Ordonnancement cumulatif avec dépassements de capacité : Contrainte globale et décompositions

Alexis de Clercq

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

ECOLE DES MINES DE NANTES

Thèse de Doctorat

Alexis De Clercq

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École des mines de Nantes
sous le label de l'Université de Nantes Angers Le Mans

Discipline : Informatique
Spécialité : Programmation par contraintes
Laboratoire : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 29 octobre 2012

École doctorale : 503 (STIM)
Thèse n°: 2012EMNA0057

Ordonnancement cumulatif avec dépassements de capacité

Contrainte globale et décompositions

JURY

Rapporteurs :	**M. Patrice BOIZUMAULT**, Professeur, Université de Caen Basse-Normandie **M. Pierre LOPEZ**, Directeur de recherche, LAAS - CNRS, Toulouse
Examinateur :	**M. Gilles TROMBETTONI**, Professeur, Université de Montpellier
Directeur de thèse :	**M. Narendra JUSSIEN**, Professeur, École des Mines de Nantes
Co-directeur de thèse :	**M. Nicolas BELDICEANU**, Professeur, École des Mines de Nantes
Encadrant scientifique :	**M. Thierry PETIT**, Maître-assistant, École des Mines de Nantes

Remerciements

Je ne saurais commencer ce manuscrit sans remercier ceux qui ont, de manière plus ou moins directe, contribué à la réussite de ces trois années de travail.

En premier lieu, je tiens à remercier tout particulièrement mon encadrant scientifique Thierry Petit, maître-assistant à l'École des Mines de Nantes, qui m'a encadré pendant ma thèse. Pour sa patience infinie, sa gentillesse, son écoute, ses conseils toujours pertinents, qu'il reçoive toute ma gratitude. Pendant ces trois années, il a su se mettre à mon niveau, en m'expliquant de nombreux concepts qui allaient parfois bien au-delà du simple cadre du sujet de ma thèse. Je tiens donc à lui exprimer toute ma reconnaissance et saluer ses grandes compétences scientifiques, pédagogiques et ses qualités personnelles.

Je voudrais également remercier mon directeur de thèse Narendra Jussien, directeur du département informatique de l'École des Mines de Nantes, qui par ses conseils, son appui et sa confiance, m'aura permis de mener au bout cette thèse.

Je remercie mon co-directeur Nicolas Beldiceanu, professeur à l'École des Mines de Nantes pour ses commentaires et conseils toujours justes, et pour sa grande connaissance du domaine.

Merci beaucoup également à Patrice Boizumault et Pierre Lopez de m'avoir fait l'honneur d'être rapporteurs de ce manuscrit de thèse. Les remarques et appréciations pertinentes qu'ils ont émis dans ces rapports m'ont été très utiles.
Merci beaucoup à Gilles Trombettoni d'avoir accepté de faire partie de mon jury de soutenance.

Je remercie Emmanuel Hébrard, Helmut Simonis, Meinolf Sellman, James Little, Nic Wilson, Jean-Charles Régin, Claude-Guy Quimper, Gilles Pesant qui m'ont fait partager leur expérience de la recherche lors de conversations plus ou moins formelles dans les diverses conférences auxquelles j'ai pu assister.

Tous mes remerciements vont également à l'ensemble des membres de l'équipe TASC qui m'ont accueilli chaleureusement et qui ont toujours été disponibles pour des conseils scientifiques, pour m'aider dans ma tâche d'enseignement. Je tiens également à les remercier chaleureusement pour la bonne humeur qu'ils ont su mettre à tout moment, pour les repas et pauses café très agréables agrémentées de discussions sérieuses... ou pas.

Je remercie tout particulièrement Arnaud qui a partagé le même bureau que moi pendant ces deux dernières années, et qui s'est montré disponible, patient, à l'écoute, et toujours prêt à discuter, de sciences, ou de toute autre chose. Merci à lui également d'avoir supporté mes fredonnements plus ou moins volontaires, d'avoir partagé des parties de basket-panier endiablées et de m'avoir supporté, plus généralement. Merci à Alban qui va me prendre mon bureau, mais qui m'a montré qu'il en serait digne.

Un grand merci également à Marie qui m'a accompagné dans mes expéditions en conférence et post-conférences. Je la remercie pour ses discussions sans fin sur la musique, et sur des tas d'autres sujets. Merci pour sa gentillesse, et bon courage pour la soutenance à venir.

Je remercie Jean-Guillaume pour ses idées fourmillantes, sa bonne compagnie et son humour.

Je tiens à remercier Aurélien pour m'avoir permis de prendre du recul au moment où j'en avais besoin et pour sa bonne humeur.

Je remercie les doctorants des autres équipes de l'École des Mines de Nantes, en particulier Flavien, Renaud, Carlos, Jérémie, Tanguy, Clément, Ludivine, Lama, pour les pauses, les verres partagés et les discussions.

Enfin, je tiens à remercier l'ensemble du département informatique de l'École des Mines de Nantes, ainsi que la direction des études, et le personnel du LINA. Un merci tout particulier à Catherine, Cécile et Diana, pour leur sympathie, leur sourire et leur disponibilité.

Ce manuscrit n'aurait sans doute pas pu voir le jour si je n'avais pas reçu le soutien et l'affection permanents de mes parents, Claudie et Fabien, de ma soeur Constance et de mon frère Raphaël. Votre présence constante malgré la distance est toujours aussi précieuse, et vous me donnez la force de me dépasser. J'en suis arrivé jusqu'ici grâce à vous et je sais que vous continuerez à me soutenir quoiqu'il advienne. Merci pour tout cela. Je remercie également mes grands-parents Éliane et Roland, eux aussi toujours présents, mais également Paulette et Lucien, partis trop tôt mais à qui je tenais à dédier ces trois années de travail.

Toute ma gratitude et mon affection vont également à Emilie, Clotilde, Kristo, Benoît, Maxime et l'autre Benoît, sans qui cette vie à Nantes n'aurait pas été aussi agréable, et sans qui je n'aurais pu aller au bout de ce travail. Pour leur soutien constant, les soirées interminables à discuter, à jouer, leur humour inépuisable, et l'affection énorme qu'ils m'inspirent, je les remercie énormément. Vous allez beaucoup me manquer.

Je veux remercier énormément Julien, Loïc et Ronan, pour leur amitié sans faille depuis nos études d'ingénieur. Les soirées à l'ancienne resteront dans les annales, et d'autres suivront. Recevez mon affection et mon amitié sincère, vous me manquerez dans la capitale.

Un grand merci à Clarisse, pour son soutien constant, pour les soirées passées à établir des théories et à refaire le monde autour d'une cervoise.

Merci à Aude pour les nombreuses pauses café-discussion entre deux paragraphes de manuscrit.

Merci, enfin, à tous mes amis (Camille, Christy, Sébastien, Benjamin, Philippe, Elsa, Faustine, Geoffroy, Julie, Fanny, Céline, Olivier, Stéphanie, Gilles, Pauline, David, Pascal, et tous ceux que j'oublie certainement), pour leur présence et pour les bons moments passés en leur compagnie.

Je vous remercie enfin, vous, cher lecteur, pour l'intérêt que vous portez à ma contribution. En espérant que la lecture de ce manuscrit pourra vous apporter ce que vous y cherchez.

Table des matières

Introduction

Objet et enjeux

Située au carrefour entre la recherche opérationnelle et l'intelligence artificielle, la programmation par contraintes est un paradigme permettant de modéliser et de résoudre de nombreux types de problèmes. Un réseau de contraintes est défini par un ensemble de variables, un ensemble de domaines, et un ensemble de contraintes. Un domaine distinct est associé à chaque variable. Il est constitué des valeurs qui peuvent être affectées à cette variable. Lorsque le domaine d'une variable est réduit à une valeur, on dit que la variable est instanciée. Une contrainte est définie sur un sous-ensemble de variables. Elle exprime une propriété qu'elles doivent satisfaire. Une solution est une instanciation des variables qui satisfait la conjonction de toutes les contraintes.

Parmi les problèmes classiquement étudiés en programmation par contraintes, nous pouvons citer les problèmes de configuration, la gestion d'emplois du temps, l'allocation de ressources, par exemple.

Dans cette thèse, nous nous intéressons à l'ordonnancement d'activités sous contraintes de ressources pouvant éventuellement être relâchées. La programmation par contraintes a prouvé qu'elle était une approche intéressante pour résoudre ce type de problèmes [18, 53, 19]. Baptiste et al [4] présentent plusieurs variantes de ces problèmes, dans le contexte de la programmation par contraintes.

Le problème cumulatif est une de ces variantes. Dans ce problème, chaque activité est définie par des variables de début et de fin et une durée, et consomme pour s'exécuter une certaine quantité de ressource. Nous nous plaçons dans le cas où les activités ne peuvent pas être interrompues. La ressource disponible à chaque instant est limitée. Cette limite s'appelle la *capacité*. Le but est de trouver un ordonnancement des activités qui respecte la capacité, c'est à dire tel qu'à n'importe quel instant la somme des consommations des activités qui s'exécutent ne dépasse pas la capacité. Généralement, on cherche également à minimiser la date de fin de l'ordonnancement, correspondant à la fin de la dernière activité exécutée.

De nombreuses situations pratiques correspondent à des problèmes d'ordonnancement cumulatif. Les problèmes d'ordonnancement de production, où la ressource peut être humaine ou matérielle, en sont un exemple intéressant. Les activités correspondent par exemple à la production, l'usinage, ou la transformation d'un produit donné.

Undes points forts de la programmation par contraintes est de pouvoir utiliser des *contraintes globales* qui peuvent représenter un sous-problème récurrent dans plusieurs domaines d'application, et qui exploitent la structure de ce sous-problème pour mieux le résoudre.

La contrainte globale Cumulative, introduite par Aggoun et Beldiceanu [1] permet de modéliser le problème cumulatif en programmation par contraintes. De nombreuses techniques de filtrage ont été dé-

finies. Deux des techniques les plus efficaces de l'état de l'art sont l'algorithme de Sweep [7, 8, 76, 20] qui est un raisonnement sur la topologie de l'ordonnancement et les algorithmes d'Edge-Finding [19, 59, 4, 89, 4, 90, 43] qui raisonnent sur *l'énergie* des activités, c'est à dire le produit de leur durée (longueur) et de leur consommation instantanée de ressource (hauteur). Le principe de l'Edge-Finding est de trouver des relations de précédence implicites entre activités, et d'en déduire des informations sur les dates de début au plus tôt et de fin au plus tard de ces activités.

Des approches par décompositions de la contrainte Cumulative ont également été proposées [1, 26, 77]. Une décomposition d'une contrainte est un réseau de contraintes, qui sont souvent plus primitives. Ce réseau représente sémantiquement une contrainte. L'intérêt principal est un gain de généricité : éviter l'implémentation et le maintien dans le temps d'algorithmes ad-hoc complexes, notamment lorsque la contrainte que l'on décompose n'existe pas dans la plupart des outils à base de contraintes. Il peut exister d'autres avantages, par exemple une plus grande robustesse à des stratégies de recherche génériques, ou encore, dans des cas assez rares, une meilleure efficacité de résolution pour certaines classes d'instances du problème [77].

En pratique, la date de fin d'un projet est souvent fixée, ou bornée par une date butoir que l'on ne peut retarder. Dans ce cas, il n'est pas toujours possible de trouver un ordonnancement des activités qui n'engendre aucun dépassement de la capacité en ressource.

On peut alors tolérer de relâcher la contrainte de capacité, dans une limite raisonnable pour obtenir une solution. Dans le contexte de l'ordonnancement de production, ceci revient à autoriser le dépassement de la capacité en ressources humaines ou matérielles de l'entreprise. La location de machines supplémentaires pour une période donnée, les heures supplémentaires effectuées par les employés, ou encore l'emploi d'intérimaires sont des solutions pratiques possibles pour absorber la surcharge ponctuelle engendrée.

La relaxation de cette contrainte de capacité nécessite également de respecter un certain nombre de contraintes métier, propres à chaque contexte applicatif. Par exemple, en ordonnancement de production, si des surcharges d'activité surviennent et que celles-ci sont absorbées par le personnel régulier de l'entreprise, par exemple avec des heures supplémentaires, alors il convient de répartir au mieux, dans le temps, ces surcharges.

En programmation par contraintes, différents cadres de relaxation ont été proposés.

Parmi ces cadres, nous trouvons les explications [42, 24, 41], certaines extensions de la programmation par contraintes utilisant des structures de valuation externes comme les CSP valués [75] ou les Semi-Ring CSP [14], et le *cadre des contraintes globales soft*, introduit par Petit [67, 62]. Dans ce dernier cadre, un problème sur-contraint est modélisé comme un problème classique de programmation par contraintes. Les variables de coût, modélisant les violations de contraintes, sont incluses dans le réseau de contraintes comme toute autre variable. Ces variables de coût ont une valeur 0 si les contraintes sont respectées, et non nulle sinon. La sémantique du problème détermine la façon dont sont calculées leur valeur dans le cas où une violation est constatée.

En nous plaçant dans ce cadre nous cherchons une sémantique de violation de contraintes qui soit la plus expressive possible. Si l'aspect modélisation de situations pratiques est très important, il est également primordial de fournir des algorithmes efficaces pour la résolution de problèmes.

Pour cela, l'introduction d'une contrainte globale, étendant la contrainte Cumulative, semble être une solution intéressante. Une contrainte globale peut à la fois encapsuler la définition du problème cumulatif relaxé, et embarquer des algorithmes de filtrage pour résoudre ce problème. Une autre approche possible est l'approche par décomposition.

Selon ces axes, nous présentons une thèse abordant la question de l'ordonnancement cumulatif sur-contraint, en proposant une contrainte globale, que nous comparons à des approches par décomposition.

Nous proposons une nouvelle contrainte globale dérivée de Cumulative pour gérer le cas sur-contraint. Nous nommons cette contrainte SoftCumulative.

Nous adaptons les algorithmes de Sweep et d'Edge-Finding au cas de l'ordonnancement cumulatif avec dépassements de capacité. Afin d'augmenter l'efficacité de notre contrainte globale, nous introduisons des méthodes de filtrage propres au cas avec dépassements. Nous proposons également une procédure de filtrage à partir des valeurs minimales dans les domaines des variables de coût modélisant les dépassements de capacité. L'idée d'imposer un minimum à une variable de coût est relativement contre-intuitive, car on cherche généralement à minimiser les coûts. Cependant, nous mettons en lumière un cas pratique où un tel filtrage est utile.

Nous présentons une stratégie de recherche adaptée à notre problème.

Nous proposons une adaptation des décompositions existantes de la contrainte Cumulative au cas cumulatif relaxé, et introduisons une nouvelle décomposition, pour la contrainte Cumulative et pour notre nouvelle contrainte SoftCumulative. Contrairement à l'existant, cette nouvelle décomposition n'engendre pas une explosion du nombre de variables : elle nécessite l'ajout d'un nombre de variables de l'ordre de $O(n)$, où n est le nombre d'activités du problème.

Notre approche sous forme de contrainte globale pour relaxer la contrainte Cumulative est nouvelle dans la formulation. A ce titre, il n'existe donc pas de jeux de données adaptés à ce problème. En conséquence, nous générons aléatoirement des problèmes pour valider notre approche.

Nous testons l'influence de différents paramètres sur la recherche d'une solution au problème cumulatif relaxé. Nous évaluons les performances de notre contrainte globale lorsque l'on cherche à minimiser les violations. Nous ajoutons des contraintes additionnelles sur les dépassements de capacité, et observons le comportement des algorithmes ad-hoc mis en place. Nous adaptons de plus une libraire de benchmarking bien connue pour les problèmes cumulatifs : la PSPLib [45], à notre relaxation et nous observons son comportement. Enfin, nous comparons notre approche aux décompositions proposées, afin de valider l'utilité pratique de notre contrainte globale.

Nous concluons ce manuscrit en évoquant quelques pistes intéressantes pour étendre notre travail.

Plan de la thèse

Etat de l'art

Chapitre 1 : Introduction à la programmation par contraintes

Après un rappel succinct des notions générales de complexité, nous présentons le paradigme de la programmation par contraintes dans le chapitre 1. Nous abordons dans un premier temps les réseaux de contraintes. Puis nous évoquons quelques algorithmes de base dédiés à la recherche de solution, optimale ou non. Nous définissons ensuite les notions de filtrage et de propagation et les stratégies de recherche de solution. Enfin, nous introduisons les contraintes globales.

Chapitre 2 : Ordonnancement cumulatif en programmation par contraintes

Dans le chapitre 2 nous présentons le problème cumulatif, après avoir introduit les différentes notions indispensables à sa compréhension : les activités, l'énergie, la capacité. Puis nous décrivons la contrainte globale Cumulative qui représente ce problème en programmation par contraintes. Depuis son introduction, cette contrainte a fait l'objet de nombreux travaux, et différents filtrages ont été introduits. Nous nous concentrons en particulier sur les algorithmes de Sweep [7, 8] et d'Edge-Finding de Vilím [90] et de Kameugne et al. [43], et évoquons d'autres algorithmes de l'état de l'art.

Chapitre 3 : Les problèmes sur-contraints

En programmation par contraintes, différents cadres existent pour modéliser et résoudre des problèmes sur-contraints. Dans le chapitre 3 nous évoquons dans un premier temps le cadre des explications [42, 24, 41]. Nous présentons ensuite les problèmes d'optimisation sur-contraints, dans lesquels nous cherchons à minimiser le coût de violation des contraintes. Pour cela deux paradigmes sont utilisés en programmation par contraintes. Nous présentons dans un premier temps les extensions de la programmation par contraintes, par une structure de valuation externe [75, 14]. Puis nous décrivons le cadre des contraintes globales soft [67, 62]. Enfin, nous décrivons succinctement des algorithmes de résolution génériques, par opposition aux algorithmes dédiés que nous utiliserons dans la suite de cette thèse.

Contributions

Chapitre 4 : Problèmes d'ordonnancement cumulatif sur-contraints

Notre but est de fournir à l'utilisateur des solutions acceptables en pratique, quand la date de fin du projet, fixée, est une contrainte dure, et qu'aucune solution parfaite n'existe à cause de cette limite. La sémantique de violation doit être adaptée. Nous présentons notre contrainte globale cumulative relaxée, la SoftCumulative, dans le chapitre 4. Nous présentons ensuite des contraintes métier usuelles sur les variables modélisant les dépassements. Nous décrivons des modèles adaptés utilisant la contrainte globale SoftCumulative.

Chapitre 5 : Résolution de problèmes d'ordonnancement cumulatif sur-contraints

Nous présentons les méthodes de résolution utilisées par la contrainte SoftCumulative dans le chapitre 5. Nous décrivons en premier lieu notre adaptation des algorithmes de Sweep et d'Edge-Finding au cas de la contrainte globale SoftCumulative. Afin d'augmenter le filtrage de notre contrainte globale, nous introduisons des méthodes de filtrage propres au cas avec dépassements. Enfin, nous présentons notre stratégie de recherche pour SoftCumulative.

Chapitre 6 : Décompositions

Dans le chapitre 6, nous décrivons les décompositions existantes de la contrainte Cumulative. Nous les adaptons ensuite au problème modélisé par la contrainte SoftCumulative. Nous introduisons une nouvelle décomposition engendrant un nombre de variables supplémentaires de l'ordre du nombre d'activités du problème, pour Cumulative et SoftCumulative.

Chapitre 7 : Expérimentations

Le chapitre 7 présente les résultats de nos expérimentations. Nous réalisons différentes tests en satisfaction de contrainte, et en optimisation. Nous travaillons sur des jeux d'instances aléatoires, et adaptons

la PSPLib [45] au cas avec dépassements de capacité. Nous expérimentons ensuite les décompositions que nous avons introduites, et les comparons aux expérimentations avec le contrainte globale SoftCumulative.

Chapitre 8 : Bilan et perspectives

Nous présentons un bilan de cette thèse dans le chapitre 8, avant de conclure sur des perspectives intéressantes pour étendre nos travaux.

Première partie

État de l'art

Chapitre 1

Introduction à la programmation par contraintes

Sommaire

La programmation par contraintes (PPC) est un paradigme permettant de résoudre des problèmes difficiles.

Nous commençons par un rappel général des notions de complexité, qui seront utiles dans cette thèse, avant de décrire les principes de base de la programmation par contraintes.

1.1 Rappels de complexité

1.1.1 Complexité d'un algorithme

La complexité d'un algorithme caractérise la quantité de ressources qu'il utilise, en fonction de la taille des données. Nous considérons la complexité temporelle et la complexité spatiale :

- la complexité temporelle exprime un ordre de grandeur du nombre d'instructions élémentaires exécutées par l'algorithme,
- la complexité spatiale exprime un ordre de grandeur de la quantité de mémoire utilisée lors de l'exécution de l'algorithme.

Définition 1 (complexité dans le pire des cas) *Soit R une ressource. La fonction de complexité dans le pire des cas $T(n, R)$ d'un algorithme pour une donnée de taille n exprime la quantité maximale de la ressource R nécessaire à l'exécution de l'algorithme.*

La notation de Knuth sert à donner un ordre de grandeur de la complexité.

Notation 1 : Knuth [44]
On exprime une complexité avec la notation O, qui est définie par : $O(f) = \{g : \mathbb{N} \to \mathbb{N}, \exists n_0 \in \mathbb{N}$ *et* $\exists c > 0$ *tels que :* $\forall n > n_0, g(n) \leq c \times f(n)\}$.

1.1.2 Classes de complexité

Les classes de complexité permettent de hiérarchiser les algorithmes et les problèmes selon leur difficulté. Nous présentons ici les classes de complexité qui seront utiles dans cette thèse. Pour plus de détails, le lecteur intéressé pourra se référer aux ouvrages de Papadimitriou [60] et de Garey et Johnson [34]. Nous supposons dans cette section que la taille des données est finie.

On distingue les problèmes de décision, retournant une réponse "oui" ou "non", des problèmes de recherche de solution.

Problèmes de décision Certains problèmes de décision peuvent être résolus par un algorithme dont la complexité temporelle est une fonction polynomiale en la taille des données d'entrée. Cet algorithme est, dans ce cas, *polynomial*. Ces problèmes constituent la classe **P**.

On dit qu'un problème est *non-déterministe polynomial* si et seulement si il existe un *certificat polynomial* pour toute instance de celui-ci. Un certificat polynomial est une donnée permettant à un algorithme polynomial de vérifier que la réponse est "oui". De tels problèmes constituent la classe **NP**. Notons que **P** $\subseteq$ **NP**.

Un problème p de **NP** est dit **NP-complet** si n'importe quelle instance d'un problème p' de **NP** peut être convertie en une instance de p à l'aide d'un algorithme polynomial en temps et en espace, avec la même réponse. Cet algorithme est une *réduction polynomiale*.

Nous avons donc les relations suivantes entre ces trois classes : **P** $\subseteq$ **NP** et **NP-complet** $\subseteq$ **NP**.

Il s'en suit la conjecture **P** $\neq$ **NP**. Prouver celle-ci est une question ouverte en théorie de la complexité. Elle serait fausse si un algorithme polynomial permettant de résoudre un problème **NP-complet** était trouvé.

Enfin, un problème p est **NP-difficile** si n'importe quelle instance d'un problème p' de **NP** peut être convertie en une instance de p à l'aide d'un algorithme polynomial en temps et en espace, avec la même réponse. Notons que p n'est pas nécessairement dans **NP**. Ainsi, tous les problèmes **NP-complets** sont des problèmes **NP-difficiles**, mais la réciproque est fausse.

Problèmes de recherche d'une solution Les problèmes de recherche de solution peuvent être classifiés de manière similaire. Papadimitriou [60] a proposé la notation suivante pour distinguer ces problèmes des problèmes de décision, où **F** correspond au préfixe "functional" : **FP**, **FNP**, **FNP-complet**. Souvent, dans la littérature, cette distinction est ignorée : les notations des problèmes de décision sont utilisées pour les problèmes de recherche de solution. Dans cette thèse, par souci de simplicité, nous abandonnons donc le préfixe **F**.

Nous pouvons maintenant introduire le cadre de la PPC. Nous présentons dans un premier temps des généralités liées à ce paradigme.

1.2 Réseaux de contraintes

Un problème de programmation par contraintes est décrit par un *réseau de contraintes*. Un réseau de contraintes est défini par un ensemble de variables, un ensemble de domaines et un ensemble de contraintes. Un domaine distinct est associé à chaque variable. Il est constitué des valeurs qui peuvent être affectées à cette variable.

Définition 2 (Réseau de contraintes) *Un réseau de contraintes $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ est défini par :*
- *un ensemble fini de variables $\mathcal{X} = \{x_0, x_1, \ldots, x_{n-1}\}$,*
- *un ensemble de domaines $\mathcal{D} = \{D_0, D_1, \ldots, D_{n-1}\}$. Chaque domaine D_i tel que $i \in \{0, 1, \ldots, n-1\}$ définit l'ensemble fini des* valeurs *que peut prendre la variable x_i. Nous pouvons aussi noter ce domaine $D(x_i)$. Pour une variable x_i, on notera $\underline{x_i}$ la valeur minimale de son domaine, et $\overline{x_i}$ sa valeur maximale. On appelle ces valeurs les* bornes *de x_i,*
- *un ensemble de contraintes $\mathcal{C} = \{C_0, C_1, \ldots, C_{m-1}\}$. Chaque contrainte C_j telle que $j \in \{0, 1, \ldots, m-1\}$ est définie sur un sous-ensemble de variables $var(C_j) \subseteq \mathcal{X}$ par un sous-ensemble $rel(C_j)$ du produit cartésien des domaines des variables $var(C_j)$, qui spécifie les combinaisons autorisées de valeurs pour ces variables. On dit que ces combinaisons* satisfont *la contrainte C_j. La contrainte C_j est aussi dite* respectée. *$var(C_j)$ est appelée* portée *de la contrainte C_j. D_{C_j} désigne l'ensemble des domaines des variables de $var(C_j)$.*

Une variable x_i est entière si son domaine D_i ne contient que des entiers relatifs. Dans cette thèse nous nous limiterons à des problèmes discrets, où les variables sont entières.

Nous définissons à présent quelques notions essentielles sur les réseaux de contraintes.

Pour tout ensemble E d'éléments, nous notons $|E|$ le nombre d'éléments contenus dans cet ensemble. Ce nombre est aussi appelé *cardinal* de E.

Définition 3 (arité d'une contrainte) *L'*arité *d'une contrainte C, $|var(C)|$, est le nombre de variables sur lesquelles elle s'applique. C est dite* binaire *si elle a une arité de 2.*

Définition 4 (degré d'une variable) *Dans un réseau $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$, le* degré *d'une variable x appartenant à $\mathcal{X}$ est le nombre de contraintes de $\mathcal{C}$ qui s'appliquent sur cette variable.*

Définition 5 (Instanciation et affectation) *Une variable x de $\mathcal{X}$ est dite* instanciée avec une valeur *v lorsque l'on fixe x à la valeur v. On dit également que v est* affectée *à la variable x. L'instanciation de la variable x à v est alors notée $I[x] = v$. Elle est* valide *si $v \in D(x)$.*

Soit $X \subseteq \mathcal{X}$. Nous notons $I[X] = (v_0, v_1, \ldots v_{|X|-1})$ son instanciation aux valeurs $v_0, v_1, \ldots, v_{|X|-1}$. Si $I[X]$ est une instanciation valide, on l'appelle un tuple *de X. Lorsque toutes les variables d'un ensemble $\mathcal{X}$ sont instanciées, on dit que l'instanciation $I[\mathcal{X}]$ est* complète. *Sinon l'instanciation $I[\mathcal{X}]$ est* partielle.

Une contrainte est décrite en *extension* lorsque la relation qu'elle définit est donnée explicitement par la liste des tuples (combinaisons de valeurs pour les variables) autorisés, ou interdits. Elle peut également être décrite implicitement par des propriétés mathématiques, ou des fonctions booléennes. Elle est alors décrite en *intension*. L'exemple 1 montre une contrainte décrite en extension et en intension.

Exemple 1 *Soient deux variables x_0 et x_1 de domaine $D_0 = [0, 2]$ et $D_1 = [0, 2]$. Nous considérons la contrainte $C : x_0 < x_1$ (elle est ici décrite en intension). C est décrite, en extension, par $rel(C) = \{(0, 1), (0, 2), (1, 2)\}$.*

La notion de consistance est essentielle pour caractériser le processus de résolution d'un problème modélisé par un réseau de contraintes.

Définition 6 : consistance
Soient $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ un réseau de contrainte et x une variable de $\mathcal{X}$. Une valeur $v \in D(x)$ est consistante *avec une contrainte $C \in \mathcal{C}$ si et seulement si $x \notin var(C)$ ou bien s'il existe un tuple τ de $rel(C)$ valide tel que $v \in \tau$.*

Définition 7 : arc-consistance
Soit $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ un réseau de contraintes et x une variable de $\mathcal{X}$:
- *une valeur $v \in D(x)$ est arc-consistante si et seulement si $\forall C \in \mathcal{C}$, v est consistante avec C,*
- *un domaine $D(x)$ est arc-consistant si et seulement si $D(x) \neq \emptyset$ et toutes les valeurs de $D(x)$ sont arc-consistantes. Par abus de langage on dira aussi que x est une variable arc-consistante,*
- *$\mathcal{R}$ est* **arc-consistant** *si toutes les variables de $\mathcal{X}$ sont arc-consistantes.*

Définition 8 : consistance globale
Soit $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C} = \{C_0, C_1, \ldots, C_{m-1}\}\}$ un réseau de contraintes et x une variable de $\mathcal{X}$. Une valeur $v \in D(x)$ est **globalement consistante** *si et seulement si elle est consistante avec la conjonction $\gamma = C_0 \wedge C_1 \wedge \ldots \wedge C_{m-1}$.*

Nous distinguons les problèmes de satisfaction (CSP : Constraint Satisfaction Problem) et les problèmes d'optimisation (COP : Constraint Optimisation Problem) en programmation par contraintes.

Définition 9 (problème de satisfaction) *Soit un réseau de contraintes $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$. Le problème de satisfaction de contraintes qui lui est associé consiste à trouver une instanciation valide de $\mathcal{X}$, globalement consistante.*

L'exemple 2 présente un problème à une contrainte et deux variables en en donnant un tuple solution et un tuple ne respectant pas la contrainte.

Exemple 2 *Soient deux variables x_0 et x_1, telles que $D_0 = [1, 5]$ et $D_1 = [4, 6]$ et la contrainte $C : x_0 \leq x_1$. Une instanciation complète $I[x_0, x_1] = (3, 5)$ est* autorisée *car elle est valide ($3 \in D_0$ et $5 \in D_1$) et respecte la contrainte ($3 \leq 5$). En revanche, l'instanciation $I[x_0, x_1] = (5, 4)$ est* interdite*, car elle ne respecte pas la contrainte C ($5 \not\leq 4$), bien qu'étant valide ($5 \in D_0$ et $4 \in D_1$).*

Un réseau de contraintes binaires $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ peut être représenté par un graphe appelé *graphe des contraintes*. Pour des problèmes impliquant des contraintes autres que des contraintes binaires, nous utilisons une représentation sous forme d'*hypergraphe*. L'exemple 3 présente un graphe des contraintes simple.

Exemple 3 *La Figure 1.1 est un exemple de graphe des contraintes d'un réseau comportant* 5 *variables et* 6 *contraintes binaires.*

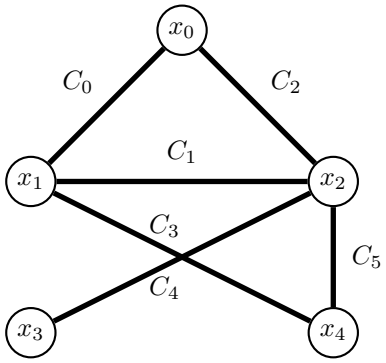


FIGURE 1.1 – Un réseau de contraintes binaires comportant 5 variables $\{x_0, x_1, \ldots, x_4\}$ et 6 contraintes binaires $\{C_0, C_1, \ldots, C_5\}$

L'exemple 4 présente un problème concret, et sa modélisation sous forme de problème de satisfaction de contraintes. Il donne une solution de ce problème.

Exemple 4 *Considérons un opérateur mobile souhaitant implanter des antennes relais dans une région divisée en plusieurs zones, situées autour des grandes villes. L'opérateur a à sa disposition plusieurs fréquences d'émission, mais, pour éviter les interférences, toutes les fréquences d'une même zone doivent être différentes. De plus, des relations de succession sont définies entre les zones : pour deux zones données* $Zone_i$ *et* $Zone_j$*, si* $Zone_i$ *est inférieure à* $Zone_j$ *alors la fréquence maximale émise dans la* $Zone_i$ *est strictement inférieure à la fréquence minimale émise dans* $Zone_j$*. Enfin, des règles métier imposent des contraintes de différence entre certaines antennes de zones différentes. Les données de l'opérateur sont les suivantes :*

Les fréquences disponibles sont des entiers appartenant à l'intervalle $[1, 25]$*. Nous numérotons les contraintes du problème. La région contient 5 zones.*

Le nombre d'antennes par zone est le suivant :

- C_0 *:* $Zone_0$ *:* 4 *antennes*
- C_1 *:* $Zone_1$ *:* 7 *antennes*
- C_2 *:* $Zone_2$ *:* 5 *antennes*
- C_3 *:* $Zone_3$ *:* 6 *antennes*
- C_4 *:* $Zone_4$ *:* 5 *antennes*

Les relations de succession entre les zones sont les suivantes :

- C_5 *:* $Zone_0 < Zone_1$
- C_6 *:* $Zone_0 < Zone_4$
- C_7 *:* $Zone_0 < Zone_3$
- C_8 *:* $Zone_2 < Zone_4$
- C_9 *:* $Zone_3 < Zone_2$

Certaines antennes de zones différentes doivent être affectées à une fréquence différente :

- C_{10} *: L'antenne* 2 *de la* $Zone_1$ *a une fréquence différente de l'antenne* 1 *de la* $Zone_3$

– C_{11} : *L'antenne* 1 *de la* $Zone_1$ *a une fréquence différente de l'antenne* 1 *de la* $Zone_4$

La contrainte ALLDIFFERENT *peut être utilisée pour modéliser ce problème.*

Définition 10 (ALLDIFFERENT) *Soit* $var(\text{ALLDIFFERENT}) = \{x_0, x_1, \ldots, x_{n-1}\}$ *un ensemble de variables. La contrainte* ALLDIFFERENT *impose que les variables soient instanciées avec des valeurs toutes différentes les unes des autres : elle est définie par l'ensemble des tuples* $rel(\text{ALLDIFFERENT}) = \{(v_0, v_1, \ldots, v_{n-1}) \in D_0 \times D_1 \times \ldots \times D_{n-1}$ *tels que* $\forall i, j \in \{0, 1, \ldots, n-1\}$ *avec* $i \neq j : v_i \neq v_j\}$.

Pour exprimer le minimum ou bien le maximum d'un ensemble de variable, nous utilisons les contraintes MIN *et* MAX *prenant en argument l'ensemble des variables dont on retient la valeur minimale ou maximale, ainsi qu'une variable "résultat" où est stockée cette valeur minimale ou maximale.*

Le réseau de contraintes $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ *modélisant le problème est alors le suivant.*

– $\mathcal{X} = \{x_{00}, x_{01}, \ldots, x_{03}, x_{10}, \ldots, x_{16}, \ldots, x_{44}, x_0^m, x_1^m, \ldots, x_4^m, x_0^M, x_1^M, \ldots, x_4^M\}$ *chaque variable* x_{ij} *représentant la fréquence d'émission de l'antenne* j *de la zone* i*, et chaque variable* x_j^m *et* x_j^M *représentant respectivement la fréquence minimale et maximale de la zone* j.

– $\mathcal{D} = \{[1, 25], [1, 25], \ldots, [1, 25]\}$

– $\mathcal{C} = \{C_0 : \text{ALLDIFFERENT}\{x_{00}, x_{01}, \ldots, x_{03}\},$
$C_1 : \text{ALLDIFFERENT}\{x_{10}, x_{11}, \ldots, x_{16}\},$
$C_2 : \text{ALLDIFFERENT}\{x_{20}, x_{21}, \ldots, x_{24}\},$
$C_3 : \text{ALLDIFFERENT}\{x_{30}, x_{31}, \ldots, x_{35}\},$
$C_4 : \text{ALLDIFFERENT}\{x_{40}, x_{41}, \ldots, x_{44}\},$
$C_5 : x_0^M < x_1^m,$
$C_6 : x_0^M < x_4^m,$
$C_7 : x_0^M < x_3^m,$
$C_8 : x_2^M < x_4^m,$
$C_9 : x_3^M < x_2^m,$
$C_{10} : x_{12} \neq x_{31},$
$C_{11} : x_{11} \neq x_{41},$
$C_{12} : \text{MIN}(\{x_{00}, x_{01}, \ldots, x_{03}\}, x_0^m),$
$C_{13} : \text{MIN}(\{x_{10}, x_{11}, \ldots, x_{16}\}, x_1^m),$
$C_{14} : \text{MIN}(\{x_{20}, x_{21}, \ldots, x_{24}\}, x_2^m),$
$C_{15} : \text{MIN}(\{x_{30}, x_{31}, \ldots, x_{35}\}, x_3^m),$
$C_{16} : \text{MIN}(\{x_{40}, x_{41}, \ldots, x_{44}\}, x_4^m),$
$C_{17} : \text{MAX}(\{x_{00}, x_{01}, \ldots, x_{03}\}, x_0^M),$
$C_{18} : \text{MAX}(\{x_{10}, x_{11}, \ldots, x_{16}\}, x_1^M),$
$C_{19} : \text{MAX}(\{x_{20}, x_{21}, \ldots, x_{24}\}, x_2^M),$
$C_{20} : \text{MAX}(\{x_{30}, x_{31}, \ldots, x_{35}\}, x_3^M),$
$C_{21} : \text{MAX}(\{x_{40}, x_{41}, \ldots, x_{44}\}, x_4^M)\}$

La figure 1.2 représente le réseau de contraintes $\mathcal{R}$*. Résoudre le problème consiste donc à affecter une fréquence (un entier entre* 1 *et* 25*) à chaque antenne (variables* $x_i j \in \mathcal{X}$*).*

Une solution de ce problème est représentée en Figure 1.3.

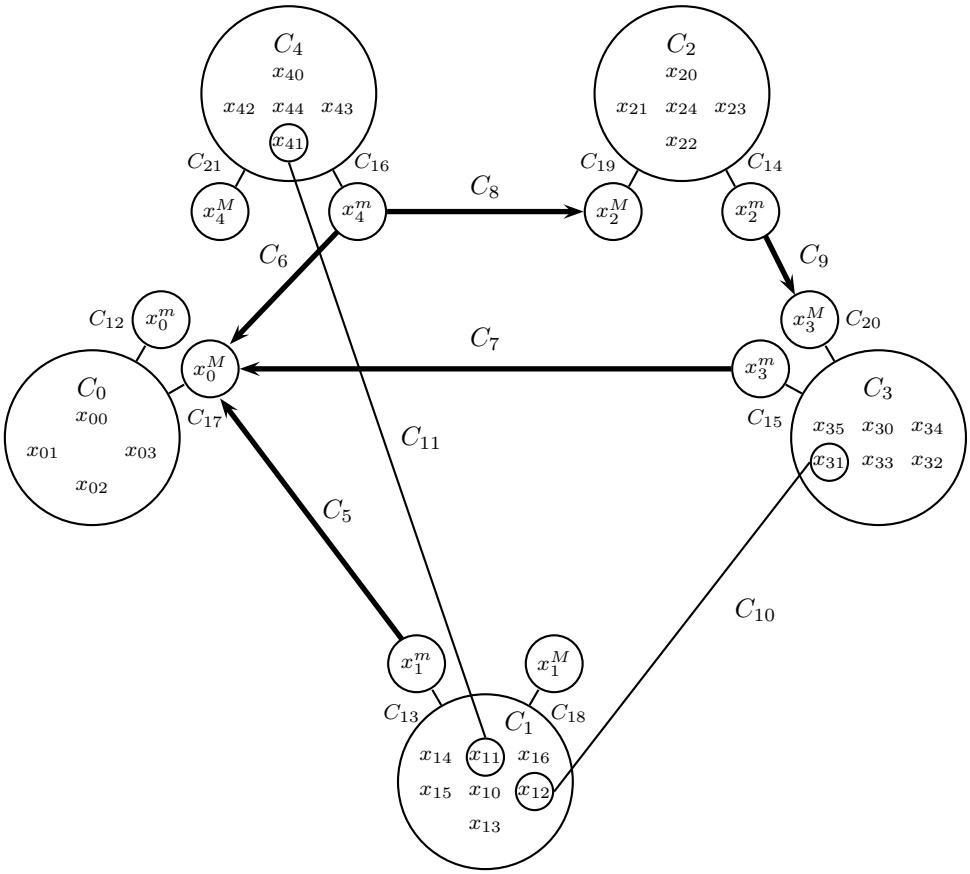


FIGURE 1.2 – Réseau de contraintes modélisant un problème d'affectation de fréquences.

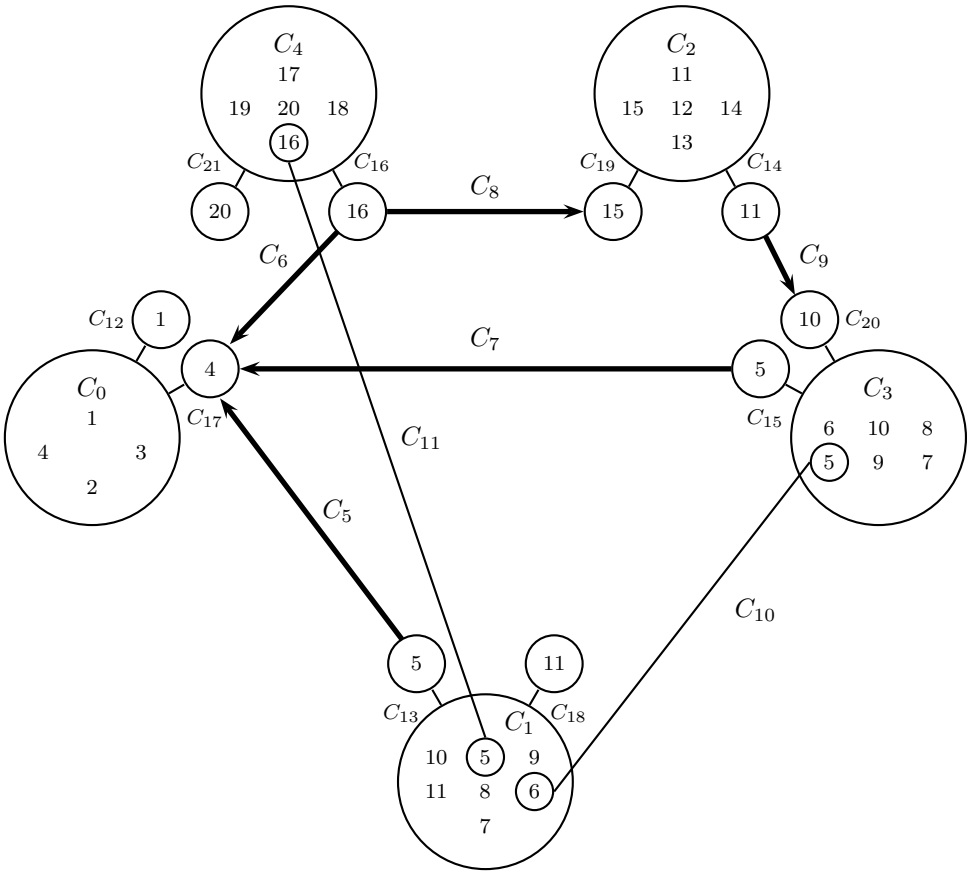


FIGURE 1.3 – Une solution au problème d'affectation de fréquences.

Dans de nombreux problèmes le but n'est pas uniquement de trouver une solution satisfaisant les contraintes, mais de fournir une *solution optimale* selon un critère relatif aux valeurs qui sont affectées aux variables. Il s'agit alors de *problèmes d'optimisation*.

Définition 11 (problème d'optimisation) *Soit un réseau de contraintes $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$. Un problème d'optimisation intègre, dans ce réseau, un* critère d'optimisation. *Celui-ci est modélisé par une variable $obj \in \mathcal{X}$, appelée* variable objectif, *contrainte avec toutes ou partie des variables $\mathcal{X} \backslash \{obj\}$. La valeur affectée à cette variable objectif définit la "qualité" d'une solution du point de vue du critère d'optimisation.*

Définition 12 (solution d'un problème d'optimisation) *Soit un problème d'optimisation représenté par le réseau de contraintes $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$, et une variable $obj \in \mathcal{X}$. Une* solution *de ce problème est une solution de $\mathcal{R}$ qui minimise (pour un problème de minimisation), ou maximise (problème de maximisation) la valeur de la variable objectif obj.*

Pour résoudre des problèmes de satisfaction ou d'optimisation, il est nécessaire d'employer une technique de recherche de solution.

1.3 Recherche de solutions

Étant donné un réseau de contraintes, différentes techniques de recherche peuvent être employées pour obtenir des solutions. Dans un premier temps, nous nous intéressons à des techniques de recherche *systématiques*. Celles-ci consistent à tester successivement toutes les instanciations, jusqu'à trouver une solution, ou trouver une inconsistance remettant en cause la dernière instanciation effectuée. Dans ce dernier cas, on continue la recherche en n'autorisant pas cette instanciation. Ces combinaisons peuvent être générées de différentes manières, et donc testées dans des ordres différents.

Arbre de recherche Une représentation usuelle du processus de recherche d'une solution est l'*arbre de recherche*, construit dynamiquement. La racine de cet arbre est le problème que l'on cherche à résoudre. Les sommets fils sont des problèmes plus petits, obtenus en décomposant le domaine d'une des variables du problème père. Des décompositions successives construisent l'arbre, jusqu'à ce qu'une inconsistance avec une contrainte soit détectée, ou qu'une solution soit trouvée. Généralement, un sommet correspond soit à un essai d'instanciation d'une variable avec une valeur, soit à l'élimination d'une valeur d'un domaine.

Définition 13 (arbre de recherche [84]) *Soit un réseau de contraintes, et P_0 le problème associé. Un* arbre de recherche *pour P_0 est un arbre tel que :*

- *ses noeuds sont des problèmes,*
- *sa racine est P_0,*
- *si $P_1, P_2, \ldots, P_{m-1}$, avec $m > 1$ sont des descendants directs d'un noeud P_0, alors l'union des solutions de $P_1, P_2, \ldots, P_{m-1}$ est égal à l'ensemble des solutions de P, pour tout noeud P.*

Techniques de recherche Pour résoudre un problème de satisfaction de contraintes, une technique consiste à générer tous les tuples de valeurs possibles, et de tester s'ils sont solutions. Cette approche porte le nom de *Generate-and-Test*. Le nombre de tuples générés est égal au produit cartésien du cardinal des domaines des variables impliquées dans le réseau. Bien que cela puisse fonctionner pour les problèmes de petite taille, cela reste impossible à envisager pour la plupart des problèmes.

Une technique de recherche plus fine est l'algorithme de *backtrack*, décrit dans [37], puis amélioré (Backmarking[35], Backjumping[36]). Nous décrivons ici le fonctionnement de cet algorithme sous forme textuelle. Pour plus de détails, le lecteur intéressé pourra se référer à [32].

Au départ, aucune variable n'est instanciée. L'algorithme de backtrack étend progressivement l'instanciation courante en instanciant une nouvelle variable à chaque étape. L'algorithme vérifie alors que l'instanciation partielle est consistante avec les contraintes du problème. Dans le cas contraire, la dernière instanciation faite est remise en cause. Nous effectuons un retour arrière (*backtrack*). C'est à dire que nous remontons au noeud père dans l'arbre de recherche. Puis nous effectuons une nouvelle instanciation. De cette manière, l'algorithme construit un arbre de recherche dont les nœuds représentent les instanciations partielles testées. Cet algorithme teste l'ensemble des instanciations possibles de manière implicite, c'est-à-dire sans les générer toutes. Le nombre d'instanciations considéré est ainsi réduit. Dans le pire des cas le nombre de noeuds parcourus dans l'arbre de recherche est égal au nombre d'instanciations complètes possibles. Ce nombre peut être élevé, même pour des problèmes de "petite taille" : il est égal au produit des tailles des domaines de chaque variable.

La résolution de *problèmes d'optimisation* peut être réalisée par des résolutions successives de problèmes de satisfaction. La méthode la plus classique vise à réduire progressivement l'objectif.

Soit $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ un réseau de contraintes. Pour simplifier la description, nous nous plaçons dans le cas où l'objectif consiste à minimiser la valeur affectée à la variable objectif $obj \in \mathcal{X}$.

Nous recherchons tout d'abord une première solution, par exemple à l'aide d'un algorithme de type *backtrack*. La valeur affectée à obj dans cette solution est notée UB. Nous résolvons alors un nouveau problème de satisfaction équivalent au précédent, à l'exception du fait que la contrainte $obj < UB$ ait été ajoutée. La valeur de obj dans la solution de ce CSP est alors à son tour affectée à UB, et nous résolvons un nouveau CSP, avec une nouvelle contrainte $obj < UB$. Le processus est répété jusqu'à ce que nous aboutissions sur un CSP n'ayant pas de solution. Dans ce cas, la solution du CSP précédant immédiatement ce dernier CSP est une solution optimale du problème d'optimisation.

L'algorithme *Branch and Bound* est une variation de ce schéma de recherche. La différence est que, à chaque étape, au lieu de résoudre un nouveau CSP "from scratch", c'est à dire à partir de la racine de l'arbre, nous poursuivons la recherche à partir du noeud précédent la feuille de l'arbre correspondant à la dernière solution trouvée.

Ces techniques de recherche de solution sont souvent insuffisantes pour traiter des problèmes d'une certaine taille. Des méthodes de retrait de valeurs inconsistantes sont alors à envisager. Nous les introduisons en section 1.4. L'ordre d'instanciation peut avoir un impact significatif sur le processus de recherche d'une solution. Nous décrivons également ce procédé en section 1.4.

1.4 Filtrage, propagation et heuristiques

Filtrage Pour accélérer le processus de résolution, des techniques de filtrage sont utilisées en PPC. Un algorithme de filtrage est associé à chaque contrainte. Il consiste à détecter et à retirer des domaines des variables, des valeurs qui ne sont pas consistantes avec la contrainte. L'implémentation de techniques de filtrage performantes, *i.e.*, qui retirent le plus de valeurs possibles des domaines des variables, joue un rôle déterminant dans la résolution de problèmes en PPC. Un filtrage est réalisé, pour un problème simple, dans l'exemple 5.

Exemple 5 *Soit le réseau de contraintes* $\mathcal{R} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *tel que :*
- $\mathcal{X} = \{x_0, x_1\}$,
- $\mathcal{D} = \{[1,3],[1,3]\}$,
- $\mathcal{C} = \{C_0 : x_0 < x_1\}$.

La valeur 1 *peut être retirée (filtrée) du domaine de* x_1*, car il n'existe aucune valeur dans le domaine de* x_0 *telle que* C_0 *soit satisfaite si* $I[x_1] = 1$.

Propagation Le filtrage d'une valeur peut créer de nouvelles inconsistances au sein du problème. Les techniques de propagation permettent de mettre à jour les domaines des variables après ces retraits, en relançant les algorithmes de filtrage.

Notons que le filtrage est contractant (le domaine d'une variable après un filtrage a une taille inférieure ou égale à celle du domaine avant le filtrage), et donc le processus est monotone.

La propagation des événements de suppression peut être répétée, et les algorithmes de filtrage relancés, jusqu'à ce qu'aucune déduction supplémentaire ne soit possible. On dit alors que *le point fixe* est atteint. La monotonie garantit l'unicité de ce point fixe. L'exemple 6 montre une propagation dans le cadre d'un problème simple.

Exemple 6 *Soit le réseau de contraintes* $\mathcal{R} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *tel que :*
- $\mathcal{X} = \{x_0, x_1, x_2\}$,
- $\mathcal{D} = \{[1,3],[1,3],[1,3]\}$,
- $\mathcal{C} = \{C_0 : x_0 < x_1, C_1 : x_1 = x_2\}$.

Nous avons vu dans l'exemple 5 que la contrainte C_0 *permet de retirer la valeur* 1 *du domaine de* x_1*. De ce fait, elle peut également être retirée du domaine de* x_2*. En effet, après ce filtrage, il n'existe pas de valeur dans le domaine de* x_1 *telle que* C_1 *soit satisfaite si* $I[x_2] = 1$*. Il y a donc une propagation de l'événement de suppression de la valeur* 1 *de* $D(x_1)$ *par* C_0*, qui aboutit à une nouvelle suppression dans* $D(x_2)$ *grâce à* C_1.

Heuristiques de recherche L'ordre dans lequel les variables sont instanciées et l'ordre dans lequel sont testées les différentes valeurs possibles d'une variable peuvent avoir une influence déterminante dans le processus de résolution d'un problème. Pour définir cet ordre, on utilise des *heuristiques* de choix de variable et de valeur. Une *heuristique de choix de variable* détermine la (les) variables(s) à fixer prioritairement. Une *heuristique de choix de valeur* détermine la (les) valeur(s) à affecter prioritairement. Idéalement, l'algorithme qui retourne la variable ou la valeur choisie ne doit pas être trop coûteux.

