Figure B-11. A comparison of the 3 necessary conditions.

In the m-Machine case, *i.e.*, when activities require exactly one unit of the resource, the three necessary conditions can be compared to several results of the literature (these results are discussed in [Perregaard, 1995]).

First, notice that the decision variant of the Preemptive m-Machine Problem is polynomial and can be formulated as a maximum flow problem (similar techniques as those depicted in Section B.2.3 for the One-Machine Problem are used), see for instance [Federgruen and Groenevelt, 1986] or [Perregaard, 1995]. As shown in [Perregaard, 1995], solving this maximum flow problem leads to a worst case complexity of $O(n^3)$. The preemptive relaxation is strictly stronger than the fully and the partially elastic relaxations. However, it is not stronger than the left-shift / right-shift necessary condition. Indeed, there exists a feasible preemptive schedule of the m-Machine instance described on Figure B-10, while the left-shift / right-shift necessary condition does not hold.

A comparison can also be made between the subset bound, a lower bound for the optimization variant of the m-Machine Problem (see for example [Carlier, 1984], [Carlier and Pinson, 1996], [Perregaard, 1995]) and the partially elastic relaxation. An instance of the optimization variant of the m-Machine Problem consists of a set of activities characterized by a release date r_i, a tail q_i and a processing time p_i, and a resource of capacity $C = m$. The objective is to find a start time assignment s_i for each activity A_i such that (1) temporal and resource constraints are met and (2) $\max_i (s_i + p_i + q_i)$ is minimal.

The subset bound is the maximum among all subsets J of at least C activities of the following expression, in which $R(J)$ and $Q(J)$ denote the sets of activities in J having respectively the C smallest release dates and the C smallest tails.

$$\frac{1}{C}\left(\sum_{A_j \in R(J)} r_j + \sum_{A_j \in J} p_j + \sum_{A_j \in Q(J)} q_j \right)$$

[Perregaard, 1995] presents an algorithm to compute the subset bound in a quadratic number of steps. Carlier and Pinson [Carlier and Pinson, 1996] describe an $O(n * \log(n) + n * C * \log(C))$ algorithm which relies on a “pseudo-preemptive” relaxation of the m-Machine Problem. Notice that the subset bound can apply, thanks to a simple transformation, as a necessary condition of existence for the decision variant of the m-Machine Problem:

$$\forall J \subseteq \{A_1, \Lambda\ , A_n\} \text{ such that } |J| \geq C, \sum_{A_j \in R(J)} r_j + \sum_{A_j \in J} p_j \leq \sum_{A_j \in D(J)} d_j$$

where $D(J)$ denotes the set of activities in J having the C largest deadlines.

Proposition B-15.
In the m-Machine case, there exists a feasible partially elastic schedule if and only if the subset bound necessary condition holds.

Proof.

- First, assume that there exists a feasible partially elastic schedule. Let J be any subset of at least C activities. Let $\rho_1, \rho_2, ..., \rho_C$ be the C smallest release dates of activities in J, and $\delta_1, \delta_2, ..., \delta_C$ be the C largest deadlines of activities in J. Since before ρ_C, at most C activities execute, and since for each of these activities A_i at most $(\rho_C - r_i)$ units are executed before ρ_C, the schedule of these activities can be reorganized so that $pes(t, i)$ is at most 1 for every $t \leq \rho_C$. Let us now replace each activity A_i in $R(J)$ with a new activity of release date ρ_1, deadline d_i, and processing time $p_i + (r_i - \rho_1)$. A feasible partially elastic schedule of the new set of activities is obtained as a simple modification of the previous schedule, by setting $pes(t, i) = 1$ for every A_i in $R(J)$ and t in $[\rho_1\ r_i)$. The same operation can be done between δ_C and δ_1 for activities in $D(J)$. We have a partially elastic schedule requiring $\Sigma_{Ai\in J} p_i + \Sigma_{Ai\in R(J)}(r_i - \rho_1) + \Sigma_{Ai\in D(J)}(\delta_1 - d_i)$ units of energy between ρ_1 and δ_1.

  Hence, $\Sigma_{Ai\in J} p_i + \Sigma_{Ai\in R(J)}(r_i - \rho_1) + \Sigma_{Ai\in D(J)}(\delta_1 - d_i) \leq C * (\delta_1 - \rho_1)$. This is equivalent to the subset bound condition for J.
- We now demonstrate the other implication. Assume that the slack S_{PE} is strictly negative for some interval. Let then $[t_1, t_2]$ be the interval over which the slack is minimal and let us define J as the set of activities A_i such that $W_{PE}(A_i, t_1, t_2) > 0$. Notice that there are at least C activities in J because $r_i + p_i \leq d_i$ implies that $W_{PE}(A_i, t_1, t_2) \leq t_2 - t_1$. In addition, at most C activities A_i in J are such that $r_i < t_1$. Otherwise, when t_1 is replaced by $t_1 - 1$, the slack decreases. Similarly, there are at most C activities A_i in J are such that $t_2 < d_i$. Let us define $X = \{A_i \in J \mid r_i < t_1\}$ and $Y = \{A_i \in J \mid t_2 < d_i\}$. According to the previous remark, we have $|X| \leq C$ and $|Y| \leq C$. Now notice that for any activity A_i in J, $W_{PE}(A_i, t_1, t_2) > 0$. Thus, we have $W_{PE}(A_i, t_1, t_2) = \Sigma_{Ai\in J} (p_i - \max(0, t_1 - r_i) - \max(0, d_i - t_2))$. This can be rewritten:

$$W_{PE}(A_i, t_1, t_2) = \Sigma_{Ai\in J}\, p_i + \Sigma_{Ai\in X}\, r_i - |X| * t_1 - \Sigma_{Ai\in Y}\, d_i + |Y| * t_2.$$

Since $S_{PE}(t_1, t_2)$ is strictly negative, we have

$$\Sigma_{Ai\in X}\, r_i + (C - |X|) * t_1 + \Sigma_{Ai\in J}\, p_i > \Sigma_{Ai\in Y}\, d_i + (C - |Y|) * t_2.$$

Moreover, because of the definition of $R(J)$ (resp. $D(J)$), and because $|X| \leq C$ and $|Y| \leq C$, we have $\Sigma_{Ai\in R(J)} r_j \geq \Sigma_{Ai\in X} r_j + (C - |X|) * t_1$ and $\Sigma_{Ai\in D(J)} d_j \leq \Sigma_{Ai\in Y} d_j + (C - |Y|) * t_2$. As a consequence, $\Sigma_{Ai\in R(J)}\ r_j + \Sigma_{Ai\in J}\ p_i > \Sigma_{Ai\in D(J)}\ dj$, which is exactly the subset bound necessary condition for the set J. $\square$

B.3.2. Time-Bound Adjustments for the CuSP

The following sections describe three time-bound adjustment schemes for the CuSP. These techniques extend the time-bound adjustments, also called edge-finding, initially proposed for the One-Machine Problem [Applegate and Cook, 1991], [Baptiste and Le Pape, 1995b], [Carlier and Pinson, 1990], [Carlier and Pinson, 1994], [Caseau and Laburthe, 1995], [Nuijten, 1994], [Pinson, 1988].

B.3.2.1. Time-Bound Adjustments for the Fully Elastic CuSP

In the fully elastic case, we rely on the reduction of the Fully Elastic CuSP to the Preemptive One-Machine Problem. We then use the time-bound adjustment algorithm for the Preemptive One-Machine Problem described in Section B.2.4.

More precisely, the adjustment scheme is:

1. Build the One-Machine Preemptive Problem instance $F(I)$ corresponding to the Fully Elastic CuSP instance I.
2. Apply the preemptive edge-finding algorithm on activities A_1', ..., A_n' of the instance $F(I)$. As explained in Section B.2.4., for each activity A_i', four time-bounds can be sharpened: the release date r_i', the latest possible start time lst_i', the earliest possible end time eet_i', and the deadline d_i'.
3. Update the four time-bounds of each A_i. $r_i = \lfloor r_i' / C \rfloor$, $lst_i = \lfloor lst_i' / C \rfloor$, $eet_i = \lceil eet_i' / C \rceil$ and $d_i = \lceil d_i' / C \rceil$.

This algorithm runs in a quadratic number of steps since (1) and (3) are linear and (2) can be done in $O(n^2)$ as detailed in Section B.2.4.

Proposition B-16.
The time-bound adjustments made by the algorithm above are the best possible ones, *i.e.*, the lower and upper bounds for the start and end time of activities can be reached by some feasible fully elastic schedules.

Proof (sketch).
The same proof applies for each of the four time-bounds. We focus on the earliest end time. The basic idea is to prove that for any A_i, (1) there is a fully elastic schedule on which A_i can end at the earliest end time computed by the fully elastic time-bound adjustment algorithm and (2) there is no fully elastic schedule on which A_i can end before the earliest end time computed by the algorithm. Both steps can be proven thanks to the reduction F, and to the fact that the preemptive edge-finding algorithm computes the best possible time-bounds for the Preemptive One-Machine Problem (*cf.*, Section B.2.4.). ☐

B.3.2.2. Time-Bound Adjustments for the CuSP Based on the Partially Elastic Relaxation

In this section, we provide an adjustment scheme for the CuSP which relies on the required energy consumptions computed in the partially elastic case. From now on, we assume that $\forall\, r_j, \forall\, d_k, W_{PE}(r_j, d_k) \leq C * (d_k - r_j)$. If not, we know that there is no feasible partially elastic schedule. As for other adjustments techniques, our basic idea is to try to order activities. More precisely, given an activity A_i and an activity A_k, we examine whether A_i can end before d_k.

Proposition B-17.[7]

If $\exists\, A_j \mid r_j < d_k$ and $W_{PE}(r_j, d_k) - W_{PE}(A_i, r_j, d_k) + c_i * \max(0, p_i - \max(0, r_j - r_i)) > C * (d_k - r_j)$ then a valid lower bound of the end time of A_i is:

$$d_k + \frac{W_{PE}(r_j, d_k) - W_{PE}(A_i, r_j, d_k) + c_i * \max(0, p_i - \max(0, r_j - r_i)) - C * (d_k - r_j)}{c_i}$$

Proof.

Notice that $W_{PE}(r_j, d_k) - W_{PE}(A_i, r_j, d_k) + c_i * \max(0, p_i - \max(0, r_j - r_i))$ is the overall required energy consumption over $[r_i\ d_k]$ when d_i is set to d_k. If this quantity is greater than $C * (d_k - r_j)$ then A_i must end after d_k. To understand the lower bound of the end time of A_i, simply notice that the numerator of the expression is the number of energy units of A_i which have to be shifted after time d_k. We can divide this number of units by the amount of resource required by A_i to obtain a lower bound of the processing time required to execute these units. ☐

As all values $W_{PE}(r_j, d_k)$ can be computed in $O(n^2)$, this mechanism leads to a simple $O(n^3)$ algorithm. For any tuple (A_i, A_j, A_k) (1) check wether A_i can end before d_k and (2) in such a case compute the corresponding time-bound adjustment. The issue is that $O(n^3)$ is a high complexity. This led us to further investigation. In the following, we show that the same adjustments can be made in $O(n^2 * \log(|\{c_i\}|))$, through successive transformations and decompositions of the adjustment scheme.

Given A_i and A_k, our goal is to find the activity A_j which will produce the best possible adjustment. Notice that if $d_i \leq d_k$, $W_{PE}(A_i, r_j, d_k) = c_i * \max(0, p_i - \max(0, r_j - r_i))$ and then no

[7] Note that the lower bound proposed in this proposition only holds for the CuSP and does not hold for the Partially Elastic CuSP. Indeed, in the partially elastic case, nothing prevents A_i from using more than c_i units of the resource at a given time point. To get a valid lower bound, one has to divide the numerator by C instead of c_i. In practice, we use only the Partially Elastic CuSP as a relaxation of the CuSP, which justifies the use of c_i to get a better bound.

adjustment can be achieved since $\forall r_j, \forall d_k, W_{PE}(r_j,d_k) \le C*(d_k-r_j)$. In the following, we only consider the case in which $d_i > d_k$. This can be written as a mathematical optimization problem (P).

$$\max_j \left(d_k + \frac{W_{PE}(r_j,d_k) - W_{PE}(A_i,r_j,d_k) + c_i * \max(0, p_i - \max(0, r_j - r_i)) - C*(d_k - r_j)}{c_i} \right)$$

$$\text{u.c.} \begin{cases} W_{PE}(r_j,d_k) - W_{PE}(A_i,r_j,d_k) + c_i * \max(0, p_i - \max(0, r_j - r_i)) > C*(d_k - r_j) \\ r_j < d_k \end{cases}$$

Let $W_{j,k} = W_{PE}(r_j, d_k)$ if $r_j < d_k$, and $-\infty$ otherwise. Note that the $W_{j,k}$ can be pre-computed in $O(n^2)$ as shown in Section B.3.1.2.3. Then, P can be reduced to the following problem.

$$\max_j \Big(W_{j,k} + c_i \big(\max(0,p_i - \max(0,r_j - r_i)) - \max(0, p_i - \max(0,r_j - r_i) - \max(0,d_i - d_k)) \big) + C\, r_j \Big)$$

$$u.c.\ W_{j,k} + c_i \big(\max(0,p_i - \max(0,r_j - r_i)) - \max(0, p_i - \max(0,r_j - r_i) - \max(0,d_i - d_k)) \big) + \mathrm{C}\, r_j > C\, d_k$$

As $C * d_k$ does not depend on j, we can first compute the maximum of the expression $W_{j,k} + c_i * (\max(0,p_i - \max(0,r_j - r_i)) - \max(0,p_i - \max(0,r_j - r_i) - \max(0,d_i - d_k))) + C * r_j$ and then check whether it is greater than $C * d_k$.

For all j such that $p_i \le \max(0,\ r_j - r_i)$, $p_i - \max(0,\ r_j - r_i)$ is smaller than or equal to $\max(0, p_i - \max(0,\ r_j - r_i))$ and $f(j) = W_{j,k} +\ C * r_j$ which, under our hypothesis does not exceed $C * d_k$. Consequently, we can replace $\max(0,p_i - \max(0,r_j - r_i))$ by $p_i - \max(0,r_j - r_i)$ in the expression above.
Thus, we seek to maximize

$$W_{j,k} + c_i * (p_i - \max(0,r_j - r_i) - \max(0,p_i - \max(0,r_j - r_i) - d_i + d_k)) + C * r_j$$

which is equivalent to

$$\max_j (W_{j,k} + c_i * (p_i - \max(0,r_j - r_i, p_i - d_i + d_k)) + C * r_j).$$

This problem can be split into two sub-problems P_1 and P_2 by adding respectively the constraint $r_j \le r_i + \max(0, p_i - d_i + d_k)$ and the constraint $r_j \ge r_i + \max(0, p_i - d_i + d_k)$. Indeed, an optimal solution j^* of the original problem is either an optimal solution j_1^* for P_1 or an optimal solution j_2^* for P_2.

$$\max_j \left(W_{j,k} + C * r_j \right) \quad \text{u.c. } r_j \le r_i + \max(0, p_i - d_i + d_k) \qquad (P_1)$$

$$\max_j \left(W_{j,k} + (C - c_i) * r_j \right) \quad \text{u.c. } r_j \ge r_i + \max(0, p_i - d_i + d_k) \qquad (P_2)$$

B.3.2.2.1. Resolution of P_1 for all i

We propose an $O(n^2)$ algorithm to compute the optima of P_1 for all pairs of activities (A_i, A_k). The basic idea consists of rewriting the constraint of P_1. In particular, $r_j \le r_i + max(0, p_i - d_i + d_k)$ is equivalent to $r_j \le r_{f(k,i)}$, where $A_{f(k,i)}$ is the activity with the largest release date such that either $r_{f(k,i)} \le r_i$ or $r_{f(k,i)} \le r_i + p_i - d_i + d_k$. Let now $\Omega_{k,u}$ denote the optimum of $W_{j,k} + C^* r_j$ under the constraint $r_j \le r_u$, then the optimum of P_1 is $\Omega_{k, f(k, i)}$. This rewriting is of great interest since computing the values of $\Omega_{k, u}$ and $f(k, i)$ can be done in linear time for a given k.

Algorithm B.5.

```
R = activities sorted in increasing order of $r_{act}$
RPD = activities sorted in inc. order of $r_{act} + p_{act} - d_{act}$
for any activity act_k
  let Ω be an array of integers (convention: Ω[0] = -∞)
  for iR = 1 to iR = n
    Ω[iR] = max(Ω[iR - 1], $W_{PE}(r_{R[iR]}, d_{act\_k})$)
  end for
  for iR = 1 to iR = n
    $f_{R[iR]}$ = iR
  end for
  iR = 0, iRPD = 0
  while (iR < n or iRPD < n)
    if (iR < n) then Rval = $r_{R[iR + 1]}$
    else Rval = ∞ end if
    if (iRPD < n) then
      RPDval = $r_{RPD[iRPD + 1]} + p_{RPD[iRPD + 1]} - d_{RPD[iRPD + 1]} + d_{act\_k}$
    else
      RPDval = ∞
    end if
    if (Rval ≤ RPDval) then
      iR = iR + 1
    else
      iRPD = iRPD + 1
      $f_{RPD[iRPD]}$ = max($f_{RPD[iRPD]}$, iR)
    end if
  end while
end for
```

Let us comment the algorithm.

- Lines 1 and 2 achieve two initial sorts of activities. Notice that since `RPD` is sorted in increasing order of $r_i + p_i - d_i$, it is also sorted in increasing order of $r_i + p_i - d_i + d_k$ for any value of d_k.
- Lines 4 to 7 compute the values of Ω. We rely on the fact that activities are taken in increasing order of release dates and thus, the recurrence property of line 6 holds.
- Lines 11 to 26 compute the values of $f(k, i)$. Since k is fixed, we use the notation f_{A_i} to denote the index of the activity in `R` such that its release date is equal to $r_{f(k, i)}$. Since $A_{f(k, i)}$ is the activity with the largest release date such that either $r_{f(k, i)} \le r_i$ or $r_{f(k, i)} \le r_i + p_i - d_i + d_k$, we first initialize $\mathtt{f}_{\mathtt{R[iR]}}$ `= iR` (lines 8 to 10) and then iterate over both lists `R` and `RPD` at the same time. `Rval` and `RPDval` correspond respectively to the release date of the current activity in R and to the "$r_i+p_i-d_i+d_k$" of the current activity in `RPD`. If it comes that `Rval > RPDval`, then the value of $\mathtt{f}_{\mathtt{RPD[iRPD]}}$ can be updated (line 24) since there the release date of the `iR`th activity in `R` has the largest release date lower than or equal to $r_i+p_i-d_i+d_k$.

Notice that this algorithm runs in $O(n^2)$ since it consists in two initial sorts and in one outer loop over the n activities and one inner loop (2 * n iterations).

B.3.2.2.2. Resolution of P_2 for all i

For each different value λ of c_i, the problem P_2 can be solved with a similar algorithm. C is replaced by $C - \lambda$ which is taken as a constant, and the optima for all pairs of activities (A_i, A_k) with $c_i = \lambda$ are computed (with a simple transformation needed to accommodate the direction of the inequality $r_j \ge r_i + max(0, p_i - d_i + d_k)$). We then have an $O(n^2 * |\{c_i\}|)$ algorithm to solve P_2. Notice that in the m-Machine case, this reduces to a simple $O(n^2)$.
We now show that this worst case complexity can be brought down to $O(n^2 * log(|\{c_i\}|))$. To simplify notations, we suppose that (1) activities are sorted in increasing order of release dates and (2) all release dates are distinct. The basic idea of the algorithm is to fix the value of k and then to solve the problem P_2 for all values of i at the same time.
First, we introduce $g(k, i)$, the first index j such that $r_j \ge r_i + \max(0, p_i - d_i + d_k)$. For the values of i for which $g(k, i)$ is not defined, the problem P_2 has no solution. As for $f(k, i)$ (*cf.* Section B.3.2.2.1), the values of $g(k, i)$ can be computed in linear time. For each activity A_u a list S_u of activities A_i such $g(k, i) = u$ can be built in linear time.
We introduce some more notations:

- For $l \ne m$, let $slope(A_l, A_m) = (W_{l,k} - W_{m,k}) / (r_m - r_l)$.
- Let $h(x) = \min(\{C - c_u, C - c_u \ge x\} \cup \{+\infty\})$.

The algorithm builds and maintains a list Q of activities initialized to $Q = (A_n)$. As we will see later on, Q contains, throughout the algorithm, the activity A_j which leads to the optimum of P_2 for some values of i.

The structure of the algorithm is the following one. We iterate from $l = n$ down to $l = 1$ (*i.e.*, activities are taken in decreasing order of release dates). For each activity A_l, we make the following computations:

1. Compute the activity A_u in Q such that slope(A_l, A_u) is minimum, *i.e.*, $slope(A_l, A_u) = \min\{slope(A_l, A_x), A_x \in Q\}$, with ties broken by taking u maximal (see Figure B-12).
2. Add A_l to the beginning of Q, *i.e.*, set $Q = (A_l, Q(A_u))$. Where $Q(A_u)$ denotes the tail of the list Q after A_u (A_u included).
3. If A_u has a successor A_{u+} in Q and if $h(slope(A_l, A_u)) = h(slope(A_u, A_{u+}))$, remove A_u from Q.
4. For each activity A_i in the list S_l, compute the maximum of $W_{j,k} + (C - c_i) * r_j$ over all activities A_j in Q.

Procedures 1 and 4 will be described later on.

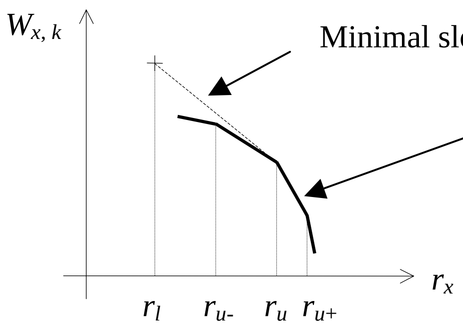


Figure B-12. Selection of A_u in Q. Notice that the values of slope(A_x, A_{x+}) and of h(slope(A_x, A_{x+})) are strictly increasing when A_x goes through Q.

Proposition B-18.

The values computed in step 4 of the algorithm are optimal values of f_i.

Proof.

We rely on the fact that Q is dominant *i.e.*, if at a step l of the algorithm, an activity A_q has been removed from Q, and if A_q is an optimal solution of (P_2) for a given i (such that $g(k, i) \leq q$), then there exists an optimal solution of (P_2) in Q for the same i. A_q can be "removed" from Q for two reasons. Either in step (2) or in step (3). We prove that in both cases, the dominance property holds when exiting respectively step 2 and step 3. To simplify notations, we introduce $f_i(A_j) = W_{j,k} + (C - c_i) * r_j$

- If A_q is removed in step 2. There exists $A_m \in Q$ such that (a) $slope(A_l, A_q) \geq slope(A_l, A_m)$ and (b) $r_q \leq r_m$. Let us compute $f_i(A_m) - f_i(A_q)$.

$$\begin{aligned}f_i(A_m)-f_i(A_q) &= f_i(A_l)-f_i(A_q)-(f_i(A_l)-f_i(A_m))\\ &=(r_q-r_l)*slope(A_l,A_q)-(r_m-r_l)*slope(A_l,A_m)+(C-c_i)*(r_m-r_q)\\ &\geq(r_q-r_l)*slope(A_l,A_m)-(r_m-r_l)*slope(A_l,A_m)+(C-c_i)*(r_m-r_q)\\ &\geq(r_m-r_q)*(C-c_i-slope(A_l,A_m))\geq(r_m-r_q)*(C-c_i-slope(A_l,A_q))\end{aligned}$$

If $slope(A_l, A_q) > C - c_i$ then it would be easy to prove that $f_i(A_q) < f_i(A_l)$; which is impossible since A_q is optimal. Thus, we can suppose that $slope(A_l, A_q) \leq C - c_i$. This leads to $f_i(A_m) - f_i(A_q) \geq 0$. Consequently, A_m is also optimal, which concludes the proof.

- If A_q is removed in step 3. In such a case, when entering step 3, A_q had one predecessor and one successor in Q. Let A_l and A_{q+} be respectively its predecessor and its successor in Q. A_q is optimal for (P_2).

  Thus, $f_i(A_q) \geq f_i(A_l)$ consequently, $W_{q,k} + (C - c_i) * r_q \geq W_{l,k} + (C - c_i) * r_l$. This leads to $slope(A_l, A_q) \leq C - c_i$. But we know that $h(slope(A_l, A_q)) = h(slope(A_q, A_{q+}))$, thus $slope(A_q, A_{q+})$ cannot be strictly greater than $C - c_i$. As a conclusion, $slope(A_q, A_{q+}) \leq C - c_i$. This directly implies $f_i(A_{q+}) \geq f_i(A_q)$. □

An interesting property of this algorithm is that the length of Q is always lower than or equal to $|\{c_i\}| + 1$, where $|\{c_i\}|$ is the number of distinct values of c_i in the instance (*cf.* step 3). Let us describe steps 1 and 4 of the algorithm. In both cases, a simple loop over activities in Q could solve the optimization problems of steps 1 and 4. This would lead to an overall complexity (for a given k) of $O(n * |\{c_i\}|)$, to compute for all i, an optimal solution of (P_2). However, this complexity can be sharpened. Notice that the values of $slope(A_x, A_{x+})$ and of $h(slope(A_x, A_{x+}))$ are strictly increasing when A_x goes through Q, and that Q can be implemented as a stack embedded in an array.

- A_u as defined in step 1 is the activity of Q such that $slope(A_l, A_u)$ is lower than or equal to $slope(A_l, A_{u-})$ and strictly lower than $slope(A_l, A_{u+})$ if A_{u-} (the predecessor of A_u in Q) and/or A_{u+} exist. Thus, a dichotomic search for A_u can be performed on Q.
- Step 4 is slightly more complex. Notice that f_i increases over a first part of the list Q and decreases in the remaining part (the value of $slope(A_l, A_{l+})$ becomes strictly greater than $C - c_i$). As a consequence, a dichotomic search for the activity A_j that leads to the optimum can be performed on Q again.

This leads to a complexity (for a given k) of $O(n * log(|\{c_i\}|))$, to compute for all i, an optimal solution of (P_2). The overall complexity of the constraint propagation algorithm is then $O(n^2 * log(|\{c_i\}|))$. Obviously, this complexity becomes quadratic when the algorithm is applied to an instance of the m-Machine Problem.

B.3.2.3. "Left-Shift/Right-Shift" Time-Bound Adjustments for the CuSP

As in Section B.3.2.2, the values of W_{Sh} can 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 B-19.
If $\exists\ t_1 \mid t_1 < t_2$ and $W_{Sh}(t_1, t_2) - W_{Sh}(A_i, t_1, t_2) + c_i * \min(t_2 - t_1, p_i^+(t_1)) > C * (t_2 - t_1)$ then a valid lower bound of the end time of A_i is

$$t_2 + \frac{W_{Sh}(t_1, t_2) - W_{Sh}(A_i, t_1, t_2) + c_i * \min(t_2 - t_1, p_i^+(t_1)) - C * (t_2 - t_1)}{c_i}.$$

Proof.
Similar to proof of Proposition B-17. □

There is an obvious $O(n^3)$ algorithm to compute all the adjustments which can be obtained on the intervals $[t_1, t_2]$ which correspond to potential local minima of the slack function. 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 $O(1)$. The overall complexity of the algorithm is then $O(n^3)$.
In spite of our efforts, we were unable to exhibit a quadratic algorithm to compute all the adjustments on the $O(n^2)$ intervals under consideration.

B.3.2.4. Synthesis of Theoretical Results

The most satisfactory result is obtained for the Fully Elastic CuSP. Indeed, time-bound adjustments are perfectly characterized (Proposition B-16).
It is easy to see that the deductions made by the fully elastic techniques (both necessary condition and time-bound adjustment) are subsumed by partially elastic techniques which are in turn subsumed by left-shift / right-shift techniques. However, the weaker the relaxation, is the cheaper the complexity is.

	Fully Elastic	Partially Elastic	Left-Sh. / Right-Sh.
Characterization	Perfect	-	-
Complexity	$O(n^2)$	$O(n^2 * log\|\{c_i\}\|)$	$O(n^3)$
Method	One-Mac. reduction	Slack computation	Slack computation

Figure B-13. A brief comparison of the three adjustment techniques

Three necessary conditions for the existence of a feasible schedule for a given instance of the Cumulative Scheduling Problem, and three deductive algorithms to adjust the time-bounds of activities have been presented. Two of the three proposed techniques correspond to well-defined relaxations of the Cumulative Scheduling Problem: the fully elastic relaxation and the partially elastic relaxation. These techniques can be used not only for standard scheduling problems, but also for preemptive scheduling problems. In addition to that, the fully elastic relaxation also applies when an activity requires an amount of resource capacity that is not fixed (*e.g.*, because the same activity can be done either by 2 people in 3 days or by 3 people in 2 days) or even allowed to vary over time (*e.g.*, 2 people on day 1 and 4 people on day 2). The third technique (left-shift/right-shift), which is the most powerful but also the most expensive, only applies to "non-elastic non-preemptive problems." Notice that the satisfiability tests are such that they provide the same answers when an activity A_i, which requires c_i units of the resource, is replaced by c_i activities A_i^j, each of which requires one unit of the resource. This is not true for the time-bound adjustments.

Several questions are still open at this point.

- First, for the left-shift/right-shift technique, we have shown that the energetic tests can be limited to $O(n^2)$ time intervals. We have also provided a precise characterization of these intervals. However, it could be that this characterization can be sharpened in order to eliminate some intervals and reduce the practical complexity of the corresponding algorithm.
- Second, it seems reasonable to think that our time-bound adjustments could be sharpened. Even though the energetic tests can be limited (without any loss) to a given set of intervals, it could be that the corresponding adjustment rules cannot. A related open question is whether the time-bound adjustment schemes proposed in the previous sections subsume the rules already presented in [Caseau and Laburthe, 1996a], [Lopez *et al.*, 1992], [Nuijten, 1994], [Nuijten and Aarts, 1996]. A partial but positive answer to this question is provided in [Le Pape and Baptiste, 1998c].

B.4. Over-loaded Resources[8]

As shown in the previous sections, various classes of strict resource constraints have been developed in the literature to enable the resolution of computationally demanding problems. These global constraints have enabled the development of many industrial applications based on constraint programming. Overloaded resources, *i.e.*, resources that can sub-contract a given amount of activities, have been less studied. The problem induced by these resources can be seen as an over-constrained problem (the strict resource constraint cannot be satisfied and one tries to satisfy it as "much" as possible). A lot of academic work has been performed on over-constrained problems, and many extensions of the constraint satisfaction paradigm have been proposed (see, *e.g.*, [Freuder and Wallace, 1992], [Bistarelli *et al.*, 1995], [Schiex *et al.*, 1995]). It appears that such extensions could be highly useful in practice. Indeed, industrial problems tend to include many "preference" constraints, that cannot be all satisfied at the same time.

In this section, we develop a resource constraint propagation algorithm that can be used for overloaded resources. If we consider this resource constraint alone (*i.e.*, without taking care of the other components of the scheduling problem), it appears that we study the decision variant of the $(1|r_j|\Sigma U_j)$ problem. Following the classical terminology for this problem, an activity is late if it is scheduled after its due-date. It is on-time otherwise. For our resource constraint, "late" corresponds to "sub-contracted" and "on-time" corresponds to "performed on the resource". We refused to use the terminology late / on-time when defining, in the introductory chapter, the resource constraint because we think it is much more restrictive than the other one. In this technical section, we come back to the classical terminology.

Recall that the number of sub-contracted (late) activities is represented by a constrained variable *reject* whose domain is $[0, n]$. Each activity A_i is described by a binary variable $in(A_i)$ that states whether the activity is performed on the resource (on-time) $in(A_i) = 1$ or sub-contracted (late) $in(A_i) = 0$, and by an integer variable $start(A_i)$ (the start time), whose domain is $[r_i, d_i - p_i]$. The first constraint to satisfy is $\Sigma(1 - in(A_i)) = reject$. Simple arc-consistency techniques can be used to propagate this constraint. The "classical" resource constraint is modified to work not only on the domains of the start time variables $start(A_i)$, but also on the activity status variables $in(A_i)$. It states that activities which must be on-time cannot overlap in time and that there are *reject* late activities:

[8] Most of the results presented in this section come from [Baptiste, 1998a], [Baptiste *et al.*, 1998c].

- $\forall t$, $|\{Ai$ such that $in(A_i) = 1$ and $start(A_i) \leq t < start(A_i) + p_i\}| \leq 1$,
- $\Sigma(1 - in(A_i)) = reject$.

To allow further pruning, the maximal value of *reject* and dominance relations between the activity status variables (of the form $in(A_i) \Rightarrow in(A_j)$) can also be optionally taken into account in this constraint (such dominance properties apply for the $(1|r_j|\Sigma U_j)$, *cf.*, Section C.4.1.3). Constraint propagation reduces the domains of both the $start(A_i)$ and $in(A_i)$ variables. O denotes the set of activities that have to be on-time ($in(A_i)$ has been bound to 1) and L denotes the set of activities that have to be late ($in(A_i) = 0$).

The propagation of the modified resource constraint consists of four interrelated parts.

1. In the first part, classical resource constraint propagation techniques are used on the on-time activities: disjunctive constraint propagation and edge-finding are applied on O (*cf.*, Section B.1.2. and Section B.1.3).
2. In the second part, for each activity A_i such that $in(A_i)$ is unbound, we try to add to the set O of activities that must be on-time (1) the activity A_i and (2), according to our dominance property, all the activities A_j such that $in(A_i) \Rightarrow in(A_j)$. The resource constraint is propagated as described in part 1. If an inconsistency is triggered, then $in(A_i)$ can be set to 0. If the propagation of the disjunctive resource constraint does not trigger a contradiction then the release date r'_i and the due date d'_i obtained after the propagation can be kept and imposed as the new release ($r_i = r'_i$) and as the new due date ($d_i = d'_i$) of the activity[9]. Notice that one pass of such a propagation scheme runs in $O(n^3)$ since for each activity the edge-finding algorithm, itself running in $O(n^2)$, is called.
3. The third part determines a lower bound for *reject* (Section B.4.1). Two techniques are proposed. The first one relies on the preemptive relaxation of the $(1|r_j|\Sigma U_j)$ problem. An $O(n^4)$ dynamic programming algorithm is proposed (Section B.4.1.1.). It improves the $O(n^5)$ algorithm described in [Lawler, 1990]. The second lower bound is itself a relaxation of the preemptive relaxation (Section B.4.1.2.). It can be computed in $O(n^2)$ and is very useful in the remaining parts of the constraint propagation.
4. The fourth part focuses on the $in(A_i)$ variables (Section B.4.2).

[9] A drawback of this mechanism is that it may strengthen the time-bounds of activities that won't execute on the machine *i.e.*, activities that will be sub-contracted. To avoid this drawback and to keep the generality of the propagation mechanism, one should introduce fictive release dates and deadlines associated to the given activity on the given machine. These release dates and deadlines would then be adjusted as described above and would be propagated to the "true" release dates and to the deadlines of activities as soon it is would be known that a given activity is on-time.

Of course, when the domain of a variable is modified by one of the four parts above, the overall propagation process restarts.

B.4.1. *Lower Bound Computation*

Any lower bound of the $(1|r_j|\Sigma U_j)$ problem is a valid lower bound of the constrained variable *reject*. Some special cases of the $(1|r_j|\Sigma U_j)$ problem are solvable in polynomial time. Moore's well-known algorithm [Moore, 1968] solves in $O(n \log(n))$ steps the special case where release dates are equal. Moreover, when release and due dates of activities are ordered similarly ($r_i < r_j \Rightarrow d_i \leq d_j$), the problem is solvable in a quadratic amount of steps ([Kise *et al.*, 1978]). This result has been extended by [Dauzère-Peres and Sevaux, 1998a] to the case where $[r_i < r_j] \Rightarrow [d_i \leq d_j$ or $r_j + p_j + p_i > d_i]$. A simple way to obtain a lower bound of the $(1|r_j|\Sigma U_j)$ problem is to relax some release dates (respectively due dates) so that the relaxed problem fits in one of the above special cases. Another special case, the $(1|r_j,p_j=p|\Sigma U_j)$ problem is solvable in $O(n^3 \log(n))$ [Carlier, 1984][10].

Lower bounding techniques have also been developed for the general problem. [Dauzère-Pérès, 1995] and [Dauzère-Peres and Sevaux, 1998a] propose several linear programming formulations of the $(1|r_j|\Sigma U_j)$ problem. The resolution of the relaxed linear programs allows to obtain lower bounds of the number of late activities. [Péridy *et al.*, 1998] propose to use a MIP formulation of the preemptive problem. Surrogate duality is used to compute a lower bound of this MIP. [Lawler, 1990] has proposed a strongly polynomial algorithm for the preemptive problem $(1|\text{pmtn},r_j|\Sigma U_j)$. Time and space bounds of this algorithm are respectively $O(n^3k^2)$ and $O(n\,k^2)$, where k is the number of distinct release dates. So, $O(n^5)$ and $O(n^3)$ if all release dates are distinct. Notice that Lawler's algorithm also applies for minimizing the weighted number of late activities $(1|\text{pmtn},r_j|\Sigma w_j U_j)$. It then becomes pseudo-polynomial in the sum W of the weights of the activities; the bounds being respectively $O(n\,k^2W^2)$ and $O(k^2W)$.

In this section, we first show that the preemptive lower bound can be reached in $O(n^4)$ (Section B.4.1.1). Despite this improvement, such a complexity remains high and we propose in Section B.4.1.2 to use a weaker lower bound that can be obtained in $O(n^2\, log(n))$.

[10] Notice that an $O(n^{10})$ algorithm is provided in Appendix 2 for solving the special case of the weighted preemptive problem where processing times are equal. The non-preemptive problem can be solved in $O(n^7)$ (Appendix 3).

B.4.1.1. The Preemptive Lower Bound

B.4.1.1.1. Reformulation of the Problem

As noticed in [Lawler, 1990], the preemptive problem reduces to finding a maximum subset of activities that is feasible, *i.e.*, which can be preemptively scheduled on a single machine. It is well known that testing the feasibility of a subset O of activities can be achieved by computing JPS_O, the Jackson Preemptive Schedule of O. We recall some fundamental properties of Jackson Preemptive Schedule (for a proof, see for instance [Carlier, 1984]).

- If an activity is scheduled on JPS_O after its due date, O is not feasible.
- The makespan C_O (*i.e.*, the time at which all activities are finished) of JPS_O is minimal among all preemptive schedules.

From now on, we assume that activities are sorted in increasing order of their due dates ($d_1 \leq d_2 \leq ... \leq d_n$). For any integer a and any activity A_k, let $S_k(a)$ be the set of activities A_i such that $a \leq r_i$ and $i \leq k$. Given an interval $[a, b]$ and a set of activities O, $slack(O, a, b)$ denotes the time during which JPS_O is idle over $[a, b]$. By convention, $C_\varnothing = -\infty$.
Let a and b be any values such that $a \leq b$. We introduce three definitions. As we will see later on, we will be mainly interested in the values of a and b that are release dates.

Definition.
Let $C_k(a, m)$ be the minimal time at which m activities in $S_k(a)$ can be completed.

$$C_k(a, m) = \min\{\{\infty\} \cup \{C_O \mid O \subseteq S_k(a), O \text{ feasible and } |O| = m\}\}$$

Definition.
Let $\pi_k(a, b)$ be the maximal number of activities in $S_{k-1}(a)$ that can be scheduled before b.

$$\pi_k(a, b) = \max\{|O| \mid O \subseteq S_{k-1}(a), O \text{ feasible and } C_O \leq b\}$$

Definition.
For any $b \geq r_k$, let $\mu_k(a, b)$ be the largest possible slack over the interval $[r_k, b]$ among sets that realize $\pi_k(a, b)$.

$$\mu_k(a, b) = \max\{slack(O, r_k, b) \mid O \subseteq S_{k-1}(a), O \text{ feasible}, |O| = \pi_k(a, b), C_O \leq b\}$$

B.4.1.1.2. Some Fundamental Properties

We prove three propositions that exhibit a strong link between the values of C, π and μ. We first show that C_k can be computed in function of C_{k-1}, π_k and μ_k. To write a compact formula, we introduce $f_k(x)$ which is equal to x if $x \leq d_k$ and to $+\infty$ otherwise.

Proposition B-20.

If $A_k \notin S_k(a)$ (*i.e.*, $r_k < a$) then $C_k(a, m) = C_{k\text{-}1}(a, m)$. If $A_k \in S_k(a)$ then

$$\begin{aligned}C_k(a, m) = f_k(\min(&C_{k\text{-}1}(a, m),\\ &\max(r_k, C_{k\text{-}1}(a, m - 1)) + p_k,\\ &\min_{r_u \geq r_k} (C_{k\text{-}1}(r_u, m - 1 - \pi_k(a, r_u)) + \max(0, p_k - \mu_k(a, r_u)))))\end{aligned}$$

Proof.

The first part of the proposition is obvious. Consider now that $A_k \in S_k(a)$. Let C' be the value corresponding to the right term in the equation above.

We first prove that $C' \leq C_k(a, m)$. We can suppose that $C_k(a, m)$ has a finite value (if not, the result is obvious). Several cases can occur:

- Either there is a set O that realizes $C_k(a, m)$ such that $A_k \notin O$. Then $C_k(a, m)$ is equal to $C_{k\text{-}1}(a, m)$. Consequently, $C' \leq C_k(a, m)$.
- Or A_k belongs to all the sets that realize $C_k(a, m)$ but there is one, say O, such that A_k is fully executed on JPS_O after all other activities. Then, we have $C_k(a, m) = \max(C_{O - \{Ak\}}, r_k) + p_k$. Moreover, $C_{O - \{Ak\}} \geq C_{k\text{-}1}(a, m - 1)$. Thus, $C_k(a, m) \geq \max(r_k, C_{k\text{-}1}(a, m - 1)) + p_k \geq C'$.
- Or A_k belongs to all the sets that realize $C_k(a, m)$ and A_k is never fully executed after all the other activities. Let O be a set that realizes $C_k(a, m)$ and let t be the maximal time point such that (a) A_k executes in $[t - 1, t]$ on JPS_O and (b) another activity executes after time t on JPS_O. Given our hypothesis, such a time point exists. Moreover, because of the particular structure of Jackson Preemptive Schedules, the first time point at which an activity executes after t is the release date r_u of an activity A_u (with $r_u \geq r_k$).

  Consider now an activity A_i ($i \neq k$) which starts on JPS_O before r_u. Since $d_i < d_k$, A_i ends before r_u (otherwise A_i would be scheduled at time $t - 1$ on JPS_O instead of A_k). Let then O_1 be the set of activities in $O - \{A_k\}$ that end before r_u on JPS_O. Let O_2 be the set of activities in $O - \{A_k\}$ that start after or at r_u.

  If O_1 is not maximal (*i.e.*, $|O_1| < \pi_k(a, r_u)$), then consider the set O'_1 that realizes $\pi_k(a, r_u)$. It is obvious that $O'_1 \cup O_2$ is feasible and that it contains as many activities as O. Moreover, $C_{O'1 \cup O2}$ is lower than or equal to C_O; which contradicts our hypothesis that A_k is in all the sets that realize $C_k(a, m)$. We know that O_1 contains $\pi_k(a, r_u)$ activities, moreover $slack(O_1, r_k, r_u)$ time units are available before r_u to schedule A_k. Thus,

$$\begin{aligned}C_O &= \max(0, p_k - slack(O_1, r_k, r_u)) + C_{O2}\\ &\geq C_{k\text{-}1}(r_u, m - 1 - \pi_k(a, r_u)) + \max(0, p_k - \mu_k(a, r_u)) \geq C'\end{aligned}$$

We now prove that $C' \geq C_k(a, m)$. Notice that if $C' = \infty$, the result is obvious. We can then suppose that C' is the minimum of one of the three terms.

- If $C' = C_{k-1}(a, m)$. Since $C_k(a, m) \leq C_{k-1}(a, m)$, $C' \geq C_k(a, m)$.
- If $C' = \max(r_k, C_{k-1}(a, m-1)) + p_k$. Let O be the set that realizes $C_{k-1}(a, m-1)$. A_k is not late if it is "added" at the end of JPS_O (because C' is finite). Thus, the set $O \cup \{A_k\}$ contains m activities and is feasible. Moreover, it is a subset of $S_k(a)$. As a consequence, $C' \geq C_{O \cup \{Ak\}} \geq C_k(a, m)$.
- If $\exists\, r_u \geq r_k \mid C' = C_{k-1}(r_u, m - 1 - \pi_k(a, r_u)) + \max(0, p_k - \mu_k(a, r_u))$. Let O_1 be the set that realizes $\mu_k(a, r_u)$ and O_2 be the set that realizes $C_{k-1}(r_u, m - 1 - \pi_k(a, r_u))$. First, notice that $O_1 \cup O_2 \cup \{A_k\}$ obviously belongs to $S_k(a)$. Second, notice that $O_1 \cup O_2 \cup \{A_k\}$ is feasible. Indeed, on $JPS_{O1 \cup O2}$, there are $\mu_k(a, r_u)$ "holes" between r_k and r_u and thus, the quantity $C_{k-1}(r_u, m-1-\pi_k(a,r_u))+max(0,p_k-\mu_k(a,r_u))$ is the makespan of $JPS_{O1 \cup O2}$ on which A_k has been scheduled as soon as possible in the "holes". This makespan is lower than d_k (otherwise $C' = \infty$). Third, notice that there are m activities in $O_1 \cup O_2 \cup \{A_k\}$. Indeed, according to the definition of μ, before r_u, $\pi_k(a, r_u)$ activities are scheduled. After r_u, $m - 1 - \pi_k(a, r_u)$ activities are scheduled; which means that $m - 1$ activities in $O_1 \cup O_2$ are scheduled.

We have proven that $C' \geq C_{O1 \cup O2 \cup \{Ak\}}$; thus $C' \geq C_k(a, m)$. □

Proposition B-21.

For any $a \leq b$, $\pi_k(a, b) = \max\{m \mid C_{k-1}(a, m) \leq b\}$.

Proof.

Let $\pi' = \max\{m \mid C_{k-1}(a, m) \leq b\}$.

Let O be the set that realizes $C_{k-1}(a, \pi')$. O is a subset of $S_{k-1}(a)$, O is feasible and $C_O = C_{k-1}(a, \pi') \leq b$; consequently, $\pi' \leq \pi_k(a, b)$.

Let O be the set that realizes $\pi_k(a, b)$. According to the definition of C, $C_k(a, |O|)$ is lower than or equal to C_O. Consequently $C_k(a, |O|) \leq b$ and thus, $\pi' \geq \pi_k(a, b)$. □

Proposition B-22.

$\forall\, a, b$ and $\forall\, A_k$ such that $a \leq r_k \leq b$,

$$\mu_k(a, b) = \max\Big(b - \max(C_{k-1}(a, \pi_k(a, b)), r_k), \max_{\begin{cases} r_k \leq r_v < b \\ \pi_k(a,b)=\pi_k(a,r_v)+\pi_k(r_v,b) \\ \pi_k(r_v,b)>0 \end{cases}} (\mu_k(a, r_v) + b - C_{k-1}(r_v, \pi_k(r_v, b)))\Big)$$

Proof.
Let μ' be the value corresponding to the right term in the previous equation.

We first prove that $\mu' \geq \mu_k(a, b)$. Let O be the set that realizes $\mu_k(a, b)$. The proof relies on the fact that if there is a time point $t \in [r_k, b]$ such that JPS_O is idle in $[t - 1, t]$, the computation of the maximal slack can be decomposed into the computation of the maximal slack over $[r_k, t]$ and over $[t, b]$. In the following, we distinguish two cases.

- If $C_O \leq r_k$, then $slack(O, r_k, b) = b - r_k$. Moreover, $C_{k-1}(a, \pi_k(a, b)) \leq C_O \leq r_k$. Thus, $b - \max(C_{k-1}(a, \pi_k(a, b)), r_k)$ is equal to $b - r_k$. Consequently, $\mu' \geq \mu_k(a, b)$.
- If $C_O > r_k$, let t be the largest time point such that JPS_O is idle immediately before t and never idle in the interval $[t, C_O]$. According to the structure of a Jackson Preemptive Schedule, t is a release date; say r_v. Two cases are distinguished.

  First, if $r_v \leq r_k$ then $slack(O, r_k, b) = b - C_O \leq b - C_{k-1}(a, \pi_k(a, b))$; thus $\mu' \geq \mu_k(a, b)$. Second, suppose that $r_v > r_k$. According to the definition of r_v, $\pi_k(r_v, b) > 0$. We claim that $\pi_k(a, b) = \pi_k(a, r_v) + \pi_k(r_v, b)$. Indeed, consider O_1 the subset of O that consists of the activities ending before or at r_v on JPS_O and O_2 the subset of O that consists of the activities ending after r_v on JPS_O. On the one hand, $|O_1| + |O_2| = |O| = \pi_k(a, b)$, on the other hand, $|O_1| \leq \pi_k(a, r_v)$ and $|O_2| \leq \pi_k(r_v, b)$. Suppose that the first inequality is strict, let then O'_1 be the set that realizes $\pi_k(a, r_v)$. It is easy to see that the set $O'_1 \cup O_2$ is included in $S_{k-1}(a)$, that it is feasible and that its JPS ends before b. Moreover, $O'_1 \cup O_2$ is larger than O, which contradicts the fact that O realizes $\pi_k(a, b)$; consequently, $|O_1| = \pi_k(a, r_v)$. Similarly, we can prove that $|O_2| = \pi_k(r_v, b)$. Let us compute the slack of O over $[r_k, b]$. $slack(O, r_k, b) = slack(O_1, r_k, r_v) + slack(O_2, r_v, b)$. As a consequence, $slack(O, r_k, b) \leq \mu_k(a, r_v) + b - C_{k-1}(r_v, \pi_k(r_v, b)) \leq \mu'$.

We now prove that $\mu' \leq \mu_k(a, b)$. Let us distinguish the two following cases:

- If $\mu' = b - \max(r_k, C_{k-1}(a, \pi_k(a,b)))$, let then O be the set that realizes $C_{k-1}(a, \pi_k(a,b))$. JPS_O is idle after $C_{k-1}(a, \pi_k(a, b))$ and its makespan is lower than or equal to b, thus $slack(O, r_k, b) \geq b - \max(C_{k-1}(a, \pi_k(a, b)), r_k)$. Moreover, $O \subseteq S_{k-1}(a)$, O is feasible, $|O| = \pi_k(a, b)$ and $C_O \leq b$; thus $\mu_k(a, b) \geq slack(O, r_k, b)$. Consequently, $\mu' \leq \mu_k(a, b)$.
- If there is a release date r_v such that (1) $r_k \leq r_v < b$, (2) $\pi_k(a, b) = \pi_k(a, r_v) + \pi_k(r_v, b)$, (3) $\pi_k(r_v, b) > 0$ and (4) $\mu' = \mu_k(a, r_v) + b - C_{k-1}(r_v, \pi_k(r_v, b))$, let then O_1 be the set that realizes $\mu_k(a, r_v)$ and let O_2 be the set that realizes $C_{k-1}(r_v, \pi_k(r_v, b))$. Notice that $slack(O_1 \cup O_2, r_k, b) \geq \mu'$ since the quantity $\mu_k(a, r_v) + b - C_{k-1}(r_v, \pi_k(r_v, b))$ is the slack of O_1 before r_v plus a lower bound of the slack of O_2 in $[r_v, b]$ ($\pi_k(r_v, b) > 0$ ensures that $C_{k-1}(r_v, \pi_k(r_v, b))$ is finite). Moreover, O_1 and O_2 are disjoint because $\forall A_i \in O_1$, $r_i < r_v$ and $\forall A_i \in O_2$, $r_i \geq r_v$. Thus, $|O_1 \cup O_2| = \pi_k(a, r_v) + \pi_k(r_v, b) = \pi_k(a, b)$. It is easy to

verify that $O_1 \cup O_2 \subseteq S_{k\text{-}1}(a)$, that $O_1 \cup O_2$ is feasible and that $C_{O1 \cup O2} \leq b$; thus $\mu_k(a, b) \geq slack(O_1 \cup O_2, r_k, b)$. Consequently, $\mu' \leq \mu_k(a, b)$. □

Propositions B-20, B-21 and B-22 are the basis of the dynamic programming algorithm that we propose in the following section.