Une heuristique peut être *statique* : l'ordre dans lequel les variables sont instanciées est fixé a priori, et ne changera pas durant la recherche. Elle peut au contraire être *dynamique* : cet ordre est susceptible d'évoluer au cours de la recherche.

Nous donnons ci-après quelques heuristiques classiques qui seront utiles dans la suite.

Heuristiques de choix de variable
- **lex** est une heuristique statique qui sélectionne les variables à instancier selon l'ordre lexicographique, c'est à dire, l'ordre dans lequel elles sont énumérées dans le problème. Cette heuristique peut être utile lorsque l'on ne connait pas à l'avance la façon dont les variables se comportent. Elle permet de connaître, et de contrôler précisément, et en amont, l'ordre dans lequel les variables seront instanciées,
- **random** sélectionne une variable à instancier aléatoirement,
- **wdeg** (weighted degree) [16] est l'heuristique du degré pondéré. Durant la recherche de solution, on enregistre, pour chaque contrainte, la somme des échecs constatés, dus à cette contrainte. Ainsi, un *poids* est affecté dynamiquement à chacune des contraintes : plus celle-ci a provoqué d'échecs,

plus le poids est important. L'emploi de cette heuristique pondère les variables impliquées dans ces contraintes. La variable choisie est celle de plus grand degré pondéré.
- **dom** (min-domain) [38] choisit la variable de plus petit domaine. C'est en effet la plus "proche d'être fixée", et donc potentiellement, la plus contraignante dans le sens où on a peu de possibilités d'instanciations différentes pour cette variable. Cette heuristique est dynamique.
- **dom/wdeg** [16] combine les heuristiques précédemment présentées. La variable sélectionnée est celle dont le quotient "taille du domaine sur degré pondéré", est le plus petit.

Heuristiques de choix de valeur
- **minVal** (resp. **maxVal**), sélectionne la valeur minimale (resp. maximale) du domaine de la variable courante. Celle-ci permet de contrôler précisément l'ordre dans lequel les valeurs seront essayées,
- **randVal** sélectionne une valeur aléatoire dans le domaine de la variable courante.

Certaines stratégies combinant heuristiques de choix de variables et heuristiques de choix de valeur peuvent également être employées.

II existe un certain nombre d'heuristiques génériques en PPC, qui sont adaptées à bon nombre de problèmes. Cependant, il est souvent efficace de définir des heuristiques spécialisées au problème que l'on résout, plutôt que d'utiliser une heuristique générique. Nous verrons dans cette thèse comment définir quelques heuristiques dans le cas particulier des ordonnancements sur-contraints.

1.5 Les contraintes globales

Définition et intérêts Le terme de contrainte globale est généralement employé pour désigner une contrainte qui représente un problème pour lequel on peut définir un algorithme de filtrage efficace, qui permettra d'accélérer significativement le processus de résolution. Bessière et Van Hentenryck ont proposé une définition qui s'articule autour de trois axes : l'axe sémantique, l'axe opérationnel et l'axe algorithmique [13].

Définition 14 (contrainte globale, Bessière et Van Hentenryck [13]) *Une contrainte est globale :*
- *au sens* sémantique *si elle ne peut être décomposée en une conjonction de contraintes primitives,*
- *au sens* opérationnel *si sa puissance de filtrage est supérieure à celle de sa décomposition en contraintes plus primitives, si une telle décomposition existe,*
- *au sens* algorithmique *si c'est une contrainte globale au sens opérationnel et que, étant donnée une décomposition quelconque Δ, aucun filtrage de Δ n'a une meilleure complexité, ni algorithmique, ni spatiale et que Δ a une taille d'encodage supérieure ou égale à la complexité spatiale nécessaire à assurer la consistance.*

L'un des exemples les plus connus de contrainte globale est la contrainte AllDifferent, introduite en section 1.2, Définition 10. L'exemple 7 montre son intérêt opérationnel.

Exemple 7 (AllDifferent) *Soit une vue partielle du problème posé dans l'exemple 4 se restreignant à l'ensemble de variables $\{x_{00}, x_{01}, x_{02}, x_{03}\}$, avec les domaines $\{[0,2],[0,2],[0,2],[0,2]\}$. Nous considérons par exemple que les domaines de départ du problème ont été réduits par des contraintes extérieures. Le problème impose que ces variables soient toutes différentes deux-à-deux. La Figure 1.4 représente ce sous-problème.*

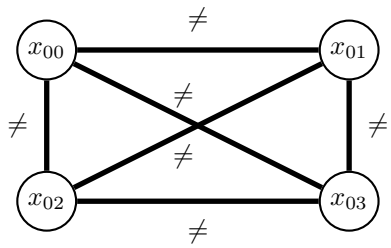


FIGURE 1.4 – Un sous-problème comportant des contraintes de différences.

La contrainte globale ALLDIFFERENT *sur les variables* $\{x_{00}, x_{01}, x_{02}, x_{03}\}$ *(*c.f. *Définition 10) exprime, en une seule contrainte, la conjonction des six contraintes de différence du graphe de contraintes.*

Rendre le réseau d'inégalités binaires arc-consistant n'est pas équivalent, en termes de suppressions de valeurs, à appliquer un filtrage complet avec la contrainte ALLDIFFERENT*.*

En effet :

- $I[\{x_{00}, x_{01}\}] = (0, 1)$ *vérifie la contrainte de différence* $x_{00} \neq x_{01}$*,*
- $I[\{x_{01}, x_{02}\}] = (1, 0)$ *vérifie également la contrainte de différence* $x_{01} \neq x_{02}$*,*
- $I[\{x_{02}, x_{03}\}] = (0, 1)$ *vérifie également la contrainte de différence* $x_{02} \neq x_{03}$*,*
- *etc.*

*L'inconsistance globale de cette contrainte n'est détectée que plus tard, alors même que la taille des domaines, et le nombre de variables (*3 *valeurs pour* 4 *variables) permettent, dès le départ, de déduire que le problème n'admet pas de solution. Cette perte d'information peut être évitée par l'emploi d'un algorithme de filtrage complet pour la contrainte* ALLDIFFERENT*, qui considère toutes les variables en même temps.*

Plusieurs algorithmes de filtrage pour la contrainte ALLDIFFERENT ont été proposés. Régin a, le premier, proposé un algorithme réalisant l'arc-consistance généralisée dans [69]. Ces différents algorithmes ont été décrits et comparés par Van Hoeve [83].

Les travaux de Bessière et al. [12] ont montré que l'on ne pouvait pas atteindre la consistance globale sur une décomposition de la contrainte ALLDIFFERENT, sans ajout de variables supplémentaires.

Chapitre 2

Ordonnancement cumulatif en programmation par contraintes

Sommaire

2.1 Problème cumulatif

Les problèmes d'ordonnancement consistent à ordonner des activités dans le temps. De nombreux problèmes différents existent en ordonnancement. Dans cette thèse, nous nous focaliserons sur le *problème cumulatif*. Dans ce problème, chaque activité est définie par sa durée et nécessite pour son exécution la disponibilité d'une certaine quantité de ressource renouvelable, sa *consommation* (ou *demande en capacité*) qu'elle utilise entre sa *date de début* et sa *date de fin*. Habituellement, l'objectif est de minimiser *l'horizon* (la date de fin maximale d'une activité), tandis qu'en chaque point de temps la consommation cumulée des activités ne doit pas excéder une limite sur la ressource disponible, la *capacité*.

Définition 15 (Activités) *Soit $A = \{a_0, a_1, \ldots, a_{n-1}\}$ un ensemble d'activités. Une activité $a_i \in A$ est définie par quatre variables :*

- $sa_i \in [\underline{sa_i}, \overline{sa_i}]$, *qui représente son début,*
- $da_i \in [\underline{da_i}, \overline{da_i}]$, *qui représente sa durée,*
- $ca_i \in [\underline{ca_i}, \overline{ca_i}]$, *qui représente sa date de fin telle que $ca_i = sa_i + da_i$,*
- $ra_i \in [\underline{ra_i}, \overline{ra_i}]$, *qui représente la quantité discrète de ressource consommée par a_i en tout point de temps entre sa date de début et sa date de fin.*

L'exemple 8 illustre la notion d'activité.

Exemple 8 (activité) *La Figure 2.1 représente une activité a_i. Pour des raisons de lisibilité nous ne représentons pas $\overline{sa_i}$ et $\underline{ca_i}$. Nous représentons les positions extrêmes entre lesquelles peut être placée a_i.*

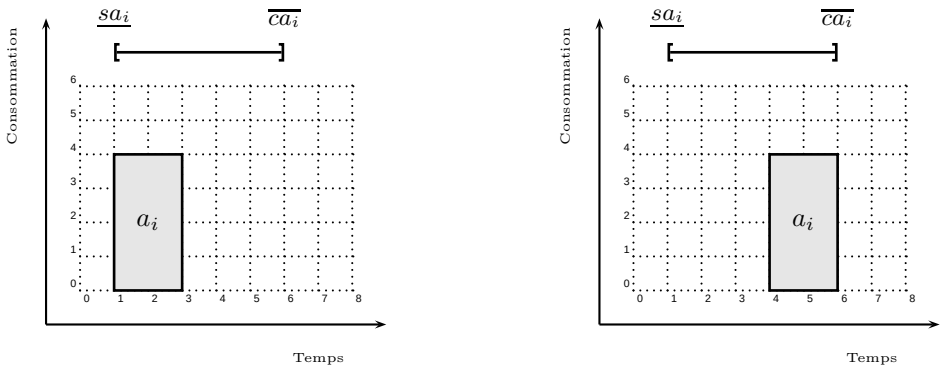


FIGURE 2.1 – Une activité de durée fixée $da_i = 2$, avec une consommation de ressource fixée $ra_i = 4$, représentée dans ses deux positions extrêmes $\underline{sa_i}$ et $\overline{ca_i}$, avec $D(sa_i) = [1, 4]$ et $D(ca_i) = [3, 6]$.

On définit l'énergie d'une activité par la quantité totale de ressource qu'elle consomme.

Définition 16 (énergie d'une activité) *Soit une activité a_i, à laquelle sont associées les variables sa_i, ca_i, da_i, ra_i. L'*énergie *d'une activité est la quantité totale de ressource que a_i consomme entre son début sa_i et sa fin ca_i. Elle est définie par la variable $ea_i = ra_i \times da_i$.*

Les caractéristiques d'une activité peuvent être étendues à des ensembles d'activités.

Notation 2 (ensembles d'activités) *Soit $A = \{a_0, a_1, \ldots, a_{n-1}\}$ un ensemble d'activités. Pour un sous ensemble d'activités $\Omega \subseteq A$, nous notons :*

- *$e(\Omega) \in [\underline{e}(\Omega), \overline{e}(\Omega)]$ l'énergie d'un sous-ensemble d'activités Ω, avec $e(\Omega) = \sum_{a_i \in \Omega} ea_i$,*
- *$s(\Omega) \in [\underline{s}(\Omega), \overline{s}(\Omega)]$ la date de début d'un sous-ensemble d'activités Ω, avec $s(\Omega) = \min_{a_i \in \Omega}(sa_i)$,*
- *$c(\Omega) \in [\underline{c}(\Omega), \overline{c}(\Omega)]$ la date de fin d'un sous-ensemble d'activités Ω, avec $c(\Omega) = \max_{a_i \in \Omega}(ca_i)$,*

Nous utiliserons également les notations $\underline{e}(\Omega)$, $\underline{s}(\Omega)$, $\underline{c}(\Omega)$ pour désigner la borne inférieure du domaine de ces variables, et $\overline{e}(\Omega)$, $\overline{s}(\Omega)$, $\overline{c}(\Omega)$ pour désigner la borne supérieure du domaine de ces variables.

Définition 17 (Hauteur cumulée) *Etant donnée une ressource et une instanciation d'un ensemble A de n activités, à chaque point de temps t la* hauteur cumulée *h_t des activités passant par t est $h_t = \sum_{\substack{a_i \in A \\ sa_i \leq t < ca_i}} ra_i$.*

Définition 18 (Problème cumulatif) *Soit une ressource avec une capacité limitée par un entier capa et une instanciation d'un ensemble A de n activités. Une solution du problème cumulatif est un ordonnancement respectant les deux contraintes suivantes :*

- *$C1$: Pour chaque activité $a_i \in A, sa_i + da_i = ca_i$,*
- *$C2$: A chaque point de temps t, $h_t \leq capa$.*

Dans cette thèse, nous considérons des activités non-préemptives, c'est à dire des activités qui ne peuvent pas être interrompues. La contrainte $C1$ de la Définition 18 garantit cette propriété.

L'exemple 9 illustre le problème cumulatif.

Exemple 9 *La Figure 2.2 illustre une solution d'un problème cumulatif à 5 activités. Chaque activité est placée en accord avec les bornes des domaines initiaux de ses variables de début et de fin, qui sont représentées dans la partie haute de la figure.*

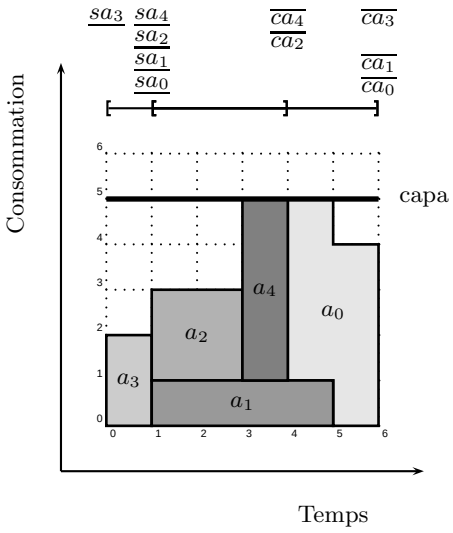


FIGURE 2.2 – Une solution d'un problème cumulatif à 5 activités. En chaque point de temps, la hauteur du profil cumulé est bien inférieure ou égale à la capacité *capa*.

Complexité Le problème cumulatif, vu comme un problème de décision, est un problème NP-complet [5], comme de nombreux problèmes d'ordonnancement [34]. Le problème d'optimisation visant à minimiser la date de fin de la dernière activité ordonnancée est un problème NP-difficile [59].

2.2 La contrainte CUMULATIVE

En PPC, la contrainte CUMULATIVE modélise les problèmes cumulatifs. Elle a été introduite par Aggoun et Beldiceanu [1].

Définition 19 (Contrainte cumulative) *Étant donnés un ensemble d'activités* $A = \{a_0, a_1, \ldots, a_{n-1}\}$ *et une capacité entière capa, la contrainte* CUMULATIVE$(A, capa)$ *modélise un problème cumulatif. Une solution de la contrainte* CUMULATIVE$(A, capa)$ *est une instanciation complète des variables représentant les activités de A, telle que les contraintes C1 et C2 de la Définition 18 soient satisfaites*[1].

Différents algorithmes de filtrage ont été définis pour le contrainte CUMULATIVE. De manière générale, nous pouvons les classifier en deux grandes catégories : des filtrages qui s'appuient sur la topologie de l'ordonnancement, et des filtrages qui s'appuient sur l'énergie d'ensembles d'activités. L'idée de *raisonnement énergétique* a été introduite et développée dans [29, 53, 54, 55].

Ces différents filtrages permettent d'effectuer des déductions différentes, et peuvent donc être complémentaires. Nous décrivons l'algorithme de Sweep, et les algorithmes d'Edge-Finding, qui sont, dans les deux catégories évoquées précédemment, parmi les plus performants.

1. Dans la version originale de la contrainte CUMULATIVE, *capa* est une *variable* représentant la hauteur maximale du profil cumulé.

2.2.1 L'algorithme de *balayage*, ou *Sweep*

Le filtrage de la contrainte Cumulative par *balayage* (connu sous le nom de *Sweep*), décrit dans [7, 8], s'appuie sur une notion essentielle en ordonnancement cumulatif : la notion de partie obligatoire. Elle a été introduite par Lahrichi [46].

L'algorithme de time-table, décrit dans [4, chapitres 2.2.1 et 3.3.1], utilise également cette notion. Il maintient un profil de parties obligatoires et filtre les débuts d'activités en fonction de ce profil, comme Sweep. Nous ne nous concentrons pas sur cet algorithme et choisissons de décrire l'algorithme de *Sweep* car le stockage en mémoire de l'ensemble du profil n'y est pas nécessaire. De plus, nous réutiliserons et adapterons cet algorithme dans le chapitre 5.

Définition 20 (partie obligatoire) *La partie obligatoire $cp(a_i)$ d'une activité $a_i \in A$ est l'intersection de toutes les instances réalisables de a_i. La partie obligatoire est définie par son support, l'intervalle $[\overline{sa_i}, \underline{ca_i}[$ (elle est donc non vide si, et seulement si, $\overline{sa_i} < \underline{ca_i}$), et une hauteur égale à $\underline{ra_i}$ sur $[\overline{sa_i}, \underline{ca_i}[$, et nulle ailleurs.*

L'exemple 10 illustre la notion de partie obligatoire.

Exemple 10 *La Figure 2.3 représente une activité positionnée au plus tôt et au plus tard, et la partie obligatoire qui est délimitée par l'intersection de ces deux positions.*

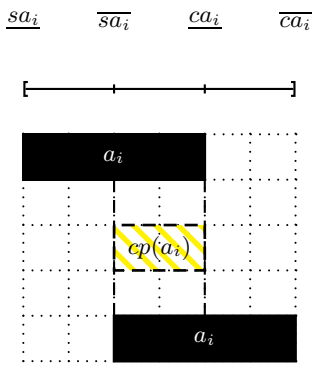


Figure 2.3 – Une activité a_i positionnée en ses deux positions extrêmes, et sa partie obligatoire $cp(a_i)$.

La partie obligatoire des activités d'un problème cumulatif permet de définir un profil cumulatif.

Définition 21 (profil cumulatif) *Le* profil cumulatif *$CumP$ représente la consommation minimale de ressource, cumulée au cours du temps, de toutes les activités. Pour un point de temps donné t, la hauteur de $CumP$ à t est égale à $\sum_{\substack{a_i \in A \\ t \in [\overline{sa_i}, \underline{ca_i}[}} \underline{ra_i}$, c'est à dire la somme des contributions des parties obligatoires de chaque activité passant par t.*

L'exemple 11 illustre la notion de profil cumulatif.

Exemple 11 *La Figure 2.4 représente le problème cumulatif introduit en Figure 2.2. Il consiste à ordonner l'ensemble d'activités* $A = \{a_0, a_1, \ldots, a_4\}$*, sans dépasser la capacité capa. Nous montrons sur la Figure 2.5, que les activités* a_1 *et* a_2 *ont une partie obligatoire. La Figure 2.6 représente le profil cumulatif associé au problème.*

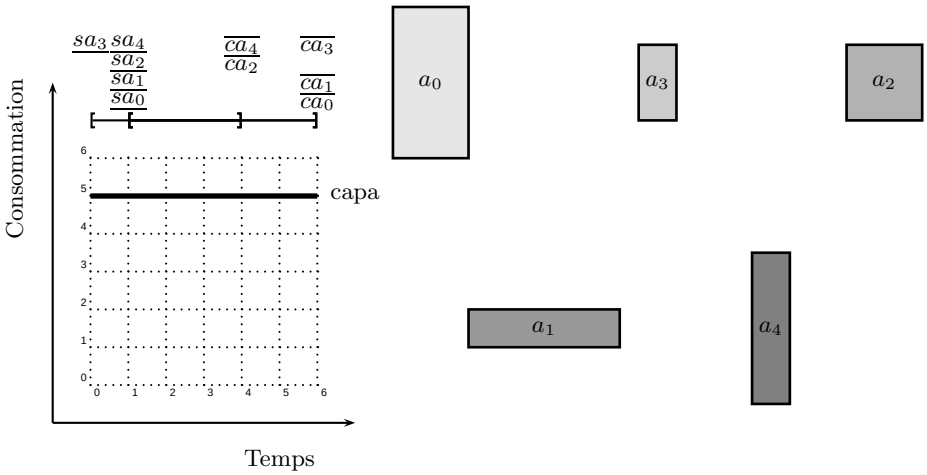


FIGURE 2.4 – Le problème cumulatif introduit en Figure 2.2.

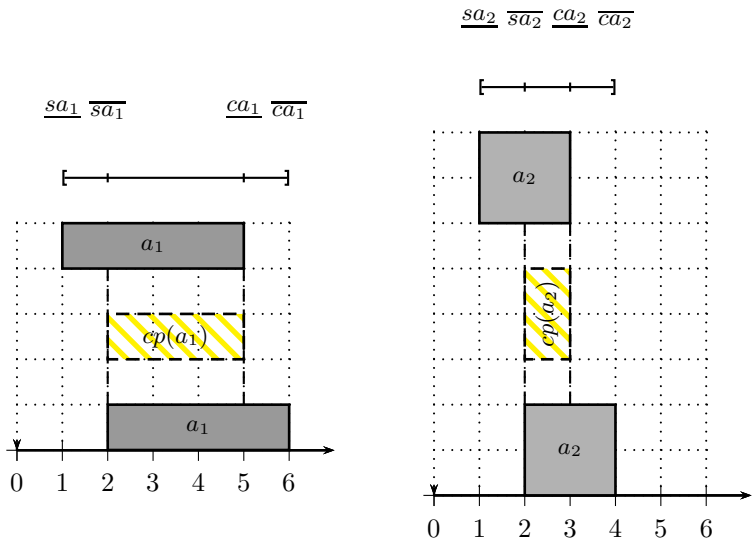


FIGURE 2.5 – Les activités a_1 et a_2 positionnées en leurs deux positions extrêmes, et leur partie obligatoire $cp(a_1)$ et $cp(a_2)$.

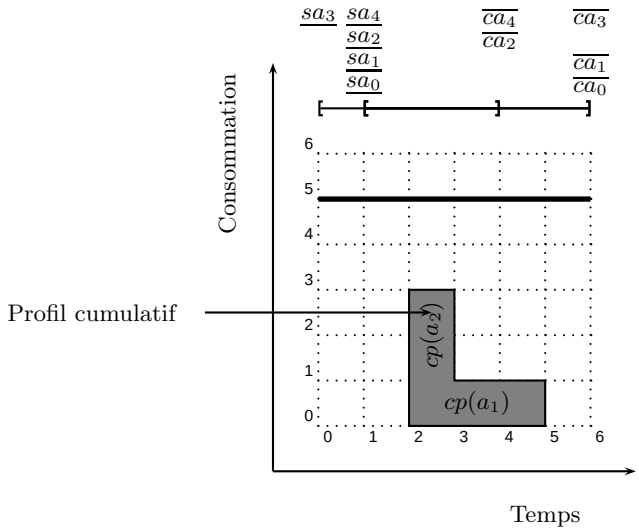


FIGURE 2.6 – Le profil cumulatif des parties obligatoires associé au problème.

Le filtrage défini en [7] s'appuie sur ce profil cumulatif pour filtrer les dates de début d'activités.

Il déplace une ligne verticale Δ sur l'axe de temps par ordre croissant de valeurs en ne considérant que certains points ou *événements*. En un balayage, il construit le profil cumulatif et filtre les activités de manière à ne pas dépasser la capacité *capa*. Un *événement* correspond au début ou à la fin d'une partie obligatoire, ou à la date de début au plus tôt $\underline{sa_i}$ d'une activité $a_i \in A$. Pour une raison qui sera énoncée dans la suite, il ne considère pas sa date de fin au plus tard $\overline{ca_i}$. Tous les événements sont initialement générés et triés par ordre croissant de leur date. On notera δ la position courante de la droite de balayage Δ. A chaque pas de l'algorithme, une liste $ActToPrune$ contient les activités à filtrer, c'est à dire celles pouvant passer par la date δ. À chaque état de la droite de balayage est associée une hauteur de profil sum_h, qui sera mise à jour incrémentalement :

- les événements de partie obligatoire servent à construire $CumP$. Ces événements, à la date δ, mettent à jour la hauteur sum_h du rectangle courant dans $CumP$, en ajoutant la hauteur si c'est le début d'une partie obligatoire, ou en l'enlevant si c'est la fin d'une partie obligatoire. Soit a_i l'activité concernée, alors un tel événement est noté $evt =< PROFILE, a_i, \overline{sa_i}, \underline{ra_i} >$ si c'est un début de partie obligatoire et $evt =< PROFILE, a_i, \underline{ca_i}, -\underline{ra_i} >$ si c'est une fin de partie obligatoire,
  Le premier événement de partie obligatoire dont la date est strictement supérieure à δ définit la fin de l'intervalle courant. On note cette date δ', et le rectangle correspondant est $\langle [\delta, \delta'[, sum_h \rangle$,
- les événements correspondant aux dates de début au plus tôt $\underline{sa_i}$ des activités $a_i \in A$ *non complètement fixées* et telles que $\delta \leq \underline{sa_i} < \delta'$ ajoutent de nouvelles activités candidates au filtrage, en fonction de $\langle [\delta, \delta'[, sum_h \rangle$ et *capa* (les activités ayant une intersection avec $\langle [\delta, \delta'[, sum_h \rangle$). Elles sont ajoutées à la liste $ActToPrune$. Soit a_i l'activité concernée, alors un tel événement est noté $evt =< PRUNING, a_i, \underline{sa_i}, 0 >$. Le paramètre 0 correspond à l'augmentation de hauteur du profil. Elle est toujours nulle dans ce type d'événement.

En chaque événement de profil, pour chaque activité $a_i \in ActToPrune$, n'ayant pas de partie obligatoire dans le rectangle $\langle [\delta, \delta'[, sum_h \rangle$, si sa hauteur ra_i est strictement supérieure à $capa - sum_h$ alors on filtre sa date de début sa_i de manière à ce que a_i ne puisse plus intersecter le rectangle courant $CumP$. Si $\overline{ca_i} \leq \delta'$ alors a_i est retirée de la liste $ActToPrune$. Une fois l'étape de filtrage terminée, δ est mise à jour à la valeur δ'.

Règle de filtrage 1 *Si* $a_i \in ActToPrune$ *n'a pas de partie obligatoire dans* $\langle [\delta, \delta'[, sum_h \rangle$ *et* $sum_h + \underline{ra_i} > capa$ *alors* $]\delta - \underline{da_i}, \delta'[$ *peut être retiré de* $D(sa_i)$.

L'algorithme 1 génère l'ensemble des événements présentés ci-dessus. L'algorithme 2 réalise la mise à jour des dates de début d'activité.

Algorithme 1: Algorithme de génération des événements du Sweep pour CUMULATIVE

Data : Un ensemble d'activités A.
Result : Une liste d'événements Evt.

List Evt // La liste d'événements générés
foreach $a_i \in A$ **do**
  if $\underline{sa_i} \neq \overline{sa_i}$ || $\underline{ca_i} \neq \overline{ca_i}$ || $\underline{da_i} \neq \overline{da_i}$ || $\underline{ra_i} \neq \overline{ra_i}$ **then**
    $Evt.add(< PRUNING, a_i, \underline{sa_i}, 0 >)$;
  if $\underline{ca_i} > \overline{sa_i}$ **then**
    $Evt.add(< PROFILE, a_i, \overline{sa_i}, \underline{ra_i} >)$;
    $Evt.add(< PROFILE, a_i, \underline{ca_i}, -\underline{ra_i} >)$;
return Evt

Algorithme 2: Algorithme de Sweep pour CUMULATIVE

Data : Un ensemble d'activités A, une capacité *capa*, une liste d'événements Evt.

List $ActToPrune$; // La liste des activités à filtrer
int $sum_h = 0$;//La hauteur du profil
int δ ; // La date courante
int δ' ; // La date du prochain événement
$evt = Evt.first()$; // Extraction du premier événement
$\delta = evt.date$; // Enregistrement de sa date
$Evt.sortByDate()$; On ordonne les événements par date croissante, pour une date égale, on place d'abord les événements $PROFILE$, puis $PRUNING$
while $evt \neq NULL$ **do**
  $a_i = evt.activity$;// L'activité correspondant à l'événement courant
  $\delta' = evt.date$; //La date de l'événement courant
  if $evt.type == PROFILE$ **then**
    if $\delta \neq \delta'$ **then**
      if $sum_h > capa$ **then**
        return fail ;
      foreach $a_j \in ActToPrune$ **do**
        if $(sum_h + \underline{ra_j} > capa)\&(\overline{sa_j} \geq \underline{ca_j} || ((\overline{sa_j} < \underline{ca_j})\&[\overline{sa_j}, \underline{ca_j}[\bigcap[\delta, \delta'[= \emptyset))$ **then**
          $D(sa_j) = \overline{D}(sa_j) \backslash]\delta - \underline{da_j}, \delta'[$;//Mise à jour de la date de début d'une activité n'ayant pas de partie obligatoire dans $[\delta, \delta'[$
        if $\overline{ca_j} < \delta'$ **then**
          $ActToPrune.remove(a_j)$;
      $\delta = \delta'$;
    $sum_h += evt.increment$;
  else if $evt.type == PRUNING$ **then**
    $ActToPrune.add(a_i)$;
  $evt = evt.next()$
if $sum_h > capa$ **then**
  return fail ;
foreach $a_j \in ActToPrune$ **do**
  if $(sum_h + \underline{ra_j} > capa)\&(\overline{sa_j} \geq \underline{ca_j} || ((\overline{sa_j} < \underline{ca_j})\&[\overline{sa_j}, \underline{ca_j}[\bigcap[\delta, \delta'[= \emptyset))$ **then**
    $D(sa_j) = \overline{D}(sa_j) \backslash]\delta - \underline{da_j}, \delta'[$;

La complexité temporelle globale de ces algorithmes (génération + mise à jour) est $O(n^2 + n \times log(n))$ où n est le nombre d'activités. Notons qu'en pratique, cette complexité est souvent atteinte car, à chaque déplacement de la droite de balayage Δ, l'algorithme parcourt l'ensemble de la liste $ActToPrune$. Pour cette raison, un algorithme de balayage plus efficace sera présenté à la conférence CP'12 [52].

L'exemple 12 illustre le fonctionnement de l'algorithme de Sweep.

Exemple 12 *La figure 2.7 représente les événements associés au problème présenté en figure 2.4, et au profil cumulatif associé. Les activités à ordonner ont les caractéristiques suivantes :*

- *activité* a_0 *:* $D(sa_0) = [1,4]$, $D(ca_0) = [3,6]$, $da_0 = 2$, $ra_0 = 4$ *;*
- *activité* a_1 *:* $D(sa_1) = [1,2]$, $D(ca_1) = [5,6]$, $da_1 = 4$, $ra_1 = 1$ *;*
- *activité* a_2 *:* $D(sa_2) = [1,2]$, $D(ca_2) = [3,4]$, $da_2 = 2$, $ra_2 = 2$ *;*
- *activité* a_3 *:* $D(sa_3) = [0,5]$, $D(ca_3) = [1,6]$, $da_3 = 1$, $ra_3 = 2$ *;*
- *activité* a_4 *:* $D(sa_4) = [1,3]$, $D(ca_4) = [2,4]$, $da_4 = 1$, $ra_4 = 4$.

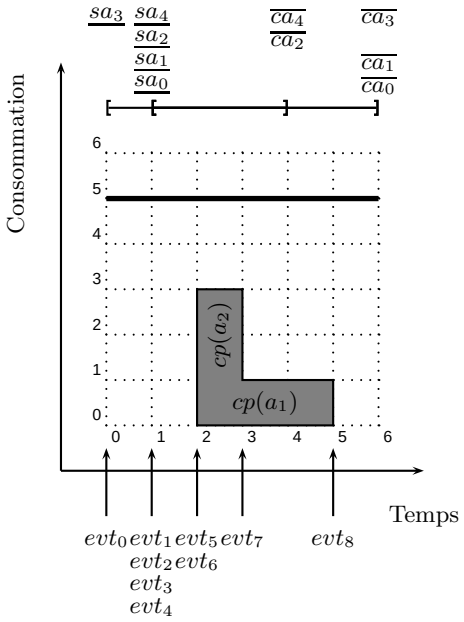


FIGURE 2.7 – Le profil cumulatif associé au problème.

9 *événements sont générés. Ils sont définis comme suit :*

- $evt_0, evt_1, \ldots, evt_4$ *sont des événements de dates de début au plus tôt. Les activités correspondantes sont alors ajoutées dans la liste* $ActToPrune$. *Ainsi, après* evt_0, $ActToPrune = \{a_3\}$. *Après* evt_4, $ActToPrune = \{a_0, a_1, a_2, a_3, a_4\}$,
- evt_5, evt_6 *sont des événements de partie obligatoire. Ils correspondent, respectivement, au début des parties obligatoires* $cp(a_1)$ *et* $cp(a_2)$. *Ils sont associés à une augmentation de la hauteur du profil cumulatif : cette augmentation est de* $\underline{ra_1} = 1$ *et* $\underline{ra_2} = 2$,
- evt_7, evt_8 *sont des événements de partie obligatoire. Ils correspondent, respectivement, à la fin des parties obligatoires* $cp(a_1)$ *et* $cp(a_2)$. *Ils sont associés à une diminution de la hauteur du profil cumulatif : cette diminution est de* $\underline{ra_1} = 1$ *et* $\underline{ra_2} = 2$.

Une fois ces événements générés, ils sont "balayés", de manière à filtrer les dates de début des activités. La Figure 2.8 illustre les différentes étapes de ce filtrage. Notons que, pour illustrer, nous n'effectuons qu'un balayage, mais que réitérer ce processus est nécessaire dans ce cas si l'on souhaite atteindre un point fixe (i.e. qu'on ne puisse plus effectuer de filtrage supplémentaire). L'algorithme décrit dans [52] permet d'éviter partiellement ce problème.

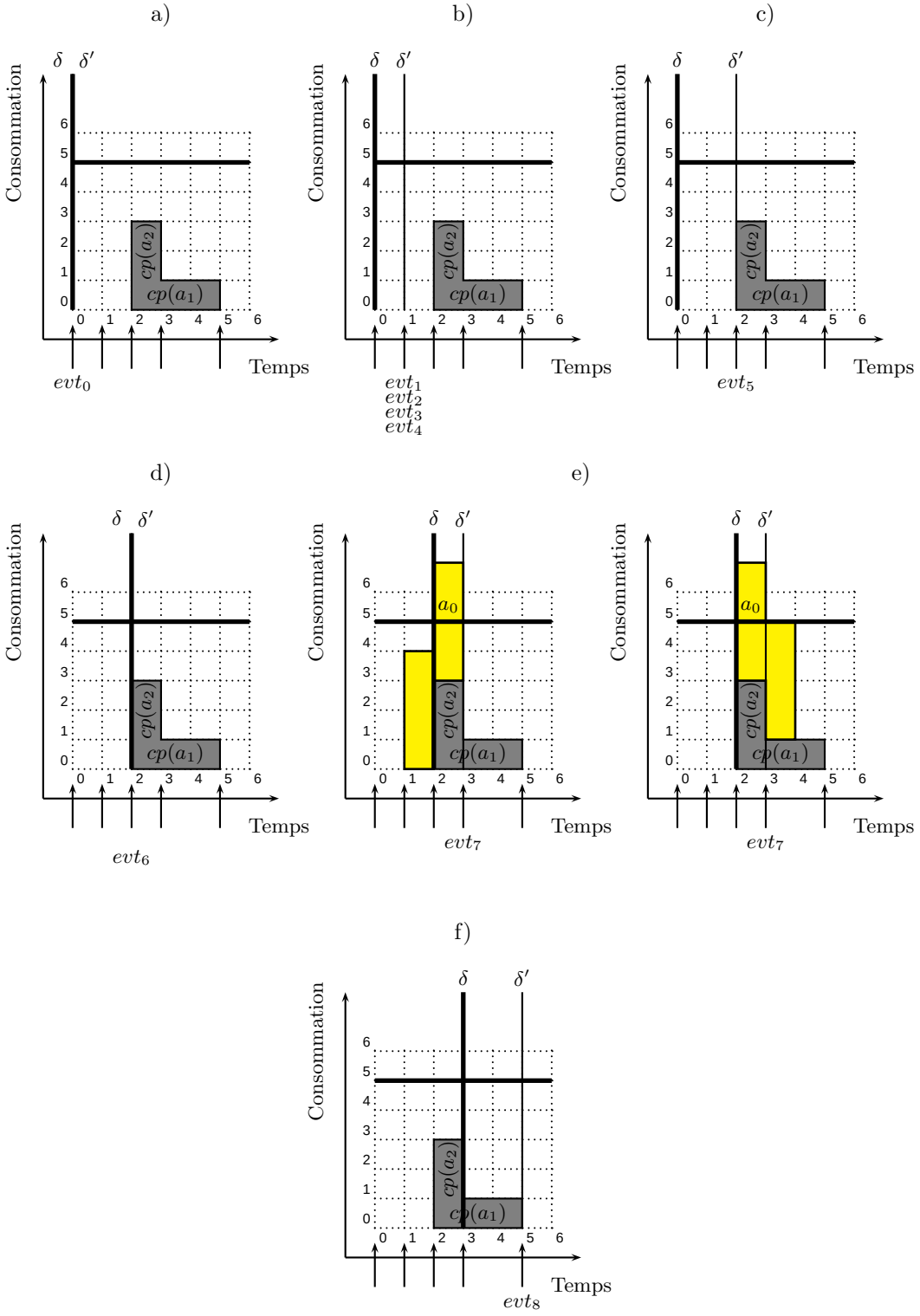


FIGURE 2.8 – différentes étapes de l'algorithme de balayage.

Nous détaillons à présent les différentes étapes de filtrage associées au balayage. Au départ, $sum_h = 0$.

- *à l'étape a)*
  1. δ *est placé à la date* 0, *et* δ' *à la date* 0,
  2. *en* 0, *il y a un événement :* $evt_0 =< PRUNING, a_3, 0, 0 >$, *qui est un événement de début au plus tôt,*
  3. *en conséquence,* $ActToPrune = \{a_3\}$,
  4. *il n'y a pas d'événement de partie obligatoire :* $sum_h = 0$,
  5. *c'est un événement de type* $PRUNING$ *: aucun filtrage n'est effectué,*
- *à l'étape b)*
  1. δ *reste en* 0, *et* δ' *est placé à la date* 1,
  2. *en* 1, *il y a quatre événements :* $evt_1 =< PRUNING, a_0, 1, 0 >$, $evt_2 =< PRUNING, a_1, 1, 0 >$, $evt_3 =< PRUNING, a_2, 1, 0 >$, $evt_4 =< PRUNING, a_4, 1, 0 >$ *qui sont des événements de début au plus tôt,*
  3. *en conséquence,* $ActToPrune = \{a_0, a_1, a_2, a_3, a_4\}$,
  4. *il n'y a pas d'événement de partie obligatoire :* $sum_h = 0$,
  5. *ce sont des événements de type* $PRUNING$ *: aucun filtrage n'est effectué,*
- *à l'étape c)*
  1. δ *est placé à la date* 0, *et* δ' *à la date* 2,
  2. *en* 2, *il y a un événement :* $evt_5 =< PROFILE, a_1, 2, 1 >$ *qui est un événement de partie obligatoire*
  3. $ActToPrune = \{a_0, a_1, a_2, a_3, a_4\}$ *n'est pas modifié,*
  4. $\delta \neq \delta'$, *alors on teste :* $sum_h = 0 \leq capa$, *l'algorithme peut donc continuer,*
  5. *aucune activité n'a de partie obligatoire dans l'intervalle* $[0, 2[$,
  6. $sum_h + \underline{ra_0} = 4 \leq capa$, $sum_h + \underline{ra_1} = 1 \leq capa$, $sum_h + \underline{ra_2} = 2 \leq capa$, $sum_h + \underline{ra_3} = 2 \leq capa$, $sum_h + \underline{ra_4} = 4 \leq capa$, *il n'y a donc pas de filtrage à effectuer,*
  7. δ *prend la valeur de* δ' *:* $\delta = 2$, $\delta' = 2$,
  8. sum_h *est mis à jour :* $sum_h = sum_h + \underline{ra_1} = 0 + 1 = 1$,
- *à l'étape d)*
  1. δ *est placé à la date* 2, *et* δ' *à la date* 2,
  2. *en* 2, *il y a un autre événement :* $evt_6 =< PROFILE, a_2, 2, 2 >$ *qui est un événement de début de partie obligatoire,*
  3. $ActToPrune = \{a_0, a_1, a_2, a_3, a_4\}$ *n'est pas modifié,*
  4. $\delta = \delta'$, *aucun filtrage n'est donc effectué,*
  5. sum_h *est mis à jour :* $sum_h = sum_h + \underline{ra_2} = 1 + 2 = 3$,
- *à l'étape e)*
  1. δ *est placé à la date* 2, *et* δ' *à la date* 3,
  2. *en* 3, *il y a un événement :* $evt_7 =< PROFILE, a_2, 3, -2 >$ *qui est un événement de fin de partie obligatoire,*
  3. $ActToPrune = \{a_0, a_1, a_2, a_3, a_4\}$ *n'est pas modifié,*
  4. $\delta \neq \delta'$, *alors on teste :* $sum_h = 3 \leq capa$, *l'algorithme peut donc continuer,*

5. a_1 *et* a_2 *ont une partie obligatoire dans l'intervalle* $[2, 3[$,
6. $\mathbf{sum_h} + \underline{\mathbf{ra_0}} = \mathbf{7} > \mathbf{capa}$, $sum_h + \underline{ra_3} = 5 \leq capa$, $\mathbf{sum_h} + \underline{\mathbf{ra_4}} = \mathbf{7} > \mathbf{capa}$, *donc on peut effectuer un filtrage des variables* sa_0 *et* sa_4,
7. *La règle de filtrage 1 implique que l'on peut ôter de* $D(sa_0)$ *l'intervalle* $[1, 3[$, *et de* $D(sa_4)$ *l'intervalle* $[2, 3[$. *Ainsi, après ce filtrage,* $D(sa_0) = [1, 4]\backslash]0, 3[= [3, 4]$. *De plus,* $D(sa_4) = [0, 3]\backslash]1, 3[= \{0, 1, 3\}$,
8. *nous représentons les positions filtrées de l'activité* a_0, *i.e. celles qui mèneraient à un dépassement de capacité,*
9. *remarquons que, par propagation, les variables de fin* ca_0 *et* ca_4 *seront également filtrées,*
10. δ *prend la valeur de* δ' : $\delta = 3$,
11. sum_h *est mis à jour :* $sum_h = sum_h - \underline{ra_2} = 3 - 2 = 1$,

– *à l'étape f)*

1. δ *est placé à la date* 3, *et* δ' *à la date* 5,
2. *en* 5, *il y a un événement :* $evt_8 =< PROFILE, a_1, 5, -1 >$ *qui est un événement de fin de partie obligatoire,*
3. $\delta \neq \delta'$, *alors on teste :* $sum_h = 1 \leq capa$, *l'algorithme peut donc continuer,*
4. a_1 *a une partie obligatoire dans l'intervalle* $[3, 5[$,
5. $sum_h + \underline{ra_0} = 5 \leq capa$, $sum_h + \underline{ra_2} = 3 \leq capa$, $sum_h + \underline{ra_3} = 3 \leq capa$, $sum_h + \underline{ra_4} = 5 \leq capa$, *donc il n'y a pas de filtrage à effectuer,*
6. $\overline{ca_2} \leq \delta'$ *et* $\overline{ca_4} \leq \delta'$. *Les activités correspondantes sont donc retirées de la liste* $ActToPrune$: $ActToPrune = \{a_0, a_1, a_3\}$,
7. δ *prend la valeur de* δ' : $\delta = 5$,
8. $sum_h = sum_h - \underline{ra_1} = 1 - 1 = 0$
9. *il n'y a plus d'événement,*

– *à la fin de l'algorithme*

1. $sum_h = 0 \leq capa$, *donc il n'y a pas d'inconsistance détectée*
2. $ActToPrune = \{a_0, a_1, a_3\}$,
3. $sum_h + \underline{ra_0} = 4 \leq capa$, $sum_h + \underline{ra_1} = 2 \leq capa$, $sum_h + \underline{ra_3} = 2 \leq capa$, *donc il n'y a pas de filtrage à effectuer.*

Nous rappelons que, dans cet exemple, suite à ce balayage, le point fixe n'a pas été atteint. En effet, par exemple, l'activité a_0 *a maintenant une partie obligatoire, elle modifie donc le profil cumulatif. De nouvelles déductions sont alors possibles. De nouveaux événements sont générés, et l'algorithme est relancé autant de fois que nécessaire pour atteindre le point fixe.*

2.2.2 Les algorithmes d'Edge Finding

Les algorithmes d'Edge-Finding cherchent des relations de précédence induites par les domaines et l'énergie des activités. Soit un problème cumulatif consistant à ordonner un ensemble d'activités A. Ces relations de précédence sont du type : un ensemble $\Omega \subseteq A$ *finit avant la fin* (ou, symétriquement, commence après le début) d'une activité $a_i \in A$.

Notons que ces algorithmes se basent sur des durées et des consommations de ressource fixes pour les activités. Pour des activités de durées et de consommations variables, nous nous baserons sur leur valeur minimale. En conséquence, l'énergie des activités est également fixe. Nous notons toujours ces quantités, pour une activité a_i, da_i, ra_i et ea_i respectivement.

Définition 22 (Précédence) *Une activité $a_i \in A$* finit avant la fin *d'une activité $a_j \in A$ ssi, dans toutes les solutions, $ca_i \leq ca_j$. On écrit : $a_i \lessdot a_j$. Cela peut être étendu à un ensemble d'activités $\Omega \subseteq A$: Ω finit avant la fin de a_j si et seulement si, dans toutes les solutions, $c(\Omega) \leq ca_j$. Nous écrivons $\Omega \lessdot a_j$.*

En se basant sur ces précédences, il est possible de filtrer les dates de début des activités impliquées.

De nombreux algorithmes d'Edge-Finding ont été développés. Carlier et Pinson [19] introduisent cet algorithme en ordonnancement *disjonctif* (les activités sont de hauteur 1 et la capacité *capa* est égale à 1). Vilím en propose également un dans [89]. Pour des problèmes disjonctifs à n activités, ces algorithmes ont une complexité en $O(n \times log(n))$.

En ordonnancement cumulatif, Nuijten [59] et Baptiste et al. [4] ont, les premiers, introduit des algorithmes d'Edge-Finding. Ceux-ci ont une complexité en $O(n^2 \times k)$ et $O(n^2)$ respectivement, n étant le nombre d'activités du problème, et k le nombre de hauteurs différentes entre toutes les activités. Mercier et Van Hentenryck démontrent dans [57] que ces approches ne sont pas correctes. Ils proposent, de plus, un algorithme de filtrage complet, en $O(n^2 \times k)$.

Nous nous concentrons sur les deux algorithmes d'Edge-Finding les plus récents, à notre connaissance, dans la littérature :

- l'algorithme de Vilím [90],
- l'algorithme de Kameugne et al. [43].

2.2.2.1 Algorithme d'Edge Finding de Vilím [90]

Vilím a proposé un algorithme d'Edge-Finding de complexité $O(k \times n \times log(n))$ [90], que nous détaillerons ici. Nous détaillons les deux phases de cet algorithme :

1. une phase de détection des précédences entre activités,
2. une phase de filtrage des dates de début au plus tôt des activités.

Première phase : détection des précédences Nous donnons d'abord quelques définitions indispensables pour comprendre cet algorithme.

Dans un premier temps, nous introduisons la notation *Area* qui correspond à l'aire disponible. Cette notation n'est pas donnée telle quelle dans les algorithmes de Vilím et Kameugne et al. Nous l'introduisons afin de généraliser ces algorithmes au cas de problèmes avec dépassement de la capacité en ressource, dans le chapitre 5.

Définition 23 (Aire disponible) *Soit un intervalle de temps $I = [a, b[$, nous notons $Area(a, b)$ la ressource maximum (aire) disponible, égale à $(b - a) \times capa$.*

L'enveloppe énergétique correspond à une surestimation maximale de l'aire occupée avant la date de fin au plus tard de l'ensemble d'activités.

Définition 24 (Enveloppe énergétique) *Soit $\Omega \subseteq A$ un ensemble d'activités. L'*enveloppe énergétique *de Ω est : $Env(\Omega) = \max_{\Theta \subseteq \Omega}(Area(0, \underline{s}(\Theta)) + \underline{e}(\Theta))$.*

L'exemple 13 illustre la notion d'enveloppe énergétique.

Exemple 13 *La Figure 2.9 montre le calcul de l'enveloppe énergétique de l'ensemble* $LCut(A, a_4) = \{a_2, a_4\}$*, avec* a_2 *en rouge et* a_4 *en violet, tels que* $D(sa_2) = [1, 2]$ *et* $D(sa_4) = [1, 3]$*. L'enveloppe énergétique de cet ensemble est une surestimation maximale de l'aire occupée avant la fin au plus tôt de l'ensemble. Ce maximum est donné par la troisième figure :* $Env(LCut(A, a_4)) = \max_{\Theta \subseteq LCut(A,a_4)}(Area(0, \underline{s}(\Theta)) + \underline{e}(\Theta)) = 13$.

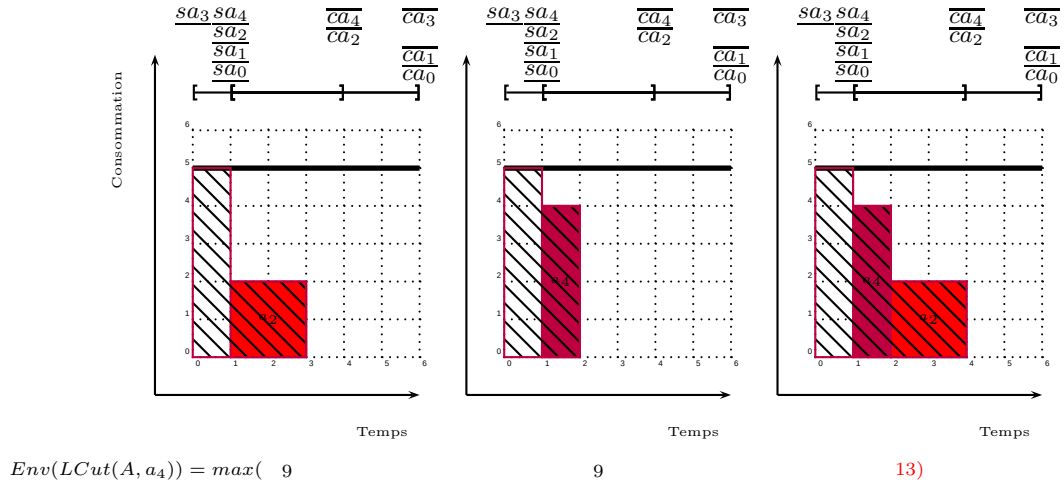


FIGURE 2.9 – L'enveloppe énergétique de l'ensemble $LCut(A, a_4) = \{a_2, a_4\}$.

De manière à détecter les précédences, impliquant une activité $a_j \in A$, nous considérons toutes les activités $a_i \in A$ dont la date de fin au plus tard $\overline{ca_i}$ est inférieure ou égale à $\overline{ca_j}$.

Définition 25 (coupe à gauche) *La coupe à gauche* $LCut(A, a_j)$ *d'un ensemble* A *d'activités, par une activité* $a_j \in A$ *est l'ensemble d'activités tel que :* $LCut(A, a_j) = \{a_i \in A, \overline{ca_i} \leq \overline{ca_j}\}$.

Soit $\Omega \subseteq A$ (dans l'algorithme utilisé, Ω est une coupe à gauche de A par une activité), l'enveloppe énergétique est utilisée pour calculer une borne inférieure $lb(\underline{c}(\Omega))$ pour la fin au plus tôt $\underline{c}(\Omega)$ de Ω. Vilím montre, dans [91], que $lb(\underline{c}(\Omega)) = \left\lceil \frac{Env(\Omega)}{capa} \right\rceil$.

Intuitivement, l'enveloppe énergétique $Env(\Omega)$ est l'énergie totale nécessairement consommée avant la fin de l'exécution de l'ensemble d'activités Ω. Pour une ressource avec une capacité limitée par *capa*, cette énergie peut être répartie sur, au minimum $\left\lceil \frac{Env(\Omega)}{capa} \right\rceil$ points de temps. Ceci donne donc une borne inférieure de la date de fin au plus tôt de l'ensemble d'activités Ω.

Une fois les enveloppes calculées, il est possible de détecter les précédences entre activités.

Règle de précédence 1 *Si* $lb(\underline{c}(LCut(A, a_j) \cup \{a_i\})) > \overline{ca_j}$*, alors* $LCut(A, a_j) \lessdot a_i$*, ce qui est équivalent à : Si* $Env(LCut(A, a_j) \cup \{a_i\}) > Area(0, \overline{ca_j})$*, alors* $LCut(A, a_j) \lessdot a_i$.

Intuitivement, cette règle correspond au fait que l'énergie nécessaire à l'exécution de l'ensemble $LCut(A, a_j)$ doit être répartie entre 0 et $\overline{c}(LCut(A, a_j))$, par définition. Si l'ajout de a_i à cet ensemble crée trop d'énergie pour que l'énergie totale soit contenue dans cet intervalle, alors a_i doit être décalée plus tard, a_i étant la seule activité non contrainte à rester dans cet intervalle.

Notons, par ailleurs, que cette règle se concentre sur le cas particulier des coupes à gauche. Cependant, sa version originelle ([59, 4]) s'écrit avec n'importe quel ensemble d'activités, de la manière suivante.

Règle de précédence 2 *Soient un ensemble* $\Omega \subseteq A$ *et une activité* $a_i \notin A$. *Si* $e(\Omega \cup a_i) > Area(\underline{s}(\Omega \cup a_i), \overline{c}(\Omega))$, *alors* $\Omega \ll a_i$.

Vilím propose une autre règle de détection qui retranscrit les règles *Earliest Finishing Time of First* et *Latest Finishing Time of Last* proposées par Caseau et Laburthe [21].

Règle de précédence 3 *Soient un ensemble* $\Omega \subseteq A$ *et une activité* $a_i \notin A$. *Si* $sa_i + da_i \geq \overline{c}(\Omega)$, *alors* $\Omega \ll a_i$.

Notons que, dans ce cas, $\Omega \subseteq LCut(A, a_i)$. Intuitivement, $sa_i + da_i \geq \overline{c}(\Omega)$ signifie que nécessairement a_i finit après la fin de Ω.

Vilím introduit une structure d'arbre : le Θ-tree, dans [89], dans le cadre de problèmes disjonctifs. Ce sont des arbres binaires qui permettent de calculer la date de fin au plus tôt d'un ensemble d'activités. Dans [91], il étend la définition de cet arbre au problème cumulatif pour obtenir une complexité temporelle pour l'algorithme d'Edge-Finding en $O(k \times n \times log(n))$.

Pour détecter les précédences :

1. il introduit un Θ-tree pour calculer l'enveloppe énergétique d'un ensemble d'activités,
2. il étend cet arbre de manière à détecter les précédences.

Calcul des enveloppes par un Θ-tree Les activités d'un ensemble Θ sont organisées dans un Θ-tree. Les feuilles de cet arbre sont les activités $a_i \in \Theta$. Celles-ci sont triées par dates de début au plus tôt $\underline{sa_i}$.

Nous notons v un noeud de l'arbre, et $Leaves(v)$ les activités correspondant aux feuilles descendantes de v. En chaque noeud v, les quantités suivantes sont maintenues :
- $e_v = e_{Leaves(v)}$ est l'énergie totale de $Leaves(v)$,
- $Env_v = Env(Leaves(v))$ est l'enveloppe énergétique de $Leaves(v)$.

En chaque feuille de l'arbre, correspondant à une activité a_i, on a :
- $e_v = ea_i$,
- $Env_v = Env(\{a_i\})$.

Pour un noeud v, le noeud fils gauche sera noté $left(v)$, et le noeud droit $right(v)$. Les noeuds internes sont ensuite calculés en fonction des noeuds fils :
- $e_v = e_{left(v)} + e_{right(v)}$,
- $Env_v = \max(Env_{left(v)} + e_{right(v)}, Env_{right(v)})$.

L'exemple 14 présente un Θ-tree.

Exemple 14 *La Figure 2.10 représente le Θ-tree correspondant au problème cumulatif que nous avons introduit sur la Figure 2.2. Les activités sont représentées par les feuilles, triées selon leur date de début au plus tôt. en chaque noeud père l'énergie totale et l'enveloppe énergétique sont calculées.*

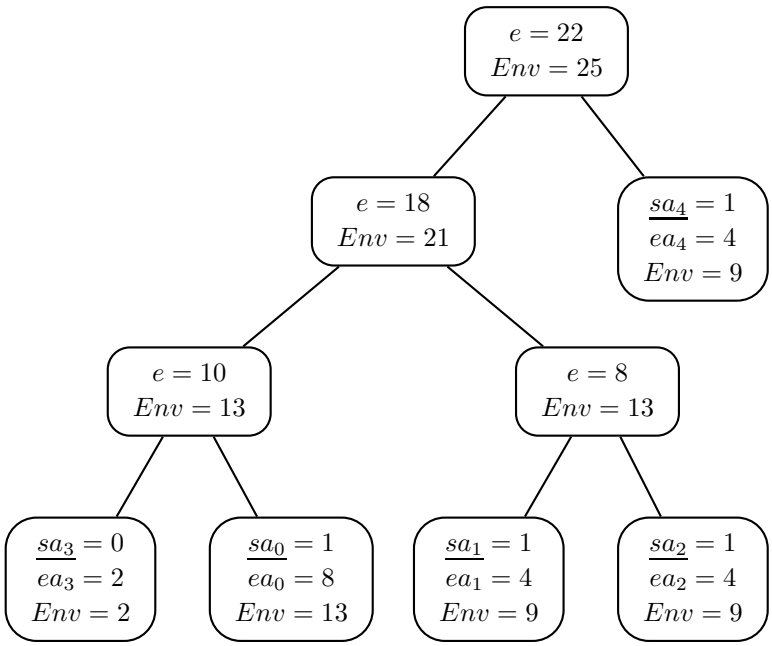


FIGURE 2.10 – Le Θ-tree correspondant au problème cumulatif introduit sur la Figure 2.2. Les 5 activités de ce problème sont représentées par les feuilles de l'arbre. en chaque noeud, l'énergie totale des feuilles descendantes et l'enveloppe énergétique sont calculées

extension du Θ-tree pour détecter les précédences Détecter une précédence revient à trouver pour chaque activité $a_i \in A$ une activité $a_j \in A$, avec une date de fin au plus tard $\overline{ca_j}$ maximale, telle que la règle de précédence 1 puisse détecter : $LCut(A, a_j) \prec a_i$.

L'algorithme d'Edge-Finding présenté par Vilím fonctionne de la manière suivante :

1. il itère sur les activités a_j dans l'ordre décroissant des dates de fin au plus tard $\overline{ca_j}$. L'ensemble Θ représente, pour chaque activité a_j, la coupe à gauche $LCut(A, a_j)$. Notons qu'au début de l'algorithme, $\Theta = A$, car la coupe à gauche de A par l'activité $a_j \in A$ qui a la plus grande date de fin au plus tard est égale à A,
2. pour chaque activité a_j, il maintient un ensemble $\Lambda \subseteq A \backslash LCut(A, a_j)$, qui contient les activités a_i pour lesquelles un ensemble précédant a_i n'a pas encore été trouvé,
3. à chaque pas de l'algorithme il vérifie si une activité $a_i \in \Lambda$ vérifie la règle de précédence 1, ce qui revient à vérifier la proposition suivante : $\max_{a_i \in \Lambda}(Env(LCut(A, a_j) \cup \{a_i\})) > Area(0, \overline{ca_j})$:
   (a) si elle est vérifiée, on cherche l'activité a_i *responsable*, c'est à dire celle qui maximise $Env(LCut(A, a_j) \cup \{a_i\})$,
   (b) sinon, a_j est déplacée de Θ dans Λ, et l'algorithme est réitéré avec une autre activité a_j.

Dans un premier temps, Vilím définit la quantité $Env(\Theta, \Lambda) = \max_{a_i \in \Lambda}(Env(LCut(A, a_j) \cup \{a_i\}))$. Cette quantité correspond à la condition dans la règle de précédence 1.

Pour calculer cette quantité, il propose une structure d'arbre, le $\Theta - \Lambda$-tree, qui est une extension du Θ-tree défini auparavant. Cette nouvelle structure d'arbre est utile pour vérifier la condition énoncée au point 3. C'est un arbre binaire équilibré, dont les feuilles sont des activités ordonnées par dates de début au plus tôt croissantes. Un $\Theta - \Lambda$-tree est ainsi construit pour chaque activité a_j telle qu'on veut tester l'existence d'une activité telle que $LCut(A, a_j)$ la précède.

Dans cette extension, chaque noeud v contient des informations semblables au Θ-tree, et des quantités supplémentaires :

- $e_v = e_{Leaves(v) \cap \Theta}$,
- $e_v^\Lambda = e_{Leaves(v) \cap \Theta} + max_{a_i \in Leaves(v) \cap \Lambda}(ea_i)$,
- $Env_v = Env(Leaves(v) \cap \Theta)$,
- $Env_v^\Lambda = Env(Leaves(v) \cap \Theta, Leaves(v) \cap \Lambda)$.

En chaque feuille v de l'arbre correspondant à une activité $a_i \in \Theta \cup \Lambda$, on a :

- $e_v = ea_i$ si $a_i \in \Theta$, $e_v = 0$ si $a_i \in \Lambda$,
- $e_v^\Lambda = -\infty$ si $a_i \in \Theta$, $ea_i = 0$ si $a_i \in \Lambda$,
- $Env_v = Env(\{a_i\})$ si $a_i \in \Theta$, $-\infty$ si $a_i \in \Lambda$,
- $Env_v^\Lambda = -\infty$ si $a_i \in \Theta$, $Env(\{a_i\})$ si $a_i \in \Lambda$.

Les noeuds internes v sont ensuite calculés en fonction des noeuds fils :

- $e_v = e_{left(v)} + e_{right(v)}$,
- $e_v^\Lambda = \max(e_{left(v)}^\Lambda + e_{right(v)}, e_{left(v)} + e_{right(v)}^\Lambda)$,
- $Env_v = \max(Env_{left(v)} + e_{right(v)}, Env_{right(v)})$,
- $Env_v^\Lambda = \max(Env_{left(v)}^\Lambda + e_{right(v)}, Env_{left(v)} + e_{right(v)}^\Lambda, Env_{right(v)}^\Lambda)$.

A la racine, la quantité Env_v^Λ est égale à $Env(\Theta, \Lambda)$. Pour le $\Theta - \Lambda$-tree correspondant à l'activité $a_j \in A$ nous pouvons alors identifier la présence ou non d'une précédence :

- si, à la racine, $Env_v^\Lambda > Area(0, \overline{ca_j})$, alors il existe une activité $a_i \in \Lambda$, telle que $LCut(A, a_j) \lessdot a_i$, on cherche alors l'activité a_i,
- sinon, aucune précédence n'est détectée.

Pour retrouver l'activité "responsable" a_i telle que $LCut(A, a_j) \lessdot a_i$, nous utilisons les expressions suivantes :

$$responsible_{e^\Lambda}(v) = \begin{cases} responsible_{e^\Lambda}(left(v)) \ si \ e^\Lambda(v) = e_{left(v)}^\Lambda + e_{right(v)} \\ responsible_{e^\Lambda}(right(v)) \ si \ e^\Lambda(v) = e_{right(v)}^\Lambda + e_{left(v)} \end{cases} \tag{2.1}$$

$$responsible_{Env^\Lambda}(v) = \begin{cases} responsible_{Env^\Lambda}(right(v)) \ si \ Env^\Lambda(v) = Env_{right(v)}^\Lambda \\ responsible_{e^\Lambda}(right(v)) \ si \ Env^\Lambda(v) = e_{right(v)}^\Lambda + Env_{left(v)} \\ responsible_{Env^\Lambda}(left(v)) \ si \ Env^\Lambda(v) = Env_{left(v)}^\Lambda + e_{right(v)} \end{cases} \tag{2.2}$$

Nous commençons la recherche par le noeud racine. Nous cherchons d'abord $responsible_{Env^\Lambda}(v)$, et cherchons en chaque noeud, le noeud responsable précédent (selon le cas, $responsible_{Env^\Lambda}(v)$, ou $responsible_{e^\Lambda}(v)$). Nous descendons jusqu'à une feuille qui correspond à l'activité a_i recherchée.

L'exemple 15 illustre une telle recherche.

Exemple 15 *La Figure 2.11 présente un $\Theta - \Lambda$-tree correspondant au problème cumulatif introduit sur la Figure 2.2, avec $\Theta = LCut(A, a_4) = \{a_2, a_4\}$, et $\Lambda = \{a_0, a_1, a_3\}$. On a, dans le noeud racine, $Env_v^\Lambda = Env(\Theta, \Lambda) = 21$. Or, $Area(0, \overline{ca_4}) = capa \times \overline{ca_4} = 20 < Env(\Theta, \Lambda)$. On en déduit donc qu'il existe une activité $a_i \in \Lambda$ telle que $LCut(A, a_4) \lessdot a_i$. Il faut maintenant trouver l'activité a_i en question, de manière à détecter la relation de précédence correspondante. Nous numérotons les noeuds sur la Figure 2.11, pour montrer le chemin emprunté. Nous détaillons le chemin dans les étapes suivantes :*

- $responsible_{Env^\Lambda}(1) = responsible_{Env^\Lambda}(2)$ *car* $Env^\Lambda(1) = Env^\Lambda_{left(1)} + e_{right(1)}$
- $responsible_{Env^\Lambda}(2) = responsible_{Env^\Lambda}(4)$ *car* $Env^\Lambda(2) = Env^\Lambda_{left(2)} + e_{right(2)}$
- $responsible_{Env^\Lambda}(4) = responsible_{Env^\Lambda}(7)$ *car* $Env^\Lambda(4) = Env^\Lambda_{right(4)}$

Le noeud 7 étant une feuille, $responsible_{Env^\Lambda}(7) = 7$, l'activité correspondante est donc l'activité a_0, correspondant à ce noeud.

Ainsi, $LCut(A, a_4) \lessdot a_0$.

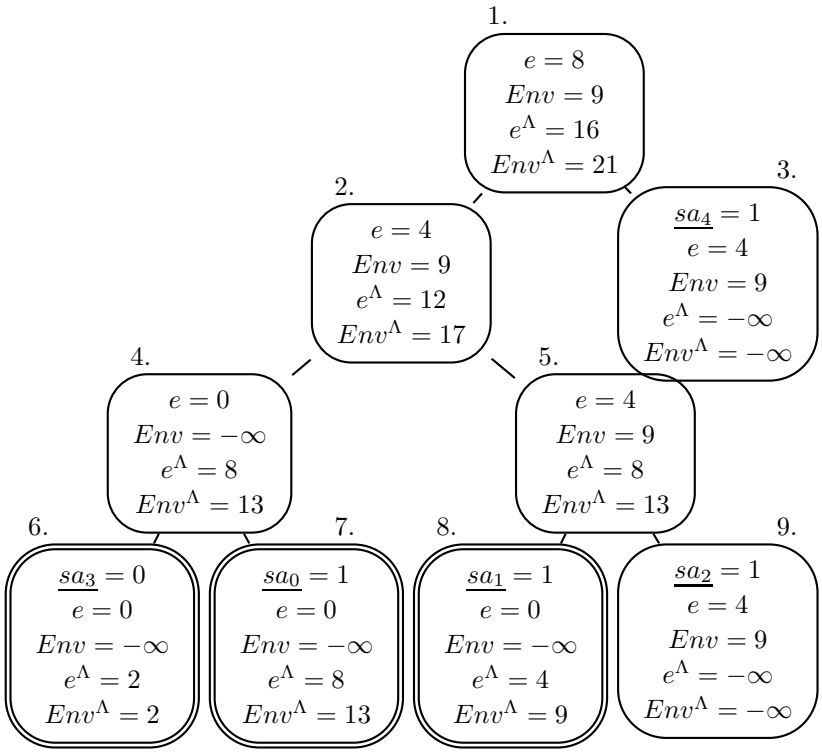


FIGURE 2.11 – Un $\Theta - \Lambda$-tree correspondant au problème cumulatif introduit sur la Figure 2.2. On a : $\Lambda = \{a_0, a_1, a_3\}$ et $\Theta = \{a_2, a_4\}$. Cet arbre permet la détection de la relation $LCut(A, a_4) \lessdot a_0$.

L'algorithme 3 décrit cette procédure de détection des précédences.

Algorithme 3: Algorithme de détection des précédences dans l'Edge-Finding de Vilím

Data : Un ensemble d'activités A.
Result : Un tableau *prec* décrivant des relations de précédence entre activités.

foreach $a_i \in A$ **do**
  $prec[i] := -\infty$;
$\Theta := A$;
$\Lambda := \emptyset$;
forall the $a_j \in A$ *triées par* $\overline{ca_j}$ *décroissantes* **do**
  while $Env(\Theta, \Lambda) > Area(0, \overline{ca_j})$ **do**
    $a_i :=$ activité de Λ responsable de $Env(\Theta, \Lambda)$;
    $prec[i] := \overline{ca_j}$; // ce qui signifie : "$LCut(A, a_j) \ll a_i$"
    $\Lambda := \Lambda \backslash \{a_i\}$;
  $\Theta := \Theta \backslash \{a_j\}$; $\Lambda := \Lambda \cup \{a_j\}$;

Seconde phase : mise à jour des bornes inférieures des dates de début au plus tôt Dans cette seconde phase, il s'agit de mettre à jour la borne inférieure du domaine des variables de début des activités dont on a identifié un ensemble d'activités la précédant.

Cette mise à jour utilise la notion de *compétition* : un ensemble Ω *est en compétition* avec une activité a_i si et seulement si $\underline{e}(\Omega) > Area(\underline{s}(\Omega), \overline{c}(\Omega)) - ra_i \times (\overline{c}(\Omega) - \underline{s}(\Omega))$.

L'idée de ce filtrage est la suivante : si un ensemble d'activités précède une activité donnée a_i, alors l'énergie nécessaire à l'exécution de cet ensemble d'activités doit être placée avant la fin de a_i. La règle de filtrage compte le nombre de point de temps nécessaire pour placer, au plus tôt, cette énergie, et met à jour la date de début de l'activité a_i en conséquence.

Règle de filtrage 2 *Si* $\Omega \ll a_i$ *et* Ω *est en compétition avec* a_i*, alors*

$$[\underline{sa_i}, \underline{s}(\Omega) + \left\lceil \frac{\underline{e}(\Omega) - Area(\underline{s}(\Omega), \overline{c}(\Omega)) + ra_i \times (\overline{c}(\Omega) - \underline{s}(\Omega))}{ra_i} \right\rceil [$$

peut être retiré de $D(sa_i)$.

L'exemple 16 illustre cette règle de filtrage.

Exemple 16 *La Figure 2.14 représente un exemple de mise à jour du début au plus tôt de l'activité a_0, en jaune, avec les activités a_2, en rouge, et a_4, en violet, telles que $D(sa_0) = [1,3]$, $D(ca_0) = [3,6]$, $D(sa_2) = [1,2]$, $D(ca_2) = [3,4]$, $D(sa_4) = [1,3]$, $D(ca_4) = [2,4]$. Comme $Env(LCut(A,a_4) \cup a_0) = 21 > Area(0,\overline{ca_4}) = 20$, une précédence est détectée : $\{a_2,a_4\} = LCut(A,a_4) \lessdot a_0$. $LCut(A,a_4)$ est en compétition avec a_0 car $\underline{e}(\{a_2,a_4\}) = 8$ et $Area(\underline{s}(\{a_2,a_4\}),\overline{c}(\{a_2,a_4\})) - \underline{ra_0} \times (\overline{c}(\{a_2,a_4\}) - \underline{s}(\{a_2,a_4\})) = 15 - 12 = 3$, donc $\underline{e}(\{a_2,a_4\}) = 8 > Area(\underline{s}(\{a_2,a_4\}),\overline{c}(\{a_2,a_4\})) - \underline{ra_0} \times (\overline{c}(\{a_2,a_4\}) - \underline{s}(\{a_2,a_4\}))$. $\underline{sa_0}$ est alors mise à jour : $\underline{sa_0} = 3$*

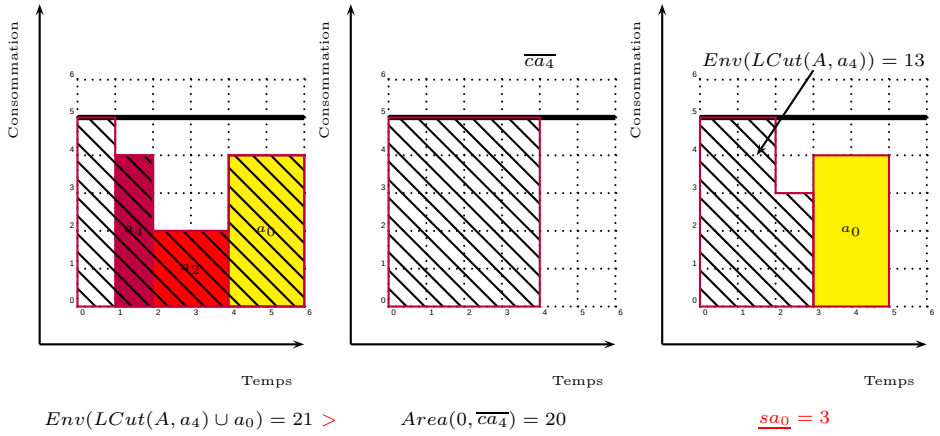


FIGURE 2.12 – Un exemple de mise à jour du début au plus tôt de l'activité a_0, grâce à la relation de précédence $\{a_2,a_4\} = LCut(A,a_4) \lessdot a_0$.

Vilím généralise la règle de filtrage 2 afin de trouver la meilleure mise à jour possible pour une activité pour laquelle a été détectée une précédence.

Règle de filtrage 3 *Soient deux activités $a_i, a_j \in A$ telles que $LCut(A,a_j) \lessdot a_i$, alors l'intervalle $[sa_i, update(a_j, ra_i)[$ peut être retiré de $D(sa_i)$, avec*

$$update(a_j, ra_i) = \max_{\substack{\Omega \subseteq LCut(A,a_j) \\ \underline{e}(\Omega) > Area(\underline{s}(\Omega),\overline{c}(\Omega)) - ra_i \times (\overline{c}(\Omega) - \underline{s}(\Omega))}} (\underline{s}(\Omega) + \left\lceil \frac{\underline{e}(\Omega) - Area(\underline{s}(\Omega),\overline{c}(\Omega)) + ra_i \times (\overline{c}(\Omega) - \underline{s}(\Omega))}{ra_i} \right\rceil)$$

Les activités sont ensuite ordonnées selon les fins au plus tard croissantes.

Considérons l'ensemble $A = \{a_0, a_1, \ldots, a_{n-1}\}$ des activités déjà ordonné comme tel. Nous considérons, pour simplifier, que toutes les dates de fin au plus tard sont différentes (la suite reste vraie dans le cas contraire). Par définition, $LCut(A,a_0) \subsetneq LCut(A,a_1) \subsetneq \ldots \subsetneq LCut(A,a_{n-1})$

A partir de ce tri, il est possible de calculer incrémentalement $update(a_j, r)$. Il n'est pas nécessaire d'itérer pour chaque a_j sur tous les sous-ensembles $\Omega \subseteq LCut(A,a_j)$. Il en effet possible d'utiliser $update(a_{j-1}, r)$, car tous les sous-ensembles de $LCut(A,a_{j-1})$ y ont déjà été considérés. Il ne reste donc qu'à itérer sur les sous-ensembles de $LCut(A,a_j)$ qui ne sont pas des sous-ensembles de $LCut(A,a_{j-1})$.

Par la suite, nous considérons une hauteur d'activité r quelconque.
Ce calcul incrémental se fait de la manière suivante :

$$update(a_j, r) = \begin{cases} diff(a_0, r) \ si \ j = 0 \\ max(update(a_{j-1}, r), diff(a_j, r)) \ si \ j > 1 \end{cases} \tag{2.3}$$

On a également :

$$diff(a_j, r) = \max_{\substack{\Omega \subseteq LCut(A, a_j) \\ \underline{e}(\Omega) > Area(\underline{s}(\Omega), \overline{ca_j}) - r \times (\overline{ca_j} - \underline{s}(\Omega))}} (\underline{s}(\Omega) + \left\lceil \frac{\underline{e}(\Omega) - Area(\underline{s}(\Omega), \overline{ca_j}) + r \times (\overline{ca_j} - \underline{s}(\Omega))}{r} \right\rceil)$$

Nous introduisons la notation $minest(a_j, r)$.

$$minest(a_j, r) = min(\underline{s}(\Omega) | \Omega \subseteq LCut(A, a_j) \& \underline{e}(\Omega) > Area(\underline{s}(\Omega), \overline{ca_j}) - r \times (\overline{ca_j} - \underline{s}(\Omega))$$

Vilím montre que

$$diff(a_j, r) = \max_{\substack{\Omega \subseteq LCut(A, a_j) \\ \underline{s}(\Omega) \leq minest(a_j, r)}} (\underline{s}(\Omega) + \left\lceil \frac{\underline{e}(\Omega) - Area(\underline{s}(\Omega), \overline{ca_j}) + r \times (\overline{ca_j} - \underline{s}(\Omega))}{r} \right\rceil)$$

qui est équivalent à

$$diff(a_j, r) = \left\lceil \frac{Env(a_j, r) - Area(0, \overline{ca_j}) + r \times \overline{ca_j}}{r} \right\rceil$$

avec :

$$Env(a_j, r) = \max_{\substack{\Omega \subseteq LCut(A, a_j) \\ \underline{e}(\Omega) > Area(\underline{s}(\Omega), \overline{ca_j}) - r \times (\overline{ca_j} - \underline{s}(\Omega))}} (Area(0, \underline{s}(\Omega)) + \underline{e}(\Omega))$$

Chaque sous-ensemble Ω peut être séparé en deux parties par $minest(a_j, r)$:

$$update(a_j, r) = \begin{cases} \Omega_1 = \{a_j | a_j \in \Omega \& \underline{sa_j} \leq minest(a_j, r)\} \\ \Omega_2 = \{a_j | a_j \in \Omega \& \underline{sa_j} > minest(a_j, r)\} \end{cases} \tag{2.4}$$

Nous pouvons écrire $Area(0, \underline{s}(\Omega)) + \underline{e}(\Omega) = Area(0, \underline{s}(\Omega_1)) + \underline{e}(\Omega_1) + \underline{e}(\Omega_2)$.
Vilím applique cette égalité aux coupes à gauche et définit

$$update(a_j, r) = \begin{cases} \alpha(a_j, r) = \{a_j | a_j \in LCut(A, a_j) \& \underline{sa_j} \leq minest(a_j, r)\} \\ \beta(a_j, r) = \{a_j | a_j \in LCut(A, a_j) \& \underline{sa_j} > minest(a_j, r)\} \end{cases} \tag{2.5}$$

Ainsi,

$$Env(a_j, r) = \max_{\substack{\Omega_1 \subseteq \alpha(a_j, r) \\ \Omega_2 \subseteq \beta(a_j, r)}} (Area(0, \underline{s}(\Omega_1)) + \underline{e}(\Omega_1) + \underline{e}(\Omega_2))$$

et

$$Env(a_j, r) = Env(\alpha(a_j, r)) + \underline{e}(\beta(a_j, r))$$

Pour un ensemble $LCut(A, a_j)$ donné, $Env(\alpha(a_j, r))$ et $\underline{e}(\beta(a_j, r))$ peuvent être calculés grâce à la structure de $\Theta - tree$ dont un exemple est donné sur la Figure 2.10. Pour cela, le Θ-tree de l'ensemble $LCut(A, a_j)$ est séparé en deux θ-trees distincts :

- nous identifions l'ensemble des feuilles correspondant aux activités $a_i \in A$ telles que $\underline{sa_i} \leq minest(a_j, r)$,
- à partir de ces feuilles nous construisons un arbre contenant uniquement leurs noeuds ancêtres,
- les noeuds ne faisant pas partie de cet arbre forment un autre θ-tree. Les feuilles de ce second arbre correspondent aux activités $a_i \in A$ telles que $\underline{sa_i} > minest(a_j, r)$.

Couper cet arbre de cette manière, c'est à dire recalculer ces deux arbres, a une complexité $O(log(n))$.

Nous rappelons que les activités représentant les feuilles de cet arbre sont triées par dates de début au plus tôt croissantes. Pour effectuer cette coupe, il faut trouver la dernière activité $a_i \in A$ telle que $\underline{sa_i} \leq minest(a_j, r)$.

Pour cela, Vilím définit une quantité Env^r, qui est une variante de l'enveloppe énergétique telle que :

$$Env^r(\Omega) = \max_{\Omega \in \Theta}(Area(0, \underline{s}(\Omega)) - r \times \underline{s}(\Omega) + \underline{e}(\Omega))$$

Cette quantité peut être également calculée par un Θ-tree. En effet, c'est une enveloppe énergétique, à ceci près que la capacité prise en compte n'est pas la capacité du problème cumulatif, mais cette capacité moins la hauteur ra_i de l'activité a_i telle que $LCut(A, a_j) \lessdot a_i$.

A partir de cet arbre coupé, les algorithmes 4 et 5 permettent de calculer les quantités *minest* puis *update*. Grâce à cette dernière, la date de début au plus tôt des activités de A peut être mise à jour.

L'algorithme 5 a une complexité globale de $k \times n \times log(n)$, avec k le nombre de hauteurs différentes des activités de A.

Algorithme 4: Algorithme de calcul de $minest(a_j, r)$.

Data : Un Θ-tree représentant $LCut(A, a_j)$.
Result : $minest(a_j, r)$.

```
v = root; // La racine de l'arbre
E := 0;
while v n'est pas une feuille do
    if Env^r(right(v)) + E > Area(0, ca_j) − r × ca_j then
        v := right(v);
    else
        E := E + e_{right(v)};
        v := left(v);
a_l := activité représentée par la feuille v ;
return sa_l;
```

Algorithme 5: Algorithme de calcul de tous les $update(a_j, r)$.

Result : l'ensemble des quantités *update* servant à mettre à jour les dates de début au plus tôt.

for $r \in \{ra_i \text{ distincts} \mid a_i \in A\}$ **do**
  $\Theta := \emptyset$;
  $upd := -\infty$;
  for $a_j \in A$ *triées par* $\overline{ca_j}$ *croissantes* **do**
    $\Theta := \Theta \cup \{a_j\}$;
    $minest := minest(a_j, r)$; // Calculé dans l'algorithme 4
    $(\alpha, \beta) := Cut(\Theta, minest)$;// L'opération de coupe en $O(log(n))$
    $Env(a_j, r) := \underline{e}(\beta) + Env(\alpha)$;
    $diff := \left\lceil \frac{Env(a_j,r)+Area(0,\overline{ca_j})-r\times\overline{ca_j}}{r} \right\rceil$;
    $upd := max(upd, diff)$;
    $update(a_j, r) := upd$;
    $\Theta := join(\alpha, \beta)$;//jointure entre les deux arbres

2.2.2.2 Algorithme d'Edge Finding de Kameugne et al. [43]

Récemment, Kameugne et al. ont proposé un autre algorithme d'Edge-Finding qui, s'il a une complexité théorique en $O(n^2)$ peut être plus intéressant que la version de Vilím présentée précédemment. En effet, les structures de données employées sont basiques, au contraire des structures de données élaborées de Vilím.

Nous rappelons que nous considérons que les durées, les hauteurs et l'énergie des activités sont constantes. Pour une activité a_i, nous les noterons, respectivement da_i, ra_i, ea_i.

De la même façon que pour l'algorithme de Vilím, l'idée est de détecter des relations de précédence, puis, en fonction de celles-ci, de mettre à jour les dates de début au plus tôt des activités ayant des activités les précédant. Les règles de précédence 2 et 3 sont utilisées pour détecter les précédences. Nous verrons, dans l'algorithme 6, comment sont effectuées ces détections.

Dans un premier temps, nous définissons les intervalles d'activités, introduits par Caseau et Laburthe [21, 22].

Définition 26 (intervalle d'activités) *Soient deux activités $a_L, a_U \in A$. L'intervalle d'activités qui lui est associé est l'ensemble $\Omega_{L,U} = \{a_i \in A | \underline{sa_L} \leq \underline{sa_i} \wedge \overline{ca_i} \leq \overline{ca_U}\}$. Il représente l'ensemble des activités nécessairement entièrement comprises entre le début au plus tôt de a_L et la fin au plus tard de a_U.*

Introduire la notion d'intervalle d'activités permet de raisonner sur les énergies : l'énergie d'un intervalle d'activités $\Omega_{L,U}$ est nécessairement consommée entre $\underline{sa_L}$ et $\overline{ca_U}$.
Nous introduisons alors la notion de *slack*. Le slack correspond à l'énergie maximale que l'on peut ajouter, en plus des activités de l'ensemble, entre le début au plus tôt, et la fin au plus tard d'un intervalle d'activités.

Définition 27 *Soit $\Omega \in A$ un ensemble d'activités. Le* slack *de Ω, que l'on notera SL_Ω est égal à : $SL_\Omega = Area(\underline{s}(\Omega), \overline{c}(\Omega)) - e(\Omega)$.*

Définition 28 *Soient deux activités $a_i, a_U \in A$. $\tau(a_U, a_i)$, avec $\underline{s}(\tau(a_U, a_i)) \leq \underline{sa_i}$, est l'intervalle d'activités de slack minimum : $\forall a_L \in A$, telle que $\underline{sa_L} \leq \underline{sa_i}$ et : $Area(\underline{s}(\tau(a_U, a_i)), \overline{ca_U}) - e(\Omega_{\tau(a_U,a_i),a_U}) \leq Area(\underline{sa_L}, \overline{ca_U}) - e(\Omega_{L,U})$*

L'algorithme 6 détecte les relations $\Omega \lessdot a_i$, en vérifiant si $SL_\Omega < ea_i$ pour tous les ensembles $\Omega = \Omega_{\tau(a_U,a_i),U}$ tels que $\overline{ca_U} < \overline{ca_i}$ et $\underline{s}(\tau(a_U, a_i)) \leq \underline{sa_i}$.

La notion de *densité* est également utilisée dans cet algorithme. La densité correspond au rapport entre l'énergie nécessairement consommée dans un intervalle, est l'énergie disponible (au sens de la Définition 23) dans cet intervalle.

Définition 29 *Soit un ensemble $\Theta \subseteq A$ d'activités. La* densité *de cet ensemble est $Dens_\Theta = \frac{e(\Theta)}{Area(\underline{s}(\Theta), \overline{c}(\Theta))}$*

Définition 30 *Soient deux activités $a_i, a_u \in A$.$\rho(a_u, a_i)$, avec $\underline{sa_i} < \underline{s}(\rho(a_u, a_i))$, définit l'intervalle d'activités de* densité maximum *: $\forall a_l \in A$ telle que $\underline{sa_i} < \underline{sa_l}$, $\frac{e(\Theta_{a_l,a_u})}{Area(\underline{sa_l}, \overline{ca_u})} \leq \frac{e(\Theta_{\rho(a_u,a_j),a_u})}{Area(\underline{s}(\rho(a_u,a_i)), \overline{ca_u})}$*

L'exemple 17 illustre les concepts de slack et de densité.

Exemple 17 *La figure 2.9 montre le calcul du slack et de la densité de l'ensemble* $\{a_2, a_4\}$*, avec* a_2 *en rouge et* a_4 *en violet, tels que* $D(sa_2) = [1, 2]$ *et* $D(sa_4) = [1, 3]$*. Le slack de l'ensemble* $\{a_2, a_4\}$ *est l'énergie laissée libre après avoir placé les activités* a_2 *et* a_4 *dans l'intervalle* $[\underline{s}(\{a_2, a_4\}), \overline{c}(\{a_2, a_4\}) = [1, 4[$*. Ici, le slack, en hâchuré, est donc* $SL(\{a_2, a_4\} = 15 - 8 = 7$*. La densité de cet ensemble est le rapport entre l'énergie* $e(\{a_2, a_4\})$*, représentée en marron, et l'aire disponible dans l'intervalle* $[\underline{s}(\{a_2, a_4\}), \overline{c}(\{a_2, a_4\})[= [1, 4[$*. Ici, la densité est donc* $Dens(\{a_2, a_4\}) = 0.53$*.*

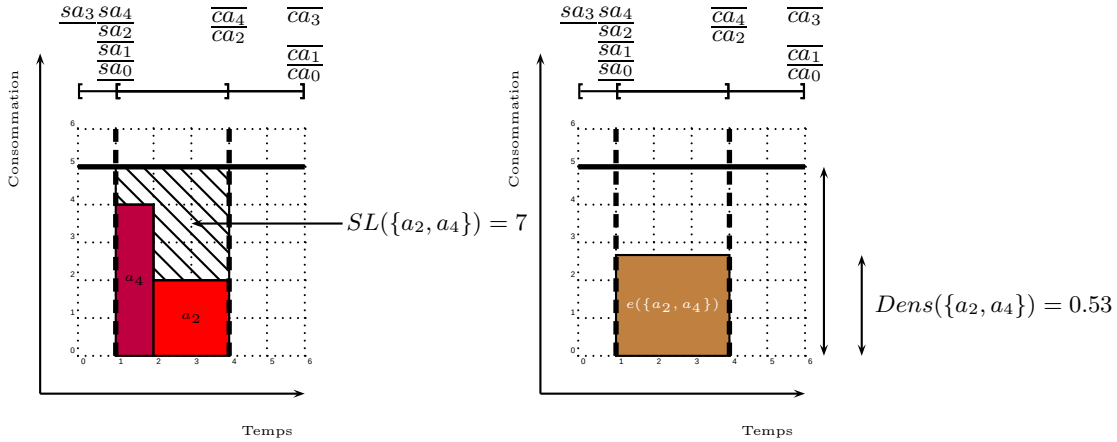


FIGURE 2.13 – Le slack et la densité de l'ensemble $\{a_2, a_4\}$.

Le slack et la densité peuvent mener à une mise à jour du début au plus tôt d'une activité $a_i \in A$. Kameugne et al. montrent les deux points suivants. Soit $\Theta_{l,u}$ l'intervalle d'activités menant à la mise à jour retirant le plus de valeurs dans le domaine de sa_i :

- si $\underline{sa_l} \leq \underline{sa_i}$, alors $\Theta_{l,u}$ est l'intervalle de slack minimum,
- sinon, $\Theta_{l,u}$ est l'intervalle de densité maximum.

L'algorithme ne vérifie pas, à l'avance, si $\underline{sa_l} \leq \underline{sa_i}$. Le slack minimum ou la densité maximum sont donc potentiellement deux méthodes de découverte de la meilleure mise à jour du début au plus tôt de a_i. Intuitivement, la meilleure mise à jour vient de l'intervalle où il y a le moins de place (slack minimum ou densité maximum) pour placer une activité, au regard des activités qui sont nécessairement dans cet intervalle.

En conséquence, l'algorithme calcule les deux quantités, et la mise à jour de $\underline{sa_i}$ se fera en fonction de la meilleure borne (la plus haute) calculée.

Notation 3 *Soient un ensemble* $\Theta \subseteq A$*, et une activité* $a_i \in A$*, de hauteur* ra_i*. On note :*

$$rest(\Theta, ra_i) = \begin{cases} e(\Theta) - Area(\underline{s}(\Theta), \overline{c}(\Theta)) + ra_i \times (\overline{c}(\Theta) - \underline{s}(\Theta)) \ si\ \Theta \neq \emptyset \\ 0\ sinon \end{cases} \tag{2.6}$$

Nous pouvons maintenant écrire les mises à jour possible des débuts d'activités au plus tôt, selon le slack minimum, ou la densité maximum, comme défini plus haut.

Proposition 1 (Mise à jour par le slack minimum) *Pour chaque activité $a_i \in A$, la mise à jour potentielle, en considérant le slack minimum, est :*

$$SLupd(a_i) = \max_{a_U : \overline{ca_U} < \overline{ca_i} \wedge rest(\Omega_{\tau(a_U,a_i),a_U}, ra_i) > 0} \left(\underline{s}(\tau(a_U, a_i)) + \left\lceil rest(\Omega_{\tau(a_U,a_i),a_U}, ra_i) \times \frac{1}{ra_i} \right\rceil \right)$$

Proposition 2 (Mise à jour par la densité maximum) *Pour chaque activité $a_i \in A$, la mise à jour potentielle, en considérant la densité maximum, est :*

$$Dupd(a_i) = \max_{a_U : \overline{ca_U} < \overline{ca_i} \wedge rest(\Theta_{\rho(a_U,a_i),a_U}, ra_i) > 0} \left(\underline{s}(\rho(a_U, a_i)) + \left\lceil rest(\Theta_{\rho(a_U,a_i),a_U}, ra_i) \times \frac{1}{ra_i} \right\rceil \right)$$

L'exemple 18 illustre la mise à jour par le slack minimum.

Exemple 18 *La Figure 2.13 représente un exemple de mise à jour du début au plus tôt de l'activité a_0, en jaune, avec les activités a_2 en rouge, et a_4 en violet, telles que $D(sa_0) = [1, 3]$, $D(ca_0) = [3, 6]$, $D(sa_2) = [1, 2]$, $D(ca_2) = [3, 4]$, $D(sa_4) = [1, 3]$, $D(ca_4) = [2, 4]$. Comme $SL_{\{a_2,a_4\}} < ea_0 = 8$, une précédence est détectée : $\{a_2, a_4\} \lessdot a_0$. L'intervalle d'activités $\{a_2, a_4\}$ est l'intervalle d'activités de slack minimum $\tau(a_4, a_0)$ Ce slack est égal à $Area(\underline{sa_2}, \overline{ca_4}) - e(\{a_2, a_4\}) = 15 - 8 = 7$. Soit $rest(\{a_2, a_4\}, ra_0) = e(\{a_2, a_4\}) - Area(\underline{s}(\{a_2, a_4\}), \overline{c}(\{a_2, a_4\})) + ra_0 \times (\overline{c}(\{a_2, a_4\}) - \underline{s}(\{a_2, a_4\})) = 8 - 15 + 12 = 5$. Alors nous avons $SLUpd(a_0) = 1 + \lceil (5 \times \frac{1}{4}) \rceil = 3$ (les variables de début et de fin activités a_2 et a_4 ayant les mêmes domaines, nous ne faisons pas apparaître le max de la Proposition 1).*

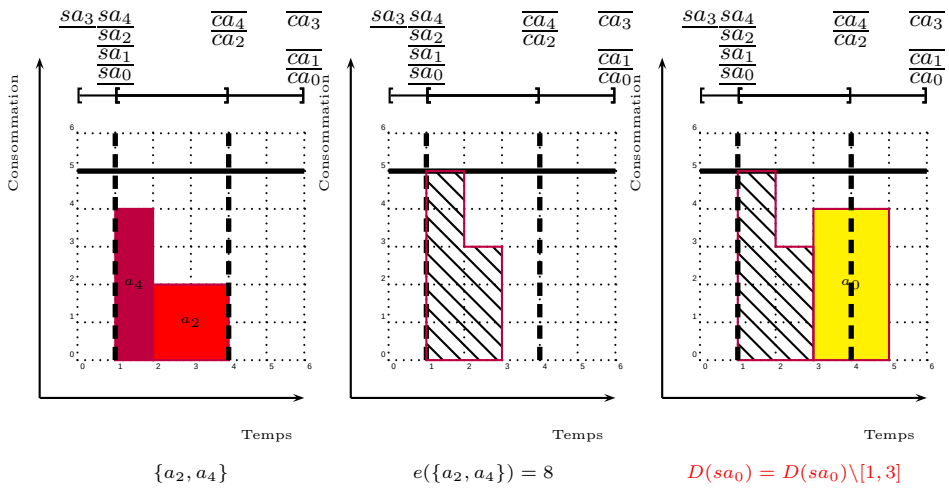


FIGURE 2.14 – Un exemple de mise à jour du début au plus tôt de l'activité a_0, grâce à la relation de précédence $e(\{a_2, a_4\}) \lessdot a_0$.

La procédure d'Edge-Finding de Kameugne et al. est donnée ci-après (algorithme 6).

Algorithme 6: Algorithme d'edge finding quadratique

Data : Un ensemble d'activités A.

Result : Une borne inférieure $lb(a_i)$ est calculée pour chaque début d'activité a_i.

foreach $a_i \in A$ **do**
  $lb(a_i) := \underline{sa_i}$, $Dupd(a_i) := -\infty$, $SLupd(a_i) := -\infty$;
forall the $a_U \in A$ *triées par* $\overline{ca_U}$ *croissantes* **do**
  $Energy := 0$, $maxEnergy := 0$, $\underline{s}(\rho) := -\infty$;
  forall the $a_i \in A$ *triées par* $\underline{sa_i}$ *décroissantes* **do**
    if $\overline{ca_i} \leq \overline{ca_U}$ **then**
      $Energy+ = ea_i$;
      if $\frac{Energy}{\overline{ca_U} - \underline{sa_i}} > \frac{maxEnergy}{\overline{ca_U} - \underline{s}(\rho)}$ **then**
        $maxEnergy := Energy$, $\underline{s}(\rho) := sa_i$;
    else
      $rest := maxEnergy - Area(\underline{s}(\rho), \overline{ca_U}) + ra_i \times (\overline{ca_U} - \underline{s}(\rho))$;
      if $rest > 0$ **then**
        $Dupd(a_i) := \max(Dupd(a_i), \underline{s}(\rho) + \lceil \frac{rest}{ra_i} \rceil)$;
    $E(a_i) := Energy$;
  $minSL := +\infty$, $\underline{s}(\tau) := \overline{ca_U}$;
  forall the $a_i \in A$ *classées par* $\underline{sa_i}$ *croissantes* **do**
    if $Area(\underline{sa_i}, \overline{ca_U}) - E(a_i) < minSL$ **then**
      $\underline{s}(\tau) := \underline{sa_i}$, $minSL := Area(\underline{s}(\tau), \overline{ca_U}) - E(a_i)$;
    if $\overline{ca_i} > \overline{ca_U}$ **then**
      $rest' := ra_i \times (\overline{ca_U} - \underline{s}(\tau)) - minSL$;
      if $\underline{s}(\tau) \leq \overline{ca_U} \wedge rest' > 0$ **then**
        $SLupd(a_i) := \max(SLupd(a_i), \underline{s}(\tau) + \lceil \frac{rest'}{ra_i} \rceil)$;
      if $\underline{sa_i} + da_i \geq \overline{ca_U} \vee minSL - ea_i < 0$ **then**
        $\overline{lb}(a_i) := \max(lb(a_i), Dupd(a_i), SLupd(a_i))$;
forall the $a_i \in A$ **do**
  $\underline{sa_i} = lb(ai)$;

2.2.3 D'autres algorithmes basés sur l'énergie

Overload checking L'*Overload checking* permet de détecter des inconsistances en calculant l'énergie maximale entre deux points de temps. Si celle-ci est inférieure à l'énergie nécessairement déployée par des activités entre ces deux points, alors une inconsistance est détectée.

La règle permettant de la détecter est la suivante.

Proposition 3 *Soit un problème cumulatif défini par un ensemble d'activités A et une capacité capa. S'il existe un ensemble non vide $\Omega \subseteq A$ tel que $\underline{e}(\Omega) > Area(\underline{s}(\Omega), \overline{c}(\Omega))$, alors, une* surcharge (overload) *est détectée, et le problème cumulatif n'admet pas de solution.*

Entre le début au plus tôt et la fin au plus tard d'un ensemble d'activités, toute l'énergie de cet ensemble doit être déployée. Cette règle vérifie qu'entre ces deux points de temps, il y a suffisamment d'espace pour l'accueillir. Sinon, le problème n'a pas de solution.

Wolf et Schrader [94] ont proposé un algorithme d'overload checking en $O(n \times log(n))$, avec n le nombre d'activités du problème cumulatif. Il réutilise la structure de $\Theta - tree$ décrite par Vilím pour les problèmes disjonctifs [88, 89]. Il teste n intervalles.

Auparavant, Caseau et Laburthe [21, 22] ont introduit la notion d'intervalles d'activités et montré qu'il suffisait de vérifier n^2 intervalles d'activités, pour que ce filtrage détecte une inconsistance. Ils proposent également un algorithme de filtrage des dates de début d'activités en utilisant l'énergie calculé dans chaque intervalle d'activités.

Vilím [88] a montré, pour des problèmes disjonctifs, que considérer n intervalles était suffisant.

Not First Not Last De nombreux algorithmes de *not first not last* dans le cas disjonctif existent dans la littérature (voir notamment [81]). Ils sont résumés dans [4, chapitres 2.1.4 et 3.3.5]. Vilím [88] a récemment proposé un algorithme en $O(n \times log(n))$. Schutt et Wolf ont étendu cet algorithme au cas cumulatif, avec une complexité $O(n^3 log(n))$ [80], puis $O(n^2 log(n))$ [79]. La détection *not first* consiste à vérifier si une tâche a_i donnée peut être exécutée avant toutes les activités d'un ensemble non-vide d'activités $\Omega \subseteq A$. Si ce n'est pas le cas, alors, cette activité doit être après au moins une activité de Ω. Ceci permet de mettre à jour le début au plus tôt de l'activité a_i. Symétriquement, la détection *not last* consiste à vérifier si une tâche a_i donnée peut être exécutée après toutes les activités d'un ensemble non-vide d'activités $\Omega \subseteq A$. Si ce n'est pas le cas, alors, cette activité doit être avant au moins une activité de Ω. Ceci permet de mettre à jour le début au plus tard de l'activité a_i.

Timetable Edge-Finding Vilím décrit, dans [92], un algorithme utilisant à la fois le raisonnement d'Edge-Finding, et le raisonnement de time-table (évoqué en section 2.2.1) sur les parties obligatoires. Dans le cadre de cette thèse, bien que nous raisonnions sur les parties obligatoires, nous n'adaptons pas cet algorithme à notre cas, mais ce pourrait être une perspective intéressante.

Chapitre 3

Les problèmes sur-contraints

Sommaire

3.1 Introduction

De nombreux problèmes réels n'admettent aucune solution satisfaisant toutes les contraintes posées. On dit qu'ils sont *sur-contraints*.

Cette thèse traite de problèmes d'ordonnancement cumulatifs avec dépassements de capacité. En un sens, il s'agit de problèmes sur-contraints, car le fait que des dépassements de capacité soient nécessaires implique qu'il n'existe aucune solution au problème cumulatif d'origine, sans dépassements.

Lorsqu'on se trouve confronté à un problème sur-contraint, le but est de fournir un compromis acceptable en pratique.

Pour obtenir ce compromis, on doit considérer une relaxation du problème : les solutions peuvent violer certaines des contraintes du problème d'origine. On souhaite naturellement limiter autant que possible les violations de contraintes, afin d'obtenir une solution aussi proche que possible de la solution idéale qu'il est impossible d'atteindre. En outre, certaines règles métier permettent de faire le tri entre les solutions acceptables d'un point de vue pratique et celles qui ne le sont pas.

Dans ce contexte, une première règle consiste à distinguer dans un problème les *contraintes dures*, qu'il faut nécessairement satisfaire, des *contraintes molles*, qui peuvent être violées.

La programmation par contraintes offre différents cadres pour traiter les problèmes sur-contraints.

L'utilisation d'*explications* permet de résoudre des problèmes sur-contraints via des résolutions successives de CSP. Dès qu'une inconsistance est détectée, sa cause est présentée à l'utilisateur. Celui-ci peut alors relaxer le problème au fur et à mesure que les inconsistances sont détectées. Le but est d'obtenir un CSP consistant et une solution *préférée*.

D'autres approches consistent à résoudre un problème d'optimisation : chaque violation d'une contrainte molle est mesurée par un coût indiquant l'amplitude de la violation, et un critère d'optimisation est défini sur l'ensemble des coûts.

Nous abordons ces différents cadres dans la section suivante.

3.2 Explications

Le cadre des explications, pour des problèmes sur-contraints, est introduit dans [42], puis étendu dans [24]. Pour un problème de programmation par contraintes donné, une *explication* est un sous-ensemble des contraintes impliquées, qui justifie le résultat de la recherche. Les explications sont une trace du comportement de la propagation, lors de la résolution du problème. L'emploi d'explications permet de justifier les réductions de domaines lors des processus de filtrage et de propagation.

Concrètement, à chaque valeur du domaine d'une variable est associée une explication, de manière à ce que, si cette valeur est filtrée, l'explication correspondante soit enregistrée.

L'explication du retrait d'une valeur n'est pas unique, et peut être plus ou moins précise. En effet, une explication contenant toutes les contraintes du problème est toujours vraie : elle contient les contraintes qui mènent au retrait de la valeur. En revanche, nous pouvons souvent extraire de cet ensemble un sous-ensemble de contraintes qui explique lui aussi le retrait d'une valeur. Celui-ci est plus précis car contenant moins de contraintes. De plus, plusieurs sous-ensembles de contraintes, indépendants ou non, peuvent expliquer le retrait d'une valeur. Ces explications peuvent être incomparables. De manière générale, obtenir, pour chaque retrait de valeur, l'explication la plus précise, revient souvent à résoudre un problème NP-difficile.

Lorsque le domaine d'une variable est vide, et qu'aucune solution n'a été trouvée, cela signifie que le problème est sur-contraint. L'usage d'explications permet d'identifier les causes du retrait de toutes les valeurs de ce domaine. Ainsi, l'utilisateur peut, en connaissance de cause, relaxer le problème en enlevant certaines de ces contraintes. Ceci peut être automatisé au cours de la recherche, à condition que l'utilisateur ait prédéfini des préférences entre les contraintes qu'il souhaite voir satisfaites.

Les explications, pour des problèmes sur-contraints, ont notamment été utilisées dans le cadre de problèmes d'ordonnancement et d'emplois du temps [27, 28]. Celles-ci ont également récemment été utilisées dans le cadre de problèmes cumulatifs, afin d'expliquer la propagation de la contrainte Cumulative [78]. Le lecteur intéressé pourra se référer aux travaux de Cambazard [17], à l'article de Verfaillie et Jussien [87], et à l'habilitation à diriger des recherches de Jussien [41].

3.3 Problèmes d'optimisation

Lorsqu'un problème est sur-contraint, certaines contraintes molles peuvent être violées dans une solution. Afin d'obtenir des solutions conservant un intérêt pratique malgré ces violations de contraintes, une approche consiste à transformer le problème de satisfaction en un problème d'optimisation.