B.4.1.1.3. Overall Algorithm

Our aim is to determine the largest value of m such that $C_n(\min_i r_i, m)$ is finite. The variables of the dynamic programming algorithm correspond to $C_k(a, m)$, $\pi_k(a, b)$ and $\mu_k(a, b)$. They are stored in multi-dimensional arrays. Actually, it is easy to understand that given Propositions 1 and 3, the relevant values of a and b are those corresponding to release dates. Thus, the values of $C_k(a, m)$, $\pi_k(a, b)$ and $\mu_k(a, b)$ are stored in indexed arrays (*e.g.*, $C_k(r_j, m)$ is stored in a 3-dimensional array at the "position" (k, A, m)).

The first step of the algorithm is the computation of $C_1(r_j, m)$ for all release date r_j and all the values of m in $[1, n]$.

- If $m = 0$, $C_1(r_j, 0) = -\infty$
- If $m = 1$ and $r_1 < r_j$, $C_1(r_j, 1) = \infty$
- If $m = 1$ and $r_1 \geq r_j$, $C_1(r_j, 1) = r_1 + p_1$
- If $m > 1$, $C_1(r_j, m) = \infty$

The second step is a loop on k from 2 to n.

- For each release date r_j and each release date r_u, compute $\pi_k(r_j, r_u)$. This computation is done in $O(n)$ thanks to Proposition B-21.
- For each release date r_j and each release date r_u (taken in increasing order), compute the values of $\mu_k(r_j, r_u)$. This is done in $O(n)$ thanks to Proposition B-22. Indeed, for a given value of r_u, we use the pre-computed values of $\mu_k(r_j, r_v)$ (with $r_v < r_u$). Moreover, the tests $r_k \leq r_v < r_u$, $\pi_k(r_j, r_u) = \pi_k(r_j, r_v) + \pi_k(r_v, r_u)$ and $\pi_k(r_v, r_u) > 0$ are computed in constant time.
- For each release date r_j and each value of m, compute $C_k(r_j, m)$. This is done in $O(n)$ thanks to Proposition B-20.

The overall algorithm then runs in $O(n^4)$. A rough analysis in terms of memory consumption leads to an $O(n^3)$ bound. Indeed, three cubic arrays are needed to store the values of $C_k(r_j, m)$, $\pi_k(r_j, r_u)$ and $\mu_k(r_j, r_u)$. However, notice that at each step of the outer loop on k, one only needs the values of C computed at the previous step (k-1). Thus, the algorithm can be implemented with 4 arrays of $n*n$ size (one for π, one for μ, one for the previous values of C and one for the current values of C); which leads to a space complexity of $O(n^2)$.

This algorithm does not exhibit a set O that realizes the optimum of the problem. A backward computation can be done to determine such a set. The space complexity then increases to $O(n^3)$ since all values taken by C must be stored. Since we are mainly interested in the computation of the optimum, which serves as a lower bound of the non-preemptive problem, we do not provide the description of how O can be computed.

B.4.1.1.4. Minimizing the Weighted Number of Late Activities

At this point, an interesting question is whether our algorithm can be extended to solve the weighted version of the problem (*i.e.*, a version of the problem where each activity A_i has a weight $w_i \geq 0$ and where the goal is to minimize the weighted number of late activities). The definitions of the variables C, π and μ can be easily extended:

- $C_k(a, w)$ is the minimal time at which a set of activities in $S_k(a)$, whose weight is greater than or equal to w, can be completed (if no such set exists, $C_k(a, w) = \infty$).
- $\pi_k(a, b)$ is the maximal weighted number of late activities in $S_{k-1}(a)$ that can be scheduled before b.
- $\mu_k(a, b)$ is the largest possible slack over $[r_k, b]$ among sets that realize $\pi_k(a, b)$.

Given these definitions, one could think that Proposition B-20 can be extended as follows:

If $A_k \notin S_k(a)$ (*i.e.*, $r_k < a$) then $C_k(a, w) = C_{k-1}(a, w)$. If $A_k \in S_k(a)$ then

$$\begin{aligned} C_k(a, w) = f_k(\min(&C_{k-1}(a, w),\\ &\max(r_k, C_{k-1}(a, w - w_k)) + p_k,\\ &\min_{r_u \geq r_k} (C_{k-1}(r_u, w - w_k - \pi_k(a, r_u)) + \max(0, p_k - \mu_k(a, r_u))))) \end{aligned}$$

Unfortunately, this extension does not hold. Intuitively, this comes from the fact that, in Proposition B-20, the expression

$$\min_{r_u \geq r_k} (C_{k-1}(r_u, m-1-\pi_k(a,r_u))+\max(0,p_k-\mu_k(a,r_u)))$$

means that it is worth scheduling between a and r_u a maximum number of activities in $S_{k-1}(a)$. On the contrary, if activities are weighted, it can be of interest to schedule a smaller amount of activities (in term of weights) between a and r_u to increase the slack and thus to leave more space to schedule A_k.

The following counter-example illustrates this phenomenon. Consider five weighted activities A_1 ($r_1 = 0$, $p_1 = 3$, $d_1 = 6$, $w_1 = 2$), A_2 ($r_2 = 0$, $p_2 = 3$, $d_2 = 6$, $w_2 = 2$), A_3 ($r_3 = 0$, $p_3 = 2$, $d_3 = 6$, $w_3 = 1$), A_4 ($r_4 = 6$, $p_4 = 1$, $d_4 = 7$, $w_4 = 2$), and A_5 ($r_5 = 0$, $p_5 = 3$, $d_5 = 9$, $w_5 = 10$). It is easy to prove by hand that $C_5(0, 15) = 9$. This is not the result obtained when applying the weighted version of Proposition B-20:

$$\begin{aligned}C_5(0, 15) = f_5(\min(&C_4(0, 15),\\ &\max(0, C_4(0, 15 - 10)) + 3,\\ &C_4(6, 15 - 10 - \pi_4(0, 6)) + \max(0, 3 - \mu_4(0, 6)))\\ = f_5(\min(&\infty, \max(0, 7) + 3, C_4(6, 1) + \max(0, 3)))\\ = f_5(\min(&\infty, 10, 7 + 3)) = \infty\end{aligned}$$

B.4.1.2. *The Relaxed Preemptive Lower Bound*

The One-Machine Problem [Carlier, 1982] is a special case of the $(1|r_j|\Sigma U_j)$ in which all activities must be on-time. Its preemptive relaxation is polynomial. It is well known that there exists a feasible preemptive schedule if and only if over any interval $[r_j, d_k]$, the sum of the processing times of the activities in $S(r_j, d_k) = \{A_i \mid r_j \le r_i \text{ and } d_i \le d_k\}$ is lower than or equal to $d_k - r_j$. As a consequence, the optimum of the following MIP is the minimum number of activities that must be late on any preemptive schedule of the machine (hence, this optimum is a lower bound of the variable *reject*). The binary variable x_i is equal to 1 when an activity is on-time, to 0 otherwise.

$$\begin{aligned}&\min \sum_{i \in \{1,\Lambda,n\}} (1 - x_i)\\ &\forall r_j, \forall d_k > r_j, \sum_{J_i \in S(r_j, d_k)} p_i x_i \le d_k - r_j \qquad\qquad (P)\\ &\forall J_i \in O, x_i = 1 \text{ and } \forall J_i \in L, x_i = 0\\ &\forall i \in \{1, \Lambda, n\}, x_i \in \{0,1\}\end{aligned}$$

The first set of constraints of P represents the resource constraints. The notation (r_j, d_k) refers to the resource constraint over the interval $[r_j, d_k]$. In the following, we focus on the continuous relaxation CP of P. We claim that CP can be solved in $O(n^2\log(n))$ steps. To achieve this result, we first provide a characterization of one vector that realizes the optimum (Proposition B-23). From now on, we suppose that activities are sorted in increasing order of processing times.

Proposition B-23.
The largest vector (according to the lexicographical order) satisfying all the constraints of *CP* realizes the optimum of *CP*.

Proof.

Let $Y = (Y_1, \ldots, Y_n)$ be the largest vector (according to the lexicographical order) satisfying all the constraints of *CP*, *i.e.*, Y_1 is maximal, Y_2 is maximal (given Y_1), Y_3 is maximal (given Y_1 and Y_2), …, Y_n is maximal (given $Y_1, \ldots, Y_{n-1}$). Moreover, let $X = (X_1, \ldots, X_n)$ be the largest (according to the lexicographical order) optimal solution of *CP*. Suppose that $X \neq Y$; let then u be the first index such that $X_u < Y_u$. Consider the set C of constraints that are saturated at X.

$$C = \{(r_j, d_k) \mid A_u \in S(r_j, d_k) \text{ and } \sum_{A_i \in S(r_j, d_k)} p_i X_i = d_k - r_j\}$$

If C is empty, then none of the constraints containing the variable x_u is saturated at the point X ($X_u < Y_u$ ensures that $X_u < 1$ and that x_u is not constrained to be equal to 0) and thus, X is not an optimum of *CP*. Hence C is not empty. Let then $(\rho_1, \delta_1) \in C$ be the pair such that ρ_1 is maximum and δ_1 is minimum (given ρ_1). Let $(\rho_2, \delta_2) \in C$ be the pair such that δ_2 is minimum and ρ_2 is maximum (given δ_2).

Suppose that $\rho_2 < \rho_1$. It is then obvious that $\rho_2 < \rho_1 \leq \rho_u < \delta_u \leq \delta_2 < \delta_1$. Let $A = S(\rho_2, \delta_2) - S(\rho_1, \delta_2)$ and let $B = S(\rho_1, \delta_1) - S(\rho_1, \delta_2)$. Because both (ρ_1, δ_1) and $(\rho_2, \delta_2) \in C$, we have:

$$\begin{cases} \sum_{A_i \in A} p_i X_i + \sum_{A_i \in S(\rho_1, \delta_2)} p_i X_i = \delta_2 - \rho_2 \\ \sum_{A_i \in S(\rho_1, \delta_2)} p_i X_i + \sum_{A_i \in B} p_i X_i = \delta_1 - \rho_1 \end{cases}$$

Since the sets A, B and $S(\rho_1, \delta_2)$ are disjoint and since $A \cup B \cup S(\rho_1, \delta_2) \subseteq S(\rho_2, \delta_1)$,

$$\sum_{A_i \in S(\rho_2, \delta_1)} p_i X_i \geq \delta_2 - \rho_2 + \delta_1 - \rho_1 - \sum_{A_i \in S(\rho_1, \delta_2)} p_i X_i \geq \delta_1 - \rho_2$$

The inequality above cannot be strict hence (ρ_1, δ_2) belongs to C. This, together with $\rho_2 < \rho_1$, contradicts our hypothesis on the choice of δ_1.

Now suppose that $\rho_1 = \rho_2 = \rho$ and $\delta_1 = \delta_2 = \delta$. The pair (ρ, δ) is the unique minimal saturated constraint containing the variable x_u. We claim that among activities in $S(\rho, \delta)$, there is one activity, say A_v, such that $v > u$ and $X_v > 0$ and $A_v \notin O$ (otherwise we could prove, because $X_u < Y_u$, that X_u can be increased; which contradicts the fact that X is optimal). Consider now X' the vector defined as follows. $\forall\ i \notin \{u, v\}$, $X'_i = X_i$ and $X'_u = X_u + \varepsilon / p_u$ and $X'_v = X_v - \varepsilon / p_v$. Where $\varepsilon > 0$ is a small value such that $\varepsilon \leq p_u (1 - X_u)$, $\varepsilon \leq p_v X_v$ and such that

$$\forall\ (r_j, d_k),\ \varepsilon \leq d_k - r_j - \sum_{A_i \in S(r_j, d_k)} p_i X_i$$

Since activities are sorted in increasing order of processing times, $\varepsilon / p_u - \varepsilon / p_v \geq 0$ and thus, $\Sigma (1 - X'_i) \leq \Sigma (1 - X_i)$. Moreover, *X'* is "better" for the lexicographical order than *X*. Second, because of the definition of ε, the constraints that were not saturated for *X* are not violated for *X'*. Third, the saturated constraints (for the vector *X*) that contain the variable x_u all contain the variables in (ρ, δ). In particular, they contain both x_u and x_v. As a consequence they are also saturated for the vector *X'*. We have proven that all constraints are satisfied. This contradicts our hypothesis on *X*. □

Proposition B-23 induces a simple algorithm (Algorithm B-6) to compute the optimum *X* of *CP*. Activities A_i that do not have to be late or on-time are considered one after another. Each time, we compute the maximum resource constraint violation if the activity is fully on-time (lines 4-11). Given this violation, the maximum value X_i that the variable x_i can take is computed (line 12). This algorithm runs in $O(n^4)$ since there are *n* activities A_i and since for each of them $O(n^2)$ violations are computed, each of them in linear time.

Algorithm B-6.

```
1  ∀ Ai, initialize Xi to 1.0 if Xi ∈ O, to 0.0 otherwise
2  for i = 1 to n
3    if Ai ∉ O and Ai ∉ L
4      Xi = 1.0, Violation = 0
5      for all constraint (rj, dk) such that Ai ∈ S(rj, dk)
6        sum = 0.0
7        for Al ∈ S(rj, dk)
8          sum = sum + pl * Xl
9        end for
10       Violation = max(Violation, sum – dk + rj)
11     end for
12     Xi = (pi – Violation) / pi
13   end if
14 end for
```

We improve this algorithm thanks to Jackson's Preemptive Schedule (*JPS*), the One-Machine preemptive schedule obtained by applying the Earliest Due Date priority dispatching rule [Carlier and Pinson, 1990]. A fundamental property of *JPS* is that it is feasible (*i.e.*, each activity ends before its due date) if and only if there exists a feasible preemptive schedule.

The procedure "`ComputeJPS`" of Algorithm B-7 is called for several values of *i*. It computes the *JPS* of the activities, assuming that the processing time of A_l ($l \neq i$) is $p_l X_l$

and that the processing time of A_i is p_i. "`EndTimeJPS[k]`" is the end time of A_k on *JPS*. *JPS* can be built in $O(n \log(n))$ [Carlier, 1982]. Algorithm B-7 then runs in $O(n^2 \log(n))$.

Algorithm B-7.

```
1  ∀ Ai, initialize Xi to 1.0 if Xi ∈ O, to 0.0 otherwise
2  for i = 1 to i = n
3    if Ai ∉ O and Ai ∉ L
4      ComputeJPS
5      ViolationJPS = 0
6      for all k such that Xk > 0
7        ViolationJPS=max(ViolationJPS,EndTimeJPS[k] – dk)
8      end for
9      Xi = (pi – ViolationJPS) / pi
10   end if
11 end for
```

Proof of the correctness of Algorithm B-7.

By induction. Suppose that at the beginning of iteration *i* (line 2), the first coordinates X_1, ..., X_{i-1} are exactly equal to those of *Y*, the maximal vector (according to the lexicographical order) satisfying the constraints of *CP*. Consider the case $Y_i = 1$ then, because of the structure of *CP*, there exists a feasible preemptive schedule of $A_1, \ldots, A_n$ (the processing time of activity A_u being $p_u Y_u$) and thus, the *JPS* computed line 4 is also feasible; which means that no violation occurs. Hence, $X_i = 1$ (line 9). Consider now the case $Y_i < 1$.

We first prove that $X_i \le Y_i$. Since $Y_i < 1$, the violation computed by Algorithm B-6 at step *i* is positive. Let then (r_j, d_k) be the constraint that realizes this violation. We then have

$$Y_i = 1 - \frac{1}{p_i}\left(p_i + \sum_{A_l \in S(r_j, d_k), l \neq i} p_l Y_l + r_j - d_k\right).$$

Moreover, at step *i* of Algorithm B-7,

$$\texttt{EndTimeJPS[k]} \ge p_i + \sum_{A_l \in S(r_j, d_k), l \neq i} p_l X_l + r_j .$$

Hence $X_i \le Y_i$ (line 9).

We now prove that $Y_i \le X_i$. Let *k* be the index of the activity such that the maximum violation on *JPS* is "`EndTimeJPS[k] – d`$_k$". Such an index exists because we have proven that $X_i \le Y_i < 1$ and thus, "`ViolationJPS`" is strictly positive. Let *t* be the largest time point lower than or equal to the end time of this activity such that immediately before *t*, *JPS* is either idle or executing an activity with a larger due date than d_k. According to the particular structure of *JPS*, *t* is a release date, say r_j. Notice that between

r_j and d_k, *JPS* is never idle and the activities that are processed are exactly those whose release date is greater than or equal to r_j and whose due date is lower than or equal to d_k. As a consequence, the end time of the k^{th} activity is

$$r_j + \sum_{\substack{A_l \in S(r_j, d_k) \\ l \neq i}} p_l X_l + p_i \, .$$

Hence, $X_i = 1 - \frac{1}{p_i}(r_j + \sum_{\substack{A_l \in S(r_j, d_k) \\ l \neq i}} p_l X_l + p_i - d_k) \geq Y_i$. ◻

The following table displays the characteristics of four activities A_1, A_2, A_3 and A_4. The last column X is the value of the activity variable at the end of Algorithm B-7. The Gantt charts display the *JPS* computed at each step of Algorithm B-7.

Activity	r	p	d	X
A_1	7	2	10	2/2
A_2	4	3	9	3/3
A_3	1	5	6	4/5
A_4	4	7	12	2/7

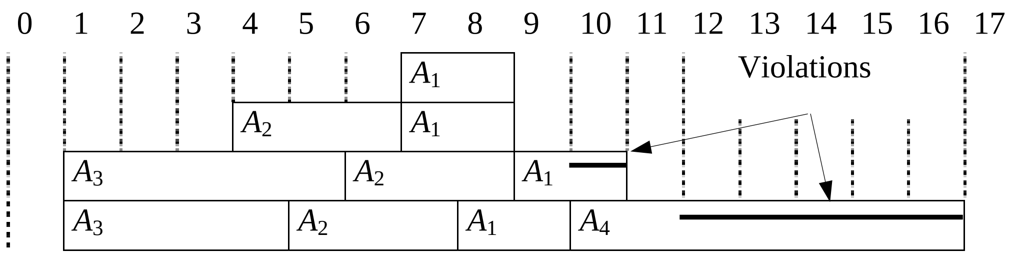


Figure B-14. The JPS computed at each step of the algorithm

B.4.2. *Resource Constraint Propagation*

In this section, we propose an algorithm which is able to detect that, given the domain of *reject*, some activities must be on time while some others must be late. A constraint propagation process is purely deductive, *i.e.*, it deduces some characteristics that any schedule satisfying all constraints must satisfy. For some problems, it happens that a particular dominance property holds. Such a dominance property can be "integrated" into the resource constraint propagation. However, it makes the propagation less generic, *i.e.*, if new constraints are added, the dominance property may not hold and thus the propagation process must be changed.

Concerning the $(1|r_j|\Sigma U_j)$ problem, a strong dominance property holds. It states that for any activity A_i,

- a set $O(A_i)$ of activities that have to be on-time if A_i is on-time
- and a set $L(A_i)$ of activities that have to be late if A_i is late

can be computed. This dominance property is detailed in Section C.4.1.3. In the following we present a constraint propagation algorithm that exploits this dominance. It can be ignored to make the constraint propagation process more generic.

B.4.2.1. Late Activity Detection

From now on, we suppose that the optimum X of CP has been computed as described in the previous section. Consider an activity A_j such that $A_j \notin O$ and $A_j \notin L$. Our objective is to compute efficiently a lower bound of the number of late activities if A_j and $O(A_j)$ are on-time. If this lower bound is greater than the maximal value in the domain of *reject*, then, A_j must be late. Algorithm B-7 could be used to compute such a lower bound. However, this would lead to a high overall complexity of $O(n^3 \log(n))$. We propose to use a slightly weaker lower bound that can be computed in linear time, for a given activity A_j. The overall filtering scheme then runs in $O(n^2)$.

Let CPo be the linear program CP to which the constraints $\forall A_i \in O(A_j)$, $x_i = 1$ have been added. Moreover, let Xo be the optimal vector of CPo obtained by Algorithm B-7 (CPo has a solution, otherwise part 2 of the propagation described at the beginning of Section B-4 would have detected that $A_i \in L$). Propositions B-24 and B-25 exhibit two relations that X and Xo satisfy. These relations are used to compute a lower bound of $\sum Xo_i$.

Proposition B-24.
$\sum p_i Xo_i \leq \sum p_i X_i$

Proof.
Let $G(Activities, Time, E)$ be a bipartite graph, where $Activities = \{Job_1, \ldots, Job_n\}$ is a set of vertices corresponding to the activities, where $Time = \{T_t, \min_i r_i \leq t < \max_i d_i\}$ is a set of vertices corresponding to all the "relevant" time-intervals $[t, t + 1]$ and where an edge (Job_i, T_t) belongs to E if and only if A_i can execute in $[t, t + 1]$ (*i.e.*, $r_i \leq t < d_i$). Consider the network flow (*cf*. Figure B-15) built from G by adding:

- two vertices S, P and an edge (P, S),
- for each node Job_i an edge (S, Job_i) whose capacity is (1) upper bounded by either 0 if $A_i \in L$ or by p_i otherwise and (2) lower bounded by either p_i if $A_i \in O$ or by 0 otherwise,

- for each node T_t an edge (T_t, P) whose capacity is upper bounded by 1.

For any feasible flow, a vector satisfying all constraints of *CP* can be built (the i^{th} coordinate of the vector is the value of the flow on (S, Job_i) divided by p_i). Since $\forall i, X_i * p_i$ is integer, a feasible flow can be derived from the *JPS* associated to the vector X (when A_i executes in $[t, t+1]$ on *JPS*, set the value of the flow to 1 on the edge (Job_i, T_t)). Suppose that $\sum p_i Xo_i > \sum p_i X_i$, then the flow corresponding to X is not maximal in G, and thus there is an augmenting path from S to P. Let then X^+ be the vector corresponding to the augmented flow. Because of the structure of G, $\forall i$, $X^+_i \geq X_i$. On top of that there exists l such that $X^+_l > X_l$. This contradicts Proposition B-23. □

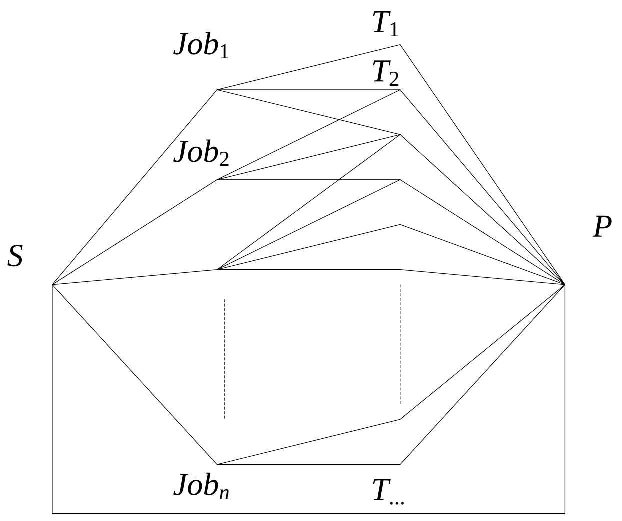


Figure B-15. The network flow built from G

Proposition B-25.

$\forall A_i \notin O(A_j)$, $Xo_i \leq X_i$

Proof (sketch).

Suppose the proposition does not hold. Let i be the first index such that $A_i \notin O(A_j)$ and $Xo_i > X_i$. We modify the instance of the problem by removing the activities A_u with $u > i$ that do not belong to O nor to $O(A_j)$. The activities that have been removed do not influence Algorithm B-7 when computing the i first coordinates of X and of Xo (*i.e.*, for the modified instance, the i first coordinates of the optimum vector are exactly those of X). Now, consider the modified instance. We still have $A_i \notin O(A_j)$ and $Xo_i > X_i$. Moreover, the activities that have a greater index than i belong to $O \cup O(A_j)$. Consider the modified network (Figure B-16) built from the bipartite graph G by adding:

- three vertices S, S', P, and two edges, (S, S') and (P, S),

- for each node Job_u ($u \neq i$) an edge (S', Job_u) whose capacity is (1) upper bounded by either 0 if $A_u \in L$ or by p_u otherwise and (2) lower bounded by either p_u if $A_u \in O$ or by 0 otherwise,
- an edge (S, Job_i) whose capacity is upper bounded by p_i and lower bounded by 0,
- for each node T_t an edge (T_t, P) whose capacity is upper bounded by 1.

For any feasible flow, a vector satisfying all constraints of *CP* can be built. Conversely, for any vector satisfying all constraints of *CP*, a feasible flow can be built. The flow corresponding to *Xo* is obviously feasible. Moreover the flow on (P, S) for vector X is greater than or equal to the one for *Xo* (see Proposition B-24). Moreover, the flow on (S, Job_i) for *Xo* is greater than the one for X. Hence, because of the conservation law at S, the flow that goes over (S, S') is not maximal for *Xo*. As a consequence, there is an augmenting path from S' to S for the flow corresponding to *Xo*. Let then Xo^+ be the vector corresponding to the augmented flow. Because of the structure of G, $\forall\ u \neq i$, $Xo^+_u \geq Xo_u$. Hence, $\forall A_u \in O \cup O(A_j)$, $Xo^+_u = 1$ and then, Xo^+ satisfies all the constraints of *CPo*. Moreover it is better than *Xo* because:

- If the edge (P, S) is in the augmenting path then $\sum p_i Xo^+_i > \sum p_i Xo_i$.
- If the edge (Job_i, S) is in the augmenting path then we claim that Xo^+ is greater, for the lexicographical order, than *Xo*. Indeed, there is an edge from S' to an activity, say Job_u, in the augmenting path. Hence, A_u neither belongs to O nor to $O(A_j)$ (otherwise, the edge would be saturated for *Xo* and it could not belong to the augmenting path). Consequently, $u < i$ and then $Xo^+_u > Xo_u$.