Le principe intuitif est le suivant. Une valeur objectif est associée à chaque solution, que l'on cherche à minimiser. Intuitivement, cette valeur indique dans quelle mesure la solution viole les contraintes.

Formellement, il existe de nombreuses manières de définir un tel problème d'optimisation, plus ou moins expressives. Le cas le plus simple est le problème **Max-CSP** [33, 31].

Dans un Max-CSP, l'objectif consiste à minimiser le *nombre* de contraintes violées dans un réseau de contraintes, sans distinction entre les contraintes dures et les contraintes molles. Il est possible de représenter ce problème avec des coûts binaires (0 ou 1) associés à chaque contrainte. Le coût est égal à 1 si et seulement si la contrainte est violée, et à 0 si et seulement si la contrainte est satisfaite. La somme des coûts est la fonction objectif à minimiser. L'exemple 19 présente un problème Max-CSP simple.

Exemple 19 *Soit un réseau de contraintes* $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ *tel que :*
- $\mathcal{X} = \{x_0, x_1, x_2\}$
- $\mathcal{D} = \{[1,3],[1,3],[1,3]\}$
- $\mathcal{C} = \{C_0 : x_0 > x_1, C_1 : x_1 > x_2, C_2 : x_2 > x_0\}$

La Figure 3.1 représente ce problème. Ce problème n'a pas de solutions. On considère alors une relaxation du problème et l'on cherche à violer le moins de contraintes possibles (Max-CSP). Une solution optimale de ce problème Max-CSP a un coût de 1.

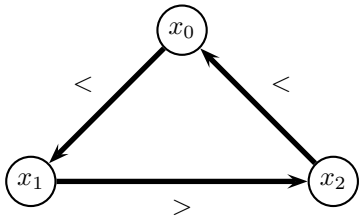


FIGURE 3.1 – Une solution optimale du Max-CSP : $I[\mathcal{X}] = (3,2,1)$. Cette solution a un coût de 1 : 1 contrainte est violée (la contrainte $x_2 > x_0$).

Plusieurs articles de la littérature présentent des algorithmes de filtrage spécifiquement dédiés à ce problème [93, 47, 48, 70]. Max-CSP est un problème somme toute assez académique, car il n'est pas très réaliste de considérer que toutes les contraintes d'un problème sont équivalentes, et parce que toutes les affectations violant une contrainte donnée n'ont pas de raison d'être considérées comme étant toutes équivalentes. Cependant, il existe dans les cas réels des sous-problèmes correspondant à des Max-CSP. Ces algorithmes ont donc un intérêt certain mais ne sont pas suffisants pour traiter les cas concrets.

Pour accroître l'expressivité du paradigme, une idée consiste à associer un coût à chaque contrainte, variant sur un spectre de valeurs plus large que $\{0,1\}$. Dans la majorité des cas pratiques un coût entier convient bien. En effet, les coûts de chaque contrainte doivent ensuite être agrégés dans l'objectif global, et il est donc préférable qu'ils soient comparables entre eux et donc définis selon un ordre total.

L'idée intuitive de cette approche peut être illustrée par un exemple simple : une violation de la contrainte $x < y$ par les affectations $x = 0$ et $y = 0$ est moins importante, d'un point de vue sémantique, qu'une violation de la même contrainte obtenue avec les affectations $x = 1000$ et $y = 0$. On associera donc un coût plus faible à la première violation.

Deux cadres d'étude principaux ont été proposés pour modéliser et résoudre les problèmes de ce type :
- étendre la PPC,
- modéliser les coûts sous forme de variables.

Nous présentons ces deux cadres ainsi que les techniques de résolution qui peuvent y être associées.

3.3.1 Paradigmes

3.3.1.1 Extension de la PPC

Nous présentons les deux paradigmes standard permettant le traitement de problèmes sur-contraints comme des problèmes d'optimisation à l'aide d'une *structure de valuations* externe au réseau de contraintes.

Ces approches étendent la PPC et nécessitent, sous leur forme originelle, l'utilisation d'outils de résolution dédiés.

Les CSP valués [75] et les Semi-ring CSP [14] sont deux paradigmes génériques aux propriétés essentiellement équivalentes [15] [1]. Leur principe commun est d'associer à chaque contrainte C une valuation dépendante des valeurs affectées aux variables de C. Cette valuation est prise dans un ensemble E ordonné. Un critère d'optimisation est alors défini sur l'ensemble des valuations. Nous décrivons brièvement, ci-dessous, les CSP valués.

Lorsque plusieurs contraintes sont violées, la combinaison des valuations est effectuée selon une loi de composition interne $\oplus$. Etant donnés $E, \oplus$ et une relation d'ordre $\succ$, on définit :

Définition 31 : structure de valuation
Une structure de valuation est un triplet $(E, \succ, \oplus)$ *tel que :*
- E *soit un ensemble totalement ordonné par* $\succ$*, muni d'un élément minimum noté* $\bot$ *et un élément maximum noté* $\top$*,*
- E *soit muni d'une loi de composition interne commutative et associative notée* $\oplus$ *qui vérifie :*
  - $\forall a, b, c \in E$ *tels que* $a \succeq c$ *on a :* $(a \oplus b) \succeq (a \oplus c)$*,*
  - *élément neutre :* $\forall a \in E, a \oplus \bot = a$*,*
  - *élément absorbant :* $\forall a \in E, a \oplus \top = \top$*.*

Définition 32 : CSP valué
Un CSP valué $VCSP = \{\mathcal{X}, \mathcal{D}, \mathcal{C}, S, \varphi\}$ *est défini par :*
- *un réseau de contraintes* $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$*,*
- *une structure de valuation* $S = (E, \succ, \oplus)$*,*
- *une application* φ *de* $\mathcal{C}$ *dans* E *associant une valuation à chaque contrainte.*

Dans la littérature, d'autres paradigmes ont été définis. Ils correspondent à des cas particuliers de CSP valués. Nous pouvons notamment citer les CSP pondérés [33], les CSP possibilistes [73] , les CSP lexicographiques [30] et les CSP flous [25].

Le tableau ci-après [62] donne les paramètres correspondant à chaque classe de problème :

Classe	$e \in E$	$\oplus$	$\bot$	$\top$	$\succ$
CSP pondérés	$[0, n]$	$+$	0	∞	$>$
CSP possibilistes	$[0, 1]$	max	0	1	$>$
CSP flous	$[0, 1]$	min	1	0	$<$
CSP lexicographiques	$[0, 1]^* \cup \top$	$\cup$	$\emptyset$	$\top$	lexicographique

Concernant les CSP lexicographiques, $[0, 1]^*$ désigne l'ensemble des multi-ensembles (c'est à dire des ensembles où l'on peut avoir plusieurs occurences des éléments) de nombres entre 0 et 1. Il est complété d'un élément spécifique $\top$; $\cup$ est l'union multi-ensembliste étendue pour traiter $\top$ comme un élément absorbant.

1. Contrairement aux CSP valués, les CSP Semi-Ring permettent la manipulation de coûts de violations qui ne définissent pas un ordre total, i.e., certains d'entre eux peuvent être incomparables.

Pour conclure ce paragraphe, il convient de noter qu'à l'origine l'ensemble de ces paradigmes considéraient exclusivement des contraintes binaires, bien qu'il n'existe pas de restriction majeure à considérer des contraintes d'arité supérieure.

3.3.1.2 Modélisation des violations par des variables

Petit et al. [67, 62] ont démontré que l'ensemble des paradigmes précédemment décrits pouvaient être adaptés au cadre standard de la PPC, sans augmentation de la complexité spatiale.

Soit $\mathcal{R} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ un réseau de contraintes. Nous avons vu que l'on pouvait définir un ensemble de contraintes dures $\mathcal{C}_d \subseteq \mathcal{C}$ et un ensemble de contraintes molles $\mathcal{C}_m \subseteq \mathcal{C}$. Petit et al. proposent d'associer, à chaque contrainte molle $C_{m_i} \in \mathcal{C}_m$, une variable de coût $cost_i$, telle que si C_{m_i} est satisfaite alors $cost_i = 0$, sinon $cost_i > 0$. Pour exprimer une telle conditionnelle il convient de remplacer chaque contrainte molle $C_{m_i} \in \mathcal{C}_m$ par une contrainte exprimant la disjonction suivante.

$$(C_{m_i} \wedge cost_i = 0) \vee (\neg C_{mi} \wedge cost_i > 0)$$

Afin de représenter aussi efficacement que possible une telle disjonction, Petit et al. ont proposé le cadre générique des "*soft global constraints*" [68]. L'idée d'utiliser une contrainte globale dédiée aux problèmes sur-contraints avait été introduite avant ces travaux, par Baptiste et al. [3], dans le cadre spécifique d'un problème d'ordonnancement disjonctif sur-contraint. La contrainte présentée dans cet article modélise un problème entier et n'a a priori pas vocation a être associée à d'autres contraintes.

Dans l'approche proposée par Petit et al., une telle contrainte globale représente directement la disjonction $(C_{m_i} \wedge cost_i = 0) \vee (\neg C_{mi} \wedge cost_i > 0)$. Elle peut être munie d'un algorithme de filtrage efficace, qui exploite sa sémantique [68, 85, 58, 39], ou encore d'algorithmes génériques [51].

Un avantage de ce modèle est le suivant : définir des coûts de violation sous forme de variables permet d'exprimer directement de nombreuses contraintes métier propres à chaque problème. Autrement dit, on peut ajouter au modèle des contraintes impliquant des variables de coût pour exprimer des critères d'acceptation des solutions plus fins qu'une simple fonction objectif [67] ; par exemple des contraintes non linéaires garantissant une répartition aussi uniforme que possible des violations. Ces aspects seront largement discutés dans la suite de ce manuscrit, dans le cadre de l'ordonnancement cumulatif.

Les valeurs de coût des instanciations valides des variables d'une contrainte sont intrinsèquement liées à la sémantique de la "soft global constraint". Aussi, des coûts génériques ont été définis pour ce type de contraintes. Nous en donnons un exemple.

Définition 33 (coût basé sur la distance de Hamming [68]) *Soit une contrainte C portant sur un ensemble de variables $\mathcal{X}$, et $I[\mathcal{X}]$ une instanciation de $\mathcal{X}$. Le coût de violation de C peut être défini comme le nombre de variables de $\mathcal{X}$ dont la valeur affectée dans $I[\mathcal{X}]$ devrait changer pour satisfaire C.*

On peut remarquer qu'avec une telle définition l'expressivité des modèles n'est pas toujours parfaite.

Par exemple, considérons la contrainte SOFTALLDIFFERENT qui enrichit une contrainte ALLDIFFERENT (*c.f.* Définition 10). Supposons qu'on la définit sur un ensemble de quatre variables $\mathcal{X} = \{x_0, x_1, x_2, x_3\}$ et une variable de coût que l'on nomme ici *cost*. Avec la définition 33, les instantiations $I_1[\mathcal{X}] = (0, 0, 1, 1)$ et $I_2[\mathcal{X}] = (0, 1, 1, 1)$ correspondent à la même valeur de coût $obj = 2$. Or si l'on considère le nombre d'inégalités binaires violées, la première instanciation semble plus proche d'un état de satisfaction que la deuxième : deux inégalités violées au lieu de trois.

La question de la définition du coût de violation d'une contrainte est fondamentale pour les problèmes sur-contraints mais aussi en *constraint based local search* [82], par exemple. Elle demeure encore assez ouverte, bien que l'on trouve dans la littérature plusieurs articles proposant des définitions génériques [68, 10, 85]. Ce sujet n'étant pas fondamentalement traité dans cette thèse, nous ne le détaillons pas davantage.

3.3.2 Résolution

Dans cette thèse, nous utiliserons des algorithmes de résolution dédiés aux problèmes d'ordonnancement cumulatifs. Ces algorithmes sont dits *dédiés* car ils sont spécifiques à cette famille de problème, par opposition aux méthodes de résolution dites *génériques*.

Notre choix est essentiellement motivé par l'état de l'art conséquent sur les algorithmes de résolution de problèmes d'ordonnancement, basés sur une prise en compte globale du problème. Cette famille de techniques de résolution nous a semblé plus prometteuse que les méthodes génériques.

De plus, nous avons constaté qu'en pratique des contraintes métier doivent être respectées sur les dépassements, de façon complémentaire au critère d'optimisation. Dans ce contexte, nous avons modélisé les coûts par des variables, ce qui, nous allons le voir, n'est pas immédiatement compatible avec l'ensemble des méthodes génériques.

Il est néanmoins intéressant de donner un aperçu des méthodes génériques, dans le cadre des paradigmes de modélisation de problèmes sur-contraints sous la forme de problèmes d'optimisation précédemment décrits dans ce chapitre.

Ces techniques suivent un schéma de type Branch and Bound (cf. Chapitre 1). Pour décrire leurs principes généraux, nous faisons abstraction du contexte. Nous considérons que certaines affectations de valeurs aux variables engendrent des coûts de violation, et qu'un objectif, noté obj, consiste à minimiser la somme de ces coûts, sans nous soucier davantage de la sémantique associée à ces valeurs de coût.

Une technique de type Branch and Bound effectue des résolutions successives de CSP, en ajoutant à chaque étape une contrainte $obj < UB$, où UB est une valeur maximale pour l'objectif, correspondant à la meilleure solution trouvée précédemment. UB diminue au fur et à mesure de la recherche, jusqu'à ce que le problème n'ait plus de solution améliorante.

Afin de réduire la taille de l'arbre de recherche, il est alors nécessaire de calculer à chaque noeud une *borne inférieure* LB de l'objectif, permettant d'éviter de poursuivre la branche courante si l'on sait qu'elle n'amènera pas à une amélioration de la valeur courante de UB. Idéalement, cette borne inférieure est calculée en prenant en compte l'ensemble des variables du problème, et non pas uniquement celles qui sont déjà instanciées. Une telle borne inférieure LB peut être également facilement intégrée au processus de filtrage, via des mécanismes de regret qui dépendent de la technique employée pour la calculer.

Notons que le calcul d'une bonne borne inférieure est un point clé pour résoudre efficacement des problèmes d'optimisation.

Projection Le principe de base des méthodes génériques consiste à évaluer pour chaque valeur v du domaine $D(x)$ d'une variable x le coût minimum induit par l'affectation de v à x, que l'on notera $cost(x, v)$. Ce coût est calculé en sommant sur l'ensemble des contraintes impliquant la variable x la valeur minimale du coût dans le cas où x prend la valeur v.

Pour chaque contrainte, l'obtention de cette valeur minimale est appelée une *projection* sur la valeur.

La borne inférieure de l'objectif LB peut alors être calculée en agrégeant ces compteurs $cost(x, v)$ de l'ensemble des variables du problème : si toutes les valeurs de $D(x)$ engendrent un coût non nul, alors on peut déduire un coût de violation nécessairement induit par l'affectation de x :

$$cost(x) = \min_{v \in D(x)} cost(x, v)$$

Sachant que chaque variable apparaît en général dans plusieurs contraintes et qu'il ne faut pas compter plusieurs fois le coût d'une même contrainte dans LB (on ferait alors une surestimation de la borne), chaque contrainte n'est prise en compte que pour un unique compteur $cost(x)$. Dans le cas de contraintes binaires, cela correspond à un graphe orienté des contraintes [47, 48] : dans le calcul de LB, une contrainte $C(x, y)$ existe pour une variable x si et seulement si x est l'origine de l'arc (x, y) dans le graphe. Elle est ignorée dans le cas contraire. Dans le cas de contraintes d'arités quelconques, cela correspond à une partition des contraintes issue des variables [70, 62]. Chaque contrainte C appartient alors à exactement une classe de la partition, celle de la variable x pour laquelle C sera prise en compte. Elle sera ignorée par toutes les autres variables.

Ce principe a été introduit dans le cadre du problème Max-CSP [31, 33] sur des contraintes binaires. Dans un Max-CSP le but est de minimiser le *nombre* de violations, ce qui revient à considérer exclusivement des coûts de violation binaires : 0 s'il n'y a pas de violation, 1 dans le cas contraire. Le principe reste valable dans le cas de coûts entiers. Le lecteur intéressé peut se référer aux articles qui décrivent de façon précise les algorithmes, dans le cas de réseaux de contraintes binaires [93, 47, 48] et dans le cas général [70, 62].

L'ensemble de ces techniques peuvent être utilisées pour filtrer les domaines. Nous n'entrerons pas dans les détails car nous n'avons pas utilisé ces techniques. Le principe est d'intégrer la borne inférieure du problème LB lorsqu'on étudie la viabilité d'une valeur $v \in D(x)$ relativement à la variable objectif. Elles peuvent être appliquées aussi bien au cadre des CSP valués qu'au paradigme consistant à représenter les coûts de violation par des variables, avec des complexités temporelles et spatiales identiques quel que soit le paradigme utilisé [62].

Extension et transfert de coûts Schiex [74] a proposé une amélioration de ces techniques, qui outre le principe de projection que nous venons de décrire, exploite un mécanisme *d'extension*, dans le but d'obtenir des bornes inférieures de meilleure qualité. Ce principe a été étendu dans de nombreux travaux (e.g., [23, 49, 71, 50, 51]). Il constitue le standard actuel concernant les méthodes génériques de résolution de CSP valués.

Nous privilégions ici une description intuitive de cette classe d'algorithmes, en nous appuyant sur un exemple de CSP valué montrant le filtrage de valeurs issu d'opérations de projection et d'extension. L'exemple 20 est tiré de l'article de Larrosa et Schiex [49].

Exemple 20 *Soient deux variables x et y pouvant chacune prendre une valeur dans $D(x) = D(y) = \{a, b, c\}$. La Figure 3.2 présente quatre CSP valués équivalents, constitués chacun d'une contrainte binaire.*

Des coûts unaires sont représentés par des entiers pour chaque couple variable-valeur. Ils correspondent à des coûts obtenus suite à des opérations de projection et d'extension précédentes (nous considérons ici que les domaines pouvaient contenir initialement davantage de valeurs).

De plus, les arêtes entre chaque couple de valeurs indiquent le coût de violation (coût binaire) de la contrainte si ses variables sont instanciées avec cette paire de valeurs. Les coûts binaires égaux à 0 sont omis. Le minimum de coût est égal à 0. Nous considérons ici que $UB = 3$, c'est à dire que l'on ne peut pas avoir un coût global supérieur ou égal à 3. Le coût global d'une instanciation est la somme des coûts impliqués (unaires et binaire).

Par exemple, l'instanciation complète (a,b) dans la figure a. a un coût global de 3 : le coût binaire pour (a,b), ajouté au coût unaire pour $I[y] = b$. Cette instanciation est donc interdite.

Nous montrons de quelle manière nous passons d'un CSP au suivant :

- *en a. l'instanciation $I[x] = c$ a un coût unaire de 3. Elle est interdite. Nous retirons donc cette valeur du domaine de x : elle est inconsistante. Nous obtenons alors le CSP b,*
- *en b. l'instanciation $I[y] = c$ a un coût unaire de 2. Nous remarquons également que, quelque soit la valeur donnée à la variable x, le coût binaire d'une instanciation complète avec $I[y] = c$ est au*

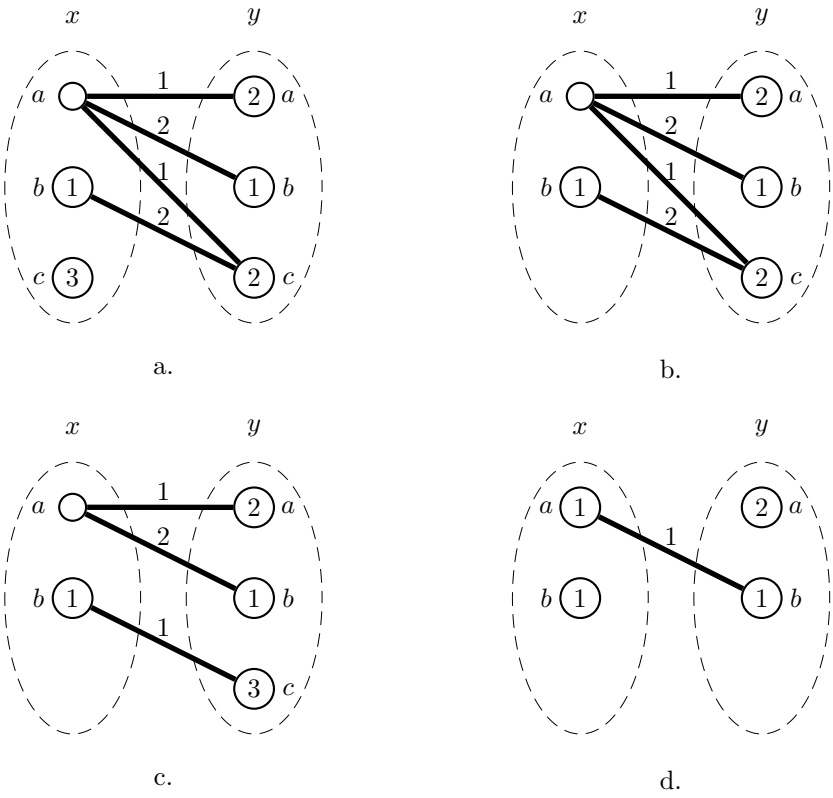


FIGURE 3.2 – Des CSP pondérés équivalents.

minimum de 1. *Nous réalisons alors une opération de projection de* 1 *vers la valeur* c. *pour* y. *Son coût unaire est alors de* 3, *le coût binaire associé au tuple* (a, c) *passe alors à* $1 - 1 = 0$, *et il est de* $2 - 1 = 1$ *pour* (b, c). *Nous obtenons le CSP c*

– *en c. l'instanciation* $I[y] = c$ *a un coût unaire de* 3. *Elle est donc interdite. Nous retirons cette valeur du domaine de* y. *De plus, par projection, le coût unaire de* $I[x] = a$ *passe à* 1, *le coût binaire de* (a, a) *est alors de* $1 - 1 = 0$, *celui de* (a, b) *est de* $2 - 1 = 1$. *Nous obtenons le CSP d.*

Par des opérations d'extension, nous pourrions également obtenir d'autres CSP équivalents.

Le CSP d. montre que le coût global est au minimum de 2. *En effet, toute instanciation de* x *a un coût unaire de* 1 *et toute instanciation de* y *a un coût unaire d'au moins* 1. *Ces coûts peuvent être additionnés pour donner la borne inférieure* LB *de l'objectif global.*

Discussion Les enchaînements d'opérations de projection et d'extension peuvent entraîner des variations non monotones des coûts associés aux tuples des contraintes. Cette propriété rend les méthodes dépendantes du paradigme : il n'est pas possible de représenter directement les coûts de violation des contraintes par des variables, munies d'un domaine classique, qui se réduit au fur et à mesure de la recherche d'une solution.

Une conséquence est qu'il n'est alors pas aisé de poser des contraintes annexes sur les coûts de violation, par exemple pour répartir les valeurs de coût non nulles de manière homogène dans le réseau de contraintes. Une idée pour contourner le problème peut être d'encapsuler cette technique dans une contrainte globale prenant en compte toutes les variables du problème, qui maintient en interne sa propre

représentation des coûts de violation. Même dans ce cas, le lien avec les variables du problème représentant les coûts de violation, indispensables pour poser les contraintes annexes, demeure à notre connaissance une problématique non triviale.

En outre, ces techniques sont basées sur la propriété suivante : pour pouvoir réaliser une opération d'extension, il est nécessaire d'avoir dans le problème au moins une valeur ayant un coût unaire non nul. Plus généralement, pour obtenir une bonne borne inférieure de l'objectif et un filtrage efficace, il convient d'avoir un nombre conséquent de coûts unaires non nuls. Si nous faisons le parallèle avec un problème cumulatif où la contrainte de capacité ne peut pas être respectée, cela signifie qu'à un certain stade de la recherche il existe plusieurs activités non fixées qui, prises chacune de façon isolée, engendrent obligatoirement un dépassement pour certaines valeurs du domaine de la variable représentant leur date de début. Cette propriété n'est pas nécessairement vérifiée, ce qui justifie l'intérêt d'autres types de raisonnements, comme par exemple celui consistant à évaluer globalement la surface requise par un *ensemble* d'activités non fixées (raisonnement énergétique).

Russian Doll Search Une autre technique de résolution générique, basée sur un principe de programmation dynamique, a été proposée par Verfaillie et al [86].

L'idée est de résoudre successivement un problème à deux variables, puis trois, puis quatre, etc., selon un ordre statique fixé, jusqu'à résoudre le problème tout entier. Les contraintes prises en compte dans chaque sous-problème sont celles qui n'impliquent que des variables présentes dans ce sous-problème. Au niveau de la résolution d'un sous-problème à k variables, les valeurs de l'objectif obj des $k-2$ sous-problèmes plus petits sont utilisés comme des bornes inférieures LB de l'objectif courant.

Par exemple, si $k = 10$ et que dans l'arbre de recherche courant 3 variables ont été instanciées, alors on utilise la valeur de l'objectif dans la solution du sous-problème précédent à 7 variables comme borne inférieure LB.

Deuxième partie

Contributions

Chapitre 4

Problèmes d'ordonnancement cumulatif sur-contraint

Sommaire

4.1 Introduction

Dans un problème cumulatif classique, l'objectif est de minimiser l'*horizon de temps*, c'est à dire la date de fin d'activité maximale dans l'ordonnancement, tandis qu'en chaque point de temps la consommation de ressource cumulée ne doit pas dépasser une limite de ressource disponible, la *capacité*.

Cependant, de nombreux cahiers des charges industriels imposent que leurs activités soient effectuées dans une fenêtre de temps de taille bornée. L'horizon de temps est fixé et ne peut être retardé.

Exemple 21 *Nous considérons une usine fonctionnant 5 jours par semaine. Dans ce problème, la ressource considérée est l'équipe d'employés. Les commandes arrivent le lundi matin et doivent être honorées avant le vendredi soir. Tous les employés n'ont pas le même régime horaire, ainsi le nombre d'employés disponibles peut varier selon les périodes de la semaine. Dans le cas où le carnet de commandes est trop important par rapport à la capacité de production obtenue avec l'équipe d'employés présente, certains employés peuvent absorber la surcharge en effectuant des heures supplémentaires, ou encore des intérimaires peuvent ponctuellement renforcer l'équipe. Ces dépassements de capacité ayant un coût, l'entreprise a intérêt à en limiter la quantité. Le problème consiste alors à déterminer un ordonnancement des activités à réaliser dans la semaine qui minimise la surcharge.*

Comme l'illustre l'exemple 21, pour obtenir une solution des dépassements peuvent être tolérés, jusqu'à une certaine limite. Dans de nombreux problèmes la fenêtre de temps doit être partitionnée : la capacité n'est pas la même pour différents intervalles de la partition. On appelle chaque classe de la partition un *intervalle utilisateur*.

La Figure 4.1 représente une telle situation.

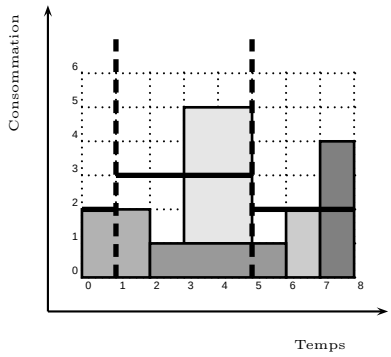


FIGURE 4.1 – Solution d'un problème cumulatif avec un horizon de temps fixé ($m = 8$) et 3 intervalles utilisateur de capacités locales respectives 2, 3 et 2. Chaque activité nécessite un certain nombre d'unités de ressource pour s'exécuter. La première commence en $t = 0$ et finit en $t = 2$, la deuxième commence en $t = 2$ et finit en $t = 6$, la troisième commence en $t = 3$ et finit en $t = 5$, la quatrième commence en $t = 6$ et finit en $t = 7$, la cinquième commence en $t = 7$ et finit en $t = 8$. Il y a deux dépassements de capacité : un dans le second intervalle utilisateur, un dans le troisième.

En outre, dans de nombreuses applications, des règles opérationnelles garantissant la faisabilité pratique doivent être satisfaites. Nous développons certains cas courants dans ce chapitre.

4.2 Modélisation : État de l'art

On trouve dans la littérature des travaux portant sur des problèmes d'ordonnancement sur-contraints de nature différente du problème cumulatif. Baptiste et al. [3] ont introduit un problème d'ordonnancement sur-contraint disjonctif, où l'on cherche à minimiser le nombre d'activités en retard. Benoist et al. [11] ont proposé une étude de cas sur un problème d'ordonnancement sur-contraint de grande taille. Ils ont également proposé une méthode de résolution à base d'explications.

Concernant les problèmes cumulatifs, Petit et Poder introduisent dans [65] les contraintes SOFTCUMULATIVEPOINT [1] et SOFTCUMULATIVESUM, qui peuvent modéliser des problèmes cumulatifs sur-contraints. Nous définissons ces contraintes, puis les illustrons dans l'exemple 22

Définition 34 (SOFTCUMULATIVEPOINT [65]) *Soit un ensemble d'activités A. Ces activités sont ordonnancées entre le point de temps 0 et l'horizon de temps m. Chacune consomme une quantité de ressource positive. La contrainte* SOFTCUMULATIVEPOINT *augmente la contrainte* CUMULATIVE *avec une seconde limite de capacité $ideal\_capa \leq capa$, et pour chaque point de temps $t < m$, une variable entière $cost_t$.* SOFTCUMULATIVE *impose :*

- $C1$ *: pour chaque activité $a_i \in A$, $sa_i + da_i = ca_i$,*
- $C2$ *: à chaque point de temps t, la hauteur du profil est telle que $h_t \leq capa$,*
- $C3$ *: pour chaque point de temps t, $cost_t = max(0, h_t - ideal\_capa)$.*

Définition 35 (SOFTCUMULATIVESUM [65]) *Soit un ensemble d'activités A. Ces activités sont ordonnancées entre le point de temps 0 et l'horizon de temps m. Chacune consomme une quantité de ressource positive. La contrainte* SOFTCUMULATIVESUM *augmente la contrainte* CUMULATIVE *avec une seconde limite*

1. Dans [65], elle est nommée SOFTCUMULATIVE. Pour éviter toute confusion avec la suite nous la renommons SOFTCUMULATIVEPOINT

de capacité ideal_capa ≤ capa, pour chaque point de temps t < m, une variable entière cost$_t$, et une variable entière obj dite "d'objectif". SoftCumulativeSum *impose :*

- *C*1, *C*2 *et C*3 *(voir la Définition 34),*
- *C*4 *: obj* $= \sum_{t\in\{0,\ldots,m-1\}} cost_t$.

Exemple 22 *La Figure 4.2 présente un problème cumulatif à* 5 *activités, avec horizon de temps fixé, modélisé par la contrainte* SoftCumulativePoint. *Aux points de temps* 3, 4 *et* 7, *la capacité ideal_capa est dépassée de* 2, 2 *et* 1 *respectivement. Les variables de coût cost*$_3$, *cost*$_4$, *et cost*$_7$ *modélisent ces dépassements. Ici, cost*$_3 = 2$, *cost*$_4 = 2$ *et cost*$_7 = 1$. *Dans le cas de la contrainte* SoftCumulativeSum, *on a, en plus, obj* $= 5$.

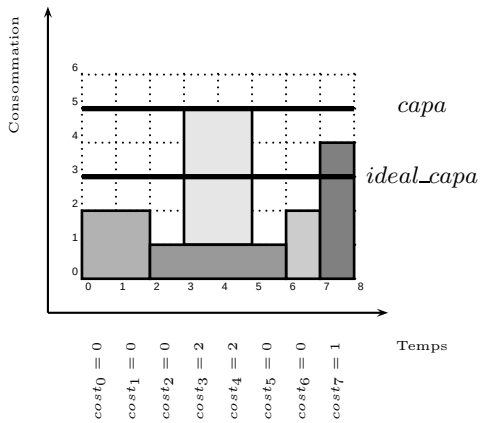


Figure 4.2 – Un problème cumulatif avec un horizon de temps fixé ($m = 8$). Deux capacités sont définies : *ideal_capa* et *capa*. Cet ordonnancement respecte la contrainte SoftCumulativePoint [65]. Les variables de coûts correspondant à chaque point de temps sont indiqués. Ils correspondent au dépassement au dessus de la capacité idéale *ideal_capa*. Dans le cas de la contrainte SoftCumulativeSum [65], on a également une variable objectif, telle que $obj = \sum_{t\in\{0,\ldots,7\}} cost_t = 5$.

Ces deux contraintes ont été introduites dans le but d'illustrer un article sur les problèmes surcontraints. Cette étude montre la nécessité de propager les violations lorsque des contraintes annexes définissent les solutions. Ces travaux ont été menés dans un cadre de "benchmarking". Ils ont un intérêt pratique modéré car, notamment, les intervalles ont tous la même taille (réduite à un point de temps dans les définitions originelles, bien que l'article de Petit et Poder propose une extension à des intervalles de taille quelconque), une unique capacité initiale *ideal_capa* est considérée sur toute la fenêtre de temps, et la définition permettant de quantifier des dépassements de capacité est unique. Dans l'exemple 21, le nombre d'employés disponible varie d'un intervalle utilisateur à un autre. La longueur de chaque intervalle peut elle aussi varier. Par exemple, si on considère des demi-journées alors le matin et l'après midi ne comportent pas nécessairement le même nombre d'heures.

Définir une nouvelle contrainte, plus flexible et mieux adaptée aux cas concrets, s'est donc avéré nécessaire.

4.3 Modélisation : une nouvelle contrainte SoftCumulative

Dans le but de modéliser des instances concrètes de problèmes, nous introduisons une nouvelle contrainte, SoftCumulative, qui étend le travail de Petit et Poder [65]. Nous considérons des problèmes cumulatifs sur-contraints où différentes capacités locales sont définies pour des intervalles de temps disjoints. A chaque intervalle est associée une variable de coût représentant un dépassement (éventuel) de la capacité.

En outre, plusieurs mesures de violation peuvent être utilisées pour quantifier ces dépassements de capacité locales.

Pour rendre des solutions d'un même problème comparables entre elles, nous introduisons une variable d'objectif qui agrège les variables de coût. Celle-ci permettra de quantifier la qualité d'une solution. La valeur de cet objectif global peut être calculée de différentes manières, que nous présentons plus loin.

Enfin, des contraintes métier peuvent être posées sur les variables de coût, comme par exemple une contrainte visant à équilibrer les dépassements sur l'ensemble des intervalles, c'est à dire à éviter qu'ils ne surviennent tous sur un petit sous-ensemble d'intervalles.

4.3.1 Définition de la contrainte SoftCumulative

Nous utilisons les notations introduites dans le chapitre 2 pour définir la contrainte SoftCumulative.

Définition 36 (SoftCumulative) *Soit une ressource avec une capacité limitée par capa et un ensemble A de n activités telles que $\overline{c}(A) \leq m$. Nous définissons :*

- *une partition $P = \{p_0, \ldots, p_{p-1}\}$ de $[0, m[$ en p intervalles consécutifs tels que chaque p_j est défini par son début et sa fin : $p_j = [sp_j, ep_j[$,*
- *une séquence de capacités locales $Loc = [lc_0, \ldots, lc_{p-1}]$ en correspondance avec P, telle qu'une capacité locale $lc_j \leq capa$ est associée avec un intervalle p_j,*
- *une séquence de variables de coût entières $Cost = [cost_0, \ldots, cost_{p-1}]$ en correspondance avec P, telle qu'une variable $cost_j$ soit associée avec l'intervalle p_j,*
- *une variable objectif obj,*
- *une constante $costC \in \{max, sum\}$ indiquant la manière dont la valeur des variables Cost sont calculées : soit le dépassement maximum, soit l'énergie au-dessus de la capacité locale,*
- *une constante $objC \in \{max, sum\}$ indiquant la manière dont la valeur de la variable obj est calculée : soit le maximum, soit la somme des valeurs des variables Cost.*

Étant donnée une instanciation de A, la contrainte SoftCumulative$(A, capa, P, Loc, Cost, obj, costC, objC)$ *est satisfaite si, et seulement si, les quatre conditions suivantes sont satisfaites :*

- *C1 : pour chaque activité $a_i \in A, sa_i + da_i = ca_i$,*
- *C2 : à chaque point de temps t, $h_t \leq capa$,*
- *C3 : $\forall j \in [0, p-1]$: $cost_j = costC_{t \in p_j}(\max(0, h_t - lc_j))$,*
- *C4 : une contrainte sur l'objectif : $obj = objC_{j \in [0,p-1]}(cost_j)$.*

La partition P définit les différentes périodes dans l'horizon temporel. Aucune restriction n'est posée sur la longueur de ces intervalles, si ce n'est qu'ils doivent partitionner exactement l'horizon de temps. Cela permet d'adapter notre contrainte à de nombreuses applications. Dans l'exemple 21, ces différents intervalles correspondent aux différentes périodes de travail.

L'ensemble Loc des capacités locales contient les capacités liées à chaque intervalle. Dans l'exemple 21, ces capacités correspondent à la ressource fixe disponible par période, c'est à dire, le nombre d'employés présent dans chaque période, hors intérimaires.

L'ensemble *Cost* des variables de coût contient les variables de coût correspondant à chaque intervalle. Ces variables représentent la valeur du dépassement de la capacité locale dans chaque intervalle. Dans l'exemple 21, les variables de *Cost* correspondent au nombre d'intérimaires par période.

La variable *obj* agrège les variables de coût et exprime la qualité globale d'une solution. Dans l'exemple 21, la variable *obj* peut représenter, par exemple, la somme du nombre d'intérimaires présents sur l'ensemble de l'horizon de temps (en considérant dans chaque intervalle le dépassement maximal).

Les constantes *costC* et *objC* permettent de définir la manière dont seront calculés les coûts locaux et l'objectif global.

Comparée à SoftCumulativeSum [65], notre contrainte partitionne l'horizon de temps en différents intervalles avec leur longueur et capacité propre. Les dépassements sont quantifiés par une variable de coût sur chaque intervalle, dépendante de *costC*, et une variable objectif agrégeant les coûts et dépendant de *objC*, comme l'illustre l'exemple 23.

Exemple 23 *La Figure 4.3 présente une problèmes cumulatif relaxé à cinq activités. Dans le deuxième, et le troisième intervalle, des dépassements sont constatés. Ces dépassements sont modélisés par les variables Cost. Les valeurs des variables Cost sont ensuite agrégées par la variable obj. Ces calculs dépendent de costC et objC :*

- *si costC= max : $cost_0 = 0$, $cost_1 = 2$, $cost_2 = 2$,*
  - *si objC= max, $obj = 2$,*
  - *si objC= sum, $obj = 4$,*
- *si costC= sum : $cost_0 = 0$, $cost_1 = 4$, $cost_2 = 2$,*
  - *si objC= max, $obj = 4$,*
  - *si objC= sum, $obj = 6$.*

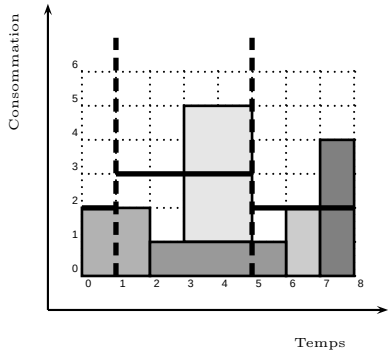


Figure 4.3 – Une instance de SoftCumulative représentant le problème de la Figure 4.1. Si $costC = max$, $cost_0 = 0$, $cost_1 = 2$, $cost_2 = 2$ et si $objC = max$, $obj = 2$, si $objC = sum$, $obj = 4$. Si $costC = sum$, $cost_0 = 0$, $cost_1 = 4$, $cost_2 = 2$. Si $objC = max$, $obj = 4$, si $objC = sum$, $obj = 6$

4.3.2 Problèmes impliquant des contraintes métier

Les problèmes cumulatifs avec un horizon fixé peuvent impliquer des contraintes additionnelles sur les variables *Cost*, de manière à distinguer les solutions ayant un réel intérêt pratique des solutions qui seront rejetées par l'utilisateur final [65].

Nous présentons trois cas usuels, correspondant à des concepts rencontrés en pratique dans les problèmes cumulatifs ainsi que, pour les deux premiers, dans de nombreux autres problèmes de recherche opérationnelle impliquant une séquence de variables de coût :

- répartition équitable des valeurs de coût élevées (notion se ramenant à de l'équilibrage),
- concentration des valeurs de coût élevées sur un nombre de zones limitées,
- lissage du profil cumulatif : limiter les variations importantes de valeurs entre une variable de coût et celle qui la précède (respectivement, qui la suit) directement.

4.3.2.1 Répartition équitable des dépassements

L'exemple 21 est un problème cumulatif à ressource humaine, communément appelé *manpower resource scheduling*. Dans un tel contexte, si des surcharges d'activité surviennent et que celles-ci sont absorbées par le personnel régulier de l'entreprise, par exemple avec des heures supplémentaires, alors il convient de répartir au mieux, dans le temps, les surcharges. Sans ce critère on peut aboutir sur des situations qui sont impossibles à gérer, en pratique, pendant certaines périodes de temps critiques.

Ce besoin a été mis en évidence dans [67]. Plus tard, différentes contraintes globales d'équilibrage ont été définies [61, 66, 72]. Elles peuvent être employées dans des problèmes dont le coeur est modélisé par notre contrainte. Pour les cas simples, des contraintes de cardinalité classiques peuvent être utilisées. Nous rappelons ici une définition de la contrainte AtLeast.

Définition 37 (AtLeast**)** *Étant donnée un entier occ, un ensemble de variables X et une valeur (entière) v, une instanciation $I[X]$ satisfait la contrainte* AtLeast(occ, X, v) *si et seulement si le nombre d'occurrences de la valeur v dans $I[X]$ est supérieur ou égal à occ.*

A l'aide de de la contrainte SoftCumulative une partition de l'ensemble de variables de coût *Cost* est définie. On peut alors, par exemple, imposer que dans chaque classe de cette partition le nombre minimal de variables de coût prenant la valeur 0 soit strictement positif, comme dans l'exemple 24.

Exemple 24 *Soit un problème cumulatif à ressource humaine. On considère un ensemble d'activités A et une capacité absolue capa (sémantiquement la charge maximale possible à un instant donné, avec ou sans surcharge). P désigne les périodes de travail découpant la fenêtre de temps, Loc le nombre d'employés disponibles par période. Nous introduisons des variables Cost, correspondant à la surcharge concrètement effectuée si des employés doivent avoir un rendement supérieur à une capacité locale. Nous fixons costC= max et objC= sum. Pour la clarté du modèle, nous supposons que l'ensemble P contient un nombre d'intervalles multiple de 3. Soit P^c un ensemble de classes d'intervalles, de longueur q, telles que : $P^c = \{P^c_0 = \{p_0, p_1, p_2\}, P^c_1 = \{p_3, p_4, p_5\}, \ldots, P^c_{q-1} = \{p_{3\times(q-1)}, p_{3\times(q-1)+1}, p_{3\times(q-1)+2}\}\}$. Avec une contrainte simple de répartition, le modèle suivant peut représenter ce problème :*

SoftCumulative$(A, capa, P, Loc, Cost, obj, max, sum)$
$\wedge\ \forall i \in [0, q-1]$, AtLeast$(1, \{cost_{3i}, cost_{3i+1}, cost_{3i+2}, \}, 0)$.

La Figure 4.4 est une solution satisfaisant à la fois SoftCumulative *et* $\forall i \in [0, 2]$, AtLeast$(1, \{cost_{3i}, cost_{3i+1}, cost_{3i+2}, \}, 0)$.

4.3.2.2 Lissage des variations de coût

Un besoin fréquent consiste à limiter le nombre de variations importantes dans les dépassements. Pour obtenir des solutions respectant une telle règle, on peut poser la contrainte Smooth$(N, tol, Cost)$ [6] sur les variables dans *Cost*, où N est une variable et *tol* un entier. Récemment, Petit et al. ont proposé un algorithme de filtrage pour cette contrainte [64]. Cet algorithme a une complexité temporelle en la somme des taille des domaines. Le filtrage réalisé est incomplet.

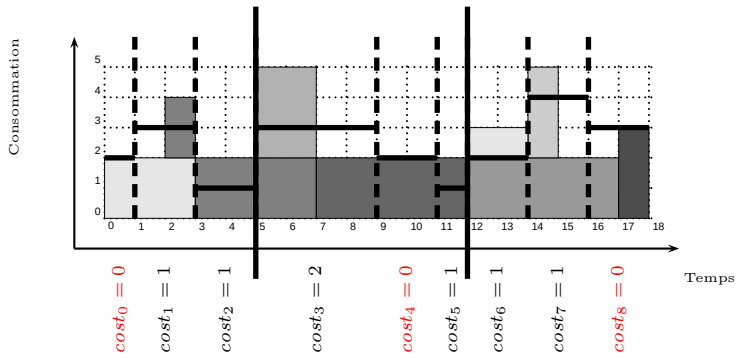


Figure 4.4 – Une solution satisfaisant des contraintes de cardinalité additionnelles sur la partition $\{(cost_0, cost_1, cost_2), (cost_3, cost_4, cost_5), (cost_6, cost_7, cost_8)\}$ avec $costC = max$: au moins un coût égal à 0 dans chaque classe de la partition : elles sont colorées en rouge. Elle satisfait aussi la contrainte smooth$(1, 1, Cost)$. Le nombre de fois où la valeur absolue de la différence entre deux variables de coût consécutives est strictement supérieure à 1 est égale à 1 ($|cost_4 - cost_3| = 2$). Enfin, elle satisfait également la contrainte Focus$(I[Cost], 2, 3, 0)$: nous avons deux séquences de variables de coût à valeur strictement positive, de longueur 3.

Définition 38 (Smooth**)** *Soit N une variable, tol un entier, et $Cost = [cost_0, \ldots, cost_{k-1}]$ un ensemble de variables. La contrainte* Smooth$(N, tol, Cost)$ *est satisfaite si et seulement si le nombre de fois où la contrainte $|cost_{i+1} - cost_i| > tol$ est vérifiée est inférieur ou égal à la valeur de N.*

Dans certains cas, ou l'on n'autorise aucune variation supérieure à un certain seuil, imposer des contraintes primitives de type $|cost_{i+1} - cost_i| \leq tol$ suffit.

Si nous nous référons à nouveau à l'exemple d'un problème à ressource humaine, lorsqu'une entreprise embauche des intérimaires leur nombre ne doit pas varier de manière trop importante d'un jour à l'autre, comme dans l'exemple 25.

Exemple 25 *Nous reprenons l'exemple 24, en considérant cette fois le cas où des employés intérimaires peuvent être embauchés en cas de surcharge. Nous ne cherchons plus à répartir les dépassements des capacités locales. La valeur maximale de dépassement dans un intervalle correspond ici au nombre d'employés embauchés. Une requête utilisateur peut alors consister à éviter des variations trop importantes dans le nombre d'employés supplémentaires embauchés, soit, par exemple, au plus une variation de plus d'un employé entre deux périodes de temps (intervalles) consécutifs. Cette règle s'exprime en autorisant au plus une différence strictement supérieure à un entre deux variables de coûts consécutives dans l'ensemble Cost. Le modèle est alors le suivant :*

SoftCumulative$(A, capa, P, Loc, Cost, obj, max, sum)$
$\wedge$ smooth $(1, 1, Cost)$

La Figure 4.4 est une solution satisfaisant à la fois SoftCumulative *et* Smooth$(N, tol, Cost)$ *avec $N = 1$ et $tol = 1$.*

4.3.2.3 Concentration des coûts

Une autre contrainte imposée dans des cas pratiques peut être la concentration des coûts non nuls. Cette règle apparaît par exemple pour les problèmes cumulatifs où l'on est contraint de louer une machine supplémentaire afin de disposer de plus de ressource pendant certaines périodes. Dans ce contexte, il peut être intéressant de louer par forfait : une location de dix jours engendre un coût financier plus faible que cinq locations de deux jours, toutes séparées dans le temps. On peut utiliser la contrainte Focus introduite par Petit [63], fournie avec un algorithme complet de complexité temporelle linéaire en nombre de variables.

Définition 39 (Focus**)** *Soit une séquence de variables Cost, y_c une variable entière telle que $0 \leq y_c \leq |Cost|$, len un entier tel que $1 \leq len \leq |Cost|$, et $k \geq 0$ un entier. Étant donnée une instanciation $I[Cost] = (v_0, v_1, \ldots, v_{k-1})$, et une valeur v_c affectée à la variable y_c.* Focus$(I[Cost], v_c, len, k)$ *est satisfaite, si et seulement si, il existe un ensemble S de séquences* **disjointes** *de variables consécutives de (Cost) telles que les trois conditions suivante soient satisfaites :*

- $|S| \leq v_c$
- $\forall j \in \{0, 1, \ldots, k-1\}, v_j > k \Leftrightarrow \exists s_i \in S$ *tel que* $cost_j \in s_i$
- $\forall s_i \in S, 1 \leq |s_i| \leq len$

L'exemple 26 présente un exemple d'utilisation de cette contrainte.

Exemple 26 *Supposons que l'on souhaite avoir, dans un problème cumulatif avec dépassements, au plus deux séquences de coûts strictement positifs, chacune de taille au plus trois. Le modèle suivant permet de représenter ce problème :*

SoftCumulative$(A, capa, P, Loc, Cost, obj, max, sum)$
$\wedge$ Focus$(I[Cost], 2, 3, 0)$

La Figure 4.4 est une solution satisfaisant à la fois SoftCumulative *et* Focus$(I[Cost], 2, 3, 0)$.

4.3.2.4 Discussion

Les problèmes cumulatifs avec dépassements de capacité et horizon fixé peuvent être vus comme des problèmes sur-contraints.

Dans [65], les expérimentations montrent que, dans les problèmes sur-contraints avec des contraintes additionnelles sur des variables représentant des violations, propager efficacement ces contraintes additionnelles est indispensable pour résoudre des instances.

Dans ce contexte, de manière générale, il est admis que les violations de contraintes doivent toujours être minimisées et ne sont jamais imposées.

Or, dans notre cas, la présence de contraintes annexe peut générer, par propagation, le besoin de forcer une variable de coût a prendre une valeur non nulle. L'exemple du lissage du profil est sans doute le plus parlant. Considérons le cas où la contrainte primitive $|cost_{i+1} - cost_i| \leq 1$ est imposée. Si $cost_i = 2$ alors, nécessairement, $cost_{i+1} \geq 1$. Ainsi, pour être efficace dans différents contextes, le filtrage de SoftCumulative doit gérer les événements sur les valeurs minimales des variables dans *Cost*.

La Section 5.1.4 présente un algorithme de filtrage dédié à de tels événements.

Il convient de noter que, dans la littérature, Hebrard et al. [40] étudient la propagation d'événements survenant sur la borne inférieure d'une variable de coût représentant une violation, pour les contraintes SoftAllDifferent et SoftAllEqual.

Chapitre 5

Résolution de problèmes d'ordonnancement cumulatif sur-contraint

Sommaire

5.1 Algorithmes de filtrage

Nous détaillons dans ce chapitre des méthodes de résolution des problèmes modélisés par la contrainte SoftCumulative. Par souci de clarté, nous rappelons tout d'abord la signature de la contrainte SoftCumulative décrite dans la Définition 36, page 68, ainsi que sa sémantique.

Soient une ressource avec une capacité limitée par $capa$, un ensemble A de n activités telles que $\overline{c}(A) \leq m$, une partition $P = \{p_0, \ldots, p_{p-1}\}$ de $[0, m[$ en p intervalles consécutifs tels que chaque p_j est défini par son début et sa fin : $p_j = [sp_j, ep_j[$, une séquence de capacités locales $Loc = [lc_0, \ldots, lc_{p-1}]$ en correspondance avec P, telle qu'une capacité locale $lc_j \leq capa$ est associée avec un intervalle p_j, une séquence de variables de coût entières $Cost = [cost_0, \ldots, cost_{p-1}]$ en correspondance avec P, telle qu'une variable $cost_j$ soit associée avec l'intervalle p_j, une variable objectif obj, une constante $costC \in \{max, sum\}$ indiquant la manière dont la valeur des variables $Cost$ sont calculées : soit le dépassement maximum, soit l'énergie au-dessus de la capacité locale, une constante $objC \in \{max, sum\}$ indiquant la manière dont la valeur de la variable obj est calculée : soit le maximum, soit la somme des valeurs des variables $Cost$.

La contrainte SoftCumulative(A, $capa$, P, Loc, $Cost$, obj,$costC$,$objC$) est satisfaite si et seulement si les quatre conditions suivantes sont satisfaites :

- C1 : Pour chaque activité $a_i \in A$, $sa_i + da_i = ca_i$,
- C2 : A chaque point de temps t, $h_t \leq capa$,
- C3 : Pour tout intervalle utilisateur $p_j \in P$: $cost_j = costC_{t \in p_j}(\max(0, h_t - lc_j))$,
- C4 : Une contrainte sur l'objectif : $obj = objC_{j \in [0,p-1]}(cost_j)$.

Les différents algorithmes de filtrage décrits dans ce chapitre sont résumés par la Figure 5.1. Un filtrage est effectué des variables à l'origine d'une flèche vers les variables à la pointe de la flèche.