This contradicts the fact that *Xo* is optimal. □

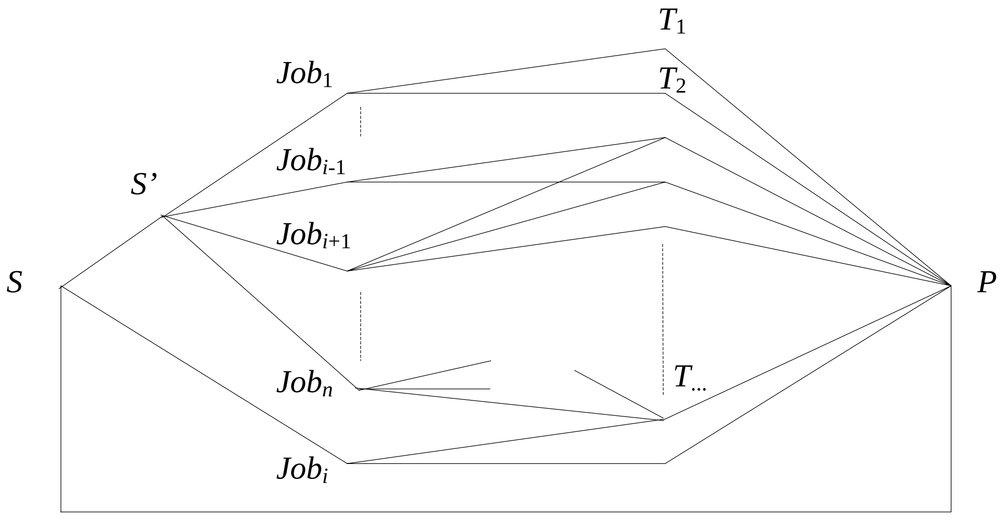


Figure B-16. The modified network flow

Thanks to Propositions B-24 and B-25, we can add the constraints $\sum p_i Xo_i \leq \sum p_i X_i$ and $\forall A_i \in O(A_j)$, $x_i \leq X_i$ to the linear program *CPo*. Since we are interested in a lower bound of *CPo*, we can also relax the resource constraints. As a consequence, we seek to solve the following program. It is solved in linear time by Algorithm B-8.

$$\begin{array}{l} \min \sum (1 - x_i) \\ \sum p_i x_i \leq \sum p_i X_i \\ \forall J_i \notin o, x_i \leq X_i \\ \forall J_i \in O \cup O(J_j), x_i = 1 \text{ and } \forall J_i \in L, x_i = 0 \\ \forall i \in \{1, \Lambda, n\},\ x_i \in [0,1] \end{array}$$

Algorithm B-8.

```
∀ Ai set Xoi to 1.0 if Ai ∈ O ∪ O(Aj), to 0.0 otherwise
MaxVal = Σ pi Xi - Σ pi Xoi
for all activity Ai ∉ O ∪ O(Aj)
  Xoi = min(Xi, MaxVal / pi)
  MaxVal = MaxVal - pi * Xoi
End for
```

B.4.2.2. On-Time Activity Detection

Let A_j be an activity that is neither late nor on-time. We want to compute a lower bound of the number of late activities if all activities in $L(A_j)$ are late. Let X_l be the optimal vector of *CPl*, the linear program *CP* to which the constraints $\forall A_i \in L(A_j)$, $x_i = 0$ have been added. We claim that $\Sigma\ p_i\ Xl_i \leq \Sigma\ p_i\ X_i$ and that $\forall A_i \notin L(A_j)$, $Xl_i \geq X_i$ (proofs are similar to the proofs of propositions B-24 and B-25). The same mechanism as for the late activity detection then applies: The new constraints are entered in *CPl* while the resource constraints are removed. The resulting linear program can be also solved in linear time.

Chapter C. Problem Solving and Experimental Results

In this chapter, we perform an experimental evaluation of the efficiency of the resource constraint propagation algorithms proposed in Chapter B. Exact problem solving procedures have been built to tackle the four problems outlined in the introductory chapter, namely

- the Job-Shop Scheduling Problem (Section C.1),
- the Preemptive Job-Shop Scheduling Problem (Section C.2),
- the Resource-Constrained Project Scheduling Problem (Section C.3),
- the minimization of the number of late jobs on a single machine (Section C.4).

The branch and bound procedures that have been designed rely not only on the resource constraints of Chapter B but also on dominance properties that allow to restrict the search space, and on more or less complex branching schemes.

Experimental results are provided for each problem. They allow us to compare the efficiency of different resource constraint propagation algorithms on the same problem. In particular, we will see that for some problems, the efficiency varies a lot from a “type” of instance to another. We also compare our approaches to the most efficient procedures of the literature.

C.1. The Job-Shop Scheduling Problem

The aim of this section is to very briefly recall a widely used branching scheme for the non-preemptive Job-Shop Scheduling Problem. We will rely on this mechanism as part of the resolution of several other combinatorial problems such as the Resource-Constrained Project Scheduling Problem (Section C.3) and the problem of minimizing the number of late activities on a single machine (Section C.4). A rather comprehensive review of the most common techniques used for solving the JSSP can be found in [Blazewicz *et al.*, 1996].

Several successful exact (branch and bound) approaches for the non-preemptive JSSP rely on successive resolutions of the decision variant of this problem:

1. Compute an obvious upper bound of the *makespan* variable and an initial lower bound.
2. Select a value V in the domain of *makespan*.
3. Constrain the *makespan* to be lower than or equal to V and run the branching procedure. If a solution is found, set *ub*(*makespan*) to the makespan of the solution; otherwise, *i.e.*, if the search procedure fails, set *lb*(*makespan*) to $V + 1$.
4. Iterate steps 2 and 3 until *makespan* is bound.

The branching procedure (step 3) often consists of ordering successively the set of activities $ACTS(M)$ which require the same machine M [Carlier and Pinson, 1990], [Carlier and Pinson, 1994], [Brucker *at al.*, 1994], [Baptiste and Le Pape, 1995b]. At each node, a machine M and a set $\Omega \subseteq ACTS(M)$ are selected. For each activity A in Ω, a new branch is created where A is constrained to execute first (or last) among the activities in Ω. The set of candidates to be first (or last) is drastically reduced by the edge finding rules applied to the set of unordered activities. The decision is then propagated, through one of the variant of the edge-finding bounding technique described in Section B.1. Such a branching scheme is known as the edge-finding branching technique [Applegate and Cook, 1991].

Its efficiency depends on the heuristic used to select the machine to schedule first, and on the heuristic used to select the unordered activity that is to be first (or last). In our experiments, we used the following heuristics:

- The machine to schedule first is the one whose slack is minimal. The slack is defined as the minimal difference between *supply* and *demand* over each time interval $[r_i, d_k]$ (release date / deadline). The resource supply over $[r_i, d_k]$ is $d_k - r_i$, which reflects the fact that the resource can perform only one activity at a time. The demand over interval $[r_i, d_k]$ is the sum of the processing times of the activities that must execute between r_i and d_k.

- The activity to schedule first is selected according to the following rule: The activity with the smallest release date is chosen; in case of ties, the activity with the smallest latest start time ($lst_i = d_i - p_i$) is chosen.

The following table C-1 provides the results reported in [Baptiste, 1995] that have been obtained with the ILOG SCHEDULE scheduling tool (version 2.0) [Le Pape, 1995] on the ten 10x10 (10 machines, 10 jobs, 100 activities) instances of the JSSP used by Applegate and Cook in their computational study of the JSSP [Applegate and Cook, 1991]. In this table, column "MAK" provides for each instance the optimal makespan, *i.e.*, the minimal total duration of the schedule. Columns "BT" and "CPU" provide the total number of backtracks and CPU time needed to find an optimal solution and prove its optimality. Columns "BT(pr)" and "CPU(pr)" provide the number of backtracks and CPU time needed for the proof of optimality. CPU times are expressed in seconds on an RS6000 workstation, rounded to the closest tenth of a second.

	MAK	BT	CPU	BT(pr)	CPU(pr)
MT10	930	69758	1076.4	7792	126.8
ABZ5	1234	17636	218.1	5145	62.6
ABZ6	943	898	15.2	291	4.7
LA19	842	21910	293.3	5618	75.8
LA20	902	74452	845.9	22567	249.2
ORB1	1059	13944	222	5382	84.7
ORB2	888	114715	1917.2	30519	500.9
ORB3	1005	190117	3193.8	25809	449
ORB4	1005	64652	1131.2	22443	395.2
ORB5	887	11629	172.8	3755	55

Table C-1. Experimental results obtained on 10 instances of the JSSP used by Applegate and Cook in their computational study.

A large amount of research has been carried on extensions or on variants of such a branching scheme. In particular, global operations also called "shaving" have been used by [Carlier and Pinson, 1994] and by [Martin and Shmoys, 1996]) to reduce the search space. The basic idea is very simple. At each node of the search tree and for each activity A_i, the earliest date x_i at which the activity can be scheduled without triggering a contradiction is computed. This basically consists of (1) iteratively trying a start time for the activity A_i, (2) propagating the consequence of this decision thanks to the edge-finding bounding technique and (3) verifying that no contradiction has been detected. The earliest date x_i is of great interest since it can serve to adjust the release date of the activity. A dichotomizing procedure can be used to determine the date x_i. It decreases both the theoretical and the practical complexities of the algorithm. Several extensions of this

mechanism are proposed in [Péridy, 1996]. The underlying idea is to impose a decision (*e.g.*, a starting time for a given activity) and to exploit the consequences of this decision in more or less complex algorithms to obtain a global information on the instance. We think that using such mechanisms is a very promising research direction.
Another promising research direction is to build approximation algorithms that exploit the extensive propagation of resource constraints. [Applegate and Cook, 1991], [Nuijten, 1994], [Baptiste *et al.*, 1995b] and [Nuijten and Le Pape, 1998] report experimental results on the job-shop scheduling problem. It is shown that very good solutions can be reached in a short amount of time.
Finally, one can also use an approximation algorithm for a given number of iterations and then proceed with an exact algorithm. The following table displays the results that have been obtained with such an approach [Baptiste *et al.*, 1995b].

	BT	CPU	BT(pr)	CPU(pr)
MT10	13684	235.8	4735	67.3
ABZ5	19303	282.1	4519	61.3
ABZ6	6227	100.6	312	4.7
LA19	18102	269.5	6561	91
LA20	40597	496.7	20626	227.2
ORB1	22725	407.3	6261	108
ORB2	31490	507.1	14123	228.7
ORB3	36729	606.1	22138	342.6
ORB4	13751	213.7	1916	23.7
ORB5	12648	210.9	2658	36.5

C.2. The Preemptive Job-Shop Scheduling Problem[11]

To evaluate the constraint propagation algorithms presented in Section B.2, we developed a branch and bound procedure for the preemptive Job-Shop Scheduling Problem (PJSSP), the variant of the Job-Shop Scheduling Problem (JSSP) in which all activities are interruptible.
For the PJSSP, the classical edge-finding branching scheme (Section C.1) is not valid since activities are interruptible, and thus cannot just be ordered. However, the dominance criterion introduced below allows the design of branching schemes which in a sense "order" the activities that require the same machine.

C.2.1. A dominance property

Definition.
For any schedule S and any activity A_i, we define the "due date of A_i in S" $d_S(A_i)$ as:

- the makespan of S if A_i is the last activity of its job;
- the start time of the successor of A_i otherwise.

Definition.
For any schedule S, an activity A_k has priority over an activity A_l in S ($A_k <_S A_l$) if and only if either $d_S(A_k) < d_S(A_l)$ or $d_S(A_k) = d_S(A_l)$ and $k \leq l$. Note that $<_S$ is a total order.

Proposition C-1.
For any schedule S, there exists a schedule $J(S)$ such that:

1. $J(S)$ meets the due dates: $\forall A$, the end time of A in $J(S)$ is at most $d_S(A)$.
2. $J(S)$ is "active": For any machine M at any time point t, if some activity A that belongs to $\textit{ACTS}(M)$ (the set of activities that execute on M), is available at time t, then M is not idle at time t (where "available" means that the predecessor of A is finished and A is not finished).
3. $J(S)$ follows the $<_S$ priority order: $\forall M, \forall t, \forall A_k \in \textit{ACTS}(M), \forall A_l \in \textit{ACTS}(M), A_l \neq A_k$, if A_k executes at time t, either A_l is not available at time t or $A_k <_S A_l$.

[11] Most of the results presented in this section come from [Baptiste, 1995], [Baptiste and Le Pape, 1996a] and [Le Pape and Baptiste, 1998a]

Proof.

We construct $J(S)$ chronologically. At any time t and on any machine M, the available activity that is the smallest (according to the $<_S$ order) is scheduled. $J(S)$ satisfies properties 2 and 3 by construction. Let us suppose $J(S)$ does not satisfy property 1. Let A denote the smallest activity (according to $<_S$) such that the end time of A in $J(S)$ exceeds $d_S(A)$. We claim that:

- the schedule of A is not influenced by the activities A_k with $A <_S A_k$ (by construction);
- for every activity $A_k <_S A$, the time at which A_k becomes available in $J(S)$ does not exceed the time at which A_k starts in S (because the predecessor of A_k is smaller than A).

Let M be the machine on which A executes. In $J(S)$, the activities $A_k \in \mathit{ACTS}(M)$ such that $A_k <_S A$ are scheduled in accordance with Jackson's rule, applied to the due dates $d_S(A_k)$. Since $d_S(A)$ is not met, and since Jackson's rule is guaranteed to meet due dates whenever it is possible to do so (*cf.* [Carlier and Pinson, 1990]), we deduce that it is impossible to schedule the activities $A_k \in \mathit{ACTS}(M)$ such that $A_k <_S A$ between their start times in S and their due dates in S. This contradicts the fact that in S these activities are scheduled between their start times and their due dates. So, the hypothesis that $J(S)$ violates property 1 is contradicted. □

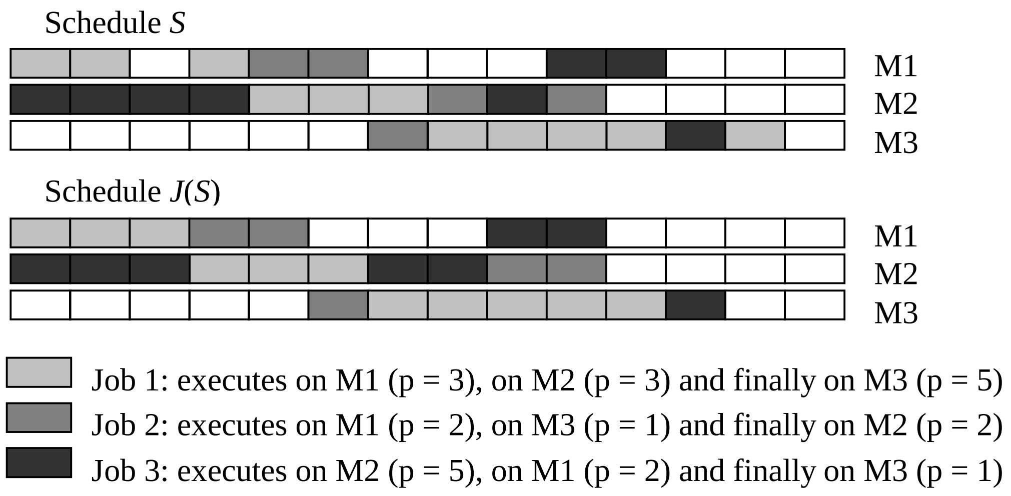


Figure C-1. A preemptive schedule and its Jackson derivation.

C.2.2. Branching scheme

We call $J(S)$ the "Jackson derivation" of S. Since the makespan of $J(S)$ does not exceed the makespan of S, at least one optimal schedule is the Jackson derivation of another schedule. Thus, in the search for an optimal schedule, we can impose the characteristics of a Jackson derivation to the schedule under construction. In this section, we present two branching procedures in which this result is used to solve the PJSSP.

Each of them is integrated in the following makespan minimization algorithm:

1. Compute an obvious upper bound of the *makespan* variable and an initial lower bound.
2. Select a value V in the domain of *makespan*.
3. Constrain the *makespan* to be lower than or equal to V and run the branching procedure. If a solution is found, set *ub*(*makespan*) to the makespan of the solution; otherwise, *i.e.*, if the search procedure fails, set *lb*(*makespan*) to $V + 1$.
4. Iterate steps 2 and 3 until *makespan* is bound.

The first branching scheme consists of ordering the activities according to an hypothetical $<_S$ order. For each machine M, an ordered list L_M of activities, initially empty, is developed as follows:

1. Select a machine M such that the set $K_M = \text{ACTS}(M) - L_M$ is not empty.
2. Select an activity A_k in K_M (*e.g.*, the one with the smallest latest end time). Add A_k to the end of the list L_M. Use Jackson's rule to schedule the activities of L_M according to the L_M priority order and impose the resulting earliest end times. Keep the other activities of K_M as alternatives to be tried upon backtracking.
3. Iterate until all the activities are ordered or until all alternatives have been tried.

This branching scheme is attractive since it mimics the edge-finding branching technique that is often used in non-preemptive disjunctive scheduling. Yet, our first experiments have been disappointing. This led us to develop another branching scheme which more heavily exploits the dominance criterion.

1. Let t be the earliest date such that there is an activity A available (and not scheduled yet!) at t.
2. Compute K, the set of activities available at t on the same machine as A.
3. Compute NDK, the set of activities which are not "dominated" in K (as explained below).
4. Select an activity A_k in NDK (*e.g.*, the one with the smallest latest end time). Schedule A_k to execute at t. Propagate the decision and its consequences according to the dominance criterion. Keep the other activities of NDK as alternatives to be tried upon backtracking.
5. Iterate until all the activities are scheduled or until all alternatives have been tried.

Needless to say, the power of this branching scheme highly depends on the rules that are used to (a) eliminate "dominated" activities in step 3 and (b) propagate "consequences" of the choice of A_k in step 4. The dominance criterion is exploited as follows:

- Whenever $A_k \in \text{ACTS}(M)$ is chosen to execute at time t, it is set to execute either up to its earliest end time or up to the earliest start time of another activity $A_l \in \text{ACTS}(M)$ which is not available at time t.

- Whenever $A_k \in K$ is chosen to execute at time t, any other activity $A_l \in K$ can be constrained not to execute between t and the end of A_k. At times $t' > t$, this reduces the set of candidates for execution: A_l is dominated by A_k, hence not included in NDK. In step 4, redundant constraints can also be added:

$$end(A_k) + rp_t(A_l) \leq end(A_l),$$

where $rp_t(A_l)$ is the remaining processing time of A_l at time t; $end(A_k) \leq start(A_l)$ if A_l is not started at time t.
- If $A_k \in \text{ACTS}(M)$ is the last activity of its job, A_k is not candidate for execution at time t if another activity $A_l \in \text{ACTS}(M)$, which is not the last activity of its job, or such that $l < k$, is available at time t (A_k is dominated by A_l).

The proof that these reductions of the search space do not eliminate all optimal schedules follows from the fact that $J(S)$ schedules are dominant. Indeed, in a $J(S)$ schedule, (1) an activity cannot be interrupted unless a new activity becomes available on the same resource, (2) an activity A_k cannot execute when another activity A_l is available, unless $A_k <_S A_l$, and (3) we cannot have $A_k <_S A_l$ if A_k is the last activity of its job and either A_l is not the last activity of its job or $l < k$.

An open question at this point is whether there exists an optimal solution S such that $J(S) = S$. This would allow us to constrain the search even more. For example, as soon as an activity A_k would be given priority over an activity A_l, we could constrain the successor of A_l not to start before the successor of A_k. This could have a dramatic impact on the search space.

C.2.3. Experimental Results

The second branching scheme was used to evaluate the various constraint propagation techniques developed in Section B.2. The disjunctive constraint $set(A) \cap set(B) = \varnothing$ and the flow-based algorithms, SCF, AEC, and GUTB, were implemented in ILOG SOLVER [Puget, 1994] on a RS6000 workstation. The mixed edge-finder was implemented in CLAIRE [Caseau and Laburthe, 1996b] on a PC Dell 200MHz running Windows NT.

Table C-2 summarizes the results on 20 well-known instances of the Job-Shop Scheduling Problem. The first two columns indicate the version of the resource constraint that was used and the problem instance(s) under consideration. This can be a unique instance like "FT06" or, for "easy" instances, a series of instances similar in size and toughness, like "LA01 to LA10." In the latter case, the table provides average results over the whole series. All the instances we use are available from the job-shop directory in the OR benchmark library (http://www.ms.ic.ac.uk/info.html), except the CAR instances which can be found in the flow-shop directory.

Column "BT" provides the total number of backtracks needed to solve the problem. Column "CPU" provides the total CPU time in seconds, on a PC for the mixed edge-finder, and on an RS6000 for the other algorithms. Columns "BT(pr)" and "CPU(pr)" provide the number of backtracks and CPU time spent in proving that the optimal solution is, indeed, optimal. Results appear only when the considered version of the resource constraint enabled the branch and bound algorithm to solve the considered instance(s) in a reasonable amount of time. (For the smallest problems (FT06 to CAR4), at most 5000 backtracks were allowed for each iteration of the makespan minimization procedure.)

Table C-2 shows that both the mixed edge-finder and the GUTB algorithm allow the resolution of "tough" problems like CAR5 (with optimal makespan 7667) and FT10 (900). Part of the differences between the edge-finder and the GUTB algorithm are due to differences in implementation, *e.g.*, different computers and different sorting functions, so further comparison is not possible. Interestingly enough, the instances that appear the most difficult in the non-preemptive case, CAR5 and FT10 [Baptiste, 1994], are also the most difficult in the preemptive case.

Table C-3 shows the results obtained by GUTB on the ten 10∗10 (*i.e.*, 10 machines ∗ 10 jobs = 100 activities) instances used by [Applegate and Cook, 1991] in their computational study of the (non-preemptive) Job-Shop Scheduling Problem. Five of these instances (ABZ6, LA19, LA20, ORB2, and ORB5) were solved to optimality in a few hours of CPU time, one (FT10) was allowed more time to terminate, and four (ABZ5, ORB1, ORB3, and ORB4) remained open. For these instances, column "OPT" provides the best lower and upper bound that have been achieved. Otherwise, column "OPT" provides the value of the optimal makespan.

Constraint	Instances	BT	CPU	BT(pr)	CPU(pr)
Disjunctive	FT06	6353	3.5	4775	2.6
Edge-finder	FT06	3	0.1	2	0.0
	LA01-10	1	0.2	1	0.0
	CAR1-4	9	0.2	1	0.0
	CAR5	97927	582.6	26034	155.3
	CAR6-8	2870	23.4	937	7.5
	FT10	140903	2105.6	41255	624.0
SCF	FT06	24	0.3	21	0.1
	LA01-10	1196	9.2	1	0.0
AEC	FT06	5	0.5	2	0.1
	LA01-10	112	25.9	1	0.1
	CAR1-4	461	61.5	11	2.0
	CAR6-8	6947	1644.9	1403	351.3
GUTB	FT06	6	0.4	2	0.0
	LA01-10	9	11.0	1	0.0
	CAR1-4	27	13.0	1	0.1
	CAR5	73135	10295.8	19265	2673.7
	CAR6-8	3593	663.6	819	146.5
	FT10	254801	97585.7	49817	19626.6

Table C-2. Results obtained on 20 instances of the preemptive Job-Shop Scheduling Problem.

	OPT	BT	CPU	BT(pr)	CPU(pr)
FT10	900	254801	97585.7	49817	19626.6
ABZ5	1159 / 1219				
ABZ6	924	17578	3955.5	10879	2268.3
LA19	812	39286	7150.1	14184	2482.4
LA20	871	5494	1483.6	1627	463.8
ORB1	991 / 1054				
ORB2	864	56863	11199.2	20203	3835.3
ORB3	951 / 1254				
ORB4	977 / 980				
ORB5	849	16457	4721.3	4496	1296.6

*Table C-3. GUTB results on ten 10*10 instances of the preemptive Job-Shop Scheduling Problem.*

	OPT	BT	CPU	BT(pr)	CPU(pr)
FT10	900	140903	2105.6	41255	624.0
ABZ5	1203	1192553	15628.0	338597	4430.9
ABZ6	924	17699	307.8	8157	134.3
LA19	812	34637	564.3	10928	176.4
LA20	871	2779	59.4	998	22.7
ORB1	1035	347647	5182.4	85085	1278.3
ORB2	864	53127	709.4	16189	220.9
ORB3	973	6804127	96917.7	1947325	27884.2
ORB4	980	97654	1201.8	37122	461.3
ORB5	849	10380	158.6	4151	61.6

*Table C-4. Edge-finding results on ten 10*10 instances of the preemptive Job-Shop Scheduling Problem.*

Table C-4 provides the results obtained with the edge-finding algorithm on the same ten instances. All of these instances have been solved to optimality. Other instances we have solved include FT20 (in 0.4 second), LA11 to LA15 (0.4 second), LA16 (145 minutes), LA17 (1 second), LA18 (4 minutes), LA21 (65 hours), LA22 (4 seconds), LA23 (1 second), LA24 (44 hours), LA26 (1 second), LA28 (1 second), LA30 (1 second), LA31 to LA35 (4 seconds), LA37 (110 minutes), ORB6 (39 minutes), ORB7 (10 minutes), ORB8 (1 second), ORB9 (3 minutes), and ORB10 (1 minute). Let us note that, in the non-preemptive case, ORB3 also appears to be one of the most difficult 10∗10 instances [Applegate and Cook, 1991], [Baptiste and Le Pape, 1995b], [Caseau and Laburthe, 1995], [Colombani, 1996], [Colombani, 1997]. Such is not the case for LA16 (also a 10∗10 instance) which is considered “easy” in the non-preemptive case.

Experimental results have shown that two of these techniques, (1) edge-finding and (2) global update of time bounds (GUTB), allow the resolution of hard instances such as the preemptive variant of the famous FT10. Let us remark that a combination of the two techniques is not likely to be useful when all the activities are interruptible and only time-bound constraints are imposed. Indeed, the characterization of the preemptive edge-finding algorithm proves that the best possible bounds are obtained. A combination might however be useful in more complex situations: on the one hand, the mixed edge-finding algorithm explicitly deals with non-interruptible activities, and thus can be more efficiently applied to the mixed case; on the other hand, if an interruptible activity cannot execute during some time intervals, the GUTB algorithm can take these intervals into account.

These results encouraged us to pursue work in the application of constraint programming to preemptive and mixed scheduling problems. [Le Pape and Baptiste, 1997b] and [Le Pape and Baptiste, 1998b] evaluate the interest of different heuristics and branching strategies to reach very good solutions of the PJSSP in a few amount of time. We think that several other research directions are of great interest for mixed scheduling problems.

- Based on our results, the PJSSP currently appears to be much harder than the non-preemptive JSSP. An important reason for this is that we have not been able to reuse the concept of “bottleneck resource” in an efficient way. An open question is how the “bottleneck” concept can be used, without throwing away the dominance criterion which appears crucial in reducing the size of the search tree.
- Most of the results presented in the preceding sections concern resources of capacity 1. More work is needed to generalize these techniques to resources of arbitrary capacity.
- Other constraint propagation techniques, such as shaving [Carlier and Pinson, 1994], [Martin and Shmoys, 1996], [Péridy, 1996] can be worth investigating.

C.3. The Resource-Constrained Project Scheduling Problem. [12]

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 processing times, (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 whether there exists a schedule of makespan smaller than a given deadline, is NP-hard in the strong sense [Garey and Johnson, 1979].

The aim of this experimental study is to test the efficiency of the constraint propagation schemes proposed in Section B.3 and also 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 $\{A_i, A_j\}$ such that either there is a chain of precedence constraints between A_i and A_j, or there is a resource constraint which is violated if A_i and A_j 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.

---

[12] Most of the results presented in this section come from [Baptiste and Le Pape, 1997a] and [Baptiste *et al.*, 1998b].

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*, 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. With this distinction in mind, we introduce several new techniques to solve the RCPSP.

Section C.3.1 presents our general approach to the resolution of the RCPSP; Section C.3.2 presents the constraint propagation techniques we use (including a redundant constraint generation scheme); Section C.3.3 presents dominance rules, which are used to dynamically decompose an instance of the RCPSP; Section C.3.4 presents experimental results, which confirm that the techniques we use exhibit different behaviors on problems with different disjunction ratios.

C.3.1. 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. As usual, 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 obvious upper bound of the *makespan* variable and an initial lower bound.
2. Select a value v in the domain of *makespan*. (*e.g.*, the middle of the domain)
3. Constrain the *makespan* to be lower than or equal to v and run the branching procedure. If a solution is found, set *ub*(*makespan*) to the makespan of the solution; otherwise, *i.e.*, if the search procedure fails, set *lb*(*makespan*) to $v+1$.
4. Iterate steps 2 and 3 until *makespan* is bound.

A branching procedure with constraint propagation at each node of the search tree is used to determine whether the problem with makespan at most *v* 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 described in Section B.3. 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 lower than or equal to *v*; 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. The deductive techniques for the CuSP have been tested on the RCPSP. Experimental results show that it is worth using such techniques 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”. As detailed in Section B.3.1.4, 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. We propose to generate 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.

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 *v*. 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 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 variable (*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]). These rules enable the reduction of the search to a limited number of nodes, which satisfy the dominance properties. Section C.3.3 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 redoing 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 solution search procedure slightly differs from the one proposed by [Caseau and Laburthe, 1996a]:

1. Select an unscheduled activity A_i of minimal release date. When several activities have the same release date, select one of the most urgent, *i.e.*, one with minimal latest start time (lst_i). Create a choice point.
2. Left branch: Schedule A_i from its release date r_i to its earliest end time eet_i (in other terms, set $start(A_i)$ to the smallest value in its domain). Propagate this decision. Apply the dominance rules. Go to step 1.
3. Right branch: If step 2 causes a backtrack, compute the set S of activities that could overlap the interval $[r_i \; eet_i]$ (according to current variable domains). Post a delaying constraint: "A_i executes after at least one activity in S". Propagate this constraint. Apply the dominance rules. Go to 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 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 Section B.3, several constraint propagation algorithms can be associated with each resource. Among these algorithms, the 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_i executes after at least one activity in S" upon backtracking is correct, because if A_i could start before the end of all activities in S, then A_i could start at the release date r_i 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 C.3.3).

C.3.2. Constraint Propagation

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 our framework are of the following types:

1. $0 \leq start(A_i)$, for every activity A_i;
2. $start(A_i) + processingTime(A_i) = end(A_i)$, for every activity A_i;
3. $end(A_i) \leq makespan$, for every activity A_i;
4. $end(A_i) \leq start(A_j)$, for every precedence constraint ($A_i \rightarrow A_j$);
5. $\Sigma_{start(Ai) \leq t < end(Ai)}(\text{capacity}(A_i, R)) \leq capacity(R)$, for every resource R and time t (cumulative constraint);
6. $makespan \leq v$;
7. "A_i executes after at least one activity in S" *i.e.*, $\min_S(end(S)) \leq start(A_i)$, where A_i 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, *UB*] where *UB* is an obvious upper bound of the optimum. As often in constraint programming, unary constraints (1, 6) are propagated by reducing the domains of the corresponding variables.
Processing time constraints (2) and precedence constraints (3, 4) are propagated using a standard arc-B-consistency algorithm [Lhomme, 1993].
The constraint "A_i executes after at least one activity in S" (7) is propagated as follows: compute $\min_{Aj\in S}eet_j$, the minimal earliest end time of all activities in S, and update r_i to $\max(r_i, \min_{Aj\in S}eet_j)$. Moreover, when there is only one activity A_j in S that can end before lst_i, then the deadline of A_j can be set to $\min(d_j, lst_i)$.
The resource constraints (5) are propagated with a timetable mechanism. Time bound adjustments based on the CuSP (*cf.*, Section B.3) are eventually applied (depending on the variant of the algorithm).

Since some project scheduling problems are highly disjunctive, we considered the generation of redundant disjunctive resource constraints as a mean to strengthen constraint propagation (see also [Brucker *et al.*, 1997]). The basic idea is simple: if a set *S* of activities is such that any two activities in *S* cannot 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 (*cf.*, Section B.1.3) can then be applied to the new resource, in order to guarantee a better update of the release dates and deadlines of these activities.

To detect the relevant sets S, we use a compatibility 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_i, A_j), such that $(A_i, A_j) \in E$ if and only if A_i and A_j 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.

- $(A_i, A_j) \in E_{cap}$ if and only if there is a resource R such that the sum of the capacities required by A_i and A_j on R is greater than the overall capacity of R.
- $(A_i, A_j) \in E_{prec}$ if and only if there is a precedence constraint between A_i and A_j (the transitive closure of the precedence graph is computed for this purpose).
- $(A_i, A_j) \in E_{time}$ if and only if either $d_i \leq r_j$ or $d_j \leq r_i$.

Any clique of the compatibility graph is a candidate disjunctive resource constraint. 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 the pot!). 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.

Example:

Let A, B, C, D, E be five activities requiring respectively 3, 2, 1, 4, 1 units of a resource of capacity 4. These activities last respectively 3, 6, 3, 2 and 5 units of time. Moreover, there are 4 precedence constraints $(A \rightarrow C)$, $(A \rightarrow E)$, $(B \rightarrow C)$, and $(B \rightarrow E)$. Let us build the incompatibility graph of this instance (see Figure C-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*, *D*, *E*} and {*A*, *B*, *D*, *C*}. Our algorithm starts with {*A*, *D*} and successively adds *B* (based on C_{cap}) and *E* (which is longer than *C*) to the clique.

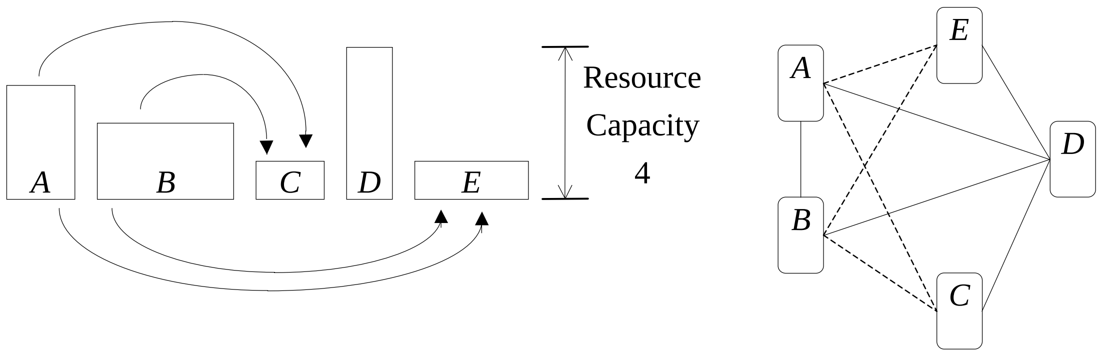


Figure C-2. The incompatibility graph of the instance described in the example above.

C.3.3. 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 satisfies also the additional constraints).

Immediate scheduling rule

Let A_i be an unscheduled activity of minimal earliest end time. Let *O* be the set of activities which can be "partially" scheduled in the interval $[r_i\ eet_i]$, *i.e.*, $O = \{A_j \mid d_j > r_i$ and $eet_i > r_j\}$.

Proposition C-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_i can be scheduled at r_i.

Proof.

Suppose that there is a schedule *S* that satisfies all the constraints posted so far. Let us examine *S*. All predecessors of A_i are scheduled before r_i since the earliest end time of A_i is minimal. Moreover, there is enough space on each resource to schedule A_i at r_i since all activities in *O* can execute in parallel. *S* can thus be modified by bringing A_i back to r_i. □

Single incompatibility rule [Demeulemeester and Herroelen, 1992]
Let t_{min} be the minimal release date among the release dates of unscheduled activities.

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

Proof.
see [Demeulemeester and Herroelen, 1992]. ☐

Incompatible set decomposition rule
We propose an extension of the single incompatibility rule based on a directed compatibility graph. Let t_{min} and t_{max} be respectively the minimal release date and the maximal deadline among unscheduled activities. Consider the directed graph $\Gamma = (X, U)$, where X is a set of vertices corresponding to the activities A_i such that $t_{min} < eet_i$ and $lst_i < t_{max}$. U is the set of directed arcs such that $(A_i, A_j) \in U$ if and only if either $(A_i, A_j) \notin E$ (*i.e.*, A_i and A_j are not incompatible as defined in Section C.3.2) or A_i precedes A_j in the transitive closure of the precedence network. Let $X_1, X_2, ..., X_m$ be the strongly connected components of Γ, *i.e.*, $\{X_1, X_2, ..., X_m\}$ is a partition of X such that any two activities A_i and A_j belong to the same X_i if and only if there is a directed path from A_i to A_j and from A_j to A_i. Let γ be the quotient graph of Γ (the "strongly connected" relation is an equivalence relation). 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 $X_1, X_2, ..., X_m$.

Proposition C-4.
For all i in $[1\ m]$, all activities in X_i can be scheduled before all activities in X_{i+1}.

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_i at which an activity A_i of X_1 is scheduled after an activity of $X - X_1$ is minimal. Let t_i' be the first time after t_i such that no activity of X_1 is scheduled at t_i'. Let t_j 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 $[t_j\ t_i]$ and $[t_i\ t_i']$ (*cf.* Figure C-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_j, t_i and t_i', 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_i exists and is minimal. □

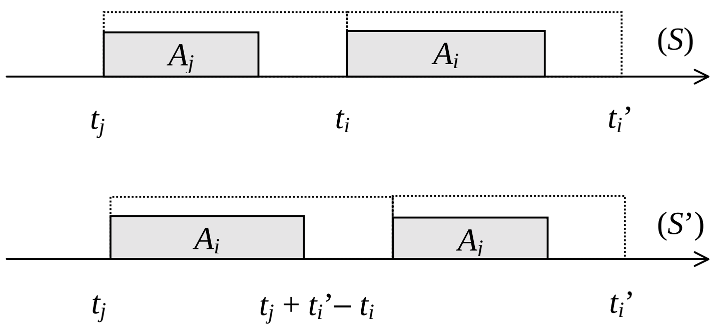


Figure C-3. The relative positions of A_i and A_j

Ordering the subsets $X_1, ..., X_m$ is interesting for two reasons. First, additional precedence constraints can be added. Second, the problem can be decomposed into m optimization problems. Indeed, since subsets $X_1, ..., X_m$ 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 $O(|U|)$ thanks to the depth first algorithm of Tarjan [Gondran and Minoux, 1995].

Example.

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 C-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 processing times are not necessary for the example.

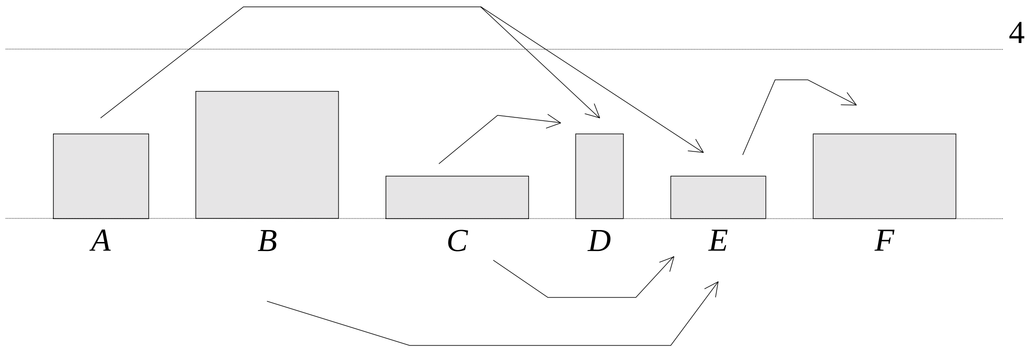


Figure C-4. A simple instance of the RCPSP

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 C-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*}.

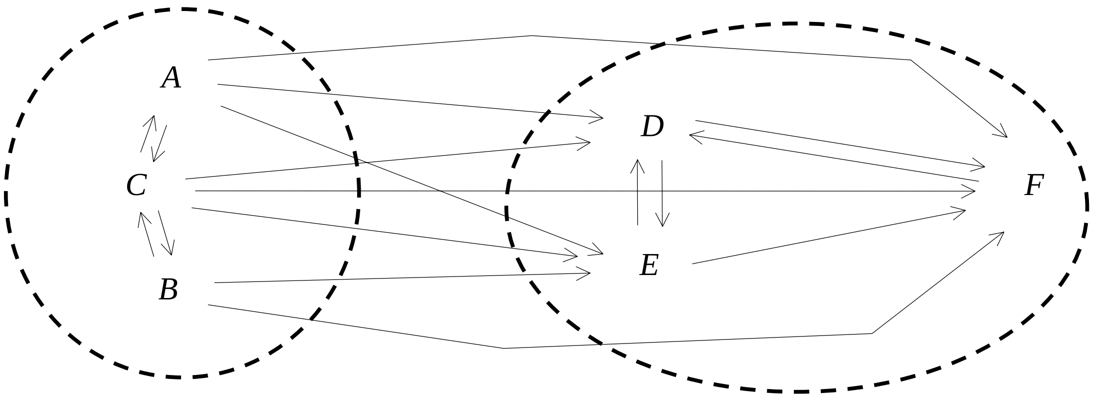


Figure C-5. The directed graph associated with activities A, B, C, D, E and F.

C.3.3. *Experimental Results*

The following tables provide the results obtained on different sets of benchmarks with four different versions of our search procedure:

- with or without using the adjustments based on the Fully Elastic relaxation of the CuSP ("FE" or "NO" in column E.F.),
- with or without the incompatible set decomposition rule ("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 PC Dell GXL 5133. Table C-5 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, in [Caseau and Laburthe, 1996a] the overall Patterson set is solved 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 fully elastic adjustments 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 *et al.*, 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, $capacity(R) = \max_i(capacity(A_i, R))$, 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 C-6 provides the results for the 120 instances of resource strength (RS) 0.2, Table C-7 provides the results for the 120 instances of resource strength 0.7, and Table C-8 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" (except one of them!), so for this subset the interest of the more complex techniques does not appear.

Table C-10 provides the average precedence ratio, 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 C-9 provides the results. It clearly shows that the fully elastic adjustment scheme is a crucial technique for solving these instances. However, one may wonder whether the versions with no cumulative adjustments 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 adjustments to solve only 4 additional instances in an average of 8173 backtracks and 146.7 seconds. With the fully elastic adjustments, 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.

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 C-5. Patterson (110 instances of average disjunctive ratio 0.67)

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 C-6. KSD RS 0.2 (120 instances of average disjunctive ratio 0.65)

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 C-7. KSD RS 0.7 (120 instances of average disjunctive ratio 0.53)

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 C-8. KSD ALL (480 instances of average disjunctive ratio 0.56)

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 C-9. BL (40 instances of average disjunctive ratio 0.33)

	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

Table C-10. Average ratios and standard deviations for different problem sets

This algorithm has two drawbacks. First, it fails to detect bottleneck resources. Moreover, it only relies on the propagation of the redundant disjunctive resource-constraints and does not take advantage of the powerful branching schemes developed for such resource constraints. We then developed a procedure which first focuses on the disjunctive resources of the instance (*i.e.*, the resources of capacity 1 that are either part of the data of

the instance or those that correspond to a redundant constraint). When all of them are ordered with the edge-finding branching technique (*cf.*, Section C.1), the previous cumulative branching scheme is used to complete the schedule. In the following experiments, we also evaluate the effect of using the more powerful time-bound adjustment techniques described in Section B.3 (Partially Elastic adjustments and Left-Shift/Right Shift adjustments).
Intuitively, the new branching scheme should benefit to highly disjunctive instances (because for such instances, the redundant disjunctive resources are very loaded) while the use of more powerful time-bound adjustment techniques should benefit to the less disjunctive ones. Our initial experiments confirmed this hypothesis. In particular, it happened that when using the edge-finding branching technique the incompatible-set decomposition rule became ineffective. This is the reason why it is not considered in the following of this experimental study.

One point of flexibility has been kept in the resulting algorithm, which corresponds to the use of a necessary condition (Section B.3.1) and of a time-bound adjustment scheme (Section B.3.2).

- The first version, NO, uses none of the necessary conditions and time-bound adjustment techniques presented in Section B.3.
- The second version, FE, relies on the fully elastic relaxation, both for the necessary condition for existence and for the time-bound.
- The third version, PE, relies on the partially elastic relaxation (Sections B.3.1.2 and B.3.2.2), both for the necessary condition for existence and for the time-bound adjustment. However, the results reported below rely on a straightforward $O(n^3)$ algorithm instead of using the more complex algorithm described in Section B.3.2.2.
- The fourth version, LSRS, relies on the necessary condition and the time-bound adjustments based on the left-shift / right-shift energy consumption (Sections B.3.1.3 and B.3.2.3). However, we quickly found out that this version was too time-consuming for producing any useful result. This is quite understandable. Indeed, the number of intervals to consider (*cf.* Proposition B-11) is multiplied by $3 * 3 + 3 + 3 = 15$ in comparison to the partially elastic case! Consequently, we tried to determine a better tradeoff, *i.e.*, to reduce the number of intervals to examine without compromising too much with respect to the effectiveness of the evaluation (detection of impossibilities and time-bound adjustments). After a few trials, we decided to reduce the set of intervals to the Cartesian product of $O_1'=\{r_i, 1 \le i \le n\} \cup \{d_i - p_i, 1 \le i \le n\}$ and $O_2'=\{d_i, 1 \le i \le n\} \cup \{r_i + p_i, 1 \le i \le n\}$.

The four versions were tested on the "Patterson" set, on the KSD set, on the BL set and also on the "Alvarez" set [Alverez-Valdès and Tamarit, 1989], which includes 48 instances with 27 activities, 48 instances with 51 activities and 48 instances with 103 activities (the last 48 instances are ignored in this computational study); the average disjunctive ratio for the Alvarez set being 0.82.

Each version of our branch and bound algorithm has been applied to each of the 726 instances, with a maximal CPU time of half an hour on a PC Dell OptiPlex GX Pro 200 MHz running Windows NT. Tables C-11, C-12, C-13 and C-14 present the results obtained respectively on the Alvarez instances, the Patterson instances, the KSD instances and the BL instances. For each version of the algorithm, each table provides the number of instances solved (including proof of optimality), the average number of backtracks to solve these instances, and the corresponding average CPU time, in seconds. For each set of instances, the last columns of the corresponding table provide the average number of backtracks and CPU time obtained on the subset of instances solved by all of the four algorithms.

Algorithm	Solved	BT	CPU	BT over 73 instances	CPU over 73 instances
NO	80	4027	78.1	244	11.2
FE	78	1810	58.4	244	14.3
PE	76	990	88.1	244	39.1
LSRS	73	243	64.7	243	64.7

Table C-11. Experimental results on the 96 Alvarez instances (27 or 51 activities)

Algorithm	Solved	BT	CPU	BT over 110 instances	CPU over 110 instances
NO	110	143	1.6	143	1.6
FE	110	139	2.3	139	2.3
PE	110	128	7.7	128	7.7
LSRS	110	111	8.3	111	8.3

Table C-12. Experimental results on the 110 Patterson instances

Algorithm	Solved	BT	CPU	BT over 451 instances	CPU over 451 instances
NO	465	4612	26.1	1181	6.8
FE	461	2593	28.9	1101	12.8
PE	453	1455	52.9	909	47.2
LSRS	451	794	52.8	794	52.8

Table C-13. Experimental results on the 480 KSD instances

Algorithm	Solved	BT	CPU	BT over 31 instances	CPU over 31 instances
NO	31	84632	187.9	84632	187.9
FE	39	17171	80.9	4839	27.1
PE	40	3757	46.7	2177	30.4
LSRS	40	3400	38.4	1868	25.6

Table C-14. Experimental results on the 40 BL instances

The effect of the different satisfiability tests and time-bound adjustment algorithms clearly depends on the set of instances. Considering only the instances solved by all algorithms, the reduction in the average number of backtracks between NO and LSRS is almost null on the Alvarez set, and equal to 22%, 33% and 98 % on the Patterson, KSD and BL sets (respectively). On the Alvarez, Patterson and KSD sets, the cost of the more complex time-bound adjustment algorithms is not balanced by the subsequent reduction of search, and the CPU time increases. On the contrary, LSRS performs much better than NO on the BL set. On the 31 instances solved by all algorithms, the number of backtracks is divided by 45, and the overall CPU time by more than 7. Figures C-6 and C-7 illustrate the behavior of NO and LSRS on the KSD instances and on the BL instances.

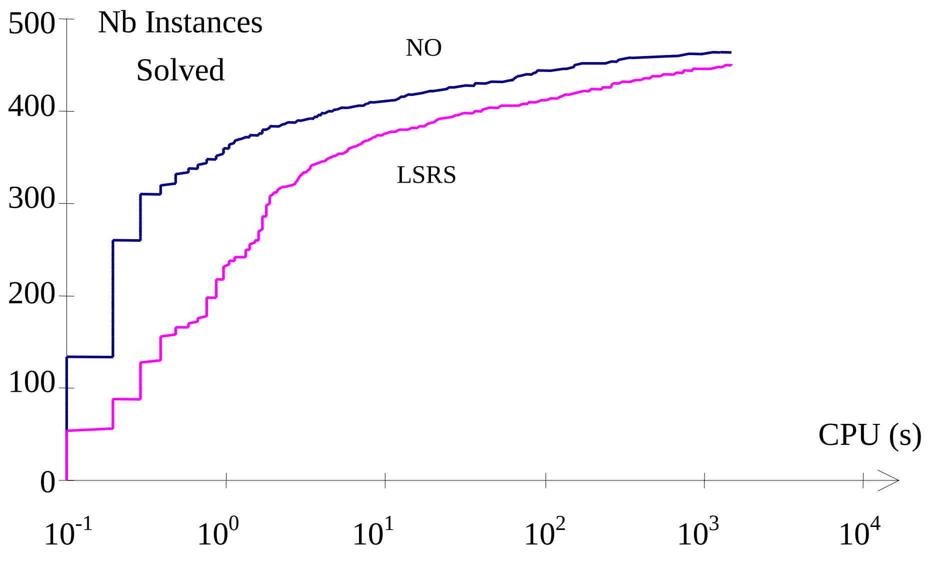


Figure C-6. The behavior of LSRS *and* NO *on the KSD instance set. Each curve shows the number of instances solved in a given amount of CPU time.*

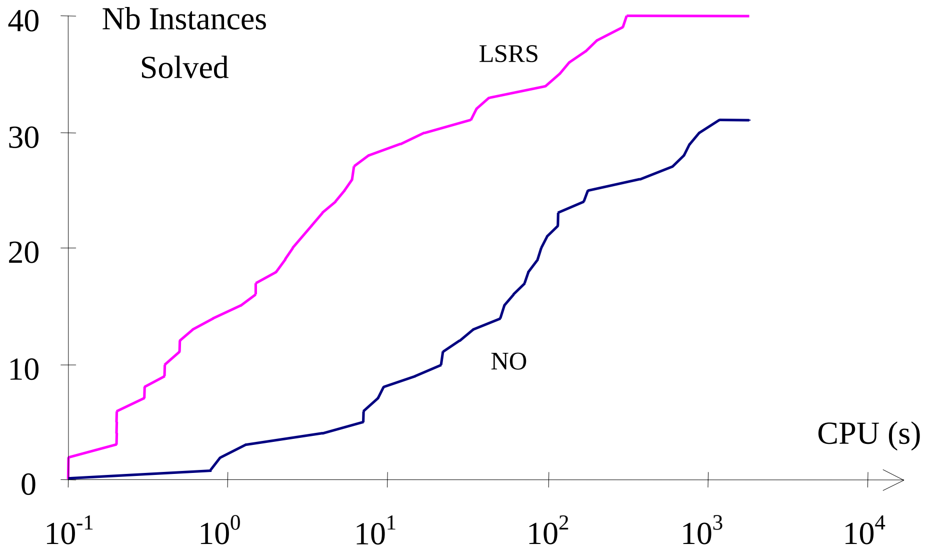


Figure C-7. The behavior of LSRS *and* NO *on the BL instances.*

The three resource constraint propagation schemes presented in Section B.3 prove to be effective on some, but not all problem instances in the cumulative scheduling class. Computational results have shown that, on "highly disjunctive" project scheduling instances, the algorithms presented in Section B.3 induce an overhead that is not balanced by the resulting reduction of search. On the other hand, the most expensive techniques prove to be highly useful for the resolution of less highly disjunctive problems. These results have been confirmed by another experimental study led on the Multi-Processor Flow-Shop (see [Vignier, 1997] for an extensive study of this problem), a special case of the RCPSP that is highly cumulative. As shown in [Néron *et.al.*, 1998], the time-bound adjustment techniques have shown to be extremely efficient for this problem. Our procedure compares very well to the best known procedures for the Multi-Processor Flow-Shop (*e.g.*, [Portmann *et al.*, 1997], [Vignier, 1997], [Carlier and Néron, 1998]).