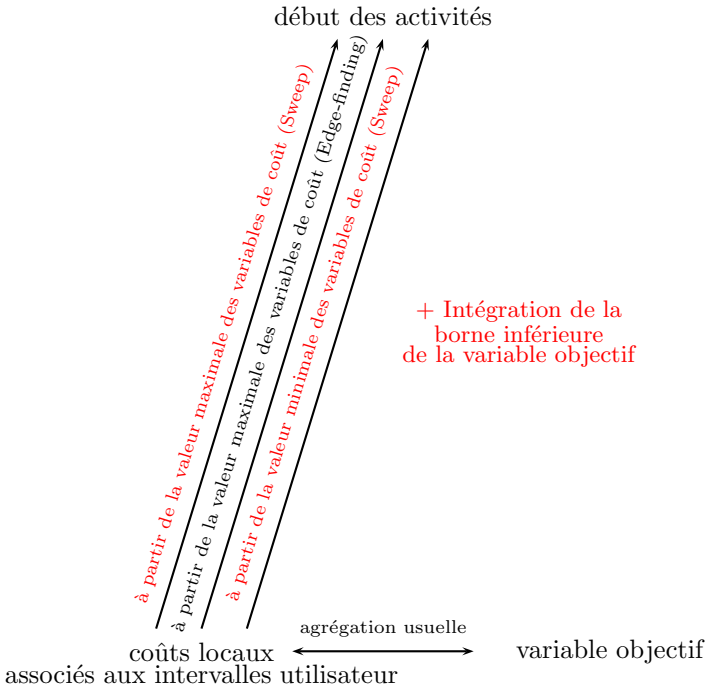


Figure 5.1 – Les différents filtrages de la contrainte SoftCumulative

Notons que certains de ces filtrages peuvent engendrer des trous dans les domaines. Or, dans certains problèmes il n'est pas envisageable de représenter les variables de début des activités avec des domaines énumérés, car le nombre de valeurs (points de temps) serait trop grand. Ce point ne constitue pas vraiment un problème car, dans la plupart des solveurs, il est possible d'imposer que les variables soient représentées exclusivement par leurs bornes. Dans ce cas, si un algorithme effectue une modification en dehors des bornes d'un domaine, celle-ci est ignorée. Cependant, ce filtrage sera redécouvert quand les bornes de la variable considérée seront incluses dans un de ces trous. Nous présentons donc nos algorithmes dans le cas le plus général, où certaines déductions peuvent être réalisées en dehors des bornes des domaines.

Le plan adopté pour présenter ces algorithmes est le suivant :

- dans un premier temps, nous adaptons en section 5.1.1 les algorithmes Sweep et Edge-Finding, présentés en section 2.2, à la contrainte SoftCumulative. Ces algorithmes permettent de filtrer les

dates de début des activités,
- la contrainte SoftCumulative représente un problème qui peut être considéré comme sur-contraint. Le but du problème d'optimisation associé est de minimiser les violations. Il est alors important de calculer des bornes inférieures des coûts associés aux intervalles utilisateur et de l'objectif global. La section 5.1.2 présente des règles de calcul de ces bornes,
- pour résoudre un problème d'optimisation, il peut être également nécessaire d'intégrer au filtrage des bornes inférieures de l'objectif au filtrage. Les bornes inférieures calculées pour l'objectif via un sous-ensemble d'activités peuvent en effet servir à filtrer les variables de début ou de fin des activités ne faisant pas partie de ce sous-ensemble. Cette intégration est présentée en section 5.1.3,
- la discussion menée en section 4.3.2.4, relative aux problèmes impliquant des contraintes métier, met en évidence le besoin de propager non seulement à partir de la valeur maximale des variables de coût, mais également à partir des événements survenant sur les bornes inférieures de leur domaine. Nous présentons une nouvelle procédure en ce sens, en section 5.1.4.

En outre, une stratégie de recherche dédiée au problème représenté par la contrainte SoftCumulative est décrite en section 5.2.

5.1.1 Filtrage à partir des bornes supérieures des domaines des variables de coût

Il est possible d'effectuer des déductions à partir d'événements sur les bornes supérieures des variables de l'ensemble *Cost*. Les interactions des variables *Cost* avec la variable *obj* correspondent à des contraintes classiques de somme ou de maximum. Il n'est donc pas nécessaire de les décrire plus en détail.

De la même manière que la capacité en ressource contraint les activités dans la contrainte Cumulative, le coût au delà de la capacité locale de chaque intervalle utilisateur peut contraindre les activités dans la contrainte SoftCumulative.

Nous nous concentrons sur le filtrage des variables de début des activités. Chaque filtrage sur une date de début d'activité est alors trivialement propagé sur la variable de fin de l'activité, par l'intermédiaire de la contrainte arithmétique C1 de la Définition 36.

5.1.1.1 Sweep pour SoftCumulative

L'algorithme de Sweep pour la contrainte Cumulative décrit en section 2.2.1 peut être adapté à la contrainte SoftCumulative. Contrairement à la contrainte Cumulative, le filtrage de SoftCumulative ne dépend pas d'une unique capacité *capa*. Différents intervalles utilisateur, avec des capacités potentiellement différentes, sont à considérer. La valeur maximale des variables *Cost* et la manière dont sont calculées les valeurs des variables de coût (définie par *costC*) interviennent dans ce filtrage.

La notion de profil cumulatif est néanmoins la même que pour les problèmes cumulatifs classiques. Nous rappelons que le profil cumulatif est construit à partir des parties obligatoires des activités du problème. Un exemple de profil dans le cas SoftCumulative est donné par l'exemple 27.

Exemple 27 *La Figure 5.2 représente un problème à 5 activités et 3 intervalles utilisateur, et son profil cumulatif des parties obligatoires associé. Dans ce problème, les activités a_1 et a_2 ont une partie obligatoire. Le profil cumulatif est construit à partir de celles-ci. L'algorithme de sweep utilisé génère des événements. La figure illustre les événements de partie obligatoire et d'intervalle pris en compte par l'algorithme de Sweep :*
- $evt_0 =< INTERVAL, p_0, 0, 0 >$,
- $evt_1 =< INTERVAL, p_1, 1, 0 >$,

- $evt_2 =< PROFILE, a_1, 2, 1 >$,
- $evt_3 =< PROFILE, a_2, 2, 2 >$,
- $evt_4 =< PROFILE, a_2, 3, -2 >$,
- $evt_5 =< PROFILE, a_1, 5, -1 >$,
- $evt_6 =< INTERVAL, p_2, 5, 0 >$.

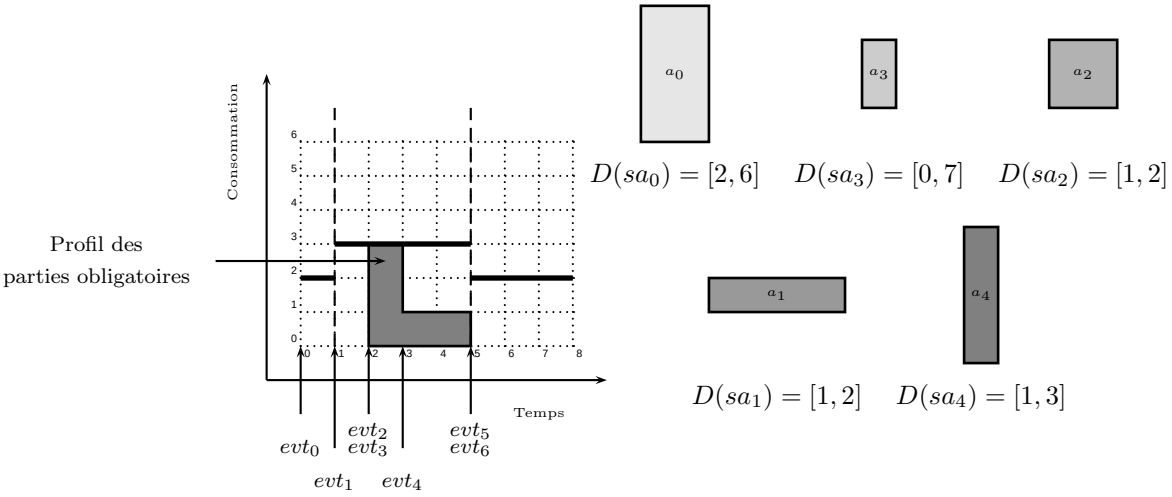


FIGURE 5.2 – Le profil cumulatif des parties obligatoires, pour cinq activités dans un problème cumulatif relaxé, représenté par la contrainte SOFTCUMULATIVE, tel que : $D(sa_0) = [2, 6]$, $D(sa_1) = [1, 2]$, $D(sa_2) = [1, 2]$, $D(sa_3) = [0, 7]$ et $D(sa_4) = [1, 3]$, avec des durées fixées pour chaque activité. Les activités a_1 et a_2 ont une partie obligatoire (respectivement : $[2, 3[$ et $[2, 5[$) qui construisent le profil cumulatif et créent des événements.

Pour prendre en compte la partition de l'horizon de temps, nous ajoutons au Sweep classique une nouvelle classe d'événements : *le début de chaque intervalle utilisateur* $p_j \in P$. Pour un intervalle $p_j \in P$, un tel événement est noté $< INTERVAL, p_j, sp_j, 0 >$. Ceci permet de garantir, lors du balayage, qu'un rectangle $\langle [\delta, \delta'[, sum_h \rangle$ soit compris dans un unique intervalle. De ce fait, la hauteur du profil cumulatif est comparée à une unique capacité locale lc_j et est liée à une variable de coût unique $cost_j$. Ces nouveaux événements ne modifient pas le calcul incrémental de la hauteur sum_h.

Nous décrivons l'algorithme de Sweep adapté, intégrant ces modifications. Nous en donnons une description pour *costC*$= max$. Pour *costC*$= sum$, seules la condition et la règle de filtrage changent.

L'algorithme déplace une ligne verticale Δ sur l'axe de temps. En un balayage, il construit le profil cumulatif et filtre les activités de manière à ne pas dépasser le coût maximum autorisé.

Un *événement* correspond soit au début ou à la fin d'une partie obligatoire, soit *au début d'un intervalle utilisateur*, soit à la date de début au plus tôt $\underline{sa_i}$ d'une activité $a_i \in A$. Tous les événements sont initialement générés et triés dans l'ordre croissant de leur date. On note δ la position courante de Δ.

A chaque pas de l'algorithme, une liste $ActToPrune$ contient les activités à filtrer.

- Les événements de partie obligatoire servent à construire $CumP$. Ces événements, à la date δ, mettent à jour la hauteur sum_h du rectangle courant dans $CumP$, en ajoutant la hauteur si c'est

le début d'une partie obligatoire, ou en l'enlevant si c'est la fin d'une partie obligatoire.
Le premier événement de partie obligatoire *ou d'intervalle utilisateur* dont la date est strictement supérieure à δ définit la fin du rectangle courant. On note cette date δ', et le rectangle correspondant est $\langle[\delta, \delta'[, sum_h\rangle$.

– Dans un intervalle p_j, les événements correspondant aux dates de début au plus tôt $\underline{sa_i}$ telles que $\delta \leq \underline{sa_i} < \delta'$ ajoutent de nouvelles activités candidates au filtrage, en fonction de $\langle[\delta, \delta'[, sum_h\rangle$ et $lc_j + \overline{cost_j}$ (les activités ayant une intersection avec $\langle[\delta, \delta'[, sum_h\rangle$). Notons qu'en chaque intervalle $p_j \in P$, $lc_j + \overline{cost_j}$ représente la hauteur maximal de profil à ne pas dépasser pour respecter la contrainte. Elles sont ajoutées à la liste $ActToPrune$.

En chaque événement de partie obligatoire ou d'intervalle utilisateur, pour chaque activité $a_i \in ActToPrune$ n'ayant pas de partie obligatoire dans le rectangle $\langle[\delta, \delta'[, sum_h\rangle$, si sa hauteur minimale $\underline{ra_i}$ est strictement supérieure à $lc_j + \overline{cost_j} - sum_h$ alors on filtre sa date de début sa_i, de manière à ce que a_i ne puisse plus provoquer un dépassement supérieur au dépassement maximal autorisé.

Si $\overline{ca_i} \leq \delta'$ alors a_i est retirée de la liste $ActToPrune$. Une fois l'étape de filtrage terminée, δ est mise à jour à la valeur δ'.

Nous donnons à présent les règles de filtrage associées aux différentes valeurs possibles pour le paramètre $costC$ de la contrainte SoftCumulative(A, $capa$, P, Loc, $Cost$, obj,$costC$,$objC$).

Cas où le paramètre $costC$ est max. Nous rappelons que, lorsque $costC = max$, le coût dans chaque intervalle utilisateur est égal au dépassement maximal de la capacité locale.

Faire passer une activité a_i par un point de temps t où la hauteur du profil cumulatif des parties obligatoires est sum_h revient à obtenir une hauteur minimale de profil en ce point égale à $sum_h + \underline{ra_i}$.

La hauteur maximale autorisée dans l'intervalle utilisateur courant p_j est égale à la somme de la capacité locale de cet intervalle, et de la borne supérieure de la variable de coût associée à cet intervalle.

Si la hauteur $sum_h + \underline{ra_i}$ est supérieure à la hauteur maximale autorisée $lc_j + \overline{cost_j}$, l'activité a_i ne peut passer par ce point de temps. Dans ce cas, $]t - \underline{da_i}, t[$ peut être retiré du domaine de sa_i.

La règle de filtrage de l'algorithme de Sweep est la suivante :

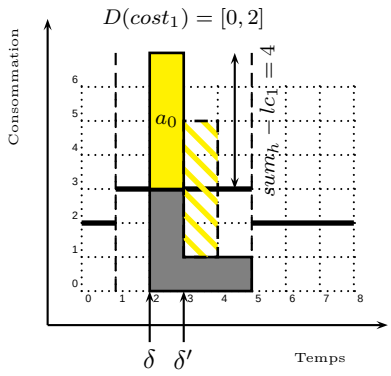
Règle de filtrage 4 ($costC = max$) *Soit une activité $a_i \in ActToPrune$, n'ayant pas de partie obligatoire dans le rectangle $\langle[\delta, \delta'[, sum_h\rangle$, et l'unique intervalle $p_j \in P$ tel que $[\delta, \delta'[\subseteq p_j$. Si $sum_h + \underline{ra_i} > lc_j + \overline{cost_j}$ alors $]\delta - \underline{da_i}, \delta'[$ peut être retiré de $D(sa_i)$.*

Preuve. A chaque pas de l'algorithme de Sweep, toute activité a_i de la liste $ActToPrune$ est telle qu'il existe au moins un point de temps $t \in [\underline{sa_i}, \overline{ca_i}[$ tel que $\delta \leq t < \delta'$. Les Définitions 17 et 21 impliquent que la hauteur h_t de toute solution étendant l'instanciation partielle courante soit telle que $h_t \geq sum_h + \underline{ra_i}$. Ainsi, si $sum_h + \underline{ra_i} - lc_j > \overline{cost_j}$, la contrainte $C3$ de la Définition 36 est violée. Le même raisonnement s'applique en tout point de temps appartenant à $[\delta, \delta'[\cap[\underline{sa_i}, \overline{ca_i}[$. Nous pouvons donc retirer de $D(sa_i)$ toutes les valeurs qui impliquent que l'activité a_i ait au moins un point de temps $t \in [\underline{sa_i}, \overline{ca_i}[$ tel que $\delta \leq t < \delta'$. Ces valeurs constituent l'intervalle $]\delta - \underline{da_i}, \delta'[$. □

L'exemple 28 donne un exemple de filtrage de date de début d'activité dans l'algorithme de Sweep.

Exemple 28 *La Figure 5.3 représente un pas de l'algorithme de Sweep associé à ce problème en considérant le paramètre costC= max. Nous supposons dans cet exemple que $\underline{sa_0} = 2$. Au point de temps 2, l'activité a_0 est donc ajoutée à la liste ActToPrune. Au point de temps 2 sont également associés deux événements de type PROFILE, c'est à dire que le profil des parties obligatoires y est modifié. Après prise en compte de cette modification, le profil a une hauteur $sum_h = 3$ au point de temps 2. Au point de*

temps $\delta' = 3$, il y a un nouvel événement de profil. $a_0 \in ActToPrune$ n'a pas de partie obligatoire dans le rectangle courant $\langle [2,3[, 3\rangle$. L'algorithme de sweep compare donc le dépassement engendré par l'ajout de l'activité a_0 en 2 au dépassement maximal autorisé dans l'intervalle utilisateur considéré, matérialisé par la borne supérieure de la variable de coût $cost_1$. La hauteur du profil si l'activité a_0 est présente au point de temps 2 est $sum_h + \underline{ra_0} = 7$. Or, la hauteur de profil maximale autorisée en ce même point de temps est $lc_1 + \overline{cost_1} = 5 < 7$. Alors a_0 ne peut passer par le point de temps 2 et l'intervalle $[2,3[$ est retiré du domaine de sa_0.



$D(sa_0) = D(sa_0) \backslash [2,3[$

durée égale à 2

FIGURE 5.3 – Un pas de l'algorithme de Sweep pour la SOFTCUMULATIVE, avec $costC = max$. L'activité a_0, représenté en jaune, est telle que $D(sa_0) = [2,6]$, $ra_0 = 4$ et $da_0 = 2$. En $\delta' = 3$, il y a un événement de profil, nous considérons alors la hauteur de profil $sum_h = 3$. Ici, $costC = max$. Nous ajoutons la consommation de ressource de a_0 : $sum_h + \underline{ra_0} = 7$. Cela est strictement supérieur à la capacité locale plus le coût maximal. Alors on peut filtrer la date de début de a_0 : $D(sa_0) = D(sa_0) \setminus [2,3[$.

Cas où le paramètre $costC$ est sum. Nous donnons d'abord une intuition de la règle de filtrage de Sweep lorsque $costC = sum$.

Nous rappelons que, lorsque $costC = sum$, dans chaque intervalle utilisateur p_j, la valeur du coût $cost_j$ doit correspondre à l'énergie totale excédant la capacité locale lc_j. Nous avons alors les deux propriétés suivantes :

1. Dans un intervalle donné $[\delta, \delta'[$ défini dans l'algorithme de Sweep adapté à SOFTCUMULATIVE, la capacité locale lc_j et la hauteur du profil sum_h restent constantes, par construction. En conséquence, l'énergie totale du profil dans cet intervalle est égale à $sum_h \times (\delta' - \delta)$.
2. L'aire totale disponible ne menant pas à un dépassement est égale à $lc_j \times (\delta' - \delta)$.

Les étapes du raisonnement menant à la règle de filtrage sont les suivantes :
- nous essayons d'ajouter une activité a_i qui a une date de début au plus tôt $\underline{sa_i} \in [\delta, \delta'[$,
- si a_i a une durée minimale telle que a_i peut être entièrement contenue dans l'intervalle $[\delta, \delta'[$, alors son énergie maximale dans cet intervalle est $\underline{ea_i}$,
- dans le cas contraire, sa date de fin au plus tôt est à l'extérieur de l'intervalle $[\delta, \delta'[$, et son énergie maximale comprise dans cet intervalle est $(\delta' - \underline{sa_i}) \times \underline{ra_i}$,
- nous comparons l'énergie totale $(\delta' - \delta) \times sum_h + \min((\delta' - \underline{sa_i}) \times \underline{ra_i}, \underline{ea_i})$ à la ressource maximale disponible $lc_j \times (\delta' - \delta) + \overline{cost_j}$,
- nous filtrons sa_i si $\overline{cost_j}$ est dépassée.

Règle de filtrage 5 ($costC = sum$) *Soit une activité $a_i \in ActToPrune$, n'ayant pas de partie obligatoire dans le rectangle $\langle[\delta, \delta'[, sum_h\rangle$ et telle que $\delta \leq \underline{sa_i} < \delta'$.*
Si $(\delta' - \delta) \times sum_h + \min((\delta' - \underline{sa_i}) \times \underline{ra_i}, \underline{ea_i}) > lc_j \times (\delta' - \delta) + \overline{cost_j}$ alors

$$[\underline{sa_i}, \min(\delta', \delta' - \left\lfloor \frac{\overline{cost_j} + (lc_j - sum_h) \times (\delta' - \delta))}{\underline{ra_i}} \right\rfloor)[$$

peut être retiré de $D(sa_i)$.

Preuve. Soit une activité $a_i \in A$ telle que $\delta \leq \underline{sa_i} < \delta'$. Fixons le début de a_i au temps $t = \underline{sa_i}$. Si $\underline{ca_i} < \delta'$, sa contribution énergétique maximale est $\underline{ea_i}$ dans $[\delta, \delta'[$, sinon, elle est égale à $\underline{ra_i} \times (\delta' - t)$. La Définition 17 implique que $(\delta' - \delta) \times h_t \geq (\delta' - \delta) \times (sum_h + \underline{ra_i})$ et $(\delta' - \delta) \times (sum_h + \underline{ra_i}) \geq (\delta' - \delta) \times sum_h + \min((\delta' - t) \times \underline{ra_i}, \underline{ea_i})$. Ainsi, si $(\delta' - \delta) \times sum_h + \min((\delta' - t) \times \underline{ra_i}, \underline{ea_i}) - lc_j \times (\delta' - \delta) > \overline{cost_j}$ (condition de filtrage), la contrainte $C3$ de la Définition 36 est violée. $\overline{cost_j} + (lc_j - sum_h) \times (\delta' - \delta)$ est l'aire disponible restante et $\left\lfloor \frac{\overline{cost_j} + (lc_j - sum_h) \times (\delta' - \delta)}{\underline{ra_i}} \right\rfloor$ est le nombre de points de temps qui peuvent être couverts par a_i dans l'intervalle $[\delta, \delta'[$ sans violer la condition de filtrage. Ceci implique que a_i ne peut commencer avant $\min(\delta', \delta' - \left\lfloor \frac{\overline{cost_j} + (lc_j - sum_h) \times (\delta' - \delta))}{\underline{ra_i}} \right\rfloor)$. L'intervalle $[\underline{sa_i}, \min(\delta', \delta' - \left\lfloor \frac{\overline{cost_j} + (lc_j - sum_h) \times (\delta' - \delta))}{\underline{ra_i}} \right\rfloor)[$ peut donc être retiré de $D(sa_i)$. □

L'exemple 29 illustre le filtrage de la date de début d'une activité, dans le cas $costC = sum$.

Exemple 29 *La Figure 5.4 représente un pas de l'algorithme de Sweep associé à ce problème en considérant le paramètre $costC = sum$. Nous supposons dans cet exemple que $\underline{sa_0} = 3$. Au point de temps 3, l'activité a_0 est donc ajoutée à la liste $ActToPrune$. Au point de temps 3 est associé un événement de type $PROFILE$, c'est à dire que le profil des parties obligatoires y est modifié. Après prise en compte de cette modification, le profil a une hauteur $sum_h = 1$ au point de temps 3.*

Le prochain événement de profil est en $\delta' = 5$. $a_0 \in ActToPrune$ n'a pas de partie obligatoire dans le rectangle courant $\langle[3, 5[, 1\rangle$. L'algorithme de Sweep compare donc le dépassement engendré par l'ajout de l'activité a_0 en 3 au dépassement maximal autorisé dans l'intervalle utilisateur considéré, matérialisé par la borne supérieure de la variable de coût $cost_1$. La hauteur du profil si l'activité a_0 est présente aux points de temps 3 et 4 est $sum_h + \underline{ra_0} = 5$. Or, la capacité lc_1 de cet intervalle utilisateur est de 3. Dans ce cas, l'aire au-dessus de cette capacité est donc égale à 4. Or, le coût maximal est $\overline{cost_1} = 2 < 4$. Alors a_0 ne peut commencer au point de temps 3 et l'intervalle $[3, 4[$ est retiré du domaine de sa_0.

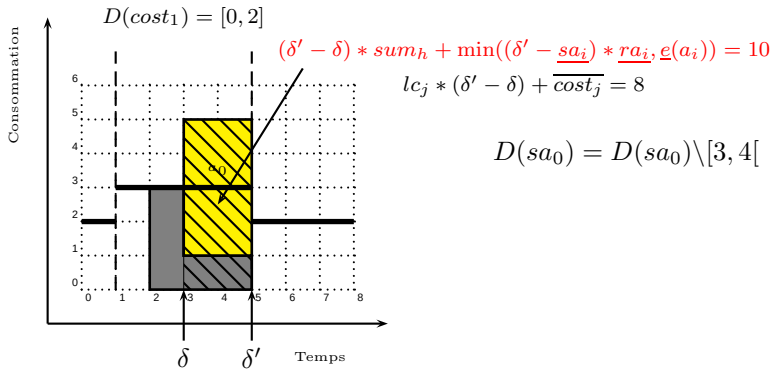


FIGURE 5.4 – Un pas de l'algorithme de Sweep pour SOFTCUMULATIVE, avec $costC = sum$. L'activité a_0, représentée en jaune, est telle que $D(sa_0) = [3, 6]$ (on considère qu'il y a eu un filtrage précédemment), $ra_0 = 4$ et $da_0 = 2$. En $t = 3$, il y a un événement, on considère alors la hauteur $sum_h = 1$. Ici, $costC = sum$. On ajoute la hauteur de a_0 dans l'intervalle $[\delta, \delta'[$: $\min((\delta' - \underline{sa_i}) * \underline{ra_i}, \underline{e}(a_i)) = 10$. Un dépassement de la capacité lc_1 est alors imposé. Il est supérieur au coût maximal autorisé dans cet intervalle utilisateur. On peut alors filtrer la date de début de a_0 : $D(sa_0) = D(sa_0) \backslash [3, 4[$.

Nous rappelons que le tri des événements et la mise à jour des dates de début dans la contrainte CUMULATIVE donnent à l'algorithme de Sweep original une complexité de $O(n \times log(n) + n^2)$. L'ajout d'événements liés à la partition influence la complexité de l'algorithme de tri de ces événements. En revanche, le nombre d'activités à potentiellement mettre à jour en chaque événement reste le même. Il s'en suit une complexité de $O((n + p) \times log(n + p) + n \times (n + p))$.

Preuve. L'algorithme de sweep, adapté à la contrainte SOFTCUMULATIVE génère au plus $3n + p$ événements :

- p événements d'intervalle
- n événements de début au plus tôt d'activités
- au maximum $2n$ événements de partie obligatoire

Ces événements sont ensuite triés. Les meilleurs algorithmes de tri ont une complexité en $O(n \times log(n))$. La complexité de la génération et du tri de ces événements est donc de $O((n + p) \times log(n + p))$.

De plus, en chaque événement de partie obligatoire ou d'intervalle, la liste $ActToPrune$ est parcourue. Celle-ci a une taille maximale de n. Ainsi, le parcours en chaque événement induit une complexité globale pour le parcours (et donc la mise à jour) de $O(n \times (n + p))$.

La complexité globale de cet algorithme est donc $O((n + p) \times log(n + p) + n \times (n + p))$. □

Nous donnons à présent explicitement l'algorithme de Sweep pour SOFTCUMULATIVE. L'algorithme 7 génère la liste d'événements. Cette liste est parcourue dans l'algorithme 8 qui implémente le principe de Sweep. Nous donnons cet algorithme pour $costC = max$. Pour $costC = sum$, l'adaptation se fait en changeant la règle de filtrage. Notons que la capacité d'un intervalle utilisateur peut être récupérée de manière statique pour toute date.

Algorithme 7: Algorithme de génération des événements du Sweep pour SoftCumulative

Data : Un ensemble d'activités A.
Result : Une liste d'événements Evt.

List Evt // La liste d'événements générés
foreach $a_i \in A$ **do**
  if $\underline{sa_i} \neq \overline{sa_i}$ || $\underline{ca_i} \neq \overline{ca_i}$ || $\underline{da_i} \neq \overline{da_i}$ || $\underline{ra_i} \neq \overline{ra_i}$ **then**
    $Evt.add(< PRUNING, a_i, \underline{sa_i}, 0 >)$;
  if $\underline{ca_i} > \overline{sa_i}$ **then**
    $Evt.add(< PROFILE, a_i, \overline{sa_i}, \underline{ra_i} >)$;
    $Evt.add(< PROFILE, a_i, \underline{ca_i}, -\underline{ra_i} >)$;
foreach $p_j \in P$ **do**
  $Evt.add(< INTERVAL, p_j, sp_j, 0 >)$;
return Evt

5.1.1.2 Edge-Finding pour SoftCumulative

Nous avons décrit dans la section 2.2.2 les algorithmes d'Edge Finding. Ces algorithmes permettent de filtrer les dates de début des activités et ce filtrage est incomparable (et complémentaire) à l'algorithme de Sweep. Nous les avons donc adaptés au cas de la contrainte SoftCumulative.

Nous étendons dans un premier temps l'algorithme de Vilím [90] au cas SoftCumulative, puis nous montrons qu'il est également possible d'étendre l'algorithme de Kameugne et al. [43].

Par la suite nous utilisons la notation suivante.

Notation 4 *Pour un point de temps t, $p_j(t) = [sp_j(t), ep_j(t)[$, $cost_j(t)$ et $lc_j(t)$ sont, respectivement, l'intervalle $p_j \in P$, la variable $cost_j \in Cost$ et la capacité locale $lc_j \in Loc$ tels que $t \in p_j$.*

Adaptation de l'algorithme d'Edge-Finding de Vilím [90] Pour étendre l'algorithme de Vilím à la contrainte SoftCumulative, nous considérons les intervalles dans P, des variables de coût $Cost$ et des capacités locales Loc, au lieu d'une seule capacité $capa$.

La Définition 23 (aire disponible) doit alors être adaptée.

Dans un intervalle $\mathcal{I} = [a, b[$, la ressource disponible dépend des capacités locales des intervalles utilisateur et des valeurs maximales des variables $Cost$.

Nous devons distinguer le cas où a et b sont dans le même intervalle utilisateur, et le cas inverse où $p_j(a) \neq p_j(b)$.

Définition 40 (Aire disponible) *Soit un intervalle de temps $\mathcal{I} = [a, b[$, $Area(a, b)$ est la quantité maximale de ressource disponible dans cet intervalle. L'aire disponible est l'aire occupable par des activités en atteignant la borne maximale de chacune des variables de coût impliquées dans $\mathcal{I}$. Nous distinguons deux cas :*

- *si $costC = max$:*
  - *si $p_j(a) = p_j(b)$, $Area(a, b) = (b - a) \times (lc_j(a) + \overline{cost_j}(a))$,*
  - *sinon, $Area(a, b) = (ep_j(a) - a) \times (lc_j(a) + \overline{cost_j}(a)) + (b - sp_j(b)) \times (lc_j(b) + \overline{cost_j}(b)) + \sum_{p_i \subseteq [ep_j(a), sp_j(b)[} (ep_i - sp_i) \times (lc_i + \overline{cost_i})$.*
- *si $costC = sum$:*
  - *si $p_j(a) = p_j(b)$, $Area(a, b) = (b - a) \times lc_j(a) + \overline{cost_j}(a)$,*
  - *sinon, $Area(a, b) = (ep_j(a) - a) \times lc_j(a) + \overline{cost_j}(a) + (b - sp_j(b)) \times lc_j(b) + \overline{cost_j}(b) + \sum_{p_i \subseteq [ep_j(a), sp_j(b)[} ((ep_i - sp_i) \times lc_i + \overline{cost_i})$.*

Il convient de noter suite à cette définition que, trivialement :

- soient 3 dates a, b, c telles que $a < b < c$. On a alors $Area(a, b) < Area(a, c)$. La réciproque est vraie,
- soient 3 dates a, b, c telles que $a < b < c$. On a alors $Area(b, c) < Area(a, c)$. La réciproque est vraie,
- soient 3 dates a, b, c. On a alors $Area(a, b) + Area(b, c) = Area(a, c)$.

Pour illustrer les principes intuitifs de calcul d'aire disponible de la Définition 40, nous considérons l'exemple 30.

Exemple 30 *La Figure 5.5 représente l'aire disponible entre deux points de temps dans le cas d'un problème à trois intervalles utilisateur avec costC= max. Nous rappelons qu'à chaque intervalle utilisateur est associé une variable de coût. Nous calculons l'aire disponible entre deux points de temps 3 et 7. Les points de temps 3 à 5 sont dans le second intervalle, de capacité $lc_1 = 3$, associé à la variable de coût $cost_1$, tel que $D(cost_1) = [0, 2]$. Avec costC= max, en chaque point de temps, le profil a une hauteur maximale de $lc_1 + \overline{cost_1} = 5$. Alors, entre 3 et 5, l'aire disponible est de 10. De la même manière, entre*

5 *et* 7 *le profil a une hauteur maximale de* 3*, l'aire disponible est alors de* 6 *entre ces deux points. L'aire totale disponible entre les points de temps* 3 *et* 7 *est donc* $Area(3, 7) = 10 + 6 = 16$.

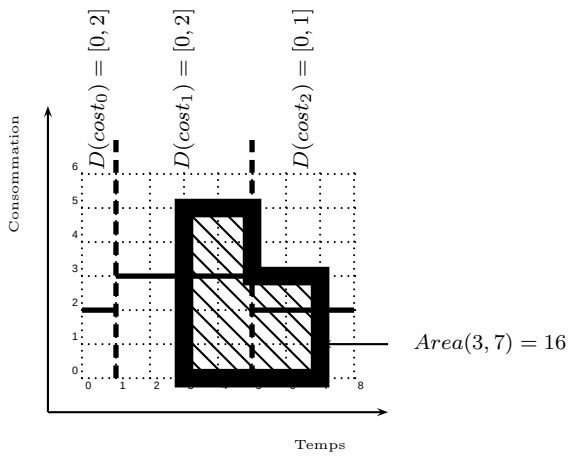


FIGURE 5.5 – Aire disponible entre deux points de temps 3 et 7, avec $costC = max$. Dans ce cas, $Area(3, 7) = 16$.

La Figure 5.6 représente le cas costC= sum. La capacité de chaque intervalle peut être dépassée par une aire, au maximum, égale à la borne supérieure de la variable de coût associée à cet intervalle. Entre 3 *et* 5*, l'aire disponible est donc égale à l'aire disponible sous la capacité, soit* 6*, à laquelle il faut ajouter* $\overline{cost_1} = 2$*. De la même manière, l'aire disponible entre* 5 *et* 7 *est égale à* 5*. Dans ce cas,* $Area(3, 7) = 8 + 5 = 13$.

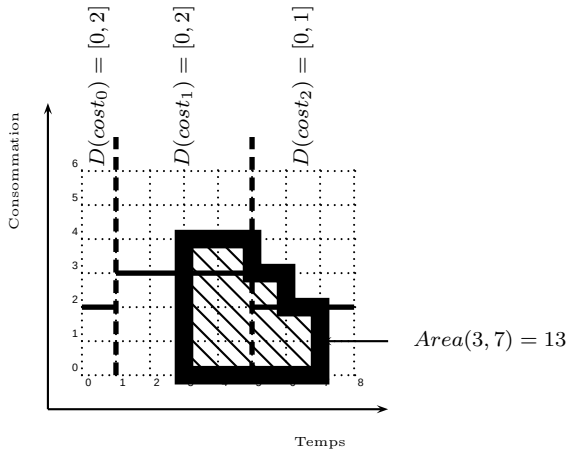


FIGURE 5.6 – Aire disponible entre deux points de temps 3 et 7, avec $costC = sum$. Dans ce cas, $Area(3, 7) = 13$.

Nous introduisons la notion d'*Aire libre* dans le cas SoftCumulative, complémentaire à la notion d'aire disponible. Contrairement à l'aire disponible, elle comptabilise l'énergie maximalement consommée dans un intervalle *sans créer de dépassement de capacité*. L'exemple 31 illustre cette notion.

Notons que le calcul de l'aire libre est indépendant de *costC*. En effet, *costC* influe sur le calcul des coûts et, dans le cas de l'aire libre, les coûts ne sont pas pris en compte.

Définition 41 (Aire libre) *Soit un intervalle de temps $I = [a, b[$, on note $FreeArea(a, b)$ la quantité maximale de ressource disponible sans créer de dépassement des capacités locales des intervalles impliqués :*

- *si $p_j(a) = p_j(b)$, $FreeArea(a, b) = (b - a) \times lc_j(a)$,*
- *sinon, $FreeArea(a, b) = (ep_j(a) - a) \times lc_j(a) + (b - sp_j(b)) \times lc_j(b) + \sum_{p_i \subseteq [ep_j(a), sp_j(b)[} (ep_i - sp_i) \times lc_i$.*

Exemple 31 *La Figure 5.7 représente l'aire libre entre deux points de temps dans le cas d'un problème* SoftCumulative *à trois intervalles. Entre les points de temps 3 et 5 ; la capacité locale est $lc_1 = 3$. En chaque point de temps, le profil cumulatif peut atteindre celle-ci sans créer de dépassement. L'aire libre entre ces 2 points de temps est donc $2 \times 3 = 6$. De la même manière, l'aire libre entre 5 et 7 est égale à 4. On obtient alors $FreeArea(3, 7) = 10$.*

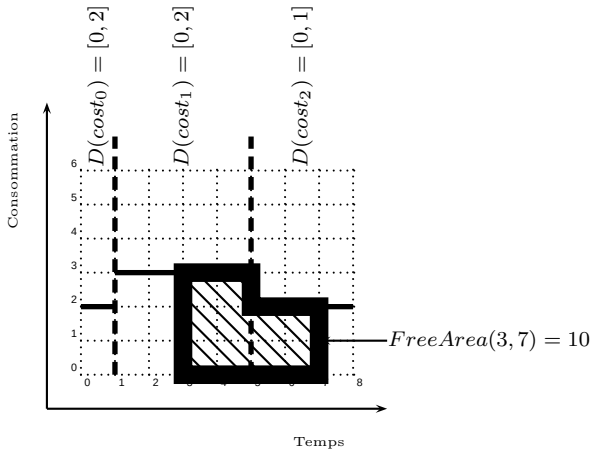


Figure 5.7 – Aire libre entre deux points de temps 3 et 7.

Les calculs de l'enveloppe énergétique et de la coupe à gauche (voir les Définitions 23 et 25) restent les mêmes que dans le cas de la contrainte Cumulative, excepté le fait que l'aire disponible est maintenant calculée conformément à la Définition 40.

Nous rappelons que l'enveloppe énergétique d'un ensemble d'activités $\Omega \subseteq A$, est $Env(\Omega) = \max_{\Theta \subseteq \Omega}(Area(0, \underline{s}(\Theta)) + \underline{e}(\Theta))$. De plus , la coupe à gauche d'un ensemble d'activités A est l'ensemble $LCut(A, a_j) = \{a_i \in A, \overline{ca_i} \leq \overline{ca_j}\}$. L'exemple 32 illustre le calcul de l'enveloppe énergétique d'un ensemble d'activités.

Exemple 32 *La Figure 5.8 représente une enveloppe énergétique dans le cas de la contrainte* SoftCumulative. *Nous considérons ici un ensemble de 2 activités (qui représente une coupe à gauche). Celles-ci sont les activités a_2 et a_4 introduites dans le problème représenté par la Figure 5.2. L'enveloppe énergétique de cet ensemble est une surestimation maximale de l'aire occupée avant la fin au plus tôt de l'ensemble. Pour la calculer, tous les sous-ensembles de $\{a_2, a_4\}$ sont considérés. En chaque sous-ensemble est calculée l'aire disponible avant le début au plus tôt du sous-ensemble, à laquelle est ajoutée l'énergie du sous-ensemble d'activités. Le maximum de cette somme donne l'enveloppe énergétique. Ici, le sous-ensemble $\{a_2, a_4\}$ donne le maximum, et $Env(\{a_2, a_4\}) = 12$.*

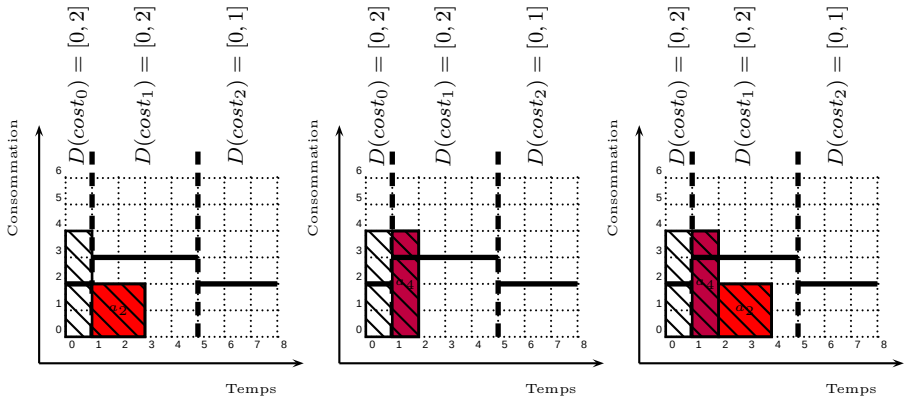


Figure 5.8 – L'enveloppe énergétique de l'ensemble d'activités $LCut(A, a_4) = \{a_2, a_4\}$, avec a_2 en rouge et a_4 en violet, telles que $D(sa_2) = [1, 2]$ et $D(sa_4) = [1, 3]$, avec $costC = max$. L'enveloppe énergétique de cet ensemble est une surestimation maximale de l'aire occupée avant la fin au plus tôt de l'ensemble. Ce maximum est donné par la troisième figure : $Env(LCut(A, a_4)) = \max_{\Theta \subseteq LCut(A,a_4)}(Area(0, \underline{mins}(\Theta)) + \underline{e}(\Theta)) = 12$.

Nous rappelons qu'avec ces données, l'algorithme d'Edge-Finding fonctionne en deux phases :

1. une phase de détection des précédences,
2. une phase de mise à jour des dates de début au plus tôt des activités.

Première phase : détection des précédences

La règle de précédence 1 reste la même dans le cas SoftCumulative.

Pour le prouver, nous introduisons la définition suivante. L'indice considéré par la Définition 42 ci-après n'a pas besoin d'être explicitement calculé dans l'algorithme.

Définition 42 (*inf*) *Soit un ensemble Ω d'activités A, et son enveloppe énergétique $Env(\Omega)$. $inf(\Omega)$ est le plus grand index j tel que $Area(sp_0, sp_j) \leq Env(\Omega)$.*

Comme dans le cas de la contrainte CUMULATIVE, étant donné un ensemble d'activités $\Omega \subseteq A$ (qui est, dans l'algorithme, systématiquement une coupe à gauche), nous adaptons le calcul d'une borne inférieure $lb(\underline{c}(\Omega))$ pour la date de fin au plus tôt $\underline{c}(\Omega)$ dans le cas où $costC = max$.[1] Nous rappelons que $sp_{inf(\Omega)}$ est le début de l'intervalle utilisateur d'indice $inf(\Omega)$.

Proposition 4 $lb(\underline{c}(\Omega)) = sp_{inf(\Omega)} + \left\lceil \frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})}{lc_{inf(\Omega)} + \overline{cost_{inf(\Omega)}}} \right\rceil$

Preuve. $lb(\underline{c}(\Omega))$ est le premier point de temps avant lequel l'énergie $Env(\Omega)$ peut avoir été entièrement consommée à partir de sp_0 (voir [90]).

D'après la Définition 42, $sp_{inf(\Omega)} \leq lb(\underline{c}(\Omega)) < sp_{inf(\Omega)+1}$. $Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})$ peut être entièrement consommée dans l'intervalle $[sp_{inf(\Omega)}, sp_{inf(\Omega)+1}[$ et est répartie sur un nombre de points de temps égal au minimum à $\left\lceil \frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})}{lc_{inf(\Omega)} + \overline{cost_{inf(\Omega)}}} \right\rceil$. □

D'après la Proposition 4, l'équivalence de la règle de précédence 1 reste vraie. Nous rappelons ici cette équivalence et la prouvons dans le cas de la contrainte SOFTCUMULATIVE.

Propriété 1 *Les deux conditions suivantes sont équivalentes :*

- $lb(\underline{c}(LCut(A, a_j) \cup \{a_i\})) > \overline{ca_j}$,
- $Env(LCut(A, a_j) \cup \{a_i\}) > Area(sp_0, \overline{ca_j})$.

Preuve. Soit l'ensemble $\Omega = LCut(A, a_j) \cup \{a_i\}$.

Nous prouvons l'équivalence dans un sens, puis dans l'autre :

Posons $lb(\underline{c}(\Omega)) > \overline{ca_j}$. La Proposition 4 indique que $sp_{inf(\Omega)} + \left\lceil \frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})}{lc_{inf(\Omega)} + \overline{cost_{inf(\Omega)}}} \right\rceil > \overline{ca_j}$. Nous avons donc : $sp_{inf(\Omega)} + \frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})}{lc_{inf(\Omega)} + \overline{cost_{inf(\Omega)}}} > \overline{ca_j}$, car $\overline{ca_j}$ est un entier positif.

Cette inégalité est équivalente à $\frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})}{lc_{inf(\Omega)} + \overline{cost_{inf(\Omega)}}} > \overline{ca_j} - sp_{inf(\Omega)}$. $lc_{inf(\Omega)} + \overline{cost_{inf(\Omega)}}$ est positif, ce qui donne : $Env(\Omega) - Area(sp_0, sp_{inf(\Omega)}) > (\overline{ca_j} - sp_{inf(\Omega)}) \times (lc_{inf(\Omega)} + \overline{cost_{inf(\Omega)}})$. Nous avons donc : $Env(\Omega) > (\overline{ca_j} - sp_{inf(\Omega)}) \times (lc_{inf(\Omega)} + \overline{cost_{inf(\Omega)}}) + Area(sp_0, sp_{inf(\Omega)})$.

Il convient alors de distinguer deux cas :

- si $\overline{ca_j} \geq sp_{inf(\Omega)}$, alors nécessairement, $\overline{ca_j} \in p_{inf(\Omega)}$ car, par la Définition 42, $sp_{inf(\Omega)} \leq lb(\underline{c}(\Omega)) < sp_{inf(\Omega)+1}$ et $lb(\underline{c}(\Omega)) > \overline{ca_j}$. Alors, si $\overline{ca_j} \in p_{inf(\Omega)}$, on a $(\overline{ca_j} - sp_{inf(\Omega)}) \times (lc_{inf(\Omega)} + \overline{cost_{inf(\Omega)}}) = Area(sp_{inf(\Omega)}, \overline{ca_j})$, par la Définition 40. Alors, l'inégalité précédente devient : $Env(\Omega) > Area(sp_{inf(\Omega)}, \overline{ca_j}) + Area(sp_0, sp_{inf(\Omega)})$, ce qui vaut : $Env(\Omega) > Area(sp_0, \overline{ca_j})$, et l'équivalence est vérifiée,
- si $\overline{ca_j} < sp_{inf(\Omega)}$, alors, par la Définition 40, $Area(sp_0, \overline{ca_j}) < Area(sp_0, sp_{inf(\Omega)})$. Or, par la Définition 42, $Env(\Omega) \geq Area(sp_0, sp_{inf(\Omega)})$. Alors, $Env(\Omega) > Area(sp_0, \overline{ca_j})$.

1. Nous ne donnons la preuve que dans le cas $costC = max$. Dans le cas $costC = sum$ la démonstration est la même, excepté le calcul de $lb(\underline{c}(\Omega))$ qui est égal à $lb(\underline{c}(\Omega)) = sp_{inf(\Omega)} + \left\lceil \frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)}) - \overline{cost_{inf(\Omega)}}}{lc_{inf(\Omega)}} \right\rceil$.

Nous prouvons maintenant l'implication inverse. Pour cela, posons $Env(\Omega) > Area(sp_0, \overline{ca_j})$. Là encore, distinguons deux cas :

- si $Area(sp_0, \overline{ca_j}) \geq Area(sp_0, sp_{inf(\Omega)})$, alors, comme $Area(sp_0, sp_{inf(\Omega)} + 1) > Env(\Omega) \geq Area(sp_0, sp_{inf(\Omega)})$, et $Env(\Omega) > Area(sp_0, \overline{ca_j})$, par la Définition 42, alors, par la Définition 40, $\overline{ca_j} \in p_{inf(\Omega)}$. Nous avons donc $Env(\Omega) - Area(sp_0, sp_{inf(\Omega)}) > Area(sp_0, \overline{ca_j}) - Area(sp_0, sp_{inf(\Omega)}) \geq 0$. Or $Area(sp_0, \overline{ca_j}) = Area(sp_0, sp_{inf(\Omega)}) + Area(sp_{inf(\Omega)}, \overline{ca_j})$, par la Définition 40. On a donc : $Env(\Omega) - Area(sp_0, sp_{inf(\Omega)}) > Area(sp_{inf(\Omega)}, \overline{ca_j})$, d'où : $\frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})}{lc_{inf(\Omega)} + \overline{cost}_{inf(\Omega)}} > \frac{Area(sp_{inf(\Omega)}, \overline{ca_j})}{lc_{inf(\Omega)} + \overline{cost}_{inf(\Omega)}}$, et, en conséquence : $\left\lceil \frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})}{lc_{inf(\Omega)} + \overline{cost}_{inf(\Omega)}} \right\rceil > \frac{Area(sp_{inf(\Omega)}, \overline{ca_j})}{lc_{inf(\Omega)} + \overline{cost}_{inf(\Omega)}}$. On obtient alors : $sp_{inf(\Omega)} + \left\lceil \frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})}{lc_{inf(\Omega)} + \overline{cost}_{inf(\Omega)}} \right\rceil > sp_{inf(\Omega)} + \frac{Area(sp_{inf(\Omega)}, \overline{ca_j})}{lc_{inf(\Omega)} + \overline{cost}_{inf(\Omega)}}$. Or, comme $\overline{ca_j} \in p_{inf(\Omega)}$, $\frac{Area(sp_{inf(\Omega)}, \overline{ca_j})}{lc_{inf(\Omega)} + \overline{cost}_{inf(\Omega)}}$ est égale au nombre de points de temps entre $sp_{inf(\Omega)}$ et $\overline{ca_j}$, par la Définition 40. De ce fait, on vérifie : $sp_{inf(\Omega)} + \left\lceil \frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})}{lc_{inf(\Omega)} + \overline{cost}_{inf(\Omega)}} \right\rceil > \overline{ca_j}$. Alors $lb(\underline{c}(\Omega)) > \overline{ca_j}$ et la première inégalité est vérifiée,
- si $Area(sp_0, \overline{ca_j}) < Area(sp_0, sp_{inf(\Omega)})$, alors, par la Définition 40, $\overline{ca_j} < sp_{inf(\Omega)}$. Or, par la Proposition 4, $lb(\underline{c}(\Omega)) > sp_{inf(\Omega)}$. Alors $lb(\underline{c}(\Omega)) > \overline{ca_j}$ et la première inégalité est vérifiée.

□

Pour toutes activités $a_i, a_j \in A$, nous avons la règle suivante :

$$Env(LCut(A, a_j) \cup \{a_i\}) > Area(sp_0, \overline{ca_j}) \Rightarrow LCut(A, a_j) \lessdot a_i$$

La règle de précédence étant la même, la détection des précédences se fait de la même manière que Cumulative. L'algorithme 3 décrit cette procédure.

Seconde phase : mise à jour des bornes inférieures des dates de début au plus tôt

Avec le nouveau calcul de l'aire disponible, la notion de compétition et la règle de filtrage 2 restent les mêmes que pour la contrainte Cumulative. Nous rappelons qu'un ensemble Ω *est en compétition* avec une activité a_i si, et seulement si, $\underline{e}(\Omega) > Area(\underline{s}(\Omega), \overline{c}(\Omega)) - \underline{ra_i} \times (\overline{c}(\Omega) - \underline{s}(\Omega))$. Nous rappelons aussi que si $\Omega \lessdot a_i$ et Ω est en compétition avec a_i, alors $[\underline{sa_i}, \underline{s}(\Omega) + \left\lceil \frac{\underline{e}(\Omega) - Area(\underline{s}(\Omega), \overline{c}(\Omega)) + \underline{ra_i} \times (\overline{c}(\Omega) - \underline{s}(\Omega))}{\underline{ra_i}} \right\rceil [$ peut être retiré de $D(sa_i)$. Ce filtrage est illustré par l'exemple 33.