We have not incorporated in our branch and bound procedure all the results obtained by other researchers for the Resource-Constrained Project Scheduling Problem (RCPSP). In particular, we have not used, until now, any "intelligent backtracking" rule such as the cut-set rule of [Demeulemeester and Herroelen, 1992]. This may seem a little "strange" given the excellent results reported in [Demeulemeester and Herroelen, 1995], in particular on the KSD instances, even with a limited use of the cut-set rule. However, it appears that many industrial scheduling problems include a variety of additional features (including, for example, elastic activities [Caseau and Laburthe, 1996a]) which seem to require the use of other techniques. Nevertheless, in the case of the pure RCPSP, it would be interesting to determine how subsequent improvements of our procedure would influence experimental results, and hence our conclusions on the usefulness of the various adjustment techniques that have been proposed.

C.4. *Minimizing the Number of Late Activities on a Single Machine*[13]

Few exact approaches have been made to solve the $(1|\,r_j\,|\Sigma U_j)$. [Dauzère-Pérès, 1995] shows that the problem can be modeled by a Mixed Integer Program (MIP). Unfortunately, instances with more than 10 jobs could not be considered because of the size of the MIP. [Dauzère-Pérès and Sevaux, 1998b] describes a branch and bound procedure that basically relies (1) on a dominance property stating that there exists an optimal schedule of on-time jobs such that for any pair of jobs (J_i, J_j) successively sequenced on the machine, either $(r_i<r_j)$ or $(d_i<d_j)$ or $(r_i=r_j$ and $d_i=d_j)$ and (2) on a set of three lower bounds. The first lower bound is obtained by solving a relaxed MIP that exploits the dominance property. The two other lower bounds are simply obtained by relaxing the release dates or conversely the due dates. (When release dates are equal the problem of minimizing the number of late jobs is polynomial.)
In this section we show that the constraint propagation methods developed in Section B.4 can be used in a branch and bound procedure for the $(1|r_j|\Sigma U_j)$ problem. Our experimental results show that this branch and bound procedure outperforms all other approaches.

To reach an optimal solution, we solve several decision variants of the problem. More precisely, we rely on the following scheme.

1. Compute an initial lower bound of $(1|r_j|\Sigma U_j)$; set the minimum of *reject* to this value (recall that number of late activities is represented by the variable *reject*).
2. Try to bound *reject* to its minimal value N.
3. If there is a feasible schedule with N late activities then stop (N is the optimum). Otherwise, backtrack and remove N from the domain of *reject* and go to step 2.

Beside the lower bound computation, our search strategy is based on three principles.

- First, at each node of the search tree, we verify that there exists a schedule of the activities that have to be on-time (if not, a backtrack occurs). Such a verification is NP-hard in the strong sense but it turns out to be "easy" in practice.
- Second, given the above verification, our branching scheme simply consists in selecting an unbound variable $in(A_i)$ to instantiate either to 1 or 0.

[13] Most of the results presented in this section come from [Baptiste *et. al.*, 1998c] and [Péridy *et al.*, 1998].

- Third, we use several dominance properties that allow us to generate constraints of the form "if A_i is on-time then A_j is also on-time" ($in(A_i) \Rightarrow in(A_j)$), which are in turn exploited as part of constraint propagation; for each activity A_i, $O(A_i)$ denotes the set of activities that have to be on time if A_i is on-time and, symmetrically, $L(A_i)$ is the set of activities that have to be late if A_i is late ($A_i \in O(A_i)$ and $A_i \in L(A_i)$).

C.4.1. Search Strategy

The search tree is built as follows. While all variables $in(A_i)$ are not instantiated,
1. select an activity A_i such that $in(A_i)$ is not instantiated,
2. impose the fact that A_i must be on-time (if a backtrack occurs, A_i must be late), *i.e.*, $in(A_i) = 1$,
3. apply dominance properties and propagate constraints
4. check that there exists a feasible One-Machine schedule of the activities that must be on-time (if not, the problem is inconsistent and a backtrack occurs).

When the branch and bound succeeds, all variables $in(A_i)$ are instantiated and at most *reject* of these variables are equal to 0 (because arc-consistency is enforced on the constraint $\Sigma(1 - in(A_i)) = reject$). Moreover there is a feasible One-Machine schedule of the on-time activities (step 4). Since the late activities can be scheduled anywhere, a solution schedule with less than *reject* late activities has been found. Let us detail the heuristic used for the activity selection and the procedure that checks whether there is a feasible One-Machine schedule of the activities that must be on-time.

C.4.1.1. Activity Selection

Let *pmin* be the minimum processing time among activities A_i such that $in(A_i)$ is not instantiated. Let then S be the set of activities A_i such that $in(A_i)$ is not instantiated and such that $p_i \leq 1.1 * pmin$. Among activities in S, we select an activity whose time window is maximum (*i.e.*, $d_i - r_i$ maximum). This heuristic "bets" that it is better to schedule small activities with large time windows rather than large activities with tight time windows.

C.4.1.2. Solving the One-Machine Problem

A large amount of work has been carried on this problem (*e.g.*, [Carlier, 1982]) because it serves as the basis for the resolution of several scheduling problems (*e.g.*, the Job-Shop problem).

Since the One-Machine resolution procedure is called at each node of the search tree, it has to be very fast. Following some initial experiments, it appeared that it is often easy to find a feasible schedule of the activities in O. As a consequence, before starting an exhaustive search, we use a simple heuristic (the Earliest Due Date dispatching rule) to test whether the obtained schedule is feasible or not. [Carlier, 1982] proposes an $O(n \log(n))$ algorithm to implement this heuristic. It appears that over all our experiments, this simple heuristic was able to find a feasible schedule for 97% of the One-Machine instances that had to be solved. For the remaining instances, the branch and bound procedure described below has been used.

To solve the One-Machine Problem we use the edge-finding branching technique (*cf,* Section C.1) combined with classical resource constraint propagation techniques (disjunctive constraint plus edge-finding bounding technique). However, this branching scheme sometimes is inefficient. In particular, it is unable to focus early in the search tree on bottlenecks that occur late in time. To avoid this drawback, and inspired by the ideas proposed in [Carlier, 1982], we slightly modify the branching scheme as follows:

- If there are some time intervals $[r_j, d_k]$ such that (1) the resource can never be idle in $[r_j, d_k]$ (because the sum of the processing times of the activities that have to execute after r_j and before d_k is equal to $d_k - r_j$) and such that (2) there is an activity A_i that can start before r_j and that can end after d_k, then we select among these intervals one whose size is maximum. The branching decision is then: Either A_i ends before r_j or A_i starts after d_k.
- Otherwise, the edge-finding branching scheme is applied.

C.4.1.3. Dominance Properties

Dominance properties allow to reduce the search space to schedules that have a particular structure. The most important dominance property relies on the idea that "it is better to schedule small activities with large time windows than large activities with small time windows". We also propose two other dominance rules that respectively fix the start times of some activities and split the problem into distinct sub-problems.

C.4.1.3.1. Dominance of Small Activities with Large Time-Windows

We rely on the observation that on any solution, if a large activity A_j is on-time and is scheduled inside the time window $[r_i, d_i]$ of a smaller activity A_i that is late, then the activities A_i and A_j can be "exchanged", *i.e.*, A_i becomes on-time and A_j becomes late. We suppose that jobs are sorted in non decreasing order of processing times. Our dominance property is based upon the binary activity-relation "<" described as follows:

$$\forall A_i, \forall A_j, A_i < A_j \Leftrightarrow \begin{cases} i < j \\ r_i + p_i \le r_j + p_j \\ d_j - p_j \le d_i - p_i \end{cases}$$

"<" is transitive and $\forall A_i, \forall A_j, A_i < A_j \Rightarrow A_i \neq A_j$. Thus, it defines a strict partial order on activities. Proposition C-5 is the theoretical basis of our dominance property.

Proposition C-5.
Recall that L is the set of activities that have to be late and that O is the set of activities that have to be on-time. We claim that there is an optimal schedule such that

$$\forall A_i, \forall A_j, (\neg(A_i < A_j)) \vee (A_j \in L) \vee (A_i \in O)$$

Proof (sketch).
Consider an optimal schedule such that the first index i, for which there exists an activity A_j that violates the above equation, is maximum. We have

$$(A_i < A_j) \wedge (A_j \in O) \wedge (A_i \in L)$$

We build a new schedule obtained by "exchanging" A_i and A_j. More precisely, A_i is scheduled at the date max(*start*(A_j), r_i) and A_j is scheduled after all other activities (it then becomes late). It is obvious to verify that the new schedule is still feasible and that A_i is now on-time. Now, suppose that there exists a late activity A_k such that $A_k < A_i$. We then have $A_k < A_i < A_j$. Moreover, A_k was also late on the initial schedule. Consequently, $k > i$ because of the choice of i. This contradicts our hypothesis on the choice of the initial schedule. □

Proposition C-5 allows us to define for each activity A_i the sets

- $L(A_i) = \{A_j \mid A_i < A_j\} \cup \{A_i\}$ and
- $O(A_i) = \{A_j \mid A_j < A_i\} \cup \{A_i\}$.

In addition, for any pair (A_i, A_j) with $i<j$, the following constraint can be added:

$(r_i + p_i > start(A_j) + p_j) \vee (start(A_j) > d_i - p_i) \vee (in(A_j) = 0) \vee (in(A_i) = 1)$.

Arc-B-consistency is achieved on this new constraint. It allows to prove that some activities must be late or on-time and it tightens the domains of the start variables.

C.4.1.3.2. Straight Scheduling Rule

We propose a simple dominance property which schedules automatically a set of activities if they "fit" in a particular interval.

Proposition C-6.
Given a time-interval $[t_1, t_2]$, let $J(t_1, t_2) = \{A_i \notin L \mid t_1 < d_i \text{ and } r_i < t_2\}$ be the set of activities that may execute (even partially) in $[t_1, t_2]$. Moreover, suppose that there exists a feasible

schedule S_J of $J(t_1, t_2)$ that is idle before t_1 and after t_2. Then there exists an optimal overall schedule S such that between t_1 and t_2 the schedules S and S_J are identical.

Proof.
Obvious. □

Consider now any time point t_1 and let $J(t_1)$ be the set of activities A_i that do not have to be late and that can end after t_1 ($t_1 < d_i$). We use the following algorithm to look for a time-point t_2 that satisfies the conditions of Proposition C-6. In this algorithm, we assume that $J(t_1)$ is not empty.

Algorithm C-1.

```
1  S = J(t1), t2 = max(t1, min({ru, Au ∊ S}))
2  stop = false, success = false
3  while (S ≠ ∅ and stop = false)
4    Select Ai in S with ri ≤ t2 and with di minimal
5    S = S – Ai, t2 = t2 + pi
6    if (di < t2)
7      stop = true
8    else if (t2 ≤ min({ru, Au ∊ S})
9      stop = true, success = true
10   end if
11 end while
```

At the end of Algorithm C-1, if the Boolean "`success`" is true, the conditions of Proposition C-6 hold for the points (t_1, t_2). Indeed, all activities in $J(t_1)$ that can start strictly before t_2 are scheduled and do not end after their due date on this schedule (lines 6-10). These activities are exactly those in $J(t_1, t_2)$. For a given value of t_1, Algorithm C-1 runs in $O(n^2)$ since there are $O(n)$ activities in S and since each time, "`min({ru, Au ∊ S}`" has to be computed (line 8). Now remark that if t_1 is not a due date of an activity then $J(t_1 – 1, t_2) = J(t_1, t_2)$ and a schedule that can fit in $[t_1, t_2]$ can also fit in $[t_1 - 1, t_2]$. Hence we decided to apply Algorithm C-1 for $t_1 = \min_i(r_i)$ and for t_1 in $\{d_1, d_2, …, d_n\}$. This leads to an overall complexity of $O(n^3)$.

C.4.1.3.3. Decomposition Rule

The basic idea of the decomposition is to detect some cases in which the problem can be split into two sub-problems. Each of them being solved independently.

Proposition C-7.
Let t_1 be a time point such that $\forall\ A_i \notin L$, either $d_i \leq t_1$ or $t_1 \leq r_i$. Any optimal schedule is the "sum" of an optimal schedule of $\{A_i \notin L \mid d_i \leq t_1\}$ and of an optimal schedule of $\{A_i \notin L \mid t_1 \leq r_i\}$

Proof.
Obvious because activities of $\{A_i \notin L \mid d_i \leq t_1\}$ and of $\{A_i \notin L \mid t_1 \leq r_i\}$ do not compete for the machine. $\square$

We only consider the values of t_1 that are release dates (if the problem can be decomposed at time t_1, it is easy to verify that it can also be decomposed at the next release date immediately after t_1). There are $O(n)$ distinct release dates and the decomposition test (at a given time point) obviously runs in linear time. Consequently, the overall decomposition test runs in $O(n^2)$.
Once the problem has been decomposed, the optimum of each sub-problem is computed, and we simply have to verify that the sum of these optima is lower than or equal to N.

C.4.2. *Experimental Results*

Our implementation of the branch and bound is based on CLAIRE SCHEDULE [Le Pape and Baptiste, 1997a]. To test the efficiency of our branch and bound procedure, instances of the $(1|r_j|\Sigma U_j)$ have been generated. Our intuition is that one shall pay attention to at least three important characteristics:

- The distribution of the processing times.
- The overall "*load*" of the machine; where the load is the ratio between the sum of the processing times of the activities and (max d_i – min r_i).
- The margin $m_i = d_i - r_i - p_i$ of each activity.

Our generation scheme is based on 4 parameters: The number of activities n and the three statistical laws followed by the release dates, the processing times and the margins (given r_i, p_i and m_i, the due date can be immediately computed $d_i = m_i + r_i + p_i$).

- Processing times are generated from the uniform distribution $[p_{min}, p_{max}]$.
- Release dates are generated from the normal distribution $(0, \sigma)$.
- Margins are generated from the uniform distribution $[0, m_{max}]$

Given these parameters and relying on the fact that most of the release dates are in $[-2\sigma, 2\sigma]$, the load is approximately equal to $(0.5\,n\,(p_{min} + p_{max}))\,/\,(4\sigma + p_{max} + m_{max})$. Given n, p_{min}, p_{max}, m_{max} and *load*, this allows us to determine a correct value of σ.
One instance has been generated for each of the 960 combinations of the parameters (n, (p_{min}, p_{max}), m_{max}, *load*) in the Cartesian product

$$\{10,20,40,60,\ldots,140\} * \{(0,100),(25,75)\} * \{50,100,\ldots,500\} * \{0.5,0.8,\ldots,2.0\}.$$

Table C-15 reports the results obtained for different values of n; each line corresponding to 120 instances. The column "%" provides the percentage of instances that have been solved within one hour of CPU time on a PC Dell OptiPlex GX Pro 200 MHz running Windows NT. The columns "Avg CPU" and "Max CPU" report respectively the average and the maximum CPU time in seconds required to solve the instances of each group (n = 10, n = 20, …, n = 140) that could be solved within the time limit. Figure C-8 illustrates the (reasonable) loss of efficiency of our algorithm when the number of activities increases. Instances are solved within a few amount of backtracks (35 backtracks on the average for n = 60, 154 for n = 100 and 259 for n = 140). The extensive use of pruning techniques can explain the good behavior of the algorithm. The large time spent per node is of course the drawback of such an approach.

Throughout our experiments, we discovered that the efficiency of the algorithm varies from one instance to another (which is not very surprising for an NP-complete problem). To characterize, from an experimental point of view, the instances that our algorithm found too hard to solve, we generated 25 instances of 60 activities for each combination of the parameters $(p_{min}, p_{max}) \in \{(0,100),(25,75)\}, m_{max} \in \{25,50,\ldots,500\}$ and $load \in \{0.2, 0.3, 0.4, \ldots, 2.2\}$. 5 minutes have been allotted to each instance. Figures C-9 and C-10 report the average CPU time required to solve the different group of instances (for the 2% of instances that could not be solved within the time limit, we decided to count 5 minutes when computing the average CPU time). The study of Figures C-9 and C-10 leads us to the following remarks:

- The tighter the processing time distribution is, the harder the instance is.
- The hardest instances seem to be those for which the *load* ratio is between 0.7 and 1.2 and when the margin parameter m_{max} varies in [275, 475] (for an average processing time of 50).
- When the *load* ratio becomes very high, instances are quite easy. An hypothesis is that the search tree becomes small (because very few activities can be on-time and thus very few decisions have to be taken). Conversely, when the load ratio is very low, instances are easy (because few activities are late on an optimal solution).
- When both *load* and m_{max} are low, the behavior of the algorithm is somewhat surprising. Further investigations have shown that for these instances the number of backtracks is kept very small but the time spent per node increases a lot. This is, we think, a side-effect of our current implementation: The decomposition rule is likely to be triggered very often for such instances. Hence, a large amount of (small and easy) sub-problems are solved one after another. This is a source of inefficiency, because in the current implementation, the data structures representing each sub-problem have to be initialized at each decomposition.

n	%	Avg CPU	Max CPU
10	100.0	0.0	0.1
20	100.0	0.2	0.7
40	100.0	3.1	27.5
60	100.0	23.2	184.5
80	96.7	117.3	2903.2
100	90.0	273.5	2395.3
120	84.2	538.2	3263.4
140	72.5	1037.3	3087.8

Table C-15. Behavior of the algorithm for several sizes of instances

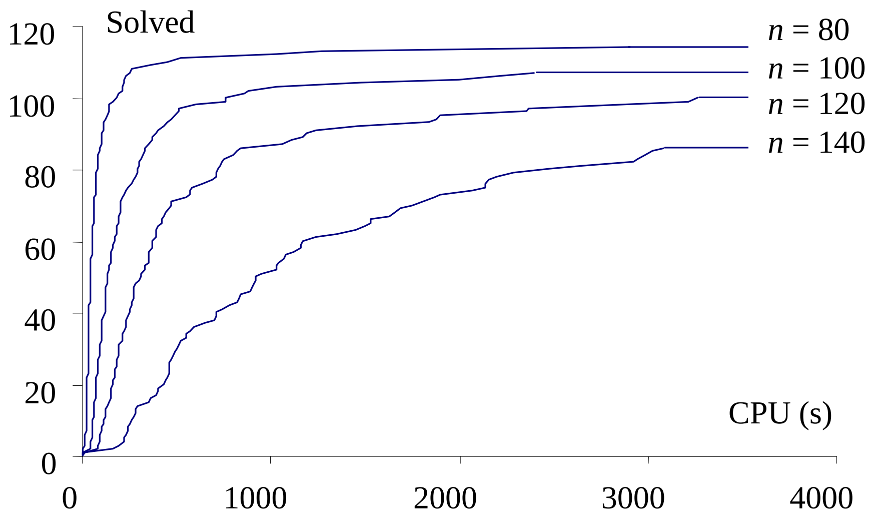


Figure C-8. Number of instances solved within a time limit in seconds

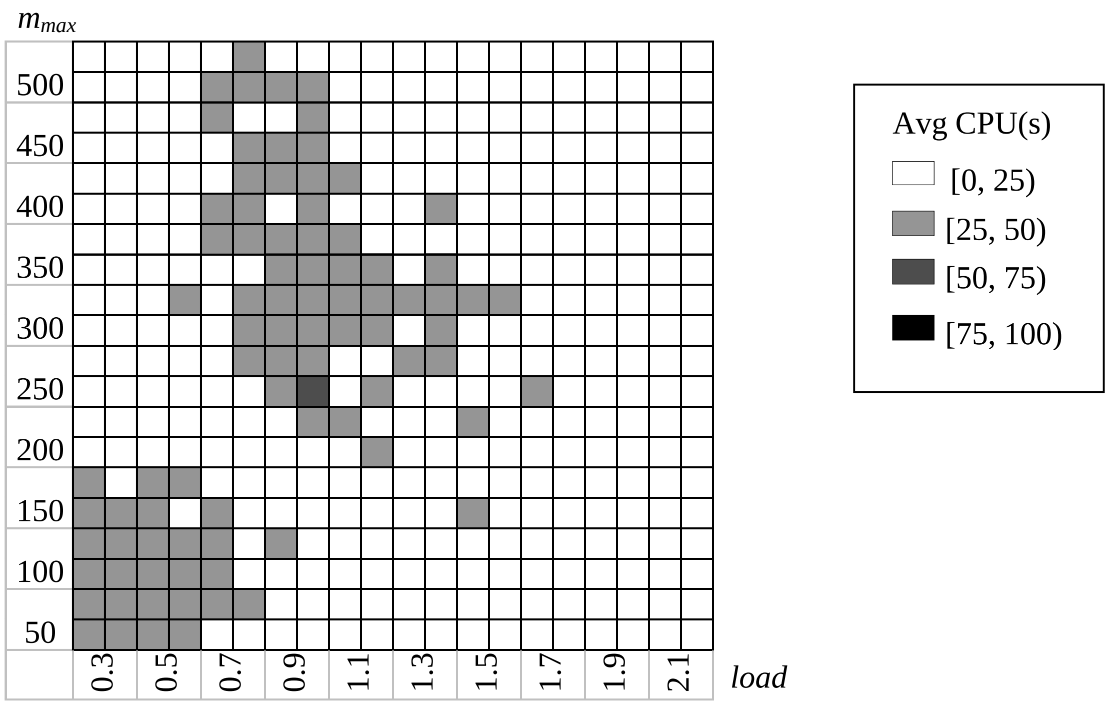


Figure C-9. The behavior of the algorithm on 60-activities instances with different characteristics (parameters p_{min} and and p_{max} kept to 0 and 100).

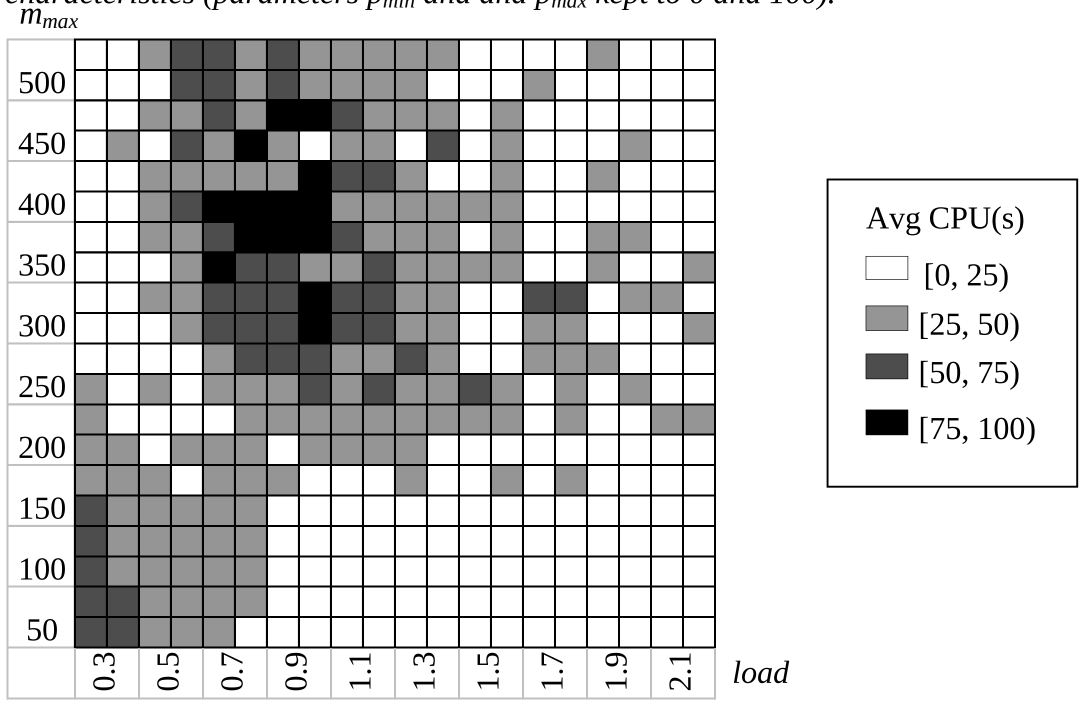


Figure C-10. The behavior of the algorithm on 60-activities instances with different characteristics (parameters p_{min} and and p_{max} kept to 25 and 75).

The branch and bound procedure has also been tested on four sets of 160 instances (instances with 80, 100, 120 and 140 jobs) generated by Stéphane Dauzère-Pérès and Marc Sevaux. Table C-16 reports the results obtained on these sets by our procedure and by the procedure described in [Dauzère-Pérès and Sevaux, 1998b]. Column *n* provides the number of jobs in each of the sets of 160 instances. Columns "%" provide the percentage of problems solved for both methods. A time limit of one hour (with a Pentium 200) has been set on our procedures. The branch and bound of [Dauzère-Pérès and Sevaux, 1998b] has been stopped after 100000 nodes (which corresponds to 1000 seconds on a SUN UltraSparc workstation for instances with 140 jobs). The columns "Avg CPU", "Avg Bck" and "Avg Nds" report respectively the average CPU time, the average number of backtracks and the average number of nodes used to solve the instances that could be solved by both methods.

At this point several remarks can be made.

- In terms of number of instances solved, our procedure compares very well to the one of [Dauzère-Pérès and Sevaux, 1998b].
- The search tree that we explore is far smaller than the one of [Dauzère-Pérès and Sevaux, 1998b].
- On "easy" instances, our procedure is significantly slower as its competitor.

This can be explained by the high amount of propagation that is performed at each node of our branch and bound. Given the theoretical complexity of the propagation algorithms, it appears that even the generation of an initial solution (where no backtrack occurs) can be costly. However, the extensive use of propagation allows us to solve hard instances in a very few amount of nodes.

	[Baptiste *et al.*, 1998c]			[Dauzère-Pérès and Sevaux, 1998b]		
n	%	Avg CPU	Avg Bck	%	Avg CPU	Avg Nds
80	100.0	20.7	14	95.0	7.0	1631
100	97.5	55.0	35	88.1	27.5	4353
120	98.1	98.0	17	82.5	21.4	2467
140	93.1	121.2	5	80.6	63.8	4557

Table C-16. A comparison of two branch and bound procedures on four sets of instances.

We have proposed a set of techniques, including global constraint propagation, to solve a particular partial CSP. This is of course a first step and a lot of work remains to adapt and to develop global constraints in over-constrained frameworks. In particular, we think that a large part of our results can be extended to the cumulative case. Studying more general situations where activities do not have the same importance (*i.e.*, each activity has a weight) or where some activities have several due dates (with a different weight for each due date) is another exciting research direction.

Chapitre D. Conclusion (en français)

Nous avons présenté dans ce mémoire un ensemble original de techniques de propagation de contraintes de ressources. Pour chaque type de ressource étudié, nous avons mené une comparaison théorique et expérimentale des différents algorithmes de propagation.

- Dans le cas préemptif comme dans le cas des ressources surchargées, les outils déductifs que nous avons mis en place sont totalement originaux et ouvrent aux systèmes de programmation par contraintes de nouveaux champs d'application en ordonnancement.
- Dans le cas cumulatif, les méthodes déductives que nous avons proposées reprennent en partie des résultats de la littérature. Nous avons particulièrement porté notre effort sur la caractérisation théorique de ces méthodes ainsi que sur leur coût algorithmique. Nous avons aussi cherché à comparer ces méthodes à des calculs de bornes inférieures classiques de Recherche Opérationnelle.

L'efficacité des algorithmes de propagation a été démontrée sur un ensemble de problèmes classiques de Recherche Opérationnelle : la procédure arborescente pour le Job-Shop préemptif résout toutes les instances de la littérature de taille 10*10. La méthode exacte pour le RCPSP est particulièrement efficace sur un ensemble d'instances. Enfin, les résultats que nous obtenons sur le problème de la minimisation du nombre de jobs en retard sont les meilleurs connus à ce jour.

Il nous semble que les travaux présentés dans ce mémoire peuvent être prolongés dans plusieurs directions.

- Notons tout d'abord que les contraintes de ressources que nous avons décrites dans l'introduction n'ont pas encore toutes été étudiées. En particulier, nous n'avons que peu travaillé sur les problèmes ayant à la fois une dimension préemptive et cumulative : si certains des algorithmes de propagation que nous avons proposés s'adaptent bien à ce cas, il n'en est pas de même des schémas de séparation. Pour ce type de problème, il faut bien avouer que nous sommes quelque peu démuni. D'autre part, il nous semble important de généraliser les résultats obtenus dans le cas d'une machine surchargée au cas de m-machines surchargées, et, pourquoi pas, en intégrant une notion de poids sur chaque activité.
- Malgré nos efforts, la complexité des algorithmes de propagation que nous proposons reste élevée (quadratique dans le meilleur des cas). Si cette complexité ne pèse pas

d’un poids trop lourd sur nos méthodes de résolution lorsque les instances sont de taille raisonnable, il est évident que sur des instances de grande taille, une telle approche est déraisonnable. Il ne nous reste alors plus qu’à décomposer de façon heuristique le problème en sous problèmes de tailles plus modestes… Vaste champ de recherche !

Nous espérons poursuivre nos recherches sur les problèmes d’ordonnancement, à la frontière entre Recherche Opérationnelle et Intelligence Artificielle. La synergie qui existe entre ces disciplines permet de résoudre dans un cadre flexible des problèmes d’ordonnancement complexes dont la taille augmente chaque jour.

Chapter D. Conclusion

Along this thesis, we have presented a set of original techniques for the propagation of resource constraints. For each type of resource, a theoretical and experimental comparison of the propagation algorithms, has been performed.

- Both for the preemptive case and for overloaded resources, the deductive tools that we have proposed are fully original. They could allow constraint based scheduling tools to tackle some new classes of problem.
- For the cumulative case, the deductive techniques that we have proposed are based upon several results of the literature. We have tried to characterize theoretically these techniques and we have paid much attention to the worst case complexities of the algorithms.

The efficiency of the propagation algorithms has been proven on a set of classical problems of the literature. Both for the preemptive Job-Shop and for the minimization of the number of late jobs, our results outperform the other approaches (if any) of the literature. Concerning the RCPSP, we have shown that the set of techniques proposed is very efficient for a particular class of instances, namely the highly cumulative class.

Following this thesis, it seems to us that several research directions could be of great interest:

- First, notice that all types of resource constraints described in the introduction have not been studied along this thesis. In particular, almost no work has been performed on problems that have both a cumulative and a preemptive dimension. Some propagation algorithms proposed in this thesis apply well to such problems. However, one must admit that it not the case for the branching schemes. On top of that, it seems to us that it could be of great practical interest to generalize the results obtained on a disjunctive overloaded resource to cumulative resources.
- Second, we have seen that, despite our efforts, the theoretical complexities of the algorithms we propose remains high (at least quadratic). A direct consequence is that when increasing the size of instances, such an approach becomes inefficient. The time spent per node being too high. One way to tackle the whole problem is to decompose it heuristically in several problems of reasonable sizes.... Wide research area!

We hope that we will be able to pursue our research on scheduling problems, at the border line between Operations Research and Artificial Intelligence. We believe that the synergy between these disciplines will allow us to solve larger and larger complex problems.

Bibliography

Abderrahmane Aggoun and Nicolas Beldiceanu [1993]. *Extending CHIP in Order to Solve Complex Scheduling and Placement Problems*. Mathematical and Computer Modelling, 17(7):57-73, 1993.

R. Alvarez-Valdès and J.M. Tamarit [1989]. *Heuristic Algorithms for Resource-Constrained Project Scheduling: A Review and an Empirical Analysis*. Chapter 5 in Advances in Project Scheduling, R. Slowinski and J. Weglarz editors, Elsevier, 1989.

David Applegate and William Cook [1991]. *A Computational Study of the Job-Shop Scheduling Problem*. ORSA Journal on Computing, 3(2):149-156, 1991.

Kenneth R. Baker [1974]. *Introduction to Sequencing and Scheduling*. John Wiley and Sons, 1974.

Philippe Baptiste [1994]. *Constraint-Based Scheduling: Two Extensions*. MSc Thesis, University of Strathclyde, 1994.

Philippe Baptiste [1995]. *Resource Constraints for Preemptive and Non-Preemptive Scheduling*. MSc Thesis, University Paris VI, 1995.

Philippe Baptiste [1998a]. *An $O(n^4)$ Algorithm for Preemptive Scheduling of a Single Machine to Minimize the Number of Late Jobs*. Technical Report 98-98, Université de Technologie de Compiègne, 1998.

Philippe Baptiste [1998b]. *Polynomial Time Algorithms for Minimizing the Weighted Number of Late Jobs on a Single Machine when Processing Times are Equal.* Technical Report 98-138, Université de Technologie de Compiègne, submitted, 1998

Philippe Baptiste, Yves Caseau, Tibor Kökény, Claude Le Pape and Robert Rodosek [1998a]. *Creating and Evaluating Hybrid Algorithms for Inventory Management Problems*. Proceedings of the Fourth National Meeting on Practical Approaches to NP-Complete Problems, Nantes, France, 1998.

Philippe Baptiste and Claude Le Pape [1995a]. *Disjunctive Constraints for Manufacturing Scheduling: Principles and Extensions*. Proceedings of the Third International Conference on Computer Integrated Manufacturing, Singapore, 1995. Also in: International Journal of Computer Integrated Manufacturing, 9(4):306-310, 1996.

Philippe Baptiste and Claude Le Pape [1995b]. *A Theoretical and Experimental Comparison of Constraint Propagation Techniques for Disjunctive Scheduling*. Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, Montréal, Québec, 1995.

Philippe Baptiste and Claude Le Pape [1996a]. *A Constraint-Based Branch and Bound Algorithm for Preemptive Job-Shop Scheduling*. Proceedings of the International Workshop on Production Planning and Control, Mons, Belgium, 1996.

Philippe Baptiste and Claude Le Pape [1996b]. *Edge-Finding Constraint Propagation Algorithms for Disjunctive and Cumulative Scheduling*. Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group, Liverpool, United Kingdom, 1996.

Philippe Baptiste and Claude Le Pape [1997a]. *Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems*. Proceedings of the Third International Conference on Principles and Practice of Constraint Programming, Schloss Hagenberg, Austria, published in Lecture Notes of Computer Science 1330, Springer-Verlag, 1997.

Philippe Baptiste and Claude Le Pape [1997b]. *Adjustments of Release and Due Dates for Cumulative Scheduling Problems*. Proceedings of the Third International Conference on Industrial Engineering and Production Management, Lyon, France, 1997.

Philippe Baptiste, Claude Le Pape and Wim Nuijten [1995a]. *Incorporating Efficient Operations Research Algorithms in Constraint-Based Scheduling*. Proceedings of the First International Joint Workshop on Artificial Intelligence and Operations Research, Timberline Lodge, Oregon, 1995.

Philippe Baptiste, Claude Le Pape and Wim Nuijten [1995b]. *Constraint-Based Optimization and Approximation for Job-Shop Scheduling*. Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, IJCAI, Montréal, Québec, 1995.

Philippe Baptiste, Claude Le Pape and Wim Nuijten [1998b]. *Satisfiability Tests and Time-Bound Adjustments for Cumulative Scheduling Problems*. Technical Report 98-97, Université de Technologie de Compiègne, 1998. To appear in Annals of Operations Research.

Philippe Baptiste, Claude Le Pape and Laurent Péridy [1998c]. *Global Constraints for Partial CSPs: A Case Study of Resource and Due-Date Constraints*. To appear in the Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming, Pisa, Italy, 1998.

Nicolas Beldiceanu and Evelyne Contejean [1994]. *Introducing Global Constraints in CHIP*. Mathematical and Computer Modelling, 20(12):97-123, 1994.

Christian Bessière, Eugene Freuder and Jean-Charles Régin [1995]. *Using Inference to Reduce Arc Consistency Computation.* Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, Montréal, Québec, 1995.

Stefano Bistarelli, Ugo Montanari and Francesca Rossi [1995]. *Constraint Solving over Semirings*. Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, Montréal, Québec, 1995.

Jacek Blazewicz, Wolfgang Domschke and Erwin Pesch [1996]. *The Job-Shop Scheduling Problem: Conventional and New Solution Techniques*. European Journal of Operational Research, 93(1):1-33, 1996.

Peter Brucker, Bernd Jurisch and Bernd Sievers [1994]. *A Branch and Bound Algorithm for the Job-Shop Scheduling Problem*. Discrete Applied Mathematics, 49(1):107-127, 1994.

Peter Brucker [1995]. *Scheduling Algorithms*. Springer Lehrbuch, 1995.

Peter Brucker and Olaf Thiele [1996]. *A Branch and Bound Method for the General-Shop Problem with Sequence-Dependent Setup Times*. OR Spektrum, 18:145-161, 1996.

Peter Brucker, Sigrid Knust, Arno Schoo and Olaf Thiele [1998]. *A Branch and Bound Algorithm for the Resource-Constrained Project Scheduling Problem*. European Journal of Operational Research, 107:272-288, 1998.

Peter Brucker, Svetlana A. Kravchenko and Yuri N. Sotskov [1999], *Preemptive Job-Shop Scheduling Problems with a Fixed Number of Jobs*. To appear in ZOR-Mathematical Methods of OR, first issue of 1999.

Jacques Carlier [1982]. *The One-Machine Sequencing Problem*. European Journal of Operational Research, 11(1):42-47, 1982.

Jacques Carlier [1984]. *Problèmes d'ordonnancement à contraintes de ressources : algorithmes et complexité*. Thèse de Doctorat d'Etat, Université Paris VI, 1984.

Jacques Carlier and Philippe Chrétienne [1988]. *Problèmes d'ordonnancement : Modélisation / Complexité / Algorithmes*. Masson, 1988.

Jacques Carlier and Eric Pinson [1989]. *An Algorithm for Solving the Job-Shop Problem*. Management Science, 35(2):164-176, 1989.

Jacques Carlier and Eric Pinson [1990]. *A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem*. Annals of Operations Research, 26:269-287, 1990.

Jacques Carlier and Bruno Latapie [1991]. *Une méthode arborescente pour résoudre les problèmes cumulatifs*. RAIRO Recherche Opérationnelle, 25(3):311-340, 1991.

Jacques Carlier and Eric Pinson [1994]. *Adjustment of Heads and Tails for the Job-Shop Problem*. European Journal of Operational Research, 78(2):146-161, 1994.

Jacques Carlier and Eric Pinson [1996]. *Jackson's Pseudo-Preemptive Schedule for the Pm/ri,qi/Cmax Scheduling Problem*. Technical Report, Université de Technologie de Compiègne, 1996.

Jacques Carlier and Emmanuel Néron [1996]. *A New Branch and Bound Method for Solving the Resource-Constrained Project Scheduling Problem*. Proceedings of the International Workshop on Production Planning and Control, Mons, Belgium, 1996.

Jacques Carlier and Emmanuel Néron [1998]. *An Exact Method for Solving the Multi-Processor Flow-Shop*. Submitted to RAIRO, 1998.

Yves Caseau [1996]. *Contraintes et algorithmes, petit précis d'optimisation combinatoire pratique*. Notes de cours du Magistère de Mathématiques Fondamentales et Appliquées et d'Informatique, Ecole Normale Supérieure, 1996.

Yves Caseau and François Laburthe [1994]. *Improved CLP Scheduling with Task Intervals*. Proceedings of the Eleventh International Conference on Logic Programming, Santa Margherita Ligure, Italy, 1994.

Yves Caseau and François Laburthe [1995]. *Disjunctive Scheduling with Task Intervals*. Technical Report, Ecole Normale Supérieure, 1995.

Yves Caseau and François Laburthe [1996a]. *Cumulative Scheduling with Task Intervals*. Proceedings of the Joint International Conference and Symposium on Logic Programming, Bonn, Germany, 1996.

Yves Caseau and François Laburthe [1996b]. *CLAIRE: A Parametric Tool to Generate C++ Code for Problem Solving*. Working Paper, Bouygues, Direction Scientifique, 1996.

Amedeo Cesta and Angelo Oddi [1996]. *Gaining Efficiency and Flexibility in the Simple Temporal Problem*. Proceedings of the Third International Workshop on Temporal Representation and Reasoning, Key West, Florida, 1996.

Edward G. Coffman Jr. (editor) [1976]. *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons, 1976.

Yves Colombani [1996]. *Constraint Programming: An Efficient and Practical Approach to Solving the Job-Shop Problem*. Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, 1996.

Yves Colombani [1997]. *Un modèle de résolution de contraintes adapté aux problèmes d'ordonnancement : un prototype et une application*. Thèse de l'Université de la Méditerranée Aix-Marseille II, 1997.

Stéphane Dauzère-Pérès [1995]. *Minimizing Late Jobs in the General One-Machine Scheduling Problem*. European Journal of Operational Research, 81:134-142, 1995.

Stéphane Dauzère-Pérès and Marc Sevaux [1998a]. *Various Mathematical Programming Formulations for a General One Machine Sequencing Problem*. Rapport de recherche 98/3/AUTO, Ecole des Mines de Nantes, 1998.

Stéphane Dauzère-Pérès and Marc Sevaux [1998b]. *A Branch and Bound Method to Minimize the Number of Late Jobs on a Single Machine*. Rapport de recherche 98/5/AUTO, Ecole des Mines de Nantes, 1998.

B. De Reyck and W. Herroelen [1995]. *Assembly Line Balancing by Resource-Constrained Project Scheduling Techniques: A Critical Appraisal.* Technical Report, Katholieke Universiteit Leuven.

Erik Demeulemeester [1992]. *Optimal Algorithms for Various Classes of Multiple Resource-Constrained Project Scheduling Problems*. PhD Thesis, Katholieke Universiteit Leuven, 1992.

Erik Demeulemeester and Willy Herroelen [1992]. *A Branch and Bound Procedure for the Multiple Resource-Constrained Project Scheduling Problem*. Management Science, 38(12):1803-1818, 1992.

Erik Demeulemeester and Willy Herroelen [1995]. *New Benchmark Results for the Resource-Constrained Project Scheduling Problem*. Technical Report, Katholieke Universiteit Leuven, 1995.

Jacques Erschler [1976]. *Analyse sous contraintes et aide à la décision pour certains problèmes d'ordonnancement*. Thèse de Doctorat d'Etat, Université Paul Sabatier, 1976.

Jacques Erschler, Pierre Lopez and Catherine Thuriot [1991]. *Raisonnement temporel sous contraintes de ressource et problèmes d'ordonnancement*. Revue d'Intelligence Artificielle, 5(3):7-32, 1991.

Patrick Esquirol [1987]. *Règles et processus d'inférence pour l'aide à l'ordonnancement de tâches en présence de contraintes*. Thèse de l'Université Paul Sabatier, 1987.

Patrick Esquirol, Pierre Lopez, Hélène Fargier and Thomas Schiex [1995], *Constraint Programming*. Belgian Journal of Operations Research, Special Issue Constraint Programming, 35(2):5-36, 1995.

A. Federgruen and H. Groenevelt [1986]. *Preemptive Scheduling of Uniform Machines by Ordinary Network Flow Techniques*. Management Science, 32(3):341-349, 1986.

Barry R. Fox [1990]. *Chronological and Non-Chronological Scheduling*. Proceedings of the First Annual Conference on Artificial Intelligence, Simulation and Planning in High Autonomy Systems, Tucson, Arizona, 1990.

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

Eugene C. Freuder and Richard J. Wallace [1992]. *Partial Constraint Satisfaction*. Artificial Intelligence, 58(1):21-70, 1992.

Michael R. Garey and David S. Johnson [1979]. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

Michel Gondran and Michel Minoux [1995]. *Graphes et algorithmes*. Eyrolles, 1995.

GOThA (Groupe d'Ordonnancement Théorique et Appliqué) Jacques Carlier, Philippe Chrétienne, Jacques Erschler, Claire Hanen, Pierre Lopez, Alix Munier, Eric Pinson, Marie-Claude Portmann, Christian Prins, Christian Proust, Pierre Villon, [1993]. *Les problèmes d'ordonnancement*. RAIRO-Recherche Opérationnelle, 27(1):77-150, 1993.

Hiroshi Kise, Toshihide Ibaraki and Hisashi Mine [1978]. *A Solvable Case of the One-Machine Scheduling Problem with Ready and Due Times*. Operations Research, 26(1):121-126, 1978.

Rainer Kolisch, Arno Sprecher and Andreas Drexel [1995]. *Characterization and Generation of a General Class of Resource-Constrained Project Scheduling Problems*. Management Science, 41(10):1693-1703, 1995.

François Laburthe, Pierre Savéant, Simon de Givry and Jean Jourdan [1998]. *Eclair, a library of constraints over finite domains*. Technical Report ATS 98-2, Thomson CSF, Corporate Research Lab, 1998.

E. L. Lawler [1990]. *A Dynamic Programming Algorithm for Preemptive Scheduling of a Single Machine to Minimize the Number of Late Jobs*. Annals of Operations Research, 26:125-133, 1990.

Claude Le Pape [1988]. Des systèmes d'ordonnancement flexibles et opportunistes. Thèse de l'Université Paris XI, 1988.

Claude Le Pape [1994]. *Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems*. Intelligent Systems Engineering 3:55-66, 1994.

Claude Le Pape [1995]. *Three Mechanisms for Managing Resource Constraints in a Library for Constraint-Based Scheduling*. Proceedings of the INRIA/IEEE Conference on Emerging Technologies and Factory Automation, Paris, France, 1995.

Claude Le Pape [1996]. *Constraint-Based Scheduling: Principles and Application*. Proceedings of the IEE Colloquium on Intelligent Planning and Scheduling Solutions, London, United Kingdom, 1996.

Claude Le Pape and Philippe Baptiste [1996]. *Constraint Propagation Techniques for Disjunctive Scheduling: The Preemptive Case*. Proceedings of the Twelfth European Conference on Artificial Intelligence, Budapest, Hungary, 1996.

Claude Le Pape and Philippe Baptiste [1997a]. *A Constraint Programming Library for Preemptive and Non-Preemptive Scheduling*. Proceedings of the Third International Conference and Exhibition on the Practical Application of Constraint Technology, London, United Kingdom, 1997.

Claude Le Pape and Philippe Baptiste [1997b]. *An Experimental Comparison of Constraint-Based Algorithms for the Preemptive Job-Shop Scheduling Problem*. Proceedings of the CP Workshop on Industrial Constraint-Directed Scheduling, CP, Schloss Hagenberg, Austria, 1997.

Claude Le Pape and Philippe Baptiste [1998a]. *Resource Constraints for Preemptive Job-Shop Scheduling*. Constraints, to appear.

Claude Le Pape and Philippe Baptiste [1998b]. *Heuristic Control of a Constraint-Based Algorithm for the Preemptive Job-Shop Scheduling Problem*. Journal of Heuristics, submitted.

Claude Le Pape and Philippe Baptiste [1998c]. *Constraint-Based Scheduling: A Theoretical Comparison of Resource Constraint Propagation Rules*. Proceedings of the ECAI98 Workshop on Non Binary Constraints, 1998.

Marie-Luce Lévy [1996]. *Méthodes par décomposition temporelle et problèmes d’ordonnancement*. Thèse de l’Institut National Polytechnique de Toulouse, 1996.

Olivier Lhomme [1993]. *Consistency Techniques for Numeric CSPs*. Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, Chambéry, France, 1993.

Hendrik C. R. Lock [1996]. *An Implementation of the Cumulative Constraint*. Working Paper, University of Karlsruhe, 1996.

Pierre Lopez [1991]. *Approche énergétique pour l'ordonnancement de tâches sous contraintes de temps et de ressources*. Thèse de l'Université Paul Sabatier, 1991.

Pierre Lopez, Jacques Erschler and Patrick Esquirol [1992]. *Ordonnancement de tâches sous contraintes : une approche énergétique*. RAIRO Automatique, Productique, Informatique Industrielle, 26(6):453-481, 1992.

Alan K. Mackworth [1977]. *Consistency in Networks of Relations*. Artificial Intelligence, 8:99-118, 1977.

Paul D. Martin and David B. Shmoys [1996]. *A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem*. Proceedings of the Fifth Conference on Integer Programming and Combinatorial Optimization, Vancouver, British Columbia, 1996.

A. A. Mastor [1970]. *An Experimental Investigation and Comparative Evaluation of Production Line Balancing Techniques*. Management Science 16(11):728-746, 1970.

Roger Mohr and Thomas C. Henderson, [1986]. *Arc and Path Consistency Revisited*. Artificial Intelligence, 28:225-233, 1986.

Ugo Montanari [1974], Network of Constraints: Fundamental Properties and Applications to Picture Processing. Information Sciences, 7:95-132, 1974.

J. Michael Moore [1968]. *An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs*. Management Science, 15(1):102-109, 1968.

Emmanuel Néron, Philippe Baptiste, Jacques Carlier and Claude Le Pape, [1998]. *Global Operations for the Multi-Processor Flow-Shop*. Proceedings of the 6th international workshop on project management and scheduling, 1998.

W. P. M. Nuijten, E. H. L. Aarts, D. A. A. Van Erp Taalman Kip and K. M. Van Hee [1993]. *Randomized Constraint Satisfaction for Job-Shop Scheduling*. Proceedings of the AAAI-SIGMAN Workshop on Knowledge-Based Production Planning, Scheduling and Control, IJCAI, Chambéry, France, 1993.

W. P. M. Nuijten [1994]. *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach*. PhD Thesis, Eindhoven University of Technology, 1994.

W. P. M. Nuijten and E. H. L. Aarts [1996]. *A Computational Study of Constraint Satisfaction for Multiple Capacitated Job-Shop Scheduling*. European Journal of Operational Research, 90(2):269-284, 1996.

Wim Nuijten and Claude Le Pape [1998]. *Constraint-Based Job-Shop Scheduling with ILOG SCHEDULER*. Journal of Heuristics, 3(4):271-286, 1998.

James H. Patterson [1984]. *A Comparison of Exact Approaches for Solving the Multiple Constrained Resource Project Scheduling Problem*. Management Science, 30(7):854-867, 1984.

Mike Pegman, Nigel Forward, Brett King and Dave Teal [1997]. *Mine Planning and Scheduling at RTZ Technical Services*. Proceedings of the Third International Conference and Exhibition on the Practical Application of Constraint Technology, London, United Kingdom, 1997.

Laurent Péridy [1996]. *Le problème de job-shop : arbitrages et ajustements*. Thèse de l'Université de Technologie de Compiègne, 1996.

Laurent Péridy, Philippe Baptiste and Eric Pinson, [1998]. *Branch and Bound Method for the Problem* $1 \mid r_i \mid \Sigma U_i$. Proceedings of the 6th international workshop on project management and scheduling, 1998.

Michael Perregaard [1995]. *Branch and Bound Methods for the Multi-Processor Job-Shop and Flow-Shop Scheduling Problems*. MSc Thesis, University of Copenhagen, 1995.