Exemple 33 *La Figure 5.9 représente le filtrage avec Edge-Finding dans le cas* SoftCumulative*, avec costC= max. Il reprend le problème introduit sur la Figure 5.2. Dans cet exemple,* $LCut(A, a_4) = \{a_2, a_4\}$*. Le calcul de l'enveloppe énergétique de* $LCut(A, a_4) \cup \{a_0\}$ *et sa comparaison avec* $Area(0, \overline{ca_4})$ *permet de détecter une précédence :*

$$Env(LCut(A, a_4) \cup \{a_0\}) = 20 > Area(0, \overline{ca_4}) = 19 \Rightarrow \{a_2, a_4\} = LCut(A, a_4) \lessdot a_0$$

selon la règle de précédence 1. La date de début au plus tôt de a_0 *est donc mise à jour en suivant la règle de filtrage 2.*

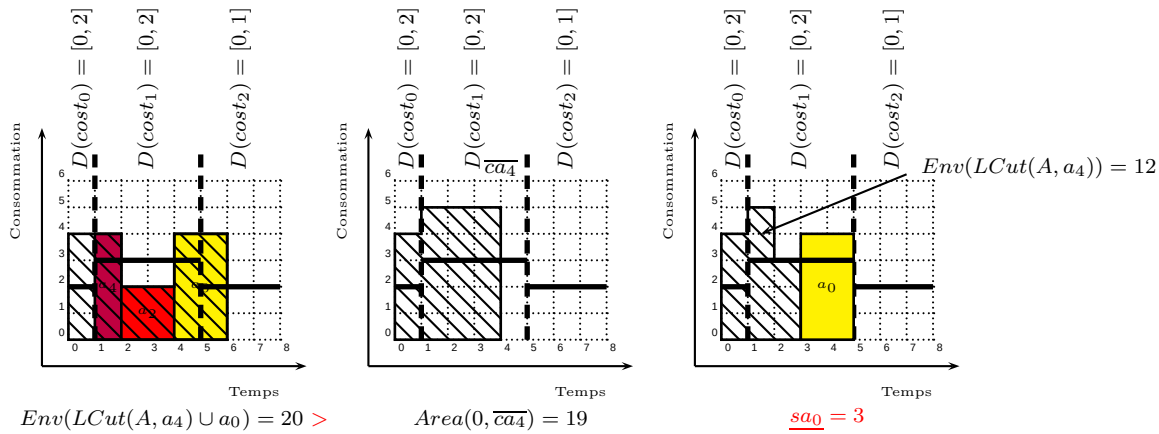


FIGURE 5.9 – Un exemple de filtrage de la date de début de l'activité a_0, en jaune, avec les activités a_2, en rouge, et a_4, en violet, telles que $D(sa_0) = [2, 6]$, $D(ca_0) = [4, 8]$, $D(sa_2) = [1, 2]$, $D(ca_2) = [3, 4]$, $D(sa_4) = [1, 3]$, $D(ca_4) = [2, 4]$, avec $costC = max$. Comme $Env(LCut(A, a_4) \cup \{a_0\}) = 20 > Area(0, \overline{ca_4}) = 19$, une précédence est détectée : $\{a_2, a_4\} = LCut(A, a_4) \lessdot a_0$. $LCut(A, a_4)$ est en compétition avec a_0, alors $\underline{sa_0}$ est mis à jour : $\underline{sa_0} = 3$.

A chaque feuille du Θ-tree de l'algorithme d'Edge-Finding de Vilím, le calcul de l'aire disponible est en $O(p)$, et non plus $O(1)$ (voir la Définition 40). Le calcul des noeuds internes reste le même que dans l'algorithme de Vilím. Ainsi, la mise à jour des dates de début des activités prend $O(n \times p)$. Cette mise à jour, ajoutée à la détection des précédences, donne une complexité globale de $O(p \times k \times n \times log(n))$, avec p le nombre d'intervalles utilisateurs, n le nombre d'activités et k le nombre de hauteurs d'activité distinctes.

Adaptation de l'algorithme d'Edge-Finding de Kameugne et al. [43] L'algorithme d'Edge-Finding de Kameugne et al. est décrit en section 2.2.2.2. En prenant en compte les modifications apportées au calcul de *Area* dans le cas SOFTCUMULATIVE, l'algorithme 6 reste le même dans le cas de la contrainte SOFTCUMULATIVE.

5.1.2 Mises à jour des bornes inférieures des coûts et de l'objectif

La mise à jour des bornes inférieures des domaines des variables de coût peut s'avérer importante pour la résolution. Nous montrons comment réaliser ces mises à jours dans les algorithmes de Sweep et d'Edge-Finding, ainsi que les bornes inférieures de la variable *obj* qui en découlent.

5.1.2.1 Mises à jour dans l'algorithme de Sweep

Les variables *Cost* peuvent être directement filtrées dans l'algorithme de Sweep, pendant que le profil cumulatif est calculé, sans modifier la complexité temporelle. La règle de filtrage est la suivante, et l'exemple 34 illustre ensuite cette règle

Règle de filtrage 6 *Considérons le rectangle courant* $\langle[\delta, \delta'[, sum_h\rangle$ *dans l'algorithme de Sweep :*
- *si* $costC = max$*, alors si* $sum_h - lc_j(\delta) > \underline{cost_j}(\delta)$*, l'intervalle* $[\underline{cost_j}(\delta), sum_h - lc_j(\delta)[$ *peut être retiré de* $D(cost_j(\delta))$*,*
- *si* $costC = sum$*, alors si* $(\delta' - \delta) \times (sum_h - lc_j(\delta)) > \underline{cost_j}(\delta)$*, l'intervalle* $[\underline{cost_j}(\delta), (sum_h - lc_j(\delta)) \times (\delta' - \delta)[$ *peut être retiré de* $D(cost_j(\delta))$*.*

Preuve. L'algorithme de Sweep maintient la donnée sum_h, qui correspond à la hauteur courante du profil cumulatif des parties obligatoires (Définition 21). Par construction, sachant que l'algorithme de Sweep pour SoftCumulative intègre les dates de début d'intervalles utilisateur dans la liste courante des événements, dans l'intervalle $[\delta, \delta'[$, sum_h est constant et lc_j est unique. On peut donc considérer de façon équivalente n'importe quel point de temps dans $[\delta, \delta'[$, par exemple δ. D'après la Définition 36, lorsque le paramètre $costC$ est max alors la variable de coût $cost_j$ de l'intervalle courant mesure la dépassement maximal au delà de la capacité locale lc_j. Donc si $sum_h - lc_j(\delta) > \underline{cost_j}(\delta)$ alors $\underline{cost_j} = \underline{cost_j}(\delta)$ peut être mis à jour à la valeur $sum_h - lc_j(\delta)$. De façon similaire, lorsqu'on considère l'aire excédant la capacité locale lc_j dans $[\delta, \delta'[$, si celle-ci est strictement supérieure à $\underline{cost_j}$ alors on peut mettre à jour $\underline{cost_j}$ à la valeur $(\delta' - \delta) \times (sum_h - lc_j(\delta))$. □

Exemple 34 *La Figure 5.10 donne un exemple de filtrage selon cette règle. Sur celle-ci, un profil cumulatif des parties obligatoires est représenté. Celui-ci induit un dépassement de* 2 *(quel que soit costC). Alors la variable de coût associée peut être mise à jour, et l'intervalle* $[0, 2[$ *peut être retiré de son domaine.*

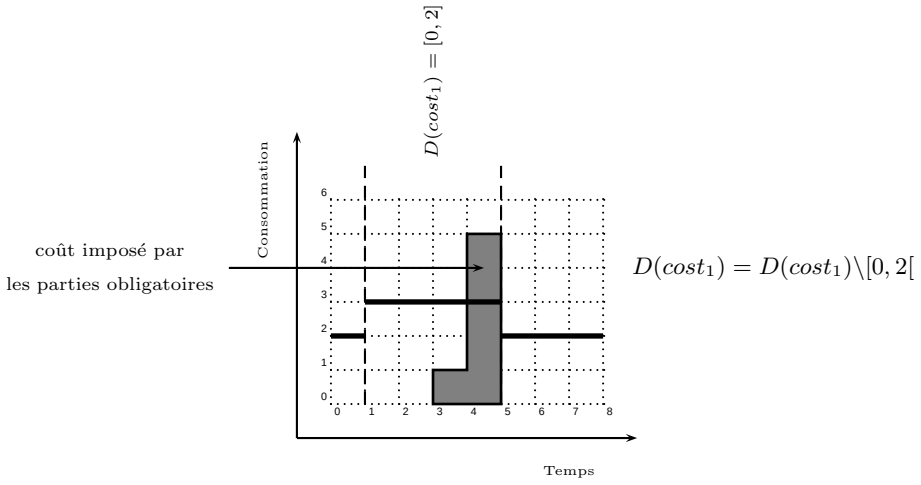


Figure 5.10 – Le polygone plein représente le profil cumulatif des parties obligatoires d'un problème softcumulative. Dans cet exemple, pour $costC = max$, ou $costC = sum$, avant le filtrage, $D(cost_1) = [0, 2]$. En $t = 3$, $sum_h = 5$, $lc_1 = 3$: $sum_h - lc_1 > \underline{cost_1}$, alors nous filtrons $D(cost_1)$ tel que $D(cost_1) = D(cost_1) \backslash [0, 2[$.

Les bornes inférieures pour la variable objectif sont :
- si $objC = sum$, $LB = \sum_{j \in [0, k-1]} \underline{cost_j}$,
- si $objC = max$, $LB = \max_{j \in [0, k-1]} (\underline{cost_j})$.

Ces bornes peuvent être calculées incrémentalement sans augmenter la complexité temporelle, et la variable *obj* est mise à jour.

5.1.2.2 Mises à jour dans l'algorithme d'Edge-Finding de Vilím

Le calcul d'une borne inférieure de la variable objectif peut également être intégré dans l'algorithme d'Edge-Finding de Vilím. Dans cet algorithme, pour chaque intervalle considéré, l'énergie totale des activités nécessairement entièrement comprises dans cet intervalle est calculée. Cette énergie peut être utilisée pour calculer une borne inférieure de la variable objectif. Nous montrons comment réaliser un tel filtrage.

Pour chaque activité $a_i \in A$, l'énergie $\underline{e}(LCut(A, a_i))$ est calculée par l'algorithme d'Edge-Finding. Elle correspond à l'énergie nécessairement contenue dans l'intervalle $[sp_0, \overline{ca_i}[$. Nous raisonnons sur cette énergie :

- nous comparons cette énergie à l'aire libre dans l'intervalle $[sp_0, \overline{ca_i}[$,
- si elle est strictement supérieure à cette aire libre, cela signifie qu'elle y engendre nécessairement un dépassement de capacité,
- ce dépassement peut alors également modifier la valeur de la variable objectif.

En suivant ce raisonnement, pour chaque activité $a_i \in A$, nous calculons une borne inférieure spécifique, notée $LB^V(a_i)$, de l'objectif.

Pour garder un calcul simple de chaque $LB^V(a_i)$, la proposition suivante considère qu'il n'y a pas de limite maximale sur le coût de chaque intervalle. Considérer les limites supérieures ne peut qu'augmenter la valeur de la borne. Notre calcul est moins fin mais nous ne faisons pas de sur-estimation de cette borne.

Nous notons $\mathcal{P}^V$ un ensemble contenant tous les intervalles utilisateurs $p_j \in P$ tels que $sp_j \leq \overline{ca_i}$. La proposition suivante donne le calcul de cette borne inférieure selon les différents paramètres *costC* et *objC*. L'exemple 35 illustre ce calcul.

Proposition 5 *Soit une activité $a_i \in A$, et $LCut(A, a_i)$ la coupe à gauche de A par a_i. Si $\underline{e}(LCut(A, a_i)) > FreeArea(sp_0, \overline{ca_i})$ alors :*

- *si $costC = max$ et $objC = sum$ alors $LB^V(a_i) = \left\lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(sp_0, \overline{ca_i})}{\max_{p_j \in \mathcal{P}^V}(min(\overline{ca_i}, sp_{j+1}) - sp_j)} \right\rceil$,*
- *si $costC = max$ et $objC = max$ alors $LB^V(a_i) = \left\lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(sp_0, \overline{ca_i})}{\overline{ca_i} - sp_0} \right\rceil$,*
- *si $costC = sum$ et $objC = sum$ alors $LB^V(a_i) = \underline{e}(LCut(A, a_i)) - FreeArea(sp_0, \overline{ca_i})$,*
- *si $costC = sum$ et $objC = max$ alors $LB^V(a_i) = \left\lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(sp_0, \overline{ca_i})}{|\mathcal{P}^V|} \right\rceil$.*

Preuve. Selon la Définition 25, $\underline{e}(LCut(A, a_i))$ est l'énergie minimale nécessairement consommée avant $\overline{ca_i}$. L'énergie maximale pouvant être placée dans l'intervalle $[sp_0, \overline{ca_i}[$ sans engendrer de dépassement est $FreeArea(sp_0, \overline{ca_i})$, selon la Définition 41. Ainsi, si $\underline{e}(LCut(A, a_i)) > FreeArea(sp_0, \overline{ca_i})$ alors cela implique un dépassement de capacité dont le surplus d'énergie est égal à $\underline{e}(LCut(A, a_i)) - FreeArea(sp_0, \overline{ca_i})$. Cela induit une nouvelle borne inférieure sur *obj*.

Considérons $costC = max$ et $objC = sum$. Sans perte de généralité, nous considérons que toute l'énergie engendrant le dépassement (c'est à dire le surplus d'énergie) peut être placée dans l'intervalle de longueur maximale $\mathcal{I} = \max_{p_j \in \mathcal{P}^V}(min(\overline{ca_i}, sp_{j+1}) - sp_j)$ commençant en sp_j. Nous nous plaçons alors dans l'intervalle $[sp_j, min(\overline{ca_i}, sp_{j+1})[$. Le surplus d'énergie est alors réparti dans cet intervalle. En effet, considérer un ou plusieurs intervalles supplémentaires ayant, par définition, une longueur inférieure ou égale à la longueur de $\mathcal{I}$, mènerait nécessairement à une somme des dépassements plus grande ou égale (nous rappelons que nous ne prenons pas en compte les limites maximales sur les coûts). Cette répartition dans

cet intervalle implique une hauteur de dépassement minimale égale à $\left\lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(sp_0,\overline{ca_i})}{min(\overline{ca_i},sp_{j+1}) - sp_j} \right\rceil$. Alors $LB^V(a_i)$ est égal à $\left\lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(sp_0,\overline{ca_i})}{\max_{p_j \in \mathcal{P}^V}(min(\overline{ca_i},sp_{j+1}) - sp_j)} \right\rceil$.

Si $costC = max$ et $objC = max$ alors le surplus d'énergie provoquant le plus petit dépassement est nécessairement réparti sur le plus grand intervalle possible, c'est à dire : $[sp_0, \overline{ca_i}[$. Alors $LB^V(a_i) = \left\lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(sp_0,\overline{ca_i})}{\overline{ca_i} - sp_0} \right\rceil$.

Si $costC = sum$ et $objC = sum$, la borne inférieure est la même, quelle que soit la répartition des dépassements sur les intervalles utilisateurs. Elle est alors $LB^V(a_i) = \underline{e}(LCut(A, a_i)) - FreeArea(sp_0, \overline{ca_i})$.

Enfin, si $costC = sum$ et $objC = max$, en répartissant le surplus d'énergie $\underline{e}(LCut(A, a_i)) - FreeArea(sp_0, \overline{ca_i})$ sur les intervalles de $\mathcal{P}^V$ on obtient une borne inférieure du coût maximal possible. On obtient donc $LB^V(a_i) = \left\lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(sp_0,\overline{ca_i})}{|\mathcal{P}^V|} \right\rceil$. □

Exemple 35

Les Figures 5.11 et 5.12 donnent un exemple de ce calcul de borne inférieure. Elles représentent un problème à 3 activités : a_2 et a_4, introduites sur la Figure 5.2, et une activité a_5 que nous introduisons pour les besoins de cet exemple. On a : $LCut(A, a_5) = \{a_2, a_4, a_5\}$, et $\underline{e}(LCut(A, a_5)) = 14$. Comme $\underline{e}(LCut(A, a_5)) > FreeArea(0, \overline{ca_5}) = 11$, une borne inférieure $LB^V(a_5)$ peut être calculée selon chacun des paramètres costC et objC tel que décrit précédemment.

Dans le cas costC= max, le dépassement maximal de la capacité locale est considéré :

- *si objC= max nous répartissons le surplus d'énergie sur l'intervalle $[sp_0, \overline{ca_5}[$, et obtenons $LB^V(a_5) = \left\lceil \frac{\underline{e}(LCut(A,a_5)) - FreeArea(sp_0,\overline{ca_5})}{\overline{ca_5} - sp_0} \right\rceil = 1$,*
- *si objC= sum, nous le répartissons sur le plus grand intervalle utilisateur et obtenons $LB^V(a_5) = \left\lceil \frac{\underline{e}(LCut(A,a_5)) - FreeArea(sp_0,\overline{ca_5})}{\max_{p_j \in \mathcal{P}}(min(\overline{ca_5},sp_{j+1}) - sp_j)} \right\rceil = 1$, avec $\mathcal{P} = \{p_0, p_1\}$ l'ensemble des intervalles utilisateur intersectés par $[sp_0, \overline{ca_5}[$.*

Dans le cas costC= sum, le surplus d'énergie est considéré :

- *si objC= max, nous répartissons ce surplus entre tous les intervalles, et obtenons $LB^V(a_5) = \lceil \frac{\underline{e}(LCut(A,a_5)) - FreeArea(sp_0,\overline{ca_5})}{|\mathcal{P}|} \rceil = 2$,*
- *si objC= sum, la répartition entre les intervalles n'importe pas et nous obtenons $LB^V(a_5) = \underline{e}(LCut(A, a_5)) - FreeArea(sp_0, \overline{ca_5}) = 3$.*

Algorithme 8: Algorithme de Sweep pour SoftCumulative

Data : Un ensemble d'activités A, une séquence d'intervalles utilisateur P, une séquence de capacités locales Loc, une séquence de variables de coût $Cost$, une liste d'événements Evt.

List $ActToPrune$; int $sum_h = 0$; int δ ; int δ' ;
int k ; //L'index de l'intervalle utilisateur courant
$evt = Evt.first()$; // Extraction du premier événement
$\delta = evt.date$; // Enregistrement de sa date
$Evt.sortByDate()$; On ordonne les événements par date croissante, pour une date égale, on place d'abord les événements de type PROFILE, puis PRUNING, puis INTERVAL
while $evt \neq NULL$ **do**
  $a_i = evt.activity$;// L'activité correspondant à l'événement courant
  $\delta' = evt.date$; //La date de l'événement courant
  $k = interval(\delta')$//Chaque date est associée statiquement à l'intervalle utilisateur dans lequel elle se trouve
  if $evt.type == PROFILE$ **then**
    if $\delta \neq \delta'$ **then**
      if $sum_h > lc_k + \overline{cost_k}$ **then**
        return fail ;
      foreach $a_j \in ActToPrune$ **do**
        if $(sum_h + \underline{ra_j} > lc_k + \overline{cost_k})\&(\overline{sa_j} \geq \underline{ca_j} || ((\overline{sa_j} < \underline{ca_j})\&[\overline{sa_j}, \underline{ca_j}[\cap[\delta, \delta'[= \emptyset))$ **then**
          $D(sa_j) = \overline{D}(sa_j)\backslash]\delta - \underline{da_j}, \delta'[$;//Mise à jour de la date de début d'une activité n'ayant pas de partie obligatoire dans $[\delta, \delta'[$
        if $\overline{ca_j} < \delta'$ **then**
          $ActToPrune.remove(a_j)$;
      $\delta = \delta'$;
    $sum_h += evt.increment$;
  else if $evt.type == INTERVAL$ **then**
    if $\delta \neq \delta'$ **then**
      if $sum_h > lc_k + \overline{cost_k}$ **then**
        return fail ;
      foreach $a_j \in ActToPrune$ **do**
        if $(sum_h + \underline{ra_j} > lc_k + \overline{cost_k})\&(\overline{sa_j} \geq \underline{ca_j} || ((\overline{sa_j} < \underline{ca_j})\&[\overline{sa_j}, \underline{ca_j}[\cap[\delta, \delta'[= \emptyset))$ **then**
          $D(sa_j) = \overline{D}(sa_j)\backslash]\delta - \underline{da_j}, \delta'[$;//Mise à jour de la date de début d'une activité n'ayant pas de partie obligatoire dans $[\delta, \delta'[$
        if $\overline{ca_j} < \delta'$ **then**
          $ActToPrune.remove(a_j)$;
      $\delta = \delta'$;
  else if $evt.type == PRUNING$ **then**
    $ActToPrune.add(a_i)$;
  $evt = evt.next()$
if $sum_h > lc_k + \overline{cost_k}$ **then** return fail ;
foreach $a_j \in ActToPrune$ **do**
  if $(sum_h + \underline{ra_j} > lc_k + \overline{cost_k})\&(\overline{sa_j} \geq \underline{ca_j} || ((\overline{sa_j} < \underline{ca_j})\&[\overline{sa_j}, \underline{ca_j}[\cap[\delta, \delta'[= \emptyset))$ **then**
  $D(sa_j) = D(\overline{sa_j})\backslash]\delta - \underline{da_j}, \delta'[$;

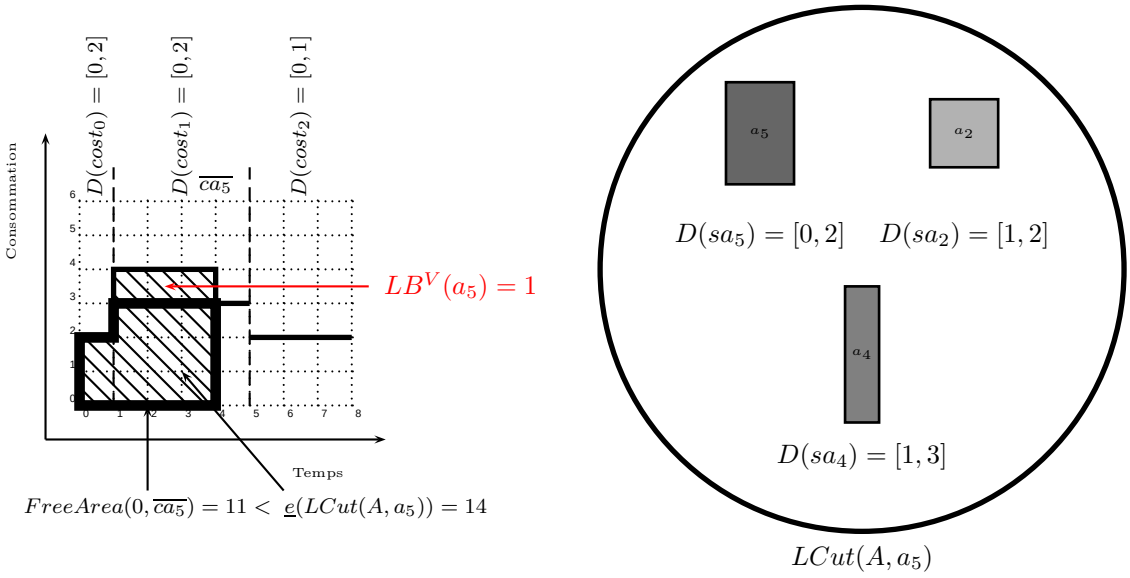


FIGURE 5.11 – Un problème cumulatif relaxé avec trois activités a_2, a_4 et a_5 telles que $D(sa_2) = [1,2]$, $D(sa_4) = [1,3]$, $D(sa_5) = [0,2]$, $D(ca_2) = [3,4]$, $D(ca_4) = [2,4]$, $D(ca_5) = [2,4]$, $da_2 = 2$, $da_4 = 1$, $da_5 = 2$, $ra_2 = 2$, $ra_4 = 4$, $ra_5 = 3$, trois intervalles utilisateur et $costC = max$. Comme $\underline{e}(LCut(A, a_5)) = 14 > FreeArea(0, \overline{ca_5}) = 11$, a_5 définit une borne inférieure sur la variable objectif. Ici, $\mathcal{P} = \{p_0, p_1\}$. Si $objC = sum$, $LB^V(a_5) = \left\lceil \frac{\underline{e}(LCut(A,a_5)) - FreeArea(sp_0, \overline{ca_5})}{\max_{p_j \in \mathcal{P}}(min(\overline{ca_5}, sp_{j+1}) - sp_j)} \right\rceil = 1$ et si $objC = max$, $LB^V(a_5) = \left\lceil \frac{\underline{e}(LCut(A,a_5)) - FreeArea(sp_0, \overline{ca_5})}{\overline{ca_5} - sp_0} \right\rceil = 1$

Le surplus d'énergie dans un des intervalles considérés dans l'Edge Finding de Vilím peut être réparti sur l'ensemble des intervalles utilisateurs intersectés par l'intervalle $[sp_0, \overline{ca_i}[$ courant, c'est à dire l'intervalle entre le point de temps 0 et la borne supérieure de la date de fin de l'activité a_i.

Des déductions sur les variables de coût peuvent être effectuées dans deux cas particuliers :

- si, pour une activité $a_i \in A$ l'intervalle $[sp_0, \overline{ca_i}[$ est entièrement contenu dans un seul intervalle utilisateur, l'intervalle p_0, et que l'énergie $\underline{e}(LCut(A, a_i))$ est strictement supérieure à l'aire libre dans l'intervalle $[sp_0, \overline{ca_i}[$, alors cela engendre nécessairement un dépassement de capacité dans l'intervalle p_0. Une borne inférieure de $cost_0$ peut alors être déduite. Nous détaillerons son calcul plus loin,
- considérons les bornes supérieures des coûts locaux. Considérons également que l'intervalle $[sp_0, \overline{ca_i}[$ intersecte plusieurs intervalles utilisateurs. Notons $\mathcal{SP}^V$ l'ensemble des dates de début d'intervalle utilisateur que l'intervalle $[sp_0, \overline{ca_i}[$ intersecte. S'il existe un intervalle $[sp_j, min(sp_{j+1}, \overline{ca_i})[$ tel que $sp_j \in \mathcal{SP}^V$ et que l'énergie $\underline{e}(LCut(A, a_i))$ soit supérieure à la somme des aires disponibles dans tous les autres intervalles $[sp_k, min(sp_{k+1}, \overline{ca_i})[$ tels que $sp_k \in \mathcal{SP}^V$ et $sp_k \neq sp_j$, alors cela induit nécessairement un dépassement de capacité dans l'intervalle $[sp_j, min(sp_{j+1}, \overline{ca_i})[$. Une borne inférieure de $cost_j$ peut alors être déduite.

Nous donnons ci-après les règles de filtrage dans ces deux cas. Les exemples 36 et 37 illustrent ces règles.

Règle de filtrage 7 *Soit une activité $a_i \in A$, et $LB^V(a_i)$ la borne inférieure de la variable objectif calculée par la proposition 5. Si $\overline{ca_i} \in p_0$ et $\underline{e}(LCut(A, a_i)) > FreeArea(sp_0, \overline{ca_i})$, alors $[\underline{cost_0}, LB^V(a_i)[$*

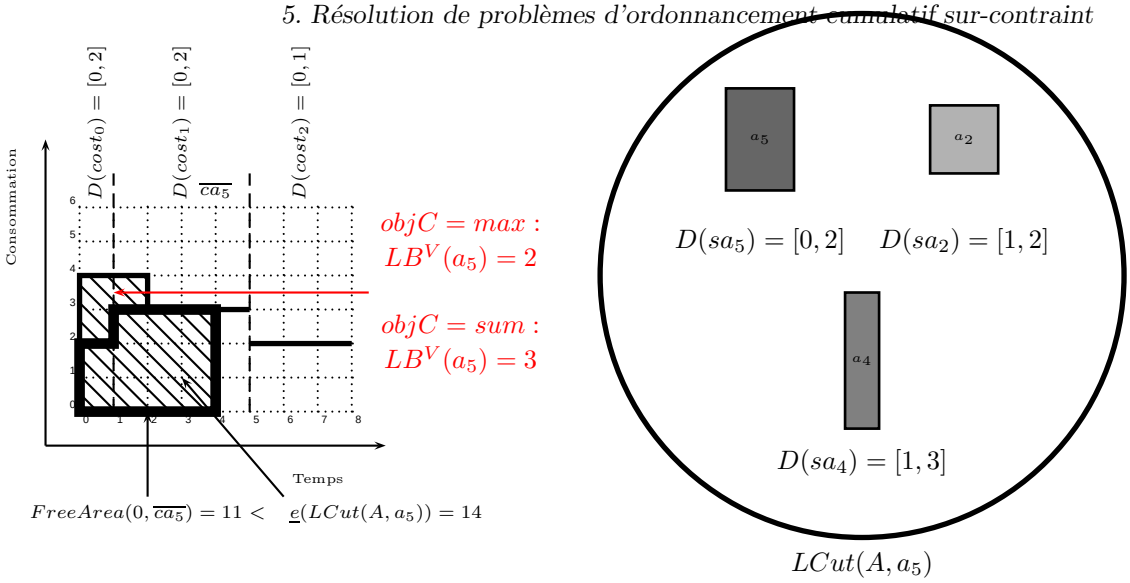


FIGURE 5.12 – Un problème cumulatif relaxé de la Figure 5.11 avec $costC = sum$. On a $\underline{e}(LCut(A, a_5)) = 14 > FreeArea(0, \overline{ca_5}) = 11$, alors a_5 définit une borne inférieure sur la variable objectif. Ici, $\mathcal{P} = \{p_0, p_1\}$. Si $objC = max$, $LB^V(a_5) = \left\lceil \frac{\underline{e}(LCut(A,a_5)) - FreeArea(sp_0, \overline{ca_5})}{|\mathcal{P}|} \right\rceil = 2$ et si $objC = sum$, $LB^V(a_5) = \underline{e}(LCut(A, a_5)) - FreeArea(sp_0, \overline{ca_5}) = 3$

peut être retiré de $D(cost_0)$.

Preuve. Pour une activité $a_i \in A$, $LB^V(a_i)$ est une borne inférieure du coût total engendré par l'énergie obligatoirement comprise dans l'intervalle $[sp_0, \overline{ca_i}[$. Si $[sp_0, \overline{ca_i}[\subseteq p_0$, alors ce coût est nécessairement engendré dans l'intervalle utilisateur p_0. Ainsi, $LB^V(a_i)$ est une borne inférieure de $cost_0$. □

Règle de filtrage 8 *Soit une activité* $a_i \in A$, *et* $\mathcal{SP}^V$ *l'ensemble des dates de début des intervalles utilisateur* $p_j \in P$ *intersectés par* $[sp_0, \overline{ca_i}[$. *Pour tout intervalle* $[sp_j, min(sp_{j+1}, \overline{ca_i})[$, *tel que* $sp_j \in \mathcal{SP}^V$, *et que* $FreeArea(sp_j, min(sp_{j+1}, \overline{ca_i})) + \sum_{\substack{sp_k \neq sp_j \\ sp_k \in \mathcal{SP}^V}} (Area(sp_k, min(sp_{k+1}, \overline{ca_i}))) < \underline{e}(LCut(A, a_i))$, *alors :*

- *si* $costC = max$ *alors*

$$[\underline{cost_j}, \left\lceil \frac{\underline{e}(LCut(A, a_i)) - FreeArea(sp_j, min(sp_{j+1}, \overline{ca_i})) - \sum_{\substack{sp_k \neq sp_j \\ sp_k \in \mathcal{SP}^V}} (Area(sp_k, min(sp_{k+1}, \overline{ca_i})))}{min(\overline{ca_i}, sp_{j+1}) - sp_j} \right\rceil [$$

  peut être retiré de $D(cost_j)$,
- *si* $costC = sum$ *alors*

$$[\underline{cost_j}, \underline{e}(LCut(A, a_i)) - FreeArea(sp_j, min(sp_{j+1}, \overline{ca_i})) - \sum_{\substack{sp_k \neq sp_j \\ sp_k \in \mathcal{SP}^V}} (Area(sp_k, min(sp_{k+1}, \overline{ca_i})))[$$

  peut être retiré de $D(cost_j)$.

Preuve. Pour une activité $a_i \in A$ donnée, $\underline{e}(LCut(A, a_i))$ est l'énergie nécessairement consommée dans l'intervalle $[sp_0, \overline{ca_i}[$. Pour un intervalle $[sp_j, min(sp_{j+1}, \overline{ca_i})[$, tel que $sp_j \in \mathcal{SP}^V$,

$$\sum_{\substack{sp_k \neq sp_j \\ sp_k \in \mathcal{SP}^V}} (Area(sp_k, min(sp_{k+1}, \overline{ca_i})))$$

est l'énergie totale qui peut être consommée par les autres intervalles $[sp_k, min(sp_{k+1}, \overline{ca_i})[$, avec $sp_k \in \mathcal{SP}^V$ et $sp_k \neq sp_j$. $FreeArea(sp_j, min(sp_{j+1}, \overline{ca_i}))$ est l'énergie maximale qui peut être consommée dans l'intervalle $[sp_j, min(sp_{j+1}, \overline{ca_i})[$ sans engendrer de dépassement de capacité, par la Définition 41. Alors, si

$$FreeArea(sp_j, min(sp_{j+1}, \overline{ca_i})) + \sum_{\substack{sp_k \neq sp_j \\ sp_k \in \mathcal{SP}^V}} (Area(sp_k, min(sp_{k+1}, \overline{ca_i}))) < \underline{e}(LCut(A, a_i))$$

cela signifie que l'énergie restante, c'est à dire la différence

$$\underline{e}(LCut(A, a_i)) - FreeArea(sp_j, min(sp_{j+1}, \overline{ca_i})) - \sum_{\substack{sp_k \neq sp_j \\ sp_k \in \mathcal{SP}^V}} (Area(sp_k, min(sp_{k+1}, \overline{ca_i})))$$

est nécessairement placée dans l'intervalle $[sp_j, min(sp_{j+1}, \overline{ca_i})[$, et qu'elle y engendre un dépassement de capacité.

Si $costC = max$, le surplus d'énergie

$$\underline{e}(LCut(A, a_i)) - FreeArea(sp_j, min(sp_{j+1}, \overline{ca_i})) - \sum_{\substack{sp_k \neq sp_j \\ sp_k \in \mathcal{SP}^V}} (Area(sp_k, min(sp_{k+1}, \overline{ca_i})))$$

peut être réparti sur l'ensemble de l'intervalle $[sp_j, min(sp_{j+1}, \overline{ca_i})[$, avec une hauteur minimale de

$$\left\lceil \frac{\underline{e}(LCut(A, a_i)) - FreeArea(sp_j, min(sp_{j+1}, \overline{ca_i})) - \sum_{\substack{sp_k \neq sp_j \\ sp_k \in \mathcal{SP}^V}} (Area(sp_k, min(sp_{k+1}, \overline{ca_i})))}{min(\overline{ca_i}, sp_{j+1}) - sp_j} \right\rceil$$

Celle-ci définit donc une borne inférieure de la variable de coût $cost_j$ associée à cet intervalle.

Si $costC = sum$, le surplus d'énergie

$$\underline{e}(LCut(A, a_i)) - FreeArea(sp_j, min(sp_{j+1}, \overline{ca_i})) - \sum_{\substack{sp_k \neq sp_j \\ sp_k \in \mathcal{SP}^V}} (Area(sp_k, min(sp_{k+1}, \overline{ca_i})))$$

est entièrement compris dans l'intervalle $[sp_j, min(sp_{j+1}, \overline{ca_i})[$, c'est donc une borne inférieure de la variable de coût $cost_j$ associée à cet intervalle. □

Exemple 36 *La Figure 5.13 donne un exemple de cas où la règle de filtrage 7 s'applique. Sur celle-ci, l'intervalle* $[0, \overline{ca_5}[$ *intersecte uniquement l'intervalle utilisateur* p_0*. L'énergie de la coupe à gauche de A par* a_5 *est telle que :* $\underline{e}(LCut(A, a_5)) > FreeArea(sp_0, \overline{ca_5})$*. Ceci engendre donc une borne inférieure sur la variable de coût* $cost_0$*. Celle-ci est égale à* $LB^V(a_5)$ *: Si* $costC = max$*,* $LB^V(a_5) = 1$ *et le domaine de la variable* $\underline{cost_0}$ *peut alors être mis à jour :* $\underline{cost_0 = 1}$ *et si* $costC = sum$*,* $\underline{cost_0 = 2}$*.*

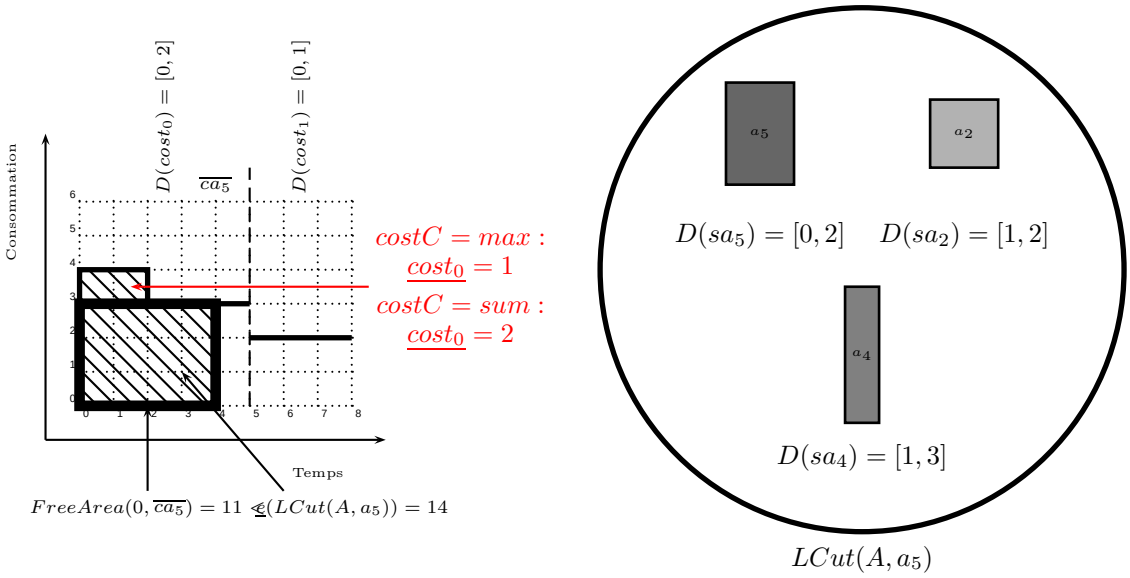


FIGURE 5.13 – Un problème cumulatif relaxé avec trois activités a_2, a_4 et a_5 telles que $D(sa_2) = [1, 2]$, $D(sa_4) = [1, 3]$, $D(sa_5) = [0, 2]$, $D(ca_2) = [3, 4]$, $D(ca_4) = [2, 4]$, $D(ca_5) = [2, 4]$, $da_2 = 2$, $da_4 = 1$, $da_5 = 2$, $ra_2 = 2$, $ra_4 = 4$, $ra_5 = 3$, et deux intervalles utilisateur. Nous modifions légèrement l'exemple de la Figure 5.11, tel que : p_0 : $[0, 5[$ et $p_1 = [5, 8[$. Nous avons : $[0, \overline{ca_5}[\subseteq p_0$ et $\underline{e}(LCut(A, a_5)) = 14 > FreeArea(0, \overline{ca_5}) = 12$, Alors a_5 définit une borne inférieure sur la variable objectif : $LB^V(a_5)$, mais également sur la variable de coût $cost_0$, telle que $\underline{cost_0} = LB^V(a_5)$. Si $costC = max$, $\underline{cost_0} = 1$, si $costC = sum$, $\underline{cost_0} = 2$.

Exemple 37 *La Figure 5.14 donne un exemple de cas où la règle de filtrage 8 s'applique. Sur celle-ci, l'intervalle* $[0, \overline{ca_5}[$ *intersecte les intervalles utilisateur* p_0 *et* p_1*. L'énergie de la coupe à gauche de A par* a_5 *est telle que :* $\underline{e}(LCut(A, a_i)) > Area(sp_0, sp_1) + FreeArea(sp_1, \overline{ca_5})$*. Ceci engendre donc une borne inférieure sur la variable de coût* $cost_1$*. Celle-ci est égale à 1 pour costC= max et pour costC= sum.*

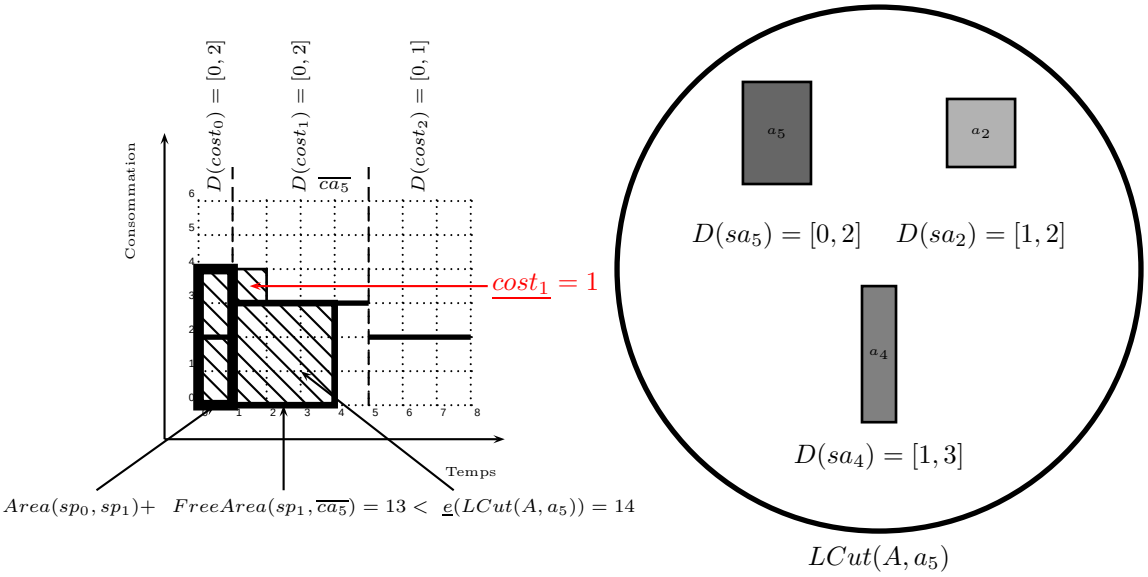


FIGURE 5.14 – Un problème cumulatif relaxé à trois activités a_2, a_4 et a_5 telles que $D(sa_2) = [1, 2]$, $D(sa_4) = [1, 3]$, $D(sa_5) = [0, 2]$, $D(ca_2) = [3, 4]$, $D(ca_4) = [2, 4]$, $D(ca_5) = [2, 4]$, $da_2 = 2$, $da_4 = 1$, $da_5 = 2$, $ra_2 = 2$, $ra_4 = 4$, $ra_5 = 3$ et trois intervalles utilisateur. Nous nous plaçons dans l'intervalle $[sp_1, \overline{ca_5}[$. Le seul autre intervalle utilisateur intersecté par $[sp_0, \overline{ca_5}[$ est p_0. Or, $\underline{e}(LCut(A, a_5)) = 14 > Area(0, sp_1) + FreeArea(sp_1, \overline{ca_5}) = 13$, a_5 définit donc une borne inférieure sur la variable de coût $cost_1$. Ici, $\mathcal{SP}^V = \{sp_0, sp_1\}$. Si $costC = max$, $\underline{cost_1} = \left\lceil \frac{\underline{e}(LCut(A,a_5)) - FreeArea(sp_1,\overline{ca_5}) - \sum_{sp_k \neq sp_1 \& sp_k \in \mathcal{SP}^V} (Area(sp_k, min(sp_{k+1}, \overline{ca_5})))}{\overline{ca_i} - sp_1} \right\rceil$, ce qui donne : $\underline{cost_1} = \left\lceil \frac{\underline{e}(LCut(A,a_5)) - FreeArea(sp_1,\overline{ca_5}) - (Area(sp_0, min(sp_1, \overline{ca_5})))}{\overline{ca_i} - sp_1} \right\rceil = 1$. Si $costC = sum$, $\underline{cost_1} = \underline{e}(LCut(A, a_5)) - FreeArea(sp_1, \overline{ca_5}) - \sum_{sp_k \neq sp_1 \& sp_k \in \mathcal{SP}^V} (Area(sp_k, min(sp_{k+1}, \overline{ca_5})))$, ce qui donne : $\underline{cost_1} = \underline{e}(LCut(A, a_5)) - FreeArea(sp_1, \overline{ca_5}) - Area(sp_0, sp_1) = 1$

Notons que le principe de ces deux dernières règles correspondent à des cas de figure somme toute très particuliers.

Le calcul de $LB^V(a_i)$, dans la règle de filtrage 5 est intégré dans l'algorithme d'Edge-Finding et est réalisé en $O(p)$, notamment à cause du calcul de $FreeArea$. Ainsi, la complexité globale, en intégrant cet algorithme dans l'Edge-Finding de Vilím devient $O(k \times p^2 \times n \times log(n))$, avec k le nombre de hauteurs d'activité dans A différentes.

Le calcul de la borne inférieure de $cost_0$ dans le cas de la règle de filtrage 7 peut être effecté en $O(1)$. Ceci n'augmente donc pas la complexité globale de l'algorithme.

Enfin, le calcul de la borne inférieure d'une variable de coût $cost_k$, pour chaque variable de coût, défini dans la règle de filtrage 7 est de l'ordre de $O(p)$: en chaque activité $a_i \in A$, nous considérons l'ensemble des intervalles utilisateur intersectant $[sp_0, \overline{ca_i}[$. Pour chacun de ces intervalles utilisateur, nous faisons la somme des aires disponibles sur les autres intervalles utilisateur appartenant à $[sp_0, \overline{ca_i}[$, cette donnée peut cependant être maintenue incrémentalement. Intégrer cette règle engendre une complexité globale de l'algorithme de $O(k \times p^2 \times n \times log(n))$.

5.1.2.3 Mises à jour dans l'algorithme d'Edge-Finding de Kameugne et al.

Le calcul d'une borne inférieure de la variable objectif peut également être intégré dans l'algorithme d'Edge-Finding de Kameugne et al. Une méthode de calcul de borne inférieure, très similaire à la méthode employée au sein de l'algorithme de Vilím, peut être mise en place au sein de cet algorithme.

Pour toutes activités $a_i, a_j \in A$, nous rappelons que l'ensemble $\Omega_{i,j}$ est l'intervalle d'activités, tel que défini par la Définition 26. L'algorithme d'Edge-Finding de Kameugne et al. intègre le calcul de l'énergie de tous les intervalles d'activités possibles.

Nous raisonnons sur cette énergie :
- nous comparons cette énergie à l'aire libre dans l'intervalle $[\underline{sa_i}, \overline{ca_j}[$,
- si elle est strictement supérieure à cette aire libre, cela signifie qu'elle y engendre nécessairement un dépassement de capacité,
- ce dépassement peut alors également modifier la valeur de la variable objectif.

En suivant ce raisonnement, pour toutes activités $a_i, a_j \in A$, nous calculons une borne inférieure spécifique, notée $LB^K(a_i, a_j)$, de l'objectif. Pour garder un calcul simple de chaque $LB^K(a_i, a_j)$, la proposition suivante considère qu'il n'y a pas de limite maximale sur le coût de chaque intervalle. Considérer les bornes supérieures des coûts ne peut qu'augmenter cette borne inférieure. Notre calcul est moins fin mais nous ne sur-estimons pas la borne inférieure calculée.

Nous rappelons que sp_k est la date de début d'un intervalle utilisateur $p_k \in P$. Notons $\mathcal{P}^K$ un ensemble contenant tous les intervalles utilisateurs $p_k \in P$ tels que $(sp_{k+1} \geq \underline{sa_i} \wedge sp_k \leq \overline{ca_j})$.

Proposition 6 *Soit deux activités $a_i, a_j \in A$, et $\Omega_{i,j}$ l'intervalle d'activités associé Si $\underline{e}(\Omega_{i,j}) > FreeArea(\underline{sa_i}, \overline{ca_j})$ alors*

- *si $costC = max$ et $objC = sum$ alors* $LB^K(a_i, a_j) = \left\lceil \frac{\underline{e}(\Omega_{i,j}) - FreeArea(\underline{sa_i}, \overline{ca_j})}{\max_{p_k \in \mathcal{P}^K}(min(\overline{ca_j}, sp_{k+1}) - max(\underline{sa_i}, sp_k))}) \right\rceil$,
- *si $costC = max$ et $objC = max$ alors* $LB^K(a_i, a_j) = \left\lceil \frac{\underline{e}(\Omega_{i,j}) - FreeArea(\underline{sa_i}, \overline{ca_j})}{\overline{ca_j} - \underline{sa_i}} \right\rceil$,
- *si $costC = sum$ et $objC = sum$ alors* $LB^K(a_i, a_j) = \underline{e}(\Omega_{i,j}) - FreeArea(\underline{sa_i}, \overline{ca_j})$,
- *si $costC = sum$ et $objC = max$ alors* $LB^K(a_i, a_j) = \left\lceil \frac{\underline{e}(\Omega_{i,j}) - FreeArea(\underline{sa_i}, \overline{ca_j})}{|\mathcal{P}^K|} \right\rceil$.

Preuve. Selon la Définition 26, $\underline{e}(\Omega_{i,j})$ est l'énergie minimale nécessairement placée entre $\underline{sa_i}$ et $\overline{ca_j}$. L'énergie maximale pouvant être placée dans l'intervalle $[\underline{sa_i}, \overline{ca_j}[$ sans engendrer de dépassement est $FreeArea(\underline{sa_i}, \overline{ca_j})$, selon la Définition 41. Ainsi, si $\underline{e}(\Omega_{i,j}) > FreeArea(\underline{sa_i}, \overline{ca_j})$ alors cela implique un dépassement de capacité avec un surplus d'énergie égal à $\underline{e}(\Omega_{i,j}) - FreeArea(\underline{sa_i}, \overline{ca_j})$. Cela peut induire une nouvelle borne inférieure sur obj.

Considérons $costC = max$ et $objC = sum$. Sans perte de généralité, nous considérons que toute l'énergie engendrant le dépassement (c'est à dire le surplus d'énergie), égal à $\underline{e}(\Omega_{i,j}) - FreeArea(\underline{sa_i}, \overline{ca_j})$ peut être placée dans l'intervalle de longueur maximale $\mathcal{I}$ commençant en $max(\underline{sa_i}, sp_k)$. Nous nous plaçons alors dans l'intervalle $[max(\underline{sa_i}, sp_k), min(\overline{ca_j}, sp_{k+1})[$. Le surplus d'énergie est alors réparti dans cet intervalle, avec une hauteur minimale de $\left\lceil \frac{\underline{e}(\Omega_{i,j}) - FreeArea(\underline{sa_i}, \overline{ca_j})}{min(\overline{ca_j}, sp_{k+1}) - max(\underline{sa_i}, sp_k)} \right\rceil$. En effet, considérer un ou plusieurs intervalles supplémentaires ayant, par définition, une longueur inférieure ou égale à la longueur de $\mathcal{I}$, mènerait nécessairement à une somme des dépassements plus grande ou égale (nous rappelons que nous ne prenons pas en compte les limites maximales sur les coûts). Alors $LB^K(a_i, a_j) = \left\lceil \frac{\underline{e}(\Omega_{i,j}) - FreeArea(\underline{sa_i}, \overline{ca_j})}{\max_{p_k \in \mathcal{P}^K}(min(\overline{ca_j}, sp_{k+1}) - max(\underline{sa_i}, sp_k))} \right\rceil$.

Si $costC = max$ et $objC = max$ alors on répartit le surplus d'énergie sur le plus grand intervalle possible, c'est à dire : $[\underline{sa_i}, \overline{ca_j}[$, avec une hauteur minimale de $\left\lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(\underline{sa_i}, \overline{ca_j})}{\overline{ca_j} - \underline{sa_i}} \right\rceil$. Alors $LB^K(a_i, a_j) = \left\lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(\underline{sa_i}, \overline{ca_j})}{\overline{ca_j} - \underline{sa_i}} \right\rceil$.

Si $costC = sum$ et $objC = sum$, la borne inférieure du dépassement est la même, quelle que soit la répartition des dépassements sur les intervalles utilisateur. Alors $LB^K(a_i, a_j) = \underline{e}(\Omega_{i,j}) - FreeArea(\underline{sa_i}, \overline{ca_j})$.

Si $costC = sum$ et $objC = max$, en répartissant le surplus d'énergie $\underline{e}(\Omega_{i,j}) - FreeArea(\underline{sa_i}, \overline{ca_j})$ sur les intervalles de $\mathcal{P}^K$ on obtient le plus petit coût maximal possible. Alors $LB^K(a_i, a_j) = \left\lceil \frac{\underline{e}(\Omega_{i,j}) - FreeArea(\underline{sa_i}, \overline{ca_j})}{|\mathcal{P}^K|} \right\rceil$. □

Le surplus d'énergie dans un des intervalles considérés dans l'Edge Finding de Kameugne et al. peut être réparti sur l'ensemble des intervalles utilisateurs intersectés par l'intervalle $[\underline{sa_i}, \overline{ca_j}[$ courant, c'est à dire l'intervalle entre la date de début au plus tôt de l'activité a_i et la date de fin au plus tard de l'activité a_j.

Des déductions sur les variables de coût peuvent être effectuées dans deux cas particuliers :

- si, pour deux activités $a_i, a_j \in A$ l'intervalle $[\underline{sa_i}, \overline{ca_j}[$ est entièrement contenu dans un seul intervalle utilisateur $p_k \in P$, et que l'énergie $\underline{e}(\Omega_{i,j})$ est supérieure à l'aire libre dans l'intervalle $[\underline{sa_i}, \overline{ca_j}[$, alors cela engendre nécessairement un dépassement de capacité dans l'intervalle p_k. Une borne inférieure de $cost_k$ peut alors être déduite. Nous détaillerons son calcul dans la suite,
- nous prenons cette fois en compte les bornes supérieures des coûts locaux. Soient deux activités $a_i, a_j \in A$. Considérons que l'intervalle $[\underline{sa_i}, \overline{ca_j}[$ intersecte plusieurs intervalles utilisateurs. Notons $\mathcal{SP}^K$ l'ensemble des dates de début d'intervalle utilisateur que l'intervalle $[\underline{sa_i}, \overline{ca_j}[$ intersecte. S'il existe un intervalle $[max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})[$ avec $sp_k \in \mathcal{SP}^K$ tel que l'énergie $\underline{e}(\Omega_{i,j})$ soit supérieure à la somme des aires disponibles dans tous les autres intervalles $[max(\underline{sa_i}, sp_l), min(sp_{l+1}, \overline{ca_j})[$ tels que $sp_l \in \mathcal{SP}^K$ et $sp_l \neq sp_k$, alors cela induit nécessairement un dépassement de capacité dans l'intervalle $[max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})[$. Une borne inférieure de $cost_k$ peut alors être déduite.