Eric Pinson [1988]. *Le problème de job-shop*. Thèse de l'Université Paris VI, 1988.

Marie-Claude Portmann, Antony Vignier, David Dardilhac and David Dezalay [1997]. *Branch and Bound Crossed with G.A. to Solve Hybrid Flowshop*. European Journal of Operational Research, to appear, 1997.

Patrick Prosser, [1993]. Hybrid Algorithms for the Constraint Satisfaction Problem, Computational intelligence 9(3):268-299, 1993.

Jean-François Puget [1994]. *A C++ Implementation of CLP*. Technical Report, ILOG S.A., 1994.

Jean-François Puget and Michel Leconte [1995]. *Beyond the Glass Box: Constraints as Objects*. Proceedings of the Twelfth International Symposium on Logic Programming, Portland, Oregon, 1995.

Jean-Charles Régin [1994]. *A Filtering Algorithm for Constraints of Difference in CSPs*. Proceedings of the Twelfth National Conference on Artificial Intelligence, Seattle, Washington, 1994.

Jean-Charles Régin [1995]. *Développement d'outils algorithmiques pour l'intelligence artificielle. Application à la chimie organique*. Thèse de l'Université Montpellier II, 1995.

Jean-Charles Régin [1996]. *Generalized Arc-Consistency for Global Cardinality Constraint*. Proceedings of the Thirteenth National Conference on Artificial Intelligence, Portland, Oregon, 1996.

Jean-Charles Régin and Jean-François Puget [1997]. *A Filtering Algorithm for Global Sequencing Constraints*. Proceedings of the Third International Conference on Principles and Practice of Constraint Programming, Schloss Hagenberg, Austria, 1997.

Thomas Schiex, Hélène Fargier and Gérard Verfaillie [1995]. *Valued Constraint Satisfaction Problems: Hard and Easy Problems*. Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, Montréal, Québec, 1995.

Stephen F. Smith [1994]. *OPIS: A Methodology and Architecture for Reactive Scheduling*. In: Monte Zweben and Mark Fox (editors), "Intelligent Scheduling," Morgan Kaufmann, 1994.

Stephen F. Smith and Cheng-Chung Cheng [1993]. *Slack-Based Heuristics for Constraint Satisfaction Scheduling*. Proceedings of the Eleventh National Conference on Artificial Intelligence, Washington, District of Columbia, 1993.

Pascal Van Hentenryck, Yves Deville and C. M. Teng [1992]. *A General Arc-Consistency Algorithm and its Specializations*. Artificial Intelligence, 57(3):291-321, 1992.

Christophe Varnier, Pierre Baptiste and Bruno Legeard [1993]. *Le traitement des contraintes disjonctives dans un problème d'ordonnancement : exemple du Hoist Scheduling Problem*. Deuxièmes journées francophones de programmation logique, Nîmes et Avignon, France, 1993.

Antony Vignier, [1997]. *Contribution à la résolution des problèmes d'ordonnancement de type monogamme, multimachine* ("*Flow-Shop hybride*"). Thèse de l'université de Tours, 1997.

Monte Zweben, Eugene Davis, Brian Daun and Michael J. Deale [1993]. *Scheduling and Rescheduling with Iterative Repair*. IEEE Transactions on Systems, Man, and Cybernetics, 23(6):1588-1596, 1993.

Appendix 1. Summary of notations

Given an integer constrained variable x,

- $d(x)$ denotes the domain of x,
- $lb(x)$ denotes the minimal value in $d(x)$,
- $ub(x)$ denotes the maximal value in $d(x)$.

Given an activity A_i,

- $start(A_i)$ denotes the constrained variable that represents the start time of A_i,
- $end(A_i)$ denotes the constrained variable that represents the end time of A_i,
- $processingTime(A_i)$ denotes the constrained variable that represents the processing time of activity A_i,
- $W(A_i, t)$ is an implicit 0-1 constrained variable representing fact that A_i executes at time t (when A_i cannot be interrupted, $W(A_i, t) = (start(A_i) \leq t) \wedge (t < end(A_i))$),
- $set(A_i)$ is a set variable that represents the set of time points at which A_i executes,
- $in(A_i)$ is a binary constrained variable representing the fact that A_i is performed on the resource or not,
- r_i denotes the release date of activity A_i, *i.e.*, $r_i = lb(start(A_i))$,
- d_i denotes the deadline of activity A_i, *i.e.*, $d_i = ub(end(A_i))$,
- $lst_i = ub(start(A_i))$, denotes the upper bound of the start time variable, *i.e.*, the latest start time of A_i,
- $eet_i = lb(end(A_i))$, denotes the lower bound of the end time variable, *i.e.*, the earliest end time of A_i,
- p_i denotes the processing time of the activity A_i, *i.e.*, $p_i = lb(processingTime(A_i))$ (in general, $processingTime(A_i)$ is bound, and thus p_i is exactly equal to $processingTime(A_i)$),
- w_i denotes the weight that is associated to the activity A_i.

Given a set of activities Ω,

- r_Ω denotes the smallest release date among release dates in Ω,
- d_Ω denotes the largest deadline among deadlines in Ω,
- p_Ω denotes the sum of the processing times of activities in Ω.

Given a resource R,

- *capacity*(*R*) denotes the constrained variable that represents the capacity of *R*, *i.e.*, the number of parallel identical machines that are available in *R*,
- *capacity*(*R*, *t*) denotes the constrained variable that represents the capacity of the resource *R* available at time *t* (such variables are useful to model a variable profile of a resource),
- C_R denotes the maximal capacity available, *i.e.*, $C_R = ub(capacity(R))$,
- *reject*(*R*) denotes the number of activities of the resource that can be sub-contracted.

Given an activity A_i and a resource *R*,

- *capacity*(A_i, *R*) denotes the constrained variable that represents the amount of resource *R* required by activity A_i.
- $c_{i,R}$ denotes the minimal amount of the capacity of the resource required by the activity, *i.e.*, $c_{i,R} = lb(capacity(A_i, R))$,
- *in*(A_i, *R*) denotes the binary variable that states whether the activity A_i is performed on the resource or not.

When a single resource is considered, the name of the resource *R* is omitted, *i.e.*, $capacity(R, t) = capacity(t)$, $C = C_R$, $capacity(A_i, R) = capacity(A_i)$, and $c_i = c_{i,R}$.

Appendix 2. Minimizing the weighted number of late jobs to be preemptively scheduled on a single machine, when processing times are equal

Given a set of weighted jobs, with release dates and due dates, the problem of minimizing the weighted number of late jobs that can be preemptively scheduled on a single machine is NP-hard. However, it is shown in [Lawler, 1990] that this problem, denoted as the ($1 \mid r_j, pmtn \mid \Sigma w_j U_j$), is solvable in $O(n^3W^2)$, where W denotes the sum of the weights. This result leaves a question open: Is there a polynomial time algorithm when all processing times are equal ($\forall i$, $p_i = p$). We answer affirmatively and we provide an $O(n^{10})$ dynamic programming algorithm[14].

From now on, we suppose that jobs are sorted in increasing order of due dates. We first introduce Jackson Preemptive Schedule and some notation. Afterwards, we provide the proposition that is the basis of our programming algorithm.

Definition.
Let $\Theta = \{t \text{ such that } \exists\, r_i, \exists\, l \in \{0, \dots, n\} \mid t = r_i + l * p\}$. □

The time points in Θ play a particular role in the structure of optimum schedules. In the following, we note $\Theta = \{t_1, t_2, \dots, t_q\}$ the ordered set of distinct time-points in Θ. Recall that $q \leq n^2$.

Definition.
The Jackson Preemptive Schedule (JPS) of a set of jobs O is the preemptive schedule obtained by applying the Earliest Due Date priority dispatching rule: whenever the machine is free and one job in O is available, schedule the job $J_i \in O$ for which d_i is the smallest. If a job J_j becomes available while J_i is in process, stop J_i and start J_j if $d_j < d_i$, otherwise continue J_i. □

[14] The result presented in this Appendix come from [Baptiste, 1998b].

Jackson Preemptive Schedule has several interesting properties (*e.g.*, [Carlier, 1984]). In particular, if a job is scheduled on JPS after its due date, there is no feasible preemptive schedule of the set of jobs. Hence, searching for a schedule on which the weighted number of late jobs is minimal, reduces to finding a set whose weight is maximal and that is feasible, *i.e.*, whose JPS is feasible.

Proposition Ap2-1.
For any subset of jobs S, the start and end times of the jobs on the JPS of S belong to the set Θ.

Proof (sketch).
We first prove that the end time of a job on the JPS of S belongs to Θ. Let J_k be any job and let s and e be respectively its start and end times on JPS. Let t be the minimal time point such that between t and s JPS is never idle. Because of the structure of JPS, t is a release date, say r_x. The jobs that execute (even partially) between s and e execute neither before s nor after e (because Jackson Preemptive schedule is based upon the EDD rule). Thus $e - s$ is a multiple of p. Two cases can occur:

- Either J_k causes an interruption and hence $s = r_k$.
- Or J_k does not cause any interruption and hence the jobs that execute between r_x and s, are fully scheduled in this interval. Consequently, $s - t$ is a multiple of p.

In both cases, there is a release date r_y (either r_k or r_x) such that between r_y and e, JPS is never idle and such that e is equal to r_y modulo p. On top of that, the distance between r_y and e is not greater than $n * p$ (because JPS is not idle). Hence, $e \in \Theta$.
Now consider the start time of any job on JPS. This time point is either the release date of the job or is equal to the end time of the “previous” one. Thus, start times also belong to Θ. $\square$

Definition.
For any time points t_u, t_v in Θ with $u<v$ and any for integer value k such that $1 \leq k \leq n$,

- let $U_k(t_u, t_v) = \{J_i \mid i \leq k \text{ and } t_u \leq r_i < t_v\}$ (as for the non-preemptive case),
- for any m such that $1 \leq m \leq n$, let $W_k(t_u, t_v, m)$ be the weight of the subset $S \subseteq U_k(t_u, t_v)$ of m jobs such that, (1) the JPS of S is feasible and ends before t_v and (2) its weight is maximal. If there is no such subset, $W_k(t_u, t_v, m)$ is arbitrarily set to $-\infty$. $\square$

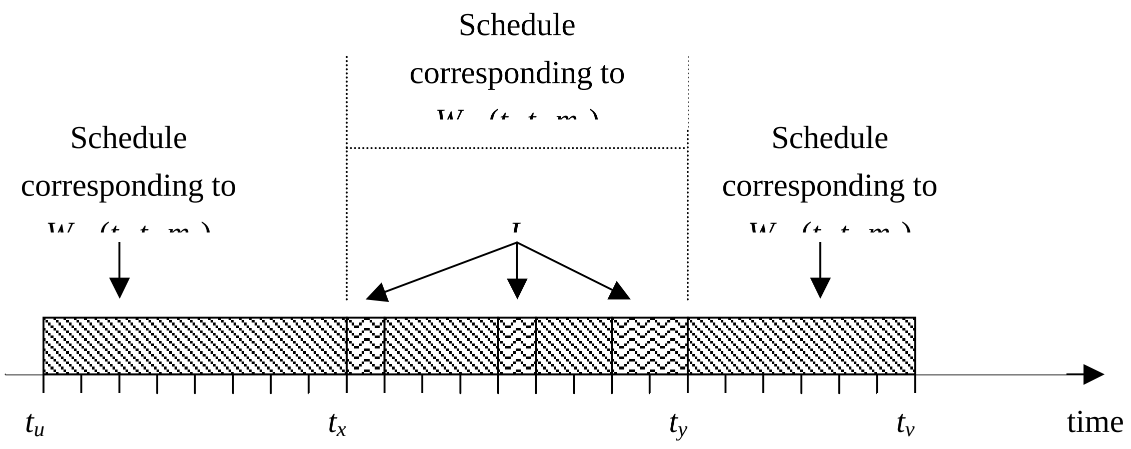


Figure Ap2-1. Illustration of Proposition Ap2-2

Proposition Ap2-2. (cf., Figure Ap2-1)

For any time points t_u, t_v in Θ with $u < v$ and any integer values k and m such that $1 < k \le n$ and $1 \le m \le n$, $W_k(t_u, t_v, m)$ is equal to $W_{k\text{-}1}(t_u,t_v,m)$ if $r_k \notin [t_u,t_v)$ and to the expression above otherwise:

$$\max\Big(W_{k\text{-}1}(t_u,t_v,m),\ \max_{\substack{t_x, t_y \in \Theta,\\ \max(r_k, t_u) \le t_x < t_y \le \min(d_k, t_v)\\ m_1+m_2+m_3=m-1,\\ p^*(m_2+1)=t_y-t_x}} (W_{k\text{-}1}(t_u,t_x,m_1)+W_{k\text{-}1}(t_x,t_y,m_2)+W_{k\text{-}1}(t_y,t_v,m_3))+w_k\Big)$$

Proof.

Let W' be the expression above. It is obvious that if $r_k \notin [t_u,t_v)$, $W_k(t_u, t_v, m)$ is equal to $W_{k\text{-}1}(t_u,t_v,m)$. In the following, we suppose that $r_k \in [t_u,t_v)$.

We first prove that $W' \le W_k(t_u,t_v,m)$.

- Consider the case where $W'=W_{k\text{-}1}(t_u,t_v,m)$. Since $U_{k\text{-}1}(t_u,t_v) \subseteq U_k(t_u,t_v)$, we have $W' \le W_k(t_u,t_v,m)$.
- Consider now the case where there exist $t_x \in \Theta, t_y \in \Theta$ and 3 integers m_1, m_2, m_3 such that
    - $\max(r_k,t_u) \le t_x < t_y \le \min(d_k,t_v)$,
    - $m_1+m_2+m_3=m\text{-}1$
    - $p^*(m_2+1)=t_y-t_x$,
    - $W'=W_{k\text{-}1}(t_u,t_x,m_1)+W_{k\text{-}1}(t_x,t_y,m_2)+W_{k\text{-}1}(t_y,t_v,m_3))+w_k$.

  Obviously, the subsets $U_{k\text{-}1}(t_u,t_x)$, $U_{k\text{-}1}(t_x,t_y)$ and $U_{k\text{-}1}(t_y,t_v)$ do not intersect. Thus, the JPS schedules of the subsets that realize $W_{k\text{-}1}(t_u,t_x,m_1)$, $W_{k\text{-}1}(t_x,t_y,m_2)$ and $W_{k\text{-}1}(t_y,t_v,m_3)$, put one after another define a valid overall schedule of a set of $m-1$ jobs taken in $U_{k\text{-}1}(t_u,t_v)$. Moreover, between t_x and t_y there is enough space to schedule J_k since m_2

jobs in $U_{k\text{-}1}(t_x,t_y)$ are scheduled and since $p*(m_2+1)=t_y-t_x$. As a consequence, we have $W'\leq W_k(t_u,t_v,m)$.

We now prove that $W_k(t_u,t_v,m)\leq W'$.

We only consider the case where $W_k(t_u,t_v,m)$ is finite otherwise the result holds. Consider a set S that realizes $W_k(t_u,t_v,m)$. If J_k does not belong to S then $W_k(t_u,t_v,m)=W_{k\text{-}1}(t_u,t_v,m)\leq W'$. Suppose now that $J_k\in S$. Let t_x and t_y be the start and end times of J_k on the JPS of S. Thanks to Proposition Ap2-1, $t_x\in\Theta$ and $t_y\in\Theta$. We also have $\max(r_k,t_u)\leq t_x<t_y\leq\min(d_k,t_v)$. Let S_1, S_2, S_3 be the partition of $S\text{-}\{J_k\}$ into the jobs that have a release date between t_u and t_x, between t_x and t_y and between t_y and t_v. Because of the structure of JPS (J_k is the job whose due date is maximal), all jobs in S_1 are scheduled before t_x. Moreover, all jobs in S_2 are scheduled after t_x and before t_y, and all jobs in S_3 are scheduled before t_v. On top of that, $p*(|S_2|+1)=t_y-t_x$ because J_k is also scheduled between t_x and t_y. Moreover, we have $|S_1|+|S_2|+|S_3|+1=m$. Finally the weight of S_1 is not greater than $W_{k\text{-}1}(t_u,t_x,|S_1|)$, the weight of S_2 is not greater than $W_{k\text{-}1}(t_x,t_y,|S_2|)$ and the weight of S_3 is not greater than $W_{k\text{-}1}(t_y,t_v,|S_3|)$. This leads to $W_k(t_u,t_v,m)\leq W'$. $\square$

Our dynamic programming algorithm relies on the above proposition. The values of $W_k(t_u,t_v,m)$ are stored in a multi-dimensional array of size $O(n^6)$ (n possible values for k, n^2 possible values for t_u, n^2 possible values for t_u, and n possible values for m).

- In the initialization phase the value of $W_1(t_u,t_v,m)$ is set to w_1 if m = 1 and if p is not greater than $\min(d_1,t_v)\text{-}\max(r_1,t_u)$ and to $-\infty$ otherwise.
- We then iterate from $k=2$ to $k=n$. Each time, W_k is computed for all the possible values of the parameters thanks to the formula of Proposition Ap2-2 and to the values of $W_{k\text{-}1}$ computed at the previous step.

The maximum weighted number of on-time jobs is equal to:

$$\max(W_n(\min(t_i),\max(t_i),1),\ W_n(\min(t_i),\max(t_i),2),\ ...,\ W_n(\min(t_i),\max(t_i),n)).$$

The overall complexity of the algorithm is $O(n^5)$ for the initialization phase. For each value of k, $O(n^5)$ values of W_k have to be computed. For each of them, a maximum among $O(n^4)$ terms has to be computed (for given values of t_x, m_1 and m_2, there is only one possible value for both t_y and m_3). This leads to an overall time complexity of $O(n^{10})$. A rough analysis of the space complexity leads to an $O(n^6)$ bound but since, at each step of the outer loop on k, one only needs the values of W computed at the previous step (k-1), the algorithm can be implemented with 2 arrays of $O(n^5)$ size (one for the current values of W and one for the previous value of W).

Appendix 3. Minimizing the weighted number of late jobs to be scheduled on a single machine, when processing times are equal

Since when a job is late, it can be scheduled arbitrarily late, the problem reduces to finding a set of jobs (1) that is feasible, *i.e.*, for which there exists a schedule that meets release dates and due dates and (2) whose weight is maximal. From now on, we suppose that jobs are sorted in increasing order of due dates. We first introduce some notation and then provide the proposition that is the basis of our dynamic programming algorithm[15].

Definition.
Let $\Theta = \{t \text{ such that } \exists\, r_i, \exists\, l \in \{0, \ldots , n\} \mid t = r_i + l * p\}$. ☐
Notice that there are at most n^2 values in Θ.

Proposition Ap3-1.
On any left-shifted schedule (*i.e.*, on any schedule on which jobs start either at their release date or immediately after another job), the starting times of jobs belong to Θ.

Proof.
Let J_k be any job. Let t be the largest time point before the start time of J_k at which the machine is idle. Since the schedule is left-shifted, t is a release date, say r_i. Between r_i and the starting time of J_k, l jobs execute ($0 \leq l \leq$ n). Hence the starting time of J_k belongs to Θ. ☐
Since any schedule can be left-shifted, Proposition Ap3-1 induces a simple dominance property: There is an optimal schedule on which jobs start at time points in Θ.

[15] The result presented in this Appendix come from [Baptiste, 1998b].

Definition.

- For any integer $k \leq n$, let $U_k(s,e)$ be the set of jobs whose index is lower than or equal to k and whose release date is in the interval $[s,e)$.
- Let $W_k(s,e)$ be the maximal weight of a subset of $U_k(s,e)$ such that there is a feasible schedule S of these jobs such that
  - S is idle before time $s+p$,
  - S is idle after time e,
  - starting times of jobs on S belong to Θ.

  Notice that if the subset of $U_k(s,e)$ is empty, $W_k(s,e)$ is equal to 0. □

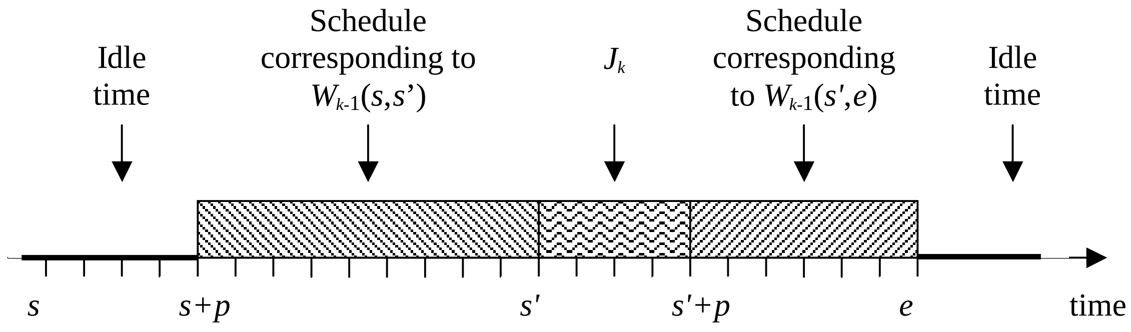

Idle time
Schedule corresponding to $W_{k-1}(s,s')$
J_k
Schedule corresponding to $W_{k-1}(s',e)$
Idle time
s
$s+p$
s'
$s'+p$
e
time

Figure Ap3-1. Illustration of Proposition Ap3-2

Proposition Ap3-2. (cf., Figure Ap3-1)

For any value of k in $[1, n]$ and for any values s, e with $s \leq e$, $W_k(s,e)$ is equal to $W_{k-1}(s,e)$ if $r_k \notin [s,e)$ and to the following expression otherwise:

$$\max\Big(W_{k-1}(s,e), \max_{\substack{s' \in \Theta \\ \max(r_k, s+p) \leq s' \leq \min(d_k, e)-p}} (w_k + W_{k-1}(s,s') + W_{k-1}(s',e))\Big).$$

Proof (sketch).

Let W' be the expression above. If $r_k \notin [s,e)$, the result obviously holds since $U_k(s,e) = U_{k-1}(s,e)$. We now consider the case where $r_k \in [s,e)$.

We first prove that $W' \leq W_k(s,e)$.

- If $W' = W_{k-1}(s,e)$ then we obviously have, $W' = W_{k-1}(s,e) \leq W_k(s,e)$.
- If there is a value s' in Θ such that $\max(r_k, s+p) \leq s' \leq \min(d_k, e)-p$ and such that $W' = w_k + W_{k-1}(s,s') + W_{k-1}(s',e)$. Let X and Y be two subsets that realize respectively $W_{k-1}(s,s')$ and $W_{k-1}(s',e)$. Because of the definition of W, the sets X and Y are disjoint and moreover, there exists a feasible schedule of X that fits in $[s+p,s']$ and there exists a feasible schedule of Y that fits in $[s'+p,e]$. Thus, $X \cup Y \cup \{J_k\}$ is a set whose weight is W' and there is a schedule of the jobs in this set that does not start before $s+p$ and that

ends before e (take the schedule of X, schedule J_k at time s' and add the schedule of Y). On top of that, starting times belong to Θ. Hence, $W' \leq W_k(s,e)$.

We now prove that $W_k(s,e) \leq W'$.

Let Z be the subset that realizes $W_k(s,e)$. If J_k does not belong to Z then $W_k(s, e) = W_{k-1}(s, e) \leq W'$. Now suppose that J_k belongs to Z. According to the definition of $W_k(s, e)$, there is a schedule S of Z that fits in $[s + p, e]$ on which starting times belong to Θ.

We claim that we can suppose that on S, the jobs executed after J_k are not available when J_k starts (*i.e.*, their release date is strictly greater than the start time of J_k). To justify our claim, we show how S can be modified to reach this property: Suppose that there is a job J_i that starts after J_k and that is available at the time where J_k starts. Let then $f(S)$ be the schedule obtained by exchanging the positions of J_i and J_k. Because $d_i \leq d_k$ and because processing times are equal, the resulting schedule is feasible. Notice that each time f is applied, the position of J_k strictly increases and that the idle time intervals of the resource remain the same. Thus, f can be applied a limited number of times only. The resulting schedule is feasible and the jobs executed after J_k are not available at the starting time of J_k. On top of that the overall set of starting times has not been modified.

Let us examine the partition induced by the starting time s' of J_k.

- The jobs scheduled before s', belong to $U_{k-1}(s, s')$ and their total weight is lower than or equal to $W_{k-1}(s, s')$.
- The jobs scheduled after J_k belong to $U_{k-1}(s', \mathrm{e})$ and their total weight is lower than or equal to $W_{k-1}(s', \mathrm{e})$.
- The weight of J_k is w_k.

Moreover s' is in Θ because it is a starting time and $\max(r_k, s + p) \leq s' \leq \min(d_k, e) - p$. Hence, the weight of the set Z is lower than or equal to W'. □

Given the dominance property induced by Proposition Ap3-1, the maximum weighted number of on-time jobs is $W_n(\min\Theta - p, \max\Theta)$. Thanks to Proposition Ap3-2, we have a straight dynamic programming algorithm to compute this value. The relevant values for s and e are exactly those in Θ (plus $\min\Theta - p$ for s). The values of $W_k(s, e)$ are stored in a multi-dimensional array of size $O(n^5)$ (n possible values for k, n^2 possible values for s and n^2 possible values for e). Our algorithm then works as follows.

- In the initialization phase, $W_0(s, e)$ is set to 0 for any values s, e ($s \leq e$) in Θ.
- We then iterate from $k = 1$ to $k = n$. Each time, W_k is computed for all the possible values of the parameters thanks to the formula of Proposition Ap3-2 and to the values of W_{k-1} computed at the previous step.

The initialization phase can be done in $O(n^4)$. Afterwards, for each value of k, $O(n^4)$ values of W_k have to be computed. For each of them, a maximum among $O(n^2)$ terms is computed. This leads to an overall time complexity of $O(n^7)$. A rough analysis of the space complexity leads to an $O(n^5)$ bound but since, at each step of the outer loop on k, one only needs the values of W computed at the previous step (k-1), the algorithm can be implemented with 2 arrays of $O(n^4)$ size (one for the current values of W and one for the previous values of W).