Nous donnons ci-après les règles de filtrage dans ces deux cas.

Règle de filtrage 9 *Soit une activité $a_i \in A$, et $LB^K(a_i, a_j)$ la borne inférieure de la variable objectif calculée par la proposition 5. Si $\underline{sa_i}$ et $\overline{ca_j}$ appartiennent au même intervalle utilisateur $p_k \in P$, et $\underline{e}(\Omega_{i,j}) > FreeArea(\underline{sa_i}, \overline{ca_j})$, alors $[\underline{cost_k}, LB^K(a_i, a_j)[$ peut être retiré de $D(cost_k)$.*

Preuve. Pour deux activités $a_i, a_j \in A$, $LB^K(a_i, a_j)$ est une borne inférieure du coût total engendré par l'énergie obligatoirement comprise dans l'intervalle $[\underline{sa_i}, \overline{ca_j}[$. S'il existe un intervalle utilisateur $p_k \in P$ tel que $[\underline{sa_i}, \overline{ca_j}[\subseteq p_k$, alors ce coût est nécessairement engendré dans l'intervalle utilisateur p_k. Ainsi, $LB^K(a_i, a_j)$ définit une borne inférieur de $cost_k$. □

Règle de filtrage 10 *Soient deux activités $a_i, a_j \in A$, et $\mathcal{SP}^K$ l'ensemble des dates de début des intervalles utilisateur $p_k \in P$ intersectés par $[\underline{sa_i}, \overline{ca_j}[$. Pour tout intervalle $[max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})[$, tel que $sp_k \in \mathcal{SP}^K$, notons $S_k(a_i, a_j)$ la quantité $FreeArea(max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})) + \sum_{\substack{sp_l \neq sp_k \\ sp_l \in \mathcal{SP}^K}} (Area(max(\underline{sa_i}, sp_l), min(sp_{l+1}, \overline{ca_i})))$, qui correspond à l'aire pouvant être placée dans l'intervalle $[\underline{sa_i}, \overline{ca_i}[$ sans engendrer de dépassement dans p_k. Si $S_k(a_i, a_j) < \underline{e}(\Omega_{i,j})$, alors :*

- *si $costC = max$ alors*

$$[\underline{cost_k}, \left\lceil \frac{\underline{e}(\Omega_{i,j}) - S_k(a_i, a_j)}{min(\overline{ca_j}, sp_{k+1}) - max(\underline{sa_i}, sp_k)} \right\rceil [$$

  peut être retiré de $D(cost_k)$,
- *si $costC = sum$ alors*

$$[\underline{cost_k}, \underline{e}(\Omega_{i,j}) - S_k(a_i, a_j)[$$

  peut être retiré de $D(cost_k)$.

Preuve. Pour deux activités $a_i, a_j \in A$ données, $\underline{e}(\Omega_{i,j})$ est l'énergie nécessairement consommée dans l'intervalle $[\underline{sa_i}, \overline{ca_j}[$. Pour un intervalle $[max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})[$, tel que $sp_k \in \mathcal{SP}^K$,

$$\sum_{\substack{sp_l \neq sp_k \\ sp_l \in \mathcal{SP}^K}} (Area(max(\underline{sa_i}, sp_l), min(sp_{l+1}, \overline{ca_j})))$$

est l'énergie totale qui peut être consommée par les autres intervalles $[max(\underline{sa_i}, sp_l), min(sp_{l+1}, \overline{ca_j})[$, avec $sp_l \in \mathcal{SP}^K$ et $sp_l \neq sp_k$. $FreeArea(sp_k, min(sp_{k+1}, \overline{ca_j}))$ est l'énergie maximale qui peut être consommée dans l'intervalle $[max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})[$ sans engendrer de dépassement de capacité, par la Définition 41. Alors, si

$$FreeArea(max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})) + \sum_{\substack{sp_l \neq sp_k \\ sp_l \in \mathcal{SP}^K}} (Area(max(\underline{sa_i}, sp_l), min(sp_{l+1}, \overline{ca_j}))) < \underline{e}(\Omega_{i,j})$$

cela signifie que l'énergie restante, c'est à dire la différence

$$\underline{e}(\Omega_{i,j}) - FreeArea(max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})) - \sum_{\substack{sp_l \neq sp_k \\ sp_l \in \mathcal{SP}^K}} (Area(max(\underline{sa_i}, sp_l), min(sp_{l+1}, \overline{ca_j})))$$

est nécessairement placée dans l'intervalle $[max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})[$, et qu'elle y engendre un dépassement de capacité.

Si $costC = max$, le surplus d'énergie

$$\underline{e}(\Omega_{i,j}) - FreeArea(max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})) - \sum_{\substack{sp_l \neq sp_k \\ sp_l \in \mathcal{SP}^K}} (Area(max(\underline{sa_i}, sp_l), min(sp_{l+1}, \overline{ca_j})))$$

peut être réparti sur l'ensemble de l'intervalle $[max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})[$, avec une hauteur minimale de

$$\left\lceil \frac{\underline{e}(\Omega_{i,j}) - FreeArea(max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})) - \sum_{\substack{sp_l \neq sp_k \\ sp_l \in \mathcal{SP}^K}} (Area(max(\underline{sa_i}, sp_l), min(sp_{l+1}, \overline{ca_j})))}{min(\overline{ca_j}, sp_{k+1}) - max(\underline{sa_i}, sp_k)} \right\rceil$$

Celle-ci définit donc une borne inférieure de la variable de coût $cost_k$ associée à cet intervalle.

Si $costC = sum$, le surplus d'énergie

$$\underline{e}(\Omega_{i,j}) - FreeArea(max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_j})) - \sum_{\substack{sp_l \neq sp_k \\ sp_l \in \mathcal{SP}^K}} (Area(max(\underline{sa_i}, sp_l), min(sp_{l+1}, \overline{ca_j})))$$

est entièrement compris dans l'intervalle $[max(\underline{sa_i}, sp_k), min(sp_{k+1}, \overline{ca_i})[$. C'est donc une borne inférieure de la variable de coût $cost_j$ associée à cet intervalle. □

Notons que ces règles correspondent là encore à des cas de figure somme toute très particuliers.

Le calcul de $LB^K(a_i, a_j)$ est intégré dans l'algorithme d'Edge-Finding de Kameugne et al. et est réalisé en $O(p)$. Ainsi, la complexité globale, en intégrant cet algorithme dans l'Edge-Finding de Kameugne et al. est $O((p \times n)^2)$.

Le calcul de la borne inférieure des variables de coût $cost_k$ dans le cas de la règle de filtrage 9 peut être effectué en $O(1)$. Ceci n'augmente donc pas la complexité globale de l'algorithme.

Enfin, le calcul de la borne inférieure d'une variable de coût $cost_k$, pour chaque variable de coût, défini dans la règle de filtrage 10 est, dans le pire des cas, de l'ordre de $O(p)$: en chaque paire d'activités $a_i, a_j \in A$, nous considérons l'ensemble des intervalles utilisateur intersectant $[\underline{sa_i}, \overline{ca_j}[$. Pour chacun de ces intervalles utilisateur, nous faisons la somme des aires disponibles sur les autres intervalles utilisateur appartenant à $[\underline{sa_i}, \overline{ca_i}[$. Cependant, cette somme peut être calculée en conservant une donnée de manière incrémentale. Intégrer cette règle engendre donc une complexité globale de l'algorithme à $O((p \times n)^2)$.

5.1.3 Intégration des bornes inférieures de l'objectif

Nous avons montré dans la section 5.1.1 comment filtrer les variables de début d'activité en fonction du coût maximal de chaque intervalle utilisateur. De manière complémentaire, il est possible de filtrer celles-ci en fonction de la borne supérieure de la variable objectif.

Dans cette section, nous considérons que les procédures de filtrage de la section 5.1.1 ont été appliquées. Notre but est de filtrer les variables de début d'activités en utilisant les bornes inférieures de la variable objectif, calculées dans la section 5.1.2.

L'intégration de ces bornes se fait via trois procédures que nous détaillons ci-après. La première s'intègre dans l'algorithme de Sweep. La seconde et la troisième se basent sur des considérations énergétiques, et sont intégrées dans les algorithmes d'Edge-Finding de Vilím et de Kameugne et al.

5.1.3.1 Dans l'algorithme de Sweep

Nous utilisons la borne inférieure LB de l'objectif, calculée en section 5.1.2.1. Nous considérons des activités n'ayant pas de partie obligatoire dans le rectangle courant $\langle [\delta, \delta'[, sum_h \rangle$. Ces activités ne participent donc pas, dans ce rectangle, au calcul du profil cumulatif des parties obligatoires. Soient l'intervalle

utilisateur $p_j \in P$ tel que $[\delta, \delta'[\subseteq p_j$ et la capacité locale correspondante lc_j.

Le principe consiste à essayer d'ajouter une activité a_i dans l'intervalle $[\delta, \delta'[$, puis à calculer à partir du profil un coût minimal sous l'hypothèse que a_i s'exécute (entièrement, ou partiellement) dans l'intervalle $[\delta, \delta'[$. Si le coût calculé est strictement supérieur à $\underline{cost_j}$, alors nous pouvons considérer une borne inférieure LB augmentée, sous la condition que a_i soit exécutée dans l'intervalle $[\delta, \delta'[$. Nous notons ce coût $cost_j^{a_i}$.

Définition 43 ($cost_j^{a_i}$) *Soit une activité $a_i \in ActToPrune$, n'ayant pas de partie obligatoire dans le rectangle $\langle [\delta, \delta'[, sum_h \rangle$, et $p_j \in P$ l'unique intervalle utilisateur contenant $[\delta, \delta'[$. $cost_j^{a_i}$ est le coût minimal dans l'intervalle utilisateur p_j sous la condition que a_i s'exécute dans p_j. Il est égal à :*

- *si $costC = max$ alors $cost_j^{a_i} = max(sum_h + \underline{ra_i} - lc_j, 0)$,*
- *si $costC = sum$ alors nous considérons uniquement les activités a_i telles que $\delta \leq \underline{sa_i} < \delta'$ et $cost_j^{a_i} = max((\delta' - \delta) \times (sum_h - lc_j) + \min((\delta' - \underline{sa_i}) \times \underline{ra_i}, \underline{ea_i}), 0)$.*

Preuve. Si $costC = max$, l'ajout de l'activité a_i dans le rectangle $\langle [\delta, \delta'[, sum_h \rangle$ augmente la hauteur du profil cumulatif d'au minimum $\underline{ra_i}$. Une borne inférieure du coût obtenue en ajoutant a_i est donc, par définition $max(sum_h + \underline{ra_i} - lc_j, 0)$. On a alors $cost_j^{a_i} = max(sum_h + \underline{ra_i} - lc_j, 0)$.

Si $costC = sum$, si $\delta \leq \underline{sa_i} < \delta'$, alors l'énergie maximale engendrée par a_i dans l'intervalle $[\delta, \delta'[$ est égale à $\min((\delta' - \underline{sa_i}) \times \underline{ra_i}, \underline{ea_i})$, et le coût minimal dans cet intervalle, sans ajouter a_i est égal à $max((\delta' - \delta) \times (sum_h - lc_j), 0)$, par définition. L'ajout de a_i engendre donc un coût minimal de $cost_j^{a_i} = max((\delta' - \delta) \times (sum_h - lc_j) + \min((\delta' - \underline{sa_i}) \times \underline{ra_i}, \underline{ea_i}), 0)$. On a alors : $cost_j^{a_i} = max((\delta' - \delta) \times (sum_h - lc_j) + \min((\delta' - \underline{sa_i}) \times \underline{ra_i}, \underline{ea_i}), 0)$. □

Une fois ce coût calculé, il est possible, à partir des bornes de la variable objectif, de réaliser un filtrage sur la date de début des activités. Nous détaillons ci-après cette règle de filtrage, en rappelant que nous pouvons l'intégrer dans l'algorithme de Sweep.

Règle de filtrage 11 *Soit une activité $a_i \in ActToPrune$, n'ayant pas de partie obligatoire dans le rectangle $\langle [\delta, \delta'[, sum_h \rangle$, et p_j l'unique intervalle contenant $[\delta, \delta'[$:*

- *si $costC = max$ and $objC = sum$ alors si $LB + max(cost_j^{a_i} - \underline{cost_j}, 0) > \overline{obj}$ alors $]\delta - \underline{da_i}, \delta'[$ peut être retiré de $D(sa_i)$,*
- *si $costC = max$ and $objC = max$ alors si $cost_j^{a_i} > \overline{obj}$ alors $]\delta - \underline{da_i}, \delta'[$ peut être retiré de $D(sa_i)$,*
- *si $costC = sum$ and $objC = sum$ alors si $LB + max(cost_j^{a_i} - \underline{cost_j}, 0) > \overline{obj}$ alors*

$$[\underline{sa_i}, \min(\delta', \delta' - \left\lfloor \frac{(\overline{obj} - LB + \underline{cost_j}) + (lc_j - sum_h) \times (\delta' - \delta)}{\underline{ra_i}} \right\rfloor)[$$

peut être retiré de $D(sa_i)$,

- *si $costC = sum$ and $objC = max$ alors si $cost_j^{a_i} > \overline{obj}$ alors*

$$[\underline{sa_i}, \min(\delta', \delta' - \left\lfloor \frac{\overline{obj} + (lc_j - sum_h) \times (\delta' - \delta)}{\underline{ra_i}} \right\rfloor)[$$

peut être retiré de $D(sa_i)$.

Preuve. $cost_j^{a_i}$ représente le coût minimum dans un intervalle $[\delta, \delta'[\subseteq p_j$ si nous ajoutons l'activité a_i à $[\delta, \delta'[$. Cette activité n'a pas de partie obligatoire dans ce rectangle. Elle n'est donc pas prise en compte dans le calcul de la borne inférieure de l'objectif.

Posons $objC = sum$.

Par la Définition 36, l'augmentation de LB est $max(cost_j^{a_i} - \underline{cost_j}, 0)$. Si cette augmentation est supérieure à la marge autorisée par $\overline{obj}$, alors nous pouvons filtrer.

Si $costC = max$, tout point de temps occupé par a_i dans $[\delta, \delta'[$ peut être responsable de l'augmentation. Alors, a_i ne peut pas, même partiellement, s'exécuter dans l'intervalle $[\delta, \delta'[$. Alors, $]\delta - \underline{da_i}, \delta'[$ peut être retiré de $D(sa_i)$.

Si $costC = sum$, le coût maximal $cost_j$ autorisé par $\overline{obj}$ est $\overline{obj} - LB + \underline{cost_j}$. Considérons $\underline{sa_i} \in [\delta, \delta'[$. Si $\underline{ca_i} < \delta'$, sa contribution énergétique minimale dans l'intervalle $[\delta, \delta'[$ si a_i est placée est placée à sa date de début au plus tôt est $\underline{ea_i}$. Sinon, le nombre maximal de points de temps où a_i s'exécute, dans l'intervalle $[\delta, \delta'[$, est égal à $\delta' - \underline{sa_i}$. Par conséquent, sa contribution énergétique minimale dans $[\delta, \delta'[$ est égale à $\underline{ra_i} \times (\delta' - \underline{sa_i})$. L'ajout de a_i à la date $\underline{sa_i}$ implique une augmentation du coût égale à $max(cost_j^{a_i} - \underline{cost_j}, 0)$. Si $LB + max(cost_j^{a_i} - \underline{cost_j}, 0) > \overline{obj}$, alors cela signifie que a_i ne peut démarrer à $\underline{sa_i}$. L'augmentation de la valeur de l'objectif engendrée par l'ajout de a_i dans $[\delta, \delta'[$ doit donc être telle que $LB + max(cost_j^{a_i} - \underline{cost_j}, 0) \leq \overline{obj}$, ce qui implique : $LB + max(max((\delta' - \delta) \times (sum_h - lc_j) + \min((\delta' - \underline{sa_i}) \times \underline{ra_i}, \underline{ea_i}), 0) - \underline{cost_j}, 0) \leq \overline{obj}$, ce qui est équivalent à $LB + (\delta' - \delta) \times (sum_h - lc_j) + \min((\delta' - \underline{sa_i}) \times \underline{ra_i}, \underline{ea_i}) - \underline{cost_j} \leq \overline{obj}$. Pour respecter cette inégalité :

- soit a_i ne peut s'exécuter dans l'intervalle $[\delta, \delta'[$, dans ce cas, $[\underline{sa_i}, \delta'[$ est retiré de $D(sa_i)$,
- dans le cas contraire, on a : $(\delta' - \delta) \times (sum_h - lc_j) + \min((\delta' - \underline{sa_i}) \times \underline{ra_i}, \underline{ea_i}) \leq \overline{obj} + \underline{cost_j} - LB$. Or, a_i ne peut être totalement comprise dans $[\delta, \delta'[$ sans engendrer un dépassement trop important, dans ce cas, on doit avoir : $(\delta' - \delta) \times (sum_h - lc_j) + (\delta' - \underline{sa_i}) \times \underline{ra_i} \leq \overline{obj} + \underline{cost_j} - LB$, ce qui est équivalent à $(\delta' - \underline{sa_i}) \times \underline{ra_i} \leq (\overline{obj} + \underline{cost_j} - LB) - (\delta' - \delta) \times (sum_h - lc_j)$, et encore : $(\delta' - \underline{sa_i}) \leq \frac{(\overline{obj} + \underline{cost_j} - LB) - (\delta' - \delta) \times (sum_h - lc_j)}{\underline{ra_i}}$, soit $\underline{sa_i} \geq \delta' - \frac{(\overline{obj} - LB + \underline{cost_j}) - (\delta' - \delta) \times (sum_h - lc_j)}{\underline{ra_i}}$. $\underline{sa_i}$ étant entière, on obtient $\underline{sa_i} \geq \delta' - \left\lfloor \frac{(\overline{obj} - LB + \underline{cost_j}) + (lc_j - sum_h) \times (\delta' - \delta)}{\underline{ra_i}} \right\rfloor [$.

Alors, si $LB + max(cost_j^{a_i} - \underline{cost_j}, 0) > \overline{obj}$, $[\underline{sa_i}, \min(\delta', \delta' - \left\lfloor \frac{(\overline{obj} - LB + \underline{cost_j}) + (lc_j - sum_h) \times (\delta' - \delta)}{\underline{ra_i}} \right\rfloor [$ peut être retiré de $D(sa_i)$.

Le raisonnement est similaire pour $objC = max$ excepté le fait que l'on ne mesure pas une augmentation, mais nous comparons directement le coût induit avec $\overline{obj}$. □

5.1.3.2 Dans l'Edge-Finding de Vilím

Nous voulons utiliser la borne inférieure $LB^V(a_i)$ de la variable objectif, calculée en section 5.1.2.2 de manière à filtrer les dates de début d'activité.

L'algorithme d'Edge-Finding de Vilím calcule, pour chaque activité $a_i \in A$, l'énergie minimale $\underline{e}(LCut(A, a_i))$. Nous nous servons de cette quantité, et essayons d'ajouter une activité a_j telle que $\underline{sa_j} < \overline{ca_i}$ et $a_j \notin LCut(A, a_i)$ dans l'intervalle $[sp_0, \overline{ca_i}[$. Nous calculons ensuite l'énergie totale obtenue en considérant que a_j s'exécute (totalement ou partiellement) dans l'intervalle $[sp_0, \overline{ca_i}[$. Celle-ci est égale à $\underline{e}(LCut(A, a_i)) + e^V(a_i, a_j)$, que nous définissons ci-après. Nous rappelons que, comme $a_j \notin LCut(A, a_i)$, elle n'a pas été prise en compte dans les calculs de $\underline{e}(LCut(A, a_i))$ et $LB^V(a_i)$.

Définition 44 *Soient a_i et a_j deux activités telles que $sp_0 \leq \underline{sa_j} < \overline{ca_i}$. Nous définissons $e^V(a_i, a_j)$ comme étant l'énergie de a_j dans l'intervalle $[sp_0, \overline{ca_i}[$ sous la condition que a_j commence en $\underline{sa_j}$: $e^V(a_i, a_j) = min(\underline{ea_j}, (\overline{ca_i} - \underline{sa_j}) \times \underline{ra_j})$.*

Si la différence entre l'énergie calculée et l'aire disponible dans cet intervalle est strictement supérieure à $LB^V(a_i)$, alors nous pouvons considérer une borne inférieure $LB^V(a_i)$ augmentée, sous la condition que a_j s'exécute dans l'intervalle $[sp_0, \overline{ca_i}[$. Alors, si cette borne inférieure est strictement supérieure à $\overline{obj}$,

nous pouvons filtrer sa_j. Nous rappelons que $\mathcal{P}^V$ est l'ensemble contenant tous les intervalles utilisateur $p_k \in P$ tels que $sp_k \leq \overline{ca_i}$.

Règle de filtrage 12 *Soient a_i et a_j deux activités dans A telles que $\underline{sa_j} < \overline{ca_i}$, $a_j \notin LCut(A, a_i)$, et $LB^V(a_i) > 0$ la borne inférieure de la variable obj calculée en 5.1.2.2 :*

- *si $costC = max$ et $objC = sum$ alors si $LB^V(a_i) + \left\lceil \frac{e^V(a_i,a_j)}{\max_{p_k \in \mathcal{P}^V}(min(\overline{ca_i}, sp_{k+1}) - sp_k))} \right\rceil > \overline{obj}$, alors*

$$[\underline{sa_j}, min(\overline{ca_i}, \overline{ca_i} - \left\lfloor \frac{(\overline{obj} - LB^V(a_i)) \times (\max_{p_j \in \mathcal{P}^V}(min(\overline{ca_i}, sp_{j+1}) - sp_j))}{\underline{ra_j}} \right\rfloor)[$$

  peut être retiré de $D(sa_j)$,
- *si $costC = max$ et $objC = max$ alors si $LB^V(a_i) + \left\lceil \frac{e^V(a_i,a_j)}{\overline{ca_i} - sp_0} \right\rceil > \overline{obj}$ alors*

$$[\underline{sa_j}, min(\overline{ca_i}, \overline{ca_i} - \left\lfloor \frac{(\overline{obj} - LB^V(a_i)) \times (\overline{ca_i} - sp_0)}{\underline{ra_j}} \right\rfloor [$$

  peut être retiré de $D(sa_j)$,
- *si $costC = sum$ et $objC = sum$ alors si $LB^V(a_i) + e^V(a_i, a_j) > \overline{obj}$ alors*

$$[\underline{sa_j}, min(\overline{ca_i}, \overline{ca_i} - \left\lfloor \frac{\overline{obj} - LB^V(a_i)}{\underline{ra_j}} \right\rfloor [$$

  peut être retiré de $D(sa_j)$,
- *si $costC = sum$ et $objC = max$ alors si $LB^V(a_i) + \left\lceil \frac{e^V(a_i,a_j)}{|\mathcal{P}^V|} \right\rceil > \overline{obj}$ alors*

$$[\underline{sa_j}, min(\overline{ca_i}, \overline{ca_i} - \left\lfloor \frac{(\overline{obj} - LB^V(a_i)) \times |\mathcal{P}^V|}{\underline{ra_j}} \right\rfloor [$$

  peut être retiré de $D(sa_j)$.

Preuve. Selon la Définition 44, $e^V(a_i, a_j)$ est l'énergie de a_j dans l'intervalle $[sp_0, \overline{ca_i}[$ sous la condition que a_j commence en $\underline{sa_j}$. $a_j \notin LCut(A, a_i)$, et a_j n'intervient pas dans le calcul de $\underline{e}(LCut(A, a_i))$, et à plus forte raison dans le calcul de $LB^V(a_i)$, dans la proposition 5. Alors, ajouter a_j dans $[sp_0, \overline{ca_i}[$ cause une augmentation de l'énergie qui y est contenue égale à $e^V(a_i, a_j)$. Si $LB^V(a_i) > 0$, alors l'énergie contenue dans cet intervalle engendre déjà un dépassement, et l'ajout de a_j ne peut que l'augmenter.

Posons $costC = max$ et $objC = sum$. Sans perte de généralité, nous considérons que toute l'énergie $e^V(a_i, a_j)$ peut être placée dans l'intervalle de longueur maximale $\mathcal{I}$ commençant en sp_k. Nous nous plaçons alors dans l'intervalle $\max_{p_k \in \mathcal{P}^V}(min(\overline{ca_i}, sp_{k+1}) - sp_k)$. $e^V(a_i, a_j)$ est alors répartie dans cet intervalle. En effet, considérer un ou plusieurs intervalles supplémentaires ayant, par définition, une longueur inférieure ou égale à la longueur de $\mathcal{I}$, mènerait nécessairement à une somme des dépassements plus grande ou égale. Alors l'augmentation minimale de l'objectif, c'est à dire la hauteur minimale engendrée par la répartition du surplus d'énergie, liée à l'ajout de a_j est égale à $\left\lceil \frac{e^V(a_i,a_j)}{\max_{p_k \in \mathcal{P}^V}(min(\overline{ca_i}, sp_{k+1}) - sp_k))} \right\rceil$. Si, ajoutée à $LB^V(a_i)$, cette augmentation dépasse la borne supérieure de la variable objectif, la contribution de a_j dans cet intervalle doit être réduite, et a_j ne peut commencer en $\underline{sa_j}$. sa_j est donc filtrée de manière à ce que sa contribution énergétique maximale dans $[sp_0, \overline{ca_i}[$ engendre une augmentation maximale de la borne inférieure de l'objectif au plus égale à $\overline{obj} - LB^V(a_i)$. Pour cela, $[\underline{sa_j}, min(\overline{ca_i}, \overline{ca_i} - \left\lfloor \frac{(\overline{obj} - LB^V(a_i)) \times (\max_{p_k \in \mathcal{P}^V}(min(\overline{ca_i}, sp_{k+1}) - sp_k))}{\underline{ra_j}} \right\rfloor))$ est retiré de $D(sa_j)$.

Si $costC = max$ et $objC = max$ alors on mesure, de la même manière, l'augmentation de la valeur d'objectif engendrée lorsque l'on ajoute a_j dans $[sp_0, \overline{ca_i}[$. Celle-ci correspond à la répartition de $e^V(a_i, a_j)$ sur le plus grand intervalle possible, c'est à dire : $[sp_0, \overline{ca_i}[$. Cette augmentation est au minimum égale à $\left\lceil \frac{e^V(a_i,a_j)}{\overline{ca_i} - sp_0} \right\rceil$. Si cette augmentation ajoutée à $LB^V(a_i)$ dépasse $\overline{obj}$, il faut retarder le début de l'activité a_j de manière à réduire sa contribution dans $[sp_0, \overline{ca_i}[$. Pour cela, $[\underline{sa_j}, min(\overline{ca_i}, \overline{ca_i} - \left\lfloor \frac{(\overline{obj} - LB^V(a_i)) \times (\overline{ca_i} - sp_0)}{\underline{ra_j}} \right\rfloor [$ est retiré de $D(sa_j)$.

Si $costC = sum$ et $objC = sum$, l'augmentation de la valeur d'objectif engendrée lorsque l'on ajoute a_j dans $[sp_0, \overline{ca_i}[$ est la même quelle que soit la distribution des dépassements sur les intervalles utilisateurs. Elle est égale à $e^V(a_i, a_j)$. Si cette augmentation ajoutée à $LB^V(a_i)$ dépasse $\overline{obj}$, il faut retarder le début de l'activité a_j de manière à réduire sa contribution dans $[sp_0, \overline{ca_i}[$. Pour cela, $[\underline{sa_j}, min(\overline{ca_i}, \overline{ca_i} - \left\lfloor \frac{\overline{obj} - LB^V(a_i)}{\underline{ra_j}} \right\rfloor [$ est retiré de $D(sa_j)$.

Si $costC = sum$ et $objC = max$, en répartissant le surplus d'énergie $e^V(a_i, a_j)$ sur les intervalles de $\mathcal{P}^V$ on obtient une borne inférieure de l'augmentation de la valeur d'objectif engendrée lorsque l'on ajoute a_j dans $[sp_0, \overline{ca_i}[$. Celle-ci est égale à $\left\lceil \frac{e^V(a_i,a_j)}{|\mathcal{P}^V|} \right\rceil$. Si cette augmentation ajoutée à $LB^V(a_i)$ dépasse $\overline{obj}$, il faut retarder le début de l'activité a_j de manière à réduire sa contribution dans $[sp_0, \overline{ca_i}[$. Pour cela, $[\underline{sa_j}, min(\overline{ca_i}, \overline{ca_i} - \left\lfloor \frac{(\overline{obj} - LB^V(a_i)) \times |\mathcal{P}^V|}{\underline{ra_j}} \right\rfloor [$ est retiré de $D(sa_j)$. □

5.1.3.3 Dans l'Edge-Finding de Kameugne et al.

Nous considérons la borne inférieure $LB^K(a_i, a_j)$ de la variable objectif, calculée en section 5.1.2.3. Nous essayons d'ajouter une activité a_l telle que $\underline{sa_i} \leq \underline{sa_l} < \overline{ca_j}$ et $a_l \notin \Omega_{i,j}$ dans l'intervalle $[\underline{sa_i}, \overline{ca_j}[$. L'algorithme d'Edge-Finding de Kameugne et al. calcule, pour toutes activités $a_i, a_j \in A$, l'énergie minimale $\underline{e}(\Omega_{i,j})$ nécessairement contenue dans l'intervalle $[\underline{sa_i}, \overline{ca_j}[$.

Nous calculons ensuite l'énergie totale obtenue en considérant que a_l s'exécute (totalement ou partiellement) dans l'intervalle $[\underline{sa_i}, \overline{ca_j}[$. Celle-ci est égale à $\underline{e}(\Omega_{i,j}) + e(a_l, a_i, a_j)$. Nous définissons ce terme ci-après. Nous rappelons que a_l n'a pas été prise en compte dans le calcul de $\underline{e}(\Omega_{i,j})$, et de $LB^K(a_i, a_j)$, car $a_l \notin \Omega_{i,j}$.

Définition 45 *Soient a_i, a_j et a_l trois activités telles que $\underline{sa_i} \leq \underline{sa_l} < \overline{ca_j}$. Nous définissons $e^K(a_l, a_i, a_j)$ comme étant l'énergie de a_l dans l'intervalle $[\underline{sa_i}, \overline{ca_j}[$ sous la condition que a_l commence en $\underline{sa_l}$: $e^K(a_l, a_i, a_j) = min(\underline{ea_l}, (\overline{ca_j} - \underline{sa_l}) \times \underline{ra_l})$.*

Si la différence entre l'énergie calculée et l'aire disponible dans cet intervalle est strictement supérieure à $LB^K(a_i, a_j)$, alors nous pouvons considérer une borne inférieure $LB^K(a_i, a_j)$ augmentée, sous la condition que a_l s'exécute dans l'intervalle $[\underline{sa_i}, \overline{ca_j}[$. Alors, si cette borne inférieure est strictement supérieure à $\overline{obj}$, nous pouvons filtrer sa_i. Nous rappelons que $\mathcal{P}^K$ est l'ensemble contenant tous les intervalles utilisateurs $p_k \in P$ tels que $(ep_k \geq \underline{sa_i} \wedge sp_k \leq \overline{ca_j})$

Règle de filtrage 13 *Soient a_i, a_j et a_l trois activités dans A telles que $\underline{sa_i} \leq \underline{sa_l} < \overline{ca_j}$, $a_l \notin \Omega_{i,j}$, et $LB^K(a_i, a_j) > 0$ la borne inférieure de la variable obj calculée en 5.1.2.3. Nous rappelons que a_l n'a pas été prise en compte dans le calcul de $\underline{e}(\Omega_{i,j})$, et de $LB^K(a_i, a_j)$, car $a_l \notin \Omega_{i,j}$. Alors :*

- *si $costC = max$ et $objC = sum$ alors si $LB^K(a_i, a_j) + \left\lceil \frac{e^K(a_l,a_i,a_j)}{\max_{p_k \in \mathcal{P}^K}(min(\overline{ca_j}, sp_{k+1}) - max(\underline{sa_i}, sp_k))} \right\rceil > \overline{obj}$, alors*

$$[\underline{sa_l}, min(\overline{ca_j}, \overline{ca_j} - \left\lfloor \frac{(\overline{obj} - LB^K(a_i, a_j)) \times (\max_{p_k \in \mathcal{P}^K}(min(\overline{ca_j}, sp_{k+1}) - max(\underline{sa_i}, sp_k)))}{\underline{ra_l}} \right\rfloor)[$$

peut être retiré de $D(sa_l)$,

– *si* $costC = max$ *et* $objC = max$ *alors si* $LB^K(a_i, a_j) + \left\lceil \frac{e^K(a_l,a_i,a_j)}{\overline{ca_j} - \underline{sa_i}} \right\rceil > \overline{obj}$ *alors*

$$[\underline{sa_l}, min(\overline{ca_j}, \overline{ca_j} - \left\lfloor \frac{(\overline{obj} - LB^K(a_i, a_j)) \times (\overline{ca_j} - \underline{sa_i})}{\underline{ra_l}} \right\rfloor [$$

peut être retiré de $D(sa_l)$,

– *si* $costC = sum$ *et* $objC = sum$ *alors si* $LB^K(a_i, a_j) + e^K(a_l, a_i, a_j) > \overline{obj}$ *alors*

$$[\underline{sa_l}, min(\overline{ca_j}, \overline{ca_j} - \left\lfloor \frac{\overline{obj} - LB^K(a_i, a_j)}{\underline{ra_l}} \right\rfloor [$$

peut être retiré de $D(sa_l)$,

– *si* $costC = sum$ *et* $objC = max$ *alors si* $LB^K(a_i, a_j) + \left\lceil \frac{e^K(a_l,a_i,a_j)}{|\mathcal{P}^K|} \right\rceil > \overline{obj}$ *alors*

$$[\underline{sa_l}, min(\overline{ca_j}, \overline{ca_j} - \left\lfloor \frac{(\overline{obj} - LB^K(a_i, a_j)) \times |\mathcal{P}^K|}{\underline{ra_l}} \right\rfloor [$$

peut être retiré de $D(sa_l)$.

Preuve. Selon la Définition 45, $e^K(a_l, a_i, a_j)$ est l'énergie de a_l dans l'intervalle $[\underline{sa_i}, \overline{ca_j}[$ sous la condition que a_l commence en $\underline{sa_l}$. Le calcul de $LB^K(a_i, a_j)$, dans la proposition 5 ne prend pas en compte l'activité $a_l \notin \Omega_{i,j}$. Alors, ajouter a_l dans $[\underline{sa_i}, \overline{ca_j}[$ cause une augmentation de l'énergie qui y est contenue égale à $e^K(a_l, a_i, a_j)$. Si $LB^K(a_i, a_j) > 0$, alors l'énergie contenue dans cet intervalle engendre déjà un dépassement, et l'ajout de a_l ne peut que l'augmenter.

Posons $costC = max$ et $objC = sum$. Sans perte de généralité, nous considérons que toute l'énergie $e^K(a_l, a_i, a_j)$ peut être placée dans l'intervalle de longueur maximale $\mathcal{I}$ commençant en $max(\underline{sa_i}, sp_k)$. Cet intervalle a une longueur égale à $\max_{p_k \in \mathcal{P}^K}(min(\overline{ca_j}, sp_{k+1}) - max(\underline{sa_i}, sp_k))$. $e^K(a_l, a_i, a_j)$ est alors répartie dans cet intervalle. En effet, considérer un ou plusieurs intervalles supplémentaires ayant, par définition, une longueur inférieure ou égale à la longueur de $\mathcal{I}$, mènerait nécessairement à une somme des dépassements plus grande ou égale (nous rappelons que nous ne prenons pas en compte les limites maximales sur les coûts). Alors l'augmentation minimale de l'objectif liée à l'ajout de a_l, c'est à dire, la hauteur minimale engendrée par la répartition du surplus d'énergie engendré par a_l, est égale à $\left\lceil \frac{e^K(a_l,a_i,a_j)}{\max_{p_k \in \mathcal{P}^K}(min(\overline{ca_j}, sp_{k+1}) - max(\underline{sa_i}, sp_k))} \right\rceil$. Si, ajoutée à $LB^K(a_i, a_j)$, cette augmentation dépasse la borne supérieure de la variable objectif, la contribution de a_l dans cet intervalle doit être réduite, et a_l ne peut commencer en $\underline{sa_l}$. sa_l est donc filtrée de manière à ce que sa contribution énergétique maximale dans $[\underline{sa_i}, \overline{ca_j}[$ engendre une augmentation maximale de la borne inférieure de l'objectif au plus égale à $\overline{obj} - LB^K(a_i, a_j)$. Pour cela, $[\underline{sa_l}, min(\overline{ca_j}, \overline{ca_j} - \left\lfloor \frac{(\overline{obj} - LB^K(a_i,a_j)) \times (\max_{p_k \in \mathcal{P}^K}(min(\overline{ca_j}, sp_{k+1}) - max(\underline{sa_i}, sp_k)))}{\underline{ra_l}} \right\rfloor))$ est retiré de $D(sa_l)$.

Si $costC = max$ et $objC = max$ alors on mesure, de la même manière, l'augmentation de la valeur d'objectif engendrée lorsque l'on ajoute a_l dans $[\underline{sa_i}, \overline{ca_j}[$. Celle-ci correspond à la répartition, avec une hauteur minimale de $e^K(a_l, a_i, a_j)$ sur le plus grand intervalle possible, c'est à dire : $[\underline{sa_i}, \overline{ca_j}[$. Cette augmentation est donc égale à $\left\lceil \frac{e^K(a_l,a_i,a_j)}{\overline{ca_j} - \underline{sa_i}} \right\rceil$. Si cette augmentation ajoutée à $LB^K(a_i, a_j)$ dépasse $\overline{obj}$, il faut retarder le début de l'activité a_l de manière à réduire sa contribution dans $[\underline{sa_i}, \overline{ca_j}[$. Pour cela, $[\underline{sa_l}, min(\overline{ca_j}, \overline{ca_j} - \left\lfloor \frac{(\overline{obj} - LB^K(a_i,a_j)) \times (\overline{ca_j} - \underline{sa_i})}{\underline{ra_l}} \right\rfloor [$ est retiré de $D(sa_l)$.

Si $costC = sum$ et $objC = sum$, l'augmentation de la valeur d'objectif engendrée lorsque l'on ajoute a_l dans $[\underline{sa_i}, \overline{ca_j}[$ est la même quelle que soit la distribution des dépassements sur les intervalles utilisateurs. Elle est égale à $e^K(a_l, a_i, a_j)$. Si cette augmentation ajoutée à $LB^K(a_i, a_j)$ dépasse $\overline{obj}$, il faut retarder le

début de l'activité a_l de manière à réduire sa contribution dans $[\underline{sa_i}, \overline{ca_j}[$. Pour cela, $[\underline{sa_l}, min(\overline{ca_j}, \overline{ca_j} - \left\lfloor \frac{\overline{obj} - LB^K(a_i,a_j)}{\underline{ra_l}} \right\rfloor [$ est retiré de $D(sa_l)$.

Si $costC = sum$ et $objC = max$, en étalant le surplus d'énergie $e^K(a_l, a_i, a_j)$ sur les intervalles de $\mathcal{P}^K$ on obtient une borne inférieure de l'augmentation de la valeur d'objectif engendrée lorsque l'on ajoute a_l dans $[\underline{sa_i}, \overline{ca_j}[$. Celle-ci est égale à $\left\lceil \frac{e^K(a_l,a_i,a_j)}{|\mathcal{P}^K|} \right\rceil$. Si cette augmentation ajoutée à $LB^K(a_i, a_j)$ dépasse $\overline{obj}$, il faut retarder le début de l'activité a_l de manière à réduire sa contribution dans $[\underline{sa_i}, \overline{ca_j}[$. Pour cela, $[\underline{sa_l}, min(\overline{ca_j}, \overline{ca_j} - \left\lfloor \frac{(\overline{obj} - LB^K(a_i,a_j)) \times |\mathcal{P}^K|}{\underline{ra_l}} \right\rfloor [$ est retiré de $D(sa_l)$. □

5.1.4 Filtrage à partir des bornes inférieures des domaines des variables de coût

En PPC, dans des problèmes sur-contraints, il est généralement admis que les violations de contraintes doivent être minimisées. Dans notre cas, les variables de coût représentent ces violations. À première vue, il semblerait donc que seul un filtrage à partir des bornes supérieures des domaines des variables de coût soit utile.
Cependant, la discussion menée dans la section 4.3.2.4 montre qu'il peut être intéressant de faire des déductions sur les valeurs minimales des domaines des variables dans *Cost*.

Dans ce but, nous introduisons une nouvelle technique de filtrage. Celle-ci filtre les dates de début d'activité en fonction de la borne inférieure des différentes variables de coût. Elle est basée sur la notion de partie enveloppante qui est duale à la notion de partie obligatoire. Nous la définissons, puis l'illustrons par l'exemple 38.

Définition 46 (Partie enveloppante) *La* Partie enveloppante $ep(a_i)$ *d'une activité $a_i \in A$, est l'union de toutes les instances réalisables de a_i. Elle est définie par $[\underline{sa_i}, \overline{ca_i}[$ et une hauteur égale à $\overline{ra_i}$ dans l'intervalle $[\underline{sa_i}, \overline{ca_i}[$, et nulle ailleurs.*

Exemple 38 *La Figure 5.15 représente en hâchures la partie enveloppante de l'activité a_3 introduite dans le problème représenté en Figure 5.2. Cette partie enveloppante s'étend de $\underline{sa_3} = 0$ à $\overline{ca_3} = 8$, avec une hauteur de $\overline{ra_3} = 2$.*

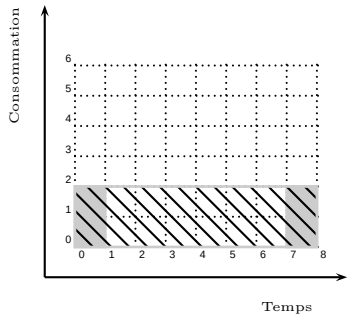


FIGURE 5.15 – La partie enveloppante de l'activité a_3 est représentée en hâchures. a_3 est telle que $D(sa_3) = [0, 7]$ $D(ca_3) = [1, 8]$ $D(ra_3) = [2, 2]$ et représentée par ses positions extrêmes sur la ligne de temps.

L'union des parties enveloppantes de toutes les activités d'un problème forme le profil cumulatif enveloppant, que nous définissons ci-après.

Définition 47 (Profil cumulatif enveloppant) *Le* Profil cumulatif enveloppant *$EnvP$ est la consommation maximale cumulée, au cours du temps, de toutes les activités. Pour un point de temps donné t, la hauteur de $EnvP$ en t est égale à $\sum_{\substack{a_i \in A \\ t \in [\underline{sa_i}, \overline{ca_i}[}} \overline{ra_i}$, soit la somme des contributions de toutes les parties enveloppantes passant en t.*

Ce profil représente, en chaque point de temps t, la hauteur maximale atteignable si toutes les activités $a_i \in A$ telles que $t \in [\underline{sa_i}, \overline{ca_i}]$ passent effectivement par t. L'exemple 39 illustre cette notion. Il présente également un problème dans lequel la valeur minimale d'une variable de coût est imposée.

Exemple 39 *La Figure 5.16 reprend le problème introduit dans la Figure 5.2, légèrement modifié. L'activité a_4 est fixée, et fait donc partie entièrement du profil des parties obligatoires. Nous ajoutons l'activité a_5 introduite sur la Figure 5.11, et l'intervalle p_1 a désormais une borne supérieure du coût égale à 4. Nous nous plaçons dans le cas costC= max. La Figure montre son profil cumulatif enveloppant, avec son dual, le profil cumulatif des parties obligatoires. Ce dernier impose un dépassement de 2 sur le deuxième intervalle utilisateur. Nous ajoutons une contrainte : $|cost_{i+1} - cost_i| \leq 1$. Par propagation de la borne inférieure de $cost_1$, un dépassement d'au moins 1 sur le premier, et le troisième intervalle est imposé.*

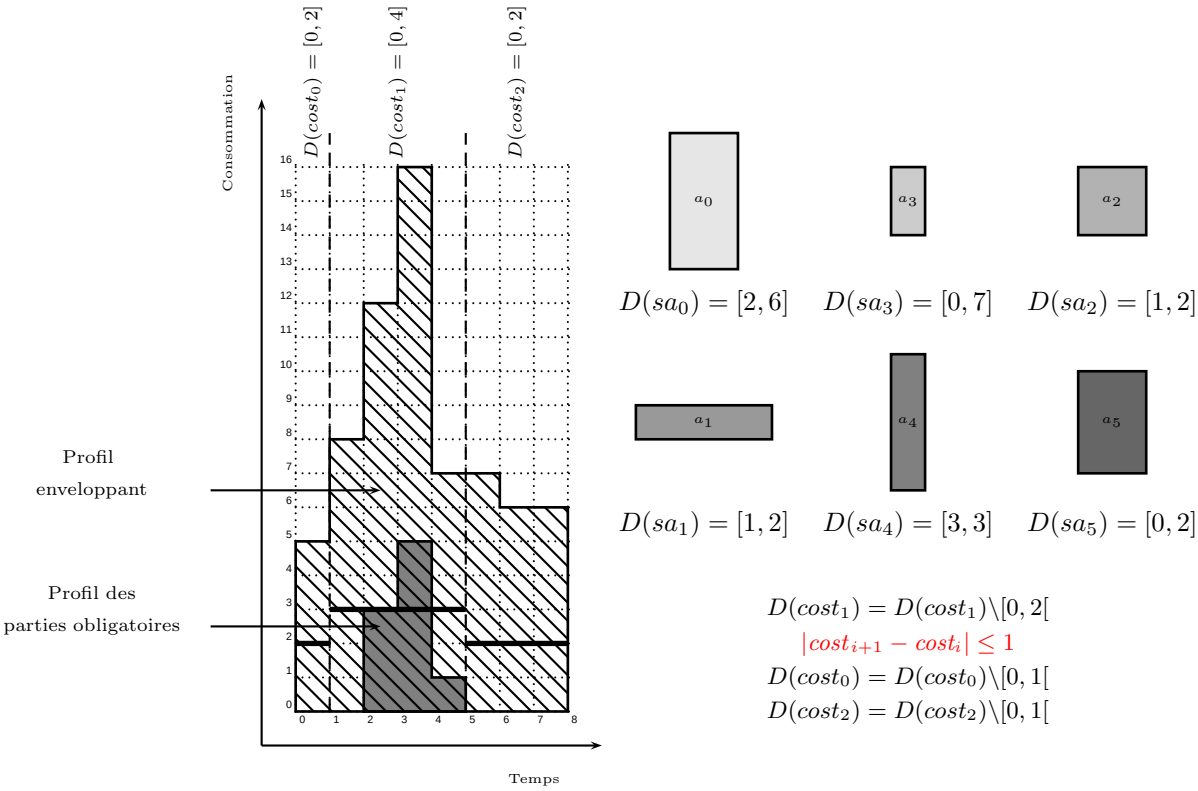


FIGURE 5.16 – Le profil cumulatif des parties obligatoires (plein), et le profil enveloppant (hâchuré), pour un problème cumulatif relaxé à six activités, représenté par la contrainte SOFTCUMULATIVE, tel que : $D(sa_0) = [2, 6]$, $D(sa_1) = [1, 2]$,$D(sa_2) = [1, 2]$, $D(sa_3) = [0, 7]$, $D(sa_4) = [3, 3]$ et $D(sa_5) = [0, 2]$, avec des durées et hauteurs fixées pour chaque activité. Les activités a_1, a_2 et a_4 ont une partie obligatoire (respectivement dans les intervalles $[2, 5[$, $[2, 3[$, et $[3, 4[$)qui construisent le profil cumulatif. Les parties enveloppantes de toutes les activités forment le profil enveloppant. Ces parties enveloppantes prennent la hauteur de chaque activité, sur les intervalles suivants, respectivement : $[2, 8[$, $[1, 6[$, $[1, 3[$, $[0, 8[$, $[3, 4[$, $[0, 4[$. 2 est une borne inférieure de $cost_1$, imposée par le profil des parties obligatoires. Nous ajoutons la contrainte suivante : $\forall i \in [0, 1]$, $|cost_{i+1} - cost_i| \leq 1$. En conséquence, 1 est une borne inférieure pour $cost_0$ et pour $cost_2$.

Pour filtrer les activités à partir des valeurs minimales des variables de coût, nous appliquons une procédure de Sweep sur le profil cumulatif enveloppant et sur le profil des parties obligatoires.

L'idée est la suivante : le profil cumulatif enveloppant donne, en chaque point de temps, la hauteur maximale atteignable par le profil cumulatif. Les variables de coût et d'objectif étant basées sur les dépassements de capacité, il faut s'assurer, pour atteindre une valeur de coût minimale, de pouvoir l'atteindre avec les domaines courants des variables de début et de fin des activités.

Nous proposons d'ajouter à l'algorithme de Sweep présenté en section 5.1.1.1 une nouvelle classe d'événements : les événements d'enveloppe. Ceux-ci correspondent au début et la fin de la partie enveloppante de chaque activité. Pour une activité $a_i \in A$, de tels événements sont notés $< ENVELOPE, a_i, \underline{sa_i}, \overline{ra_i} >$ et $< ENVELOPE, a_i, \overline{ca_i}, -\overline{ra_i} >$ respectivement.

Ces nouveaux événements ne modifient pas le calcul incrémental de sum_h dans l'algorithme de Sweep. De plus, nous considérons une nouvelle hauteur sum_e, qui est la hauteur du profil cumulatif enveloppant dans le rectangle courant.

Intuitivement, le principe du filtrage est le suivant : Lors de la construction du profil enveloppant $EnvP$, à chaque événement de début d'intervalle utilisateur sp_j, nous vérifions s'il y a un unique rectangle $\langle [\delta, \delta'[, sum_e \rangle$ de $EnvP$ entre sp_j et ep_j tel que $sum_e \geq lc_j + \underline{cost_j}$. Si c'est le cas, nous pouvons filtrer les activités a_i en suivant la règle suivante :

1. nous retirons de $EnvP$ la contribution de a_i,
2. nous filtrons les bornes de $D(sa_i)$ de manière à s'assurer que, si a_i commence en $\underline{sa_i}$ ou $\overline{sa_i}$ alors, en chaque point de temps t entre 0 et m, $EnvP$ puisse atteindre $\underline{cost_j}(t)$.

L'algorithme de Sweep, augmenté de ce filtrage, se présente comme suit. Nous en donnons une description pour $costC = max$.

L'algorithme déplace cette fois deux lignes verticales Δ et Δ^E sur l'axe de temps. En un balayage, il construit le profil cumulatif *et le profil enveloppant*. Il filtre les activités de manière à :

- ne pas dépasser le coût maximum autorisé,
- *garder la possibilité d'atteindre, dans chaque intervalle utilisateur, le coût minimal imposé.*

Un *événement* correspond au début ou à la fin d'une partie obligatoire, au début d'un intervalle utilisateur, *au début ou à la fin d'une partie enveloppante* ou bien à la date de début au plus tôt $\underline{sa_i}$ d'une activité $a_i \in A$. Tous les événements sont initialement générés et triés dans l'ordre croissant de leur date. On notera δ la position courante de Δ et δ^E la position courante de Δ^E.

A chaque pas de l'algorithme, une liste $ActToPrune$ contient les activités à filtrer à partir des bornes supérieures des variables de coût et une liste $ActToPrune^E$ contient les activités à filtrer à partir des bornes *inférieures* des variables de coût.

- Les événements de partie obligatoire servent à construire $CumP$. Ces événements, à la date δ, mettent à jour la hauteur sum_h du rectangle de parties obligatoires courant dans $CumP$, en ajoutant la hauteur minimale si c'est le début d'une partie obligatoire, ou en l'enlevant si c'est la fin d'une partie obligatoire.
  Le premier événement de partie obligatoire ou d'intervalle utilisateur dont la date est strictement supérieure à δ définit la fin du rectangle courant. On note cette date δ', et le rectangle correspondant est $\langle [\delta, \delta'[, sum_h \rangle$.
- Les événements de *partie enveloppante* servent à construire $EnvP$. Ces événements, à la date δ^E, mettent à jour la hauteur sum_e du rectangle courant dans $EnvP$, en ajoutant la hauteur maximale si c'est le début d'une partie enveloppante, ou en l'enlevant si c'est la fin d'une partie enveloppante.

- Le premier événement de partie obligatoire ou d'intervalle utilisateur dont la date est strictement supérieure à δ définit la fin du rectangle de partie obligatoire courant. On note cette date δ', et le rectangle correspondant est $\langle[\delta, \delta'[, sum_h\rangle$.
- Le premier événement *de partie enveloppante* ou d'intervalle utilisateur dont la date est strictement supérieure à δ^E définit la fin du rectangle de partie enveloppante courant. On note cette date δ', et le rectangle enveloppant correspondant est $\langle[\delta^E, \delta'[, sum_e\rangle$.
- Dans un intervalle p_j, les événements correspondant aux dates de début au plus tôt $\underline{sa_i}$ ajoutent de nouvelles activités candidates au filtrage.
  Elles sont ajoutées aux listes $ActToPrune$ et $ActToPrune^E$.

En chaque événement de partie obligatoire ou d'intervalle utilisateur, pour chaque activité $a_i \in ActToPrune$ n'ayant pas de partie obligatoire dans le rectangle $\langle[\delta, \delta'[, sum_h\rangle$, si sa hauteur minimale $\underline{ra_i}$ est strictement supérieure à $lc_j + \overline{cost_j} - sum_h$ alors on filtre sa date de début sa_i de manière à ce que a_i ne puisse plus provoquer un dépassement supérieur au dépassement maximal autorisé.

En chaque événement *de partie enveloppante* ou d'intervalle utilisateur, pour chaque activité $a_i \in ActToPrune^E$ n'ayant pas de partie obligatoire dans le rectangle $\langle[\delta^E, \delta'[, sum_e\rangle$, si sa hauteur maximale $\overline{ra_i}$ est strictement supérieure à $lc_j + \underline{cost_j} - sum_e$, l'intervalle $[\delta^E, \delta'[$ est gardé en mémoire.

En chaque événement d'intervalle utilisateur, si un seul intervalle $[\delta^E, \delta'[$ compris dans l'intervalle utilisateur précédent a été enregistré, la date de début sa_i de l'activité a_i associée est mise à jour, de manière à ce que a_i passe obligatoirement dans l'intervalle $[\delta^E, \delta'[$.

En un événement de profil ou d'intervalle utilisateur, si $\overline{ca_i} \leq \delta'$ alors a_i est retirée de la liste $ActToPrune$. Une fois l'étape de filtrage terminée, δ est mise à jour à la valeur δ'.

En un événement d'enveloppe ou d'intervalle utilisateur, si $\overline{ca_i} \leq \delta'$ alors a_i est retirée de la liste $ActToPrune^E$. Une fois l'étape de filtrage terminée, δ^E est mise à jour à la valeur δ'.

Notons que les événements de $PRUNING$ et de début de partie enveloppante pourraient être fusionnés(pour une activité $a_i \in A$, ils sont à la date $\underline{sa_i}$).

Cependant, nous conservons une distinction entre les événements de type $PRUNING$ et les événements de début de partie enveloppante pour rester homogène avec les autres algorithmes de Sweep présentés auparavant et pouvoir, en pratique, débrancher ce filtrage si on le souhaite.

Nous écrivons la règle de filtrage associée. L'exemple 40 illustre ce filtrage.

Règle de filtrage 14 ($costC = max$) *Considérons un rectangle $\langle[\delta^E, \delta'[, sum_e\rangle$ et un intervalle p_j tel que $\langle[\delta^E, \delta'[, sum_e\rangle$ est le seul rectangle contenu dans p_j satisfaisant $sum_e \geq lc_j + \underline{cost_j}$. Soit $a_i \in ActToPrune$, si $sum_e - \overline{ra_i} < lc_j + \underline{cost_j}$ alors : $[\underline{sa_i}, \delta^E - \overline{da_i}]$ et $[\delta', \overline{sa_i}]$ peuvent être retirés de $D(sa_i)$.*

Preuve. Si $sum_e - \overline{ra_i} < lc_j + \underline{cost_j}$ alors a_i doit s'exécuter (totalement ou partiellement) dans $[\delta^E, \delta'[$, sinon, dans toute solution étendant l'instanciation partielle, aucun point de temps t de $[\delta^E, \delta'[$, et donc de p_j ne pourra satisfaire $h_t \geq lc_j + cost_j$, $[\delta^E, \delta'[$ étant l'unique intervalle inclus dans p_j vérifiant $sum_e \geq lc_j + \underline{cost_j}$. La contrainte C3 de la Définition 36 serait alors violée. □

Exemple 40 *La Figure 5.17 représente un exemple de filtrage dans le cas costC= max. La variable de début de l'activité a_5 du problème présenté sur la Figure 5.16 y est filtrée. Dans un premier temps, la contribution de a_5 au profil enveloppant est retirée de celui-ci. Le profil enveloppant restant ne permettrait pas d'atteindre la valeur minimale de $cost_0$. Ceci impose donc que a_5 passe dans l'intervalle p_0. La variable sa_5 peut donc être mise à jour : $D(sa_5) = D(sa_5)\backslash[1,2]$.*

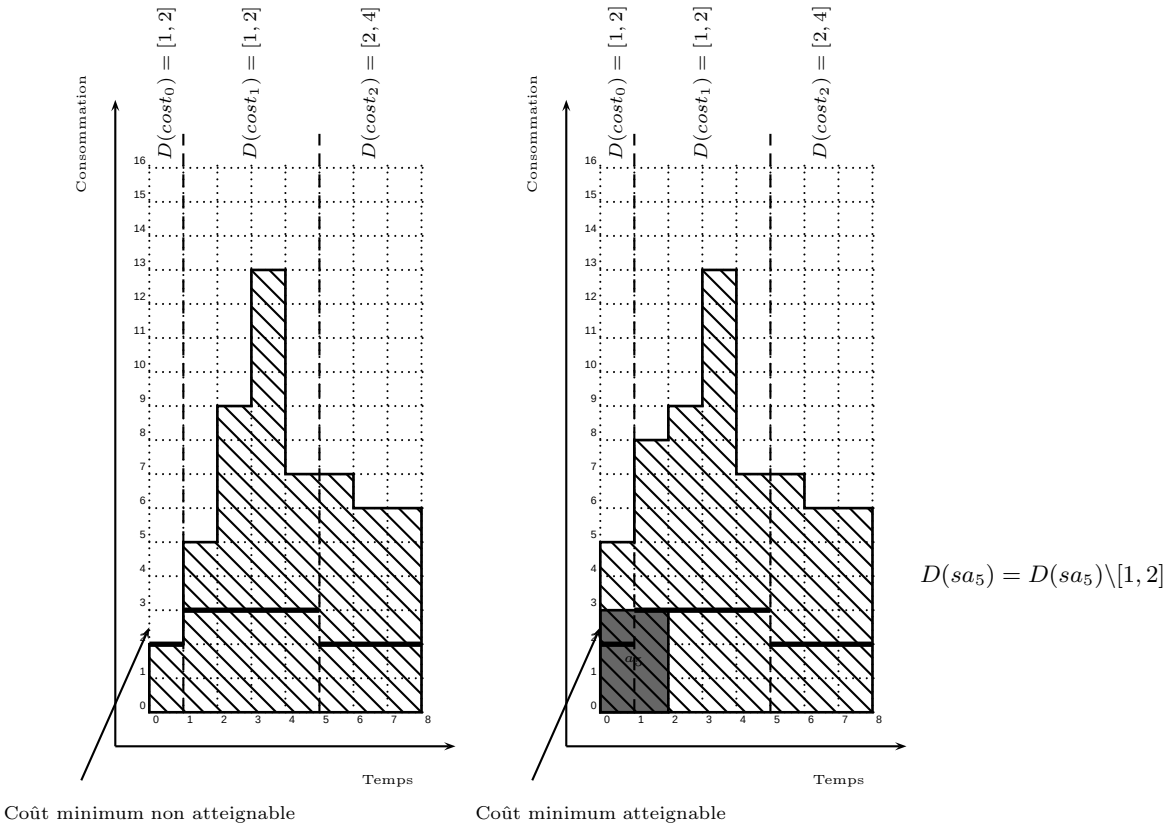


FIGURE 5.17 – Les différents pas de la procédure de filtrage de l'activité a_5 de la Figure 5.16, avec $costC = max$. Nous rappelons que $D(sa_5) = [0,2]$. La partie enveloppante de l'activité a_5 à filtrer est retirée du profil enveloppant. Sur la première figure, il apparaît que le coût minimal dans le premier intervalle $\underline{cost_0} = 1$ ne peut être atteint. Nous mettons alors à jour le début de a_5 pour s'assurer que, si a_5 débute en $\underline{sa_i}$ ou $\overline{sa_i}$, alors, en chaque point de temps, le profil enveloppant peut atteindre la valeur minimale du domaine des coûts locaux. De ce fait, $D(sa_5) = D(sa_5)\backslash[1,2]$.

Une telle règle n'est pas adaptable au cas $costC = sum$. En effet, si comparer la hauteur du profil enveloppant à chaque capacité locale est approprié dans le cas $costC = max$, ce n'est pas le cas pour $costC = sum$. Il faut alors raisonner en fonction de l'énergie. Cependant, l'aire de la partie enveloppante d'une activité $a_i \in A$, contenue dans le rectangle $\langle [\underline{sa_i}, \overline{ca_i}[, \overline{ra_i} \rangle$, n'est pas l'énergie de cette activité. Si l'on raisonne sur les énergies, comme c'est le cas avec $costC = sum$, considérer l'aire de la partie enveloppante n'a pas de sens.

De plus, dans un intervalle donné, on ne peut pas définir correctement un minimum d'énergie qui peut mener à un dépassement. Pour s'en convaincre, nous nous appuyons sur l'exemple 41.

Dans ces conditions, appliquer un raisonnement similaire à la règle de filtrage 14 ne paraît pas approprié.

Exemple 41 *La Figure 5.18 montre deux répartitions d'énergie dans un intervalle. Sur la première figure, l'énergie égale à* 12 *comprise dans l'intervalle* p_1 *n'engendre aucun coût dans cet intervalle. En revanche, l'énergie, égale à* 5 *sur la seconde figure, engendre un coût de* 2 *dans l'intervalle* p_1*.*

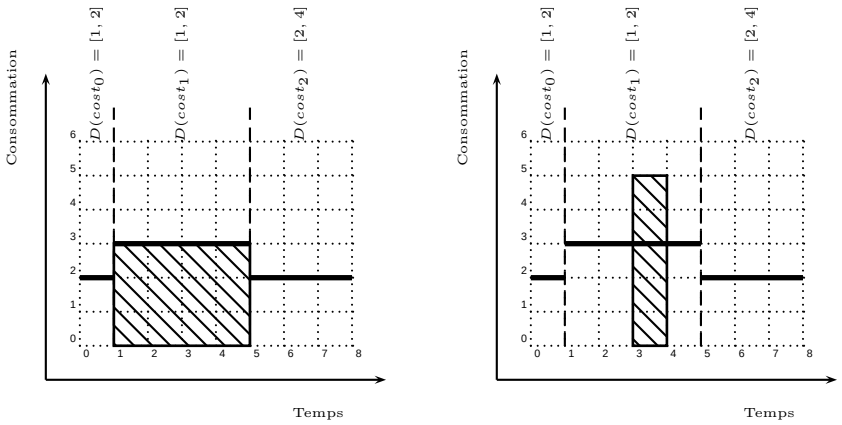


FIGURE 5.18 – Deux répartitions d'énergie dans un intervalle, dans le cas $costC = sum$. Sur la première figure, l'énergie comprise dans l'intervalle utilisateur p_1 n'engendre aucun dépassement. Elle est égale à 12. Sur la seconde figure, l'énergie comprise dans le second intervalle engendre un dépassement de 2. Cette énergie est égale à 5.

L'algorithme 9 détaille la génération des différents événements, et l'algorithme 10 montre la procédure de parcours des événements et de filtrage. Nous ne détaillons pas le filtrage effectué dans l'algorithme de Sweep présenté en section 5.1.1.1, pour des raisons de lisibilité de l'algorithme. Nous mentionnons leur place dans l'algorithme. Appliquer ce filtrage dans cet algorithme ne pose aucun problème.

Nous considérons que, pour chaque événement correspondant au début d'un intervalle utilisateur, nous essayons d'appliquer la règle de filtrage 14 à l'intervalle utilisateur précédent. Ceci peut être fait en $O(1)$. Nous ajoutons $O(n)$ événements au Sweep, et l'existence, et la position de l'unique rectangle $\langle [\delta, \delta'[, sum_e \rangle$ satisfaisant $sum_e \geq lc_j + \underline{cost_j}$ peuvent être maintenues en $O(1)$. Comme pour l'algorithme de Sweep décrit en section 5.1.1.1, la complexité globale est en $O((n+p) \times log(n+p) + n \times (n+p))$.

Algorithme 9: Algorithme de génération des événements du Sweep pour SoftCumulative intégrant les événements de partie enveloppante.

Data : Un ensemble d'activités A.
Result : Une liste d'événements Evt.

List Evt // La liste d'événements générés
foreach $a_i \in A$ **do**
  if $\underline{sa_i} \neq \overline{sa_i}$ || $\underline{ca_i} \neq \overline{ca_i}$ || $\underline{da_i} \neq \overline{da_i}$ || $\underline{ra_i} \neq \overline{ra_i}$ **then**
    $Evt.add(< PRUNING, a_i, \underline{sa_i}, 0 >)$;
  $Evt.add(< ENVELOPE, a_i, \underline{sa_i}, \overline{ra_i})$;
  $Evt.add(< ENVELOPE, a_i, \overline{ca_i}, -\overline{ra_i})$;
  if $\underline{ca_i} > \overline{sa_i}$ **then**
    $Evt.add(< PROFILE, a_i, \overline{sa_i}, \underline{ra_i} >)$;
    $Evt.add(< PROFILE, a_i, \underline{ca_i}, -\underline{ra_i} >)$;
foreach $p_j \in P$ **do**
  $Evt.add(< INTERVAL, p_j, sp_j, 0 >)$;
return Evt

Algorithme 10: Algorithme de Sweep pour SoftCumulative intégrant la règle de filtrage 14

Data : Un ensemble d'activités A, une séquence d'intervalles utilisateur P, une séquence de capacités locales Loc, une séquence de variables de coût $Cost$, une liste d'événements Evt.

List $ActToPrune$; int δ ; int δ^E ; int δ' ; int $sum_h = 0$; int k ; $evt = Evt.first()$; $\delta = evt.date$;
int $sum_e = 0$;//La hauteur du profil enveloppant
$\delta^E = evt.date$;//Enregistrement de la date pour les intervalles $[\delta^E, \delta'[$
int $nbFiltrageMin = 0$;// Sert à vérifier l'unicité mentionnée dans la règle de filtrage 14
int δ^E_{ok}, δ'^E_{ok} ;//Stockage de l'intervalle intéressant pour la règle de filtrage 14
$Evt.sortByDate()$; On ordonne les événements par date croissante, pour une date égale, on place d'abord les événements de type PROFILE, puis ENVELOPE, puis PRUNING puis INTERVAL
while $evt \neq NULL$ **do**
  $a_i = evt.activity$; $\delta' = evt.date$; $k = interval(\delta')$
  if $evt.type == ENVELOPE$ **then**
    if $\delta^E \neq \delta'$ **then**
      if $sum_e - \overline{ra_i} < lc_k + \underline{cost_k}$ **then** nbFiltrageMin ++ ; $\delta^E_{ok} = \delta^E$; $\delta'^E_{ok} = \delta'^E$;
      $\delta^E = \delta'$;
    $sum_e += evt.increment$;
  else if $evt.type == PROFILE$ **then**
    if $\delta \neq \delta'$ **then**
      if $sum_h > lc_k + \overline{cost_k}$ **then**
        return fail ;
      foreach $a_j \in ActToPrune$ **do**
        //Nous effectuons ici le filtrage décrit dans l'algorithme 8
        if $\overline{ca_j} < \delta'$ **then** $ActToPrune.remove(a_j)$;
      $\delta = \delta'$;
    $sum_h += evt.increment$;
  else if $evt.type == INTERVAL$ **then**
    if $\delta \neq \delta'$ **then**
      if $sum_h > lc_k + \overline{cost_k}$ **then** return fail ;
      foreach $a_j \in ActToPrune$ **do**
        //Nous effectuons ici le filtrage décrit dans l'algorithme 8
        if $\overline{ca_j} < \delta'$ **then** $ActToPrune.remove(a_j)$;
      if $nbFiltrageMin == 1$ **then**
        foreach $a_j \in ActToPrune^E$ **do**
          if $sum_e - \overline{ra_j} < lc_k + \underline{cost_k}$ **then**
            $D(sa_j) = D(sa_j) \backslash [\underline{sa_j}, \delta^E_{ok} - \overline{da_j}]$; $D(sa_j) = D(sa_j) \backslash [\delta'^E_{ok}, \overline{sa_j}]$;
        if $\overline{ca_j} < \delta'$ **then** $ActToPrune^E.remove(a_j)$;
      $\delta = \delta'$; $\delta^E = \delta'$; $nbFiltrageMin = 0$;
  else if $evt.type == PRUNING$ **then**
    $ActToPrune.add(a_i)$; $ActToPrune^E.add(a_i)$;
  $evt = evt.next()$;
if $sum_h > lc_k + \overline{cost_k}$ **then** return fail ;
foreach $a_j \in ActToPrune$ **do**
  if $(sum_h + ra_j > lc_k + \overline{cost_k}) \& (\overline{sa_j} \geq \underline{ca_j} || ((\overline{sa_j} < \underline{ca_j}) \& [\overline{sa_j}, \underline{ca_j}[\cap [\delta, \delta'[= \emptyset))$ **then**
  $D(sa_j) = D(\overline{sa_j}) \backslash]\delta - \underline{da_j}, \delta'[$;

5.2 Stratégies de recherche

Les stratégies de recherche consistent usuellement à choisir au cours de la recherche la prochaine variable à instancier, ainsi que la prochaine valeur à essayer pour cette variable. En ordonnancement cumulatif classique, il est admis que les stratégies génériques (e.g., la variable minimisant le rapport entre la taille de son domaine et le nombre de contraintes dans laquelle elle intervient) ne sont pas efficaces.

Il ne semble pas y avoir d'argument raisonnable laissant penser que ce constat ne reste pas valable dans le cas des problèmes modélisés via la contrainte SoftCumulative. Notamment, chaque intervalle utilisateur est associé à une variable de coût qui mesure un dépassement, et il semble peu opportun de brancher sur de telles variables avant d'avoir essayé de placer les activités.

Dans ce contexte, nous proposons une stratégie de recherche dédiée aux problèmes d'ordonnancement cumulatifs sur-contraints. Il convient de noter que ce résultat a été obtenu au cours des phases d'expérimentation qui sont présentées dans le chapitre 7.

Nous présentons d'abord une première stratégie de recherche naïve en section 5.2.1.

Nous montrons ensuite qu'elle peut être améliorée, et nous introduisons notre nouvelle stratégie de recherche en section 5.2.2

5.2.1 Une première stratégie de recherche naïve

Une première stratégie consiste à instancier, en priorité, les variables de début d'activités. Ces variables sont ordonnées selon l'ordre lexicographique des activités non encore instanciées. A ces variables, nous affectons en priorité la valeur minimale de leur domaine. Nous utilisons donc la stratégie *minVal* définie en section 1.4.

L'heuristique de choix de variable s'assure que l'on n'essaie pas d'instancier en priorité les variables de coût ou d'objectif.

Sémantiquement, les coûts sont définis *en fonction du profil cumulatif*. Le profil cumulatif est défini par les activités. Il est donc logique d'instancier les dates de début d'activités, puis d'en déduire les coûts, plutôt que l'inverse.

Cette stratégie est relativement naïve car elle ne tient pas compte de la forme du profil cumulatif des parties obligatoires.

5.2.2 Une nouvelle stratégie de recherche dédiée à SoftCumulative

Une stratégie de recherche en PPC peut suivre deux ordres distincts :

- nous pouvons choisir en premier lieu la prochaine variable à instancier, puis essayer une valeur pour cette variable,
- à l'inverse, nous pouvons, en premier lieu, choisir une valeur à essayer, puis, en fonction de celle-ci, choisir la variable à laquelle nous essayons de l'affecter.

Nous choisissons d'adopter la première solution car il nous semble plus naturel d'essayer de placer une activité, puis de trouver une place pour celle-ci, que l'inverse. Cependant, étudier la stratégie inverse pourrait représenter une perspective intéressante à notre travail.

Nous présentons donc tout d'abord notre heuristique de choix de variable en section 5.2.2.1. Nous détaillons ensuite l'heuristique de choix de valeur de notre stratégie en section 5.2.2.2.

5.2.2.1 Heuristique de choix de variable

Le principe intuitif de l'heuristique est de choisir les variables de début des activités en fonction de leur "criticité".

Dans chaque intervalle utilisateur, la forme du profil cumulatif et la capacité locale de cet intervalle déterminent le dépassement dans cet intervalle. Les dépassements de chaque intervalle utilisateur sont ensuite agrégés pour donner la valeur de la variable objectif, qui représente le dépassement global. Dans le problème représenté par cette contrainte, nous essayons de trouver une solution avec le dépassement global le plus faible. Nous essayons donc de placer, en priorité, les activités susceptibles de mener à des dépassements importants. Les autres activités pourront combler plus facilement les "trous" dans le profil cumulatif.

Cas *costC= max* Les activités les plus hautes sont les plus susceptibles d'engendrer un dépassement. Il est donc intéressant de les placer en premier : les autres activités pourront servir à combler des "trous" dans le profil. Il est plus facile de placer des activités de hauteur petites que des activités très hautes. Nous ordonnons les activités par hauteur décroissante. Si plusieurs activités ont la même hauteur, nous les ordonnons par longueur décroissante.

Cas *costC= sum* Les activités sont ordonnées par surface décroissante. Si plusieurs activités ont la même surface, nous les ordonnons par hauteur décroissante. Les activités de plus grande surface sont celles qui risquent de faire augmenter le plus les coûts locaux et donc le coût total agrégé (l'objectif).

5.2.2.2 Heuristique de choix de valeur

L'heuristique de choix de valeur se base sur deux critères :

1. nous recherchons dans le profil les intervalles de temps les plus adaptés au placement d'une activité donnée. Nous cherchons à faire débuter l'activité au point de temps qui minimise le coût globalement engendré par son placement. Nous cherchons donc dans le profil des parties obligatoires les endroits où il y a des "trous". Pour cela :
   - nous identifions des intervalles dans lesquels la hauteur du profil cumulatif est constante,
   - nous cherchons à y placer l'activité a_i courante,
   - nous évaluons le coût global engendré, afin de choisir un intervalle de temps correspondant à un coût global minimal. Nous ne tenons pas compte d'éventuelles bornes inférieures des domaines des variables de coût car nous voulons limiter l'influence du placement d'une activité sur le coût global. Considérer ces bornes inférieures pourrait sous-estimer cette influence,
2. plusieurs de ces intervalles de temps peuvent engendrer un coût égal. Nous choisissons, parmi ceux-ci, l'intervalle de temps le plus adapté à l'activité considérée. Il s'agit de celui qui resserre le plus le profil, c'est à dire qu'un placement de l'activité dans cet intervalle de temps laissera le moins d'énergie libre dans cet intervalle. Ceci permet de laisser le plus de place pour d'autres activités dans les autres intervalles considérés. Pour cela :
   - pour chaque intervalle déterminé par le critère précédent, nous plaçons l'activité a_i dans cet intervalle,
   - nous évaluons l'énergie laissée libre dans cet intervalle.

Pour définir formellement ces critères nous introduisons la notion de plateau.

Définition 48 *Nous appelons plateau un intervalle de temps $pl = [spl, epl[$ tel que,*
- *pl est inclus dans un unique intervalle utilisateur $p_j \in P$,*
- *$\forall t \in pl, lc_j - h_t$ reste constante.*
- *cet intervalle a une longueur maximale au sens de l'inclusion, c'est à dire que :*
  - *si $spl - 1 \in p_j$: $lc_j - h_{spl-1} \neq lc_j - h_{spl}$,*
  - *si $epl \in p_j$: $lc_j - h_{epl} \neq lc_j - h_{spl}$.*

Les quantités $cout(a_i, t)$ et $comble(a_i, t)$ définissent les quantités associées aux deux critères présentés précédemment, pour le placement d'une activité $a_i \in A$ à la date t. $cout(a_i, t)$ évalue le dépassement engendré par le placement sur un plateau donné (la longueur d'une activité peut faire qu'elle intersecte plusieurs plateaux). $comble(a_i, t)$ évalue l'énergie laissée libre par le placement sur ce (ces) plateau(x).

Notation 5 ($cout(a_i, t)$) *Soit $a_i \in A$ une activité, t un point de temps tel que $t \in [\underline{sa_i}, \overline{sa_i}]$, $\mathcal{P}^H$ l'ensemble des intervalles utilisateur intersectés par $[t, t + \underline{da_i}[$, et $\mathcal{PL}^t_{a_i}$ l'ensemble des plateaux $\{pl_0, pl_1, \ldots, pl_{q-1}\}$ intersectant l'intervalle $[t, t + \underline{da_i}[$. Alors on note $cout(a_i, t)$ le coût du placement de a_i au point de temps t :*
- *si costC= max et objC= sum :*

$$cout(a_i, t) = \sum_{p_j \in \mathcal{P}^H} \Big(\max_{\substack{pl_k \in \mathcal{PL}^t_{a_i} \\ pl_k \subseteq p_j}} (max(h_{spl_k} + \underline{ra_i} - lc_j, 0)) \Big)$$

- *si costC= max et objC= max :*

$$cout(a_i, t) = \max_{p_j \in \mathcal{P}^H} \Big(\max_{\substack{pl_k \in \mathcal{PL}^t_{a_i} \\ pl_k \subseteq p_j}} (max(h_{spl_k} + \underline{ra_i} - lc_j, 0)) \Big)$$

- *si costC= sum et objC= sum :*

$$cout(a_i, t) = \sum_{pl_k \in \mathcal{PL}^t_{a_i}} (max(spl_k, t), min(epl_{k+1}, t + \underline{da_i})) \times max(h_{spl_k} + \underline{ra_i} - lc_j(spl_k), 0)$$

- *si costC= sum et objC= max :*

$$cout(a_i, t) = \max_{p_j \in \mathcal{P}^H} \Big(\sum_{\substack{pl_k \in \mathcal{PL}^t_{a_i} \\ pl_k \subseteq p_j}} \big(max(spl_k, t), min(epl_{k+1}, t + \underline{da_i})) \times max(h_{spl_k} + \underline{ra_i} - lc_j(spl_k), 0)\big) \Big)$$

Notation 6 ($comble(a_i, t)$) *Soit $a_i \in A$ une activité, t un point de temps tel que $t \in [\underline{sa_i}, \overline{sa_i}]$, et $\mathcal{PL}^t_{a_i}$ l'ensemble des plateaux $\{pl_0, pl_1, \ldots, pl_{q-1}\}$ intersectant l'intervalle $[t, t + \underline{da_i}[$. Nous notons*

$$comble(a_i, t) = \sum_{pl_k \in \mathcal{PL}^t_{a_i}} (max(spl_k, t), min(epl_{k+1}, t + \underline{da_i})) \times max(lc_j(spl_k) - h_{spl_k} - \underline{ra_i}, 0))$$

Alors, pour une activité $a_i \in A$, la valeur $t \in [\underline{sa_i}, \overline{sa_i}]$ à essayer en priorité pour la variable $\underline{sa_i}$ est :

1. celle qui minimise $cout(a_i, t)$,
2. en cas d'égalité, celle qui minimise $comble(a_i, t)$.

L'exemple 42 illustre notre stratégie de recherche.

Exemple 42 *La Figure 5.19 montre un profil cumulatif avec des "trous". Nous posons ici costC= max et objC= sum. Dans cet exemple, nous considérons que les coûts et objectif ne sont pas contraint, c'est à dire qu'on ne considère pas les bornes supérieures de leur domaine. Nous considérons que les activités considérées n'ont pas de partie obligatoire. Si elles en ont, elles ont été retirées du profil avant affectation. Nous ordonnons dans un premier temps les activités selon l'ordre en section 5.2.2.1. La première activité que nous essayons de placer est la plus haute. Puis nous appliquons la stratégie de recherche décrite précédemment.*

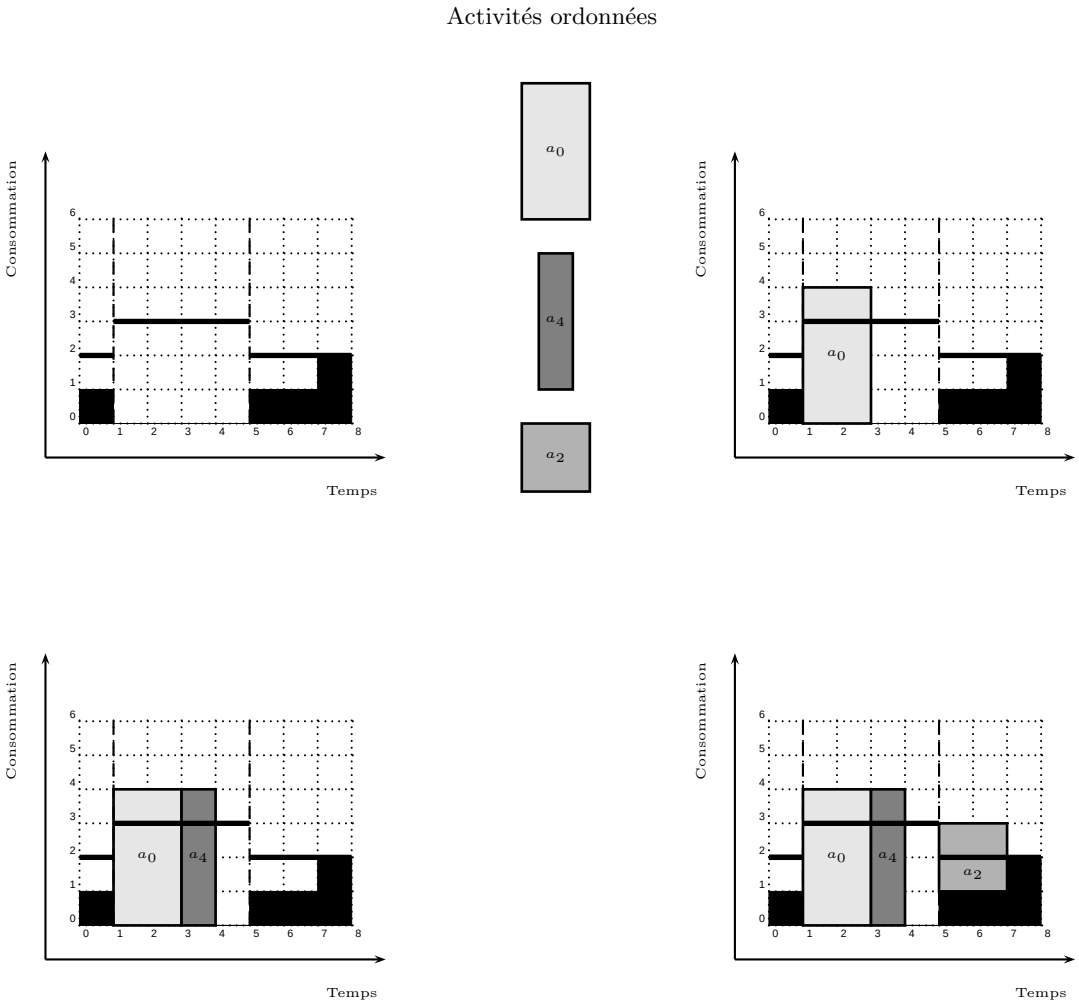


FIGURE 5.19 – Stratégie de recherche pour un problème où 3 activités ne sont pas fixées et n'ont pas de partie obligatoire. Le domaine de leurs variables de début et de fin leur permettent d'être placés dans l'ensemble de l'intervalle $[0, 8[$Le profil cumulatif des parties obligatoires à une hauteur de 1 entre les points de temps 0 et 1, et entre les points 5 et 7, et une hauteur de 2 entre les points de temps 7 et 8. Sa hauteur est nulle ailleurs. Les activités sont ordonnées par hauteur décroissante, puis nous les plaçons aux endroits du profil cumulatif qui engendrent le moins de dépassement possible. Nous utilisons les paramètres $costC = max$ et $objC = sum$. Ainsi, l'activité a_0 est placée en premier au point de temps 1 ($cout(a_0, 1) = 1$) : elle y engendre un dépassement de 1. En 2 ou 3 elle aurait également engendré le même dépassement : ces 3 solutions sont équivalentes. A tout autre endroit, le dépassement aurait été supérieur. Puis, l'activité a_4 est placée en 3 : elle engendre un dépassement de 1 ($cout(a_4, 3) = 1$). En 4 elle aurait également engendré le même dépassement : ces 2 solutions sont équivalentes. Enfin, l'activité a_2 est placée en 5, elle y engendre un dépassement de 1 ($cout(a_2) = 1$). Placer a_2 en 4 aurait engendré le même dépassement ($cout(a_2, 4) = 1$) mais aurait moins resserré le profil. En effet, $comble(a_2, 4) = 1$ et $comble(a_2, 5) = 0$: placer a_2 en 4 aurait laissé une énergie libre de 1 sous la capacité dans le second intervalle.

Chapitre 6

Décompositions

Sommaire

6.1 Introduction

Une des raisons d'être des contraintes globales peut être d'encapsuler, dans une seule contrainte, le coeur d'un problème. Nous avons présenté dans les chapitres précédents plusieurs algorithmes de filtrage pour la contrainte Cumulative, et nous avons proposé de nouveaux algorithmes de filtrage pour la contrainte SoftCumulative.

Une alternative à l'emploi de ces contraintes est l'utilisation de *décompositions*.

Une décomposition est la représentation d'une contrainte à l'aide d'un ensemble de contraintes, généralement plus primitives. Utiliser une décomposition peut nécessiter l'ajout de nouvelles variables dans le modèle. Un des avantages des décompositions est qu'elle fournissent une modélisation simple et adaptable à n'importe quel solveur dans lequel ne serait pas implémentée la contrainte concernée. On peut parfois y trouver d'autres avantages, comme par exemple un meilleur comportement avec certaines stratégies de recherche génériques, ou encore le fait d'éviter de manipuler des structures de données complexes dans un algorithme de filtrage dédié. En revanche, si le nombre de variables supplémentaires est trop élevé ou si le filtrage obtenu à l'aide de la décomposition est trop faible, alors les classes de problèmes qu'une décomposition permet de résoudre peuvent s'avérer rapidement très restreintes, en comparaison avec la contrainte d'origine.

Dans notre contexte, il semble intuitif que Cumulative et SoftCumulative donnent de meilleurs résultats que des décompositions. Cependant, Schutt et al. montrent dans [77] que décomposer la contrainte

Cumulative, en l'associant à des techniques de résolution de type SAT permet de résoudre certaines instances difficiles de l'état de l'art de taille raisonnable, de taille raisonnable (120 activités maximum). Ils ferment notamment certaines instances ouvertes de la PSPLib [45], qui est une librairie de "benchmarking" bien connue pour les problèmes cumulatifs. Dans cette thèse nous n'avons pas étudié de techniques hybridant la programmation par contrainte et SAT. En revanche, il nous a semblé intéressant d'étudier les décompositions des contraintes Cumulative et SoftCumulative dans le cadre de la programmation par contrainte.

Trois questions ouvertes nous semblaient importantes au début de cette étude.

1. Peut-on adapter facilement les décompositions existantes de Cumulative à la contrainte SoftCumulative ?
2. Peut-on concevoir une décomposition de Cumulative (respectivement de SoftCumulative) qui nécessite un nombre de variables supplémentaire demeurant raisonnable, par exemple de l'ordre de $O(n)$, où n est le nombre de variables de la contrainte d'origine ?
3. Quelle est l'efficacité pratique de ces décompositions ?

Nous présentons dans ce chapitre trois principes de décomposition des contraintes Cumulative et SoftCumulative, qui répondent aux questions 1 et 2 ci-dessus. L'analyse expérimentale des décompositions sera réalisée dans le prochain chapitre.

Dans notre étude, nous abordons uniquement les décompositions de la SoftCumulative avec les paramètres $costC = max$ et $objC = sum$. Pour plus de clarté, nous considérons des longueurs et des hauteurs constantes pour les activités.

6.2 Décompositions temps-ressource

6.2.1 Décomposition temps-ressource de Cumulative

Nous présentons la décomposition la plus intuitive de la contrainte Cumulative, appelée *décomposition temps-ressource* [1, 77].

Pour cela nous nous appuyons sur l'exemple donné par la Figure 6.1. Elle représente une solution d'un problème cumulatif à 4 activités et indique les valeurs affectées aux différentes variables de sa décomposition temps-ressource.

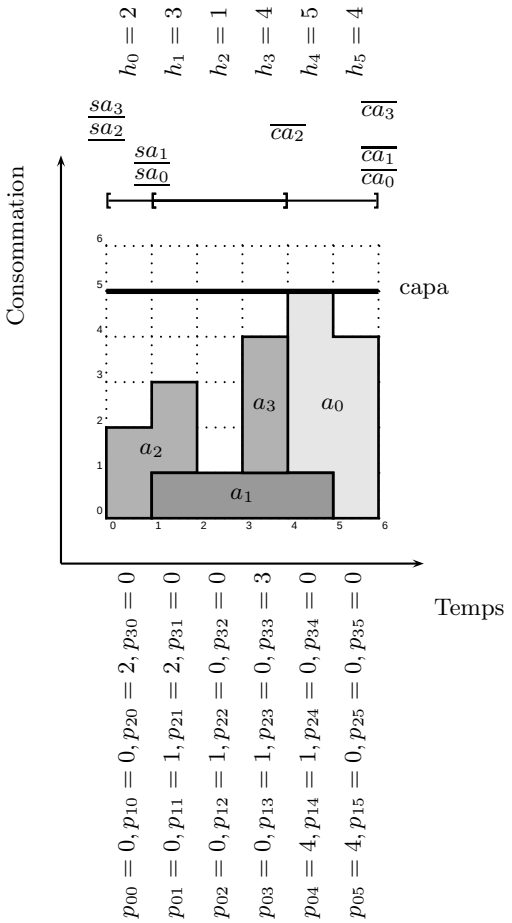


FIGURE 6.1 – Une solution d'un problème cumulatif à 4 activités modélisé par la décomposition temps-ressource de CUMULATIVE. Pour chaque activité a_i et chaque point de temps j, la variable p_{ij} prend la valeur de la hauteur de l'activité si celle-ci est présente en j, la valeur 0 sinon. La hauteur h_j du profil en chaque point de temps j est égale à $\sum_{i \in [0,3]} p_{ij}$. En chaque point de temps j, h_j est inférieure ou égale à la capacité *capa*.

Soient n le nombre d'activités, et m le nombre de points de temps du problème cumulatif représenté par la contrainte Cumulative$(A, capa)$.

Cette décomposition utilise $n \times m$ variables p_{ij} qui peuvent être instanciées aux valeurs suivantes : la hauteur ra_i de l'activité $a_i \in A$, si et seulement si a_i est présente au point de temps j, et la valeur 0 sinon. L'exemple 43 présente de telles variables dans le cas instancié.

Exemple 43 *Dans le cas instancié représenté par la Figure 6.1, les valeurs affectées aux variables p_{ij} sont résumées dans le tableau suivant, avec les colonnes représentant les points de temps j et les lignes représentant les activités a_i. Les cases grisées représentent le domaine $[sa_i, ca_i[$ de chaque activité $a_i \in A$, c'est à dire les points de temps par lesquels l'activité a_i passe.*

$a_i \backslash j$	0	1	2	3	4	5
a_0	0	0	0	0	$ra_0 = 4$	$ra_0 = 4$
a_1	0	$ra_1 = 1$	$ra_1 = 1$	$ra_1 = 1$	$ra_1 = 1$	0
a_2	$ra_2 = 2$	$ra_2 = 2$	0	0	0	0
a_3	0	0	0	$ra_3 = 3$	0	0

En chaque point de temps j, la somme des valeurs de ces variables donne la hauteur du profil h_j, comme l'illustre l'exemple 44. On peut alors comparer cette hauteur à la capacité *capa*. La contrainte est respectée si la hauteur est inférieure à *capa* pour tout point de temps j. Dans le cas contraire, elle est violée.

Exemple 44 *Dans le cas instancié représenté par la Figure 6.1, les valeurs affectées aux variables h_j,et la capacité capa sont indiqués dans le tableau suivant. En chaque point de temps j, $h_j \leq capa$. La contrainte est donc satisfaite.*

j	0	1	2	3	4	5
h_j	2	3	1	4	5	4
$capa$	5					

Nous donnons maintenant une définition formelle de cette décomposition temps-ressource.

Décomposition 1 (Décomposition temps-ressource de la contrainte Cumulative) *Soit une ressource avec une capacité limitée par un entier capa et un ensemble $A = \{a_0, a_1, \ldots, a_{n-1}\}$ de n activités, finissant au plus tard en $m \geq \overline{c}(A)$, $\overline{c}(A)$ étant la date de fin au plus tard de l'ensemble A, comme défini par la Notation 2. La contrainte Cumulative$(A, capa)$ est équivalente à la décomposition suivante.*

Variables

- *Un ensemble de variables entières $\mathcal{H} = \{h_0, h_1, \ldots, h_{m-1}\}$, tel que, pour tout $j \in [0, m-1]$, h_j représente la hauteur du profil cumulatif au point de temps j, avec $D(h_j) = [0, capa]$,*
- *un ensemble de variables entières $\mathcal{P} = \{p_{ij} | i \in [0, n-1], j \in [0, m-1]\}$, tel que, pour tout $a_i \in A$, et pour tout $j \in [0, m-1]$, p_{ij} représente la contribution de l'activité a_i au point de temps j, avec $D(p_{ij}) = \{0, ra_i\}$.*

Contraintes

- *Pour toute activité* $a_i \in A$ *:*
  - $sa_i + da_i = ca_i$,
  - *pour tout point de temps* $j \in [0, m-1]$, $([j \geq sa_i \wedge j < ca_i] \wedge [p_{ij} = ra_i]) \vee ([j < sa_i \vee j \geq ca_i] \wedge [p_{ij} = 0])$,
- *pour tout point de temps* $j \in [0, m-1]$ *:*
  - $h_j = \sum_{i \in [0,n-1]} (p_{ij})$,
  - $h_j \leq capa$.

6.2.2 Contribution : décomposition temps-ressource de SoftCumulative

Nous étendons la décomposition temps-ressource de la contrainte Cumulative, décrite en section 6.2.1, au cas SoftCumulative.

Pour décrire cette extension, nous nous appuyons sur l'exemple donné par la Figure 6.2. Elle représente une solution d'un problème cumulatif relaxé à 4 activités, modélisé par la contrainte SoftCumulative. Elle indique les valeurs affectées aux différentes variables de sa décomposition temps-ressource.

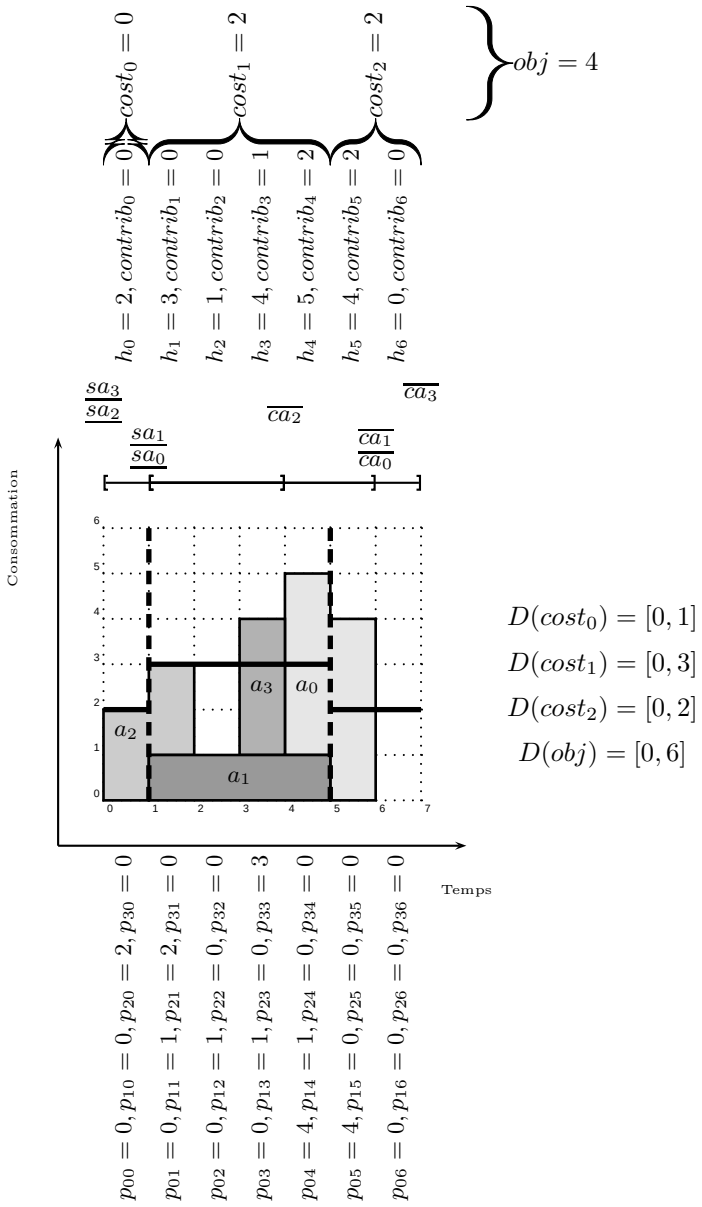


FIGURE 6.2 – Une solution d'un problème cumulatif relaxé à 4 activités et 3 intervalles utilisateur, modélisé par la décomposition temps-ressource de SOFTCUMULATIVE. Celui-ci est posé avec $D(cost_0) = [0, 1]$, $D(cost_1) = [0, 3]$, $D(cost_2) = [0, 2]$ et $D(obj) = [0, 6]$. Pour chaque activité a_i et chaque point de temps j, la variable p_{ij} prend la valeur de la hauteur de l'activité si celle-ci est présente en j, la valeur 0 sinon. La hauteur h_j du profil en chaque point de temps j est égale à $\sum_{i \in [0,3]} p_{ij}$. En chaque point de temps j de l'intervalle p_k, si h_j est inférieure ou égale à la capacité lc_k, $contrib_j = 0$, sinon, $contrib_j = h_j - lc_k$. Les coûts sont fonctions des variables contrib, et l'objectif est fonction des coûts.

Soient n le nombre d'activités, et m le nombre de points de temps du problème cumulatif relaxé représenté par la contrainte SoftCumulative(A, $capa$, P, Loc, $Cost$, obj,$costC$,$objC$). On considère ici $costC = max$, et $objC = sum$.

Cette décomposition utilise $n \times m$ variables p_{ij} qui peuvent être instanciées aux valeurs suivantes : la hauteur de l'activité $a_i \in A$ si et seulement si elle est présente au point de temps j, et la valeur 0 sinon. L'exemple 45 donne les valeurs associées à ces variables pour la Figure 6.2.

Exemple 45 *Dans le cas instancié représenté par la Figure 6.2, les valeurs affectées aux variables p_{ij} sont résumées dans le tableau suivant, avec les colonnes représentant les points de temps j et les lignes représentant les activités a_i. Les cases grisées représentent le domaine $[sa_i, ca_i[$ de chaque activité $a_i \in A$, c'est à dire les points de temps par lesquels l'activité a_i passe.*

$a_i \backslash j$	0	1	2	3	4	5	6
a_0	0	0	0	0	$ra_0 = 4$	$ra_0 = 4$	0
a_1	0	$ra_1 = 1$	$ra_1 = 1$	$ra_1 = 1$	$ra_1 = 1$	0	0
a_2	$ra_2 = 2$	$ra_2 = 2$	0	0	0	0	0
a_3	0	0	0	$ra_3 = 3$	0	0	0

Une variable h_j est associée à chaque point de temps j et donne la hauteur du profil au point de temps j en sommant les variables p_{ij}.

Il convient également de prendre en compte les intervalles utilisateur $p_k \in P$, et donc de différencier les capacités locales $lc_k \in Loc$. Pour cela, nous ajoutons au modèle les variables de coût $cost_k \in Cost$ associées aux intervalles utilisateur et une variable objectif obj. Pour tout point de temps $j \in p_k$, la hauteur du profil n'est pas contrainte à être inférieure à la capacité locale lc_k, mais le dépassement $contrib_j = h_j - lc_k$ doit être inférieur au coût maximal autorisé $\overline{cost_k}$. Le coût associé à un intervalle p_k est le maximum des variables $contrib_j$ telles que $j \in p_k$.

Enfin, la valeur de l'objectif obj est égale à la somme des valeurs des variables de $Cost$. L'ensemble des valeurs de ces variables dans le cas de la Figure 6.2 est donné dans l'exemple 46.

Exemple 46 *Dans le cas instancié représenté par la Figure 6.2, les valeurs affectées aux variables h_j sont indiquées dans le tableau suivant. Pour chaque intervalle $p_k \in P$, la capacité lc_k, les $contrib_j$ associées à chaque point de temps et les valeurs des variables de coût sont également indiquées dans le tableau suivant. Nous grisons les valeurs de h_j et $contrib_j$ responsable du coût dans chaque intervalle. Nous avons : $D(cost_0) = [0, 1]$, $D(cost_1) = [0, 3]$, $D(cost_2) = [0, 2]$ et $D(obj) = [0, 6]$. En chacun des intervalles $p_k \in P$, $cost_k \leq \overline{cost_k}$. Enfin, la valeur de obj est représentée. Elle est inférieure à $\overline{obj}$. La contrainte est donc satisfaite.*

j	0	1	2	3	4	5	6
p_k	p_0	p_1				p_2	
h_j	2	3	1	4	5	4	0
lc_k	2	3				2	
$contrib_j$	0	0	0	1	2	2	0
$cost_k$	0	2				2	
obj	4						

Nous donnons maintenant une définition formelle de l'adaptation à SoftCumulative de cette décomposition temps-ressource, dans le cas où l'on représente la contrainte SoftCumulative(A, $capa$, P, Loc, $Cost$, obj, max, sum). Les décompositions de SoftCumulative avec d'autres paramètres $costC$ et $objC$ sont similaires.

Décomposition 2 (Décomposition temps-ressource de la contrainte SoftCumulative) *Soient un ensemble $A = \{a_0, a_1, \ldots, a_{n-1}\}$ de n activités, un entier $capa$, $P = \{p_0, p_1, \ldots, p_{p-1}\}$ une partition de l'horizon de temps m, $Loc = \{lc_0, lc_1, \ldots, lc_{p-1}\}$ un ensemble d'entiers représentant les capacités locales des intervalles définis dans P, $Cost = \{cost_0, cost_1, \ldots, cost_{p-1}\}$ un ensemble de variables de coût, obj une variable d'objectif les agrégeant . La contrainte* SoftCumulative*(A, $capa$, P, Loc, $Cost$, obj, max, sum) est équivalente à la décomposition suivante.*

Variables

- *Un ensemble de variables entières $\mathcal{H} = \{h_0, h_1, \ldots, h_{m-1}\}$ tel que, pour tout $j \in [0, m-1]$, h_j représente la hauteur du profil cumulatif au point de temps j, avec $j \in p_k$, et $D(h_j) = [0, lc_k + \overline{cost_k}]$,*
- *un ensemble de variables entières $\mathcal{P} = \{p_{ij} | i \in [0, n-1], j \in [0, m-1]\}$ tel que, pour tout $a_i \in A$, et pour tout $j \in [0, m-1]$, p_{ij} représente la contribution de l'activité a_i au point de temps j, avec $D(p_{ij}) = \{0, ra_i\}$,*
- *un ensemble de variables entières $contrib = \{contrib_0, contrib_1, \ldots, contrib_{m-1}\}$, tel que, pour tout $j \in [0, m-1]$, avec $j \in p_k$, $contrib_j$ représente le dépassement de la capacité lc_k au point de temps j, avec $D(contrib_j) = [0, \overline{cost_k}]$.*

Contraintes

- *Pour toute activité $a_i \in A$:*
  - *$sa_i + da_i = ca_i$,*
  - *pour tout point de temps $j \in [0, m-1]$, $([j \geq sa_i \wedge j < ca_i] \wedge [p_{ij} = ra_i]) \vee ([j < sa_i \vee j \geq ca_i] \wedge [p_{ij} = 0])$,*
- *pour tout point de temps $j \in [0, m-1]$:*
  - *$h_j = \sum_{i \in [0, n-1]} (p_{ij})$*
  - *pour $p_k \in P$ tel que, $j \in p_k$, $contrib_j = \max(0, h_j - lc_k)$*
- *$\forall k \in [0, |Cost| - 1]$, $cost_k = \max_{j \in p_k}(contrib_j)$*
- *$obj = \sum_{k \in [0, |Cost|-1]} (cost_k)$*

6.3 Décompositions activité-ressource

6.3.1 Décomposition activité-ressource de Cumulative

El Kholy [26], Schutt et al. [77], puis Beldiceanu et al. [9] présentent une autre décomposition de la contrainte Cumulative, la décomposition *activité-ressource*. Celle-ci utilise un nombre de variables de l'ordre de $O(n^2)$, avec n le nombre d'activités du problème.

Plutôt que de créer une variable par point de temps et par activité, le nombre de points de temps considérés par la décomposition est ici égal au nombre d'activités.

Pour la présenter, nous nous appuyons sur l'exemple donné par la Figure 6.3. Elle représente une solution d'un problème cumulatif à 4 activités, modélisé par la contrainte Cumulative. Elle spécifie les valeurs affectées aux différentes variables de sa décomposition temps-ressource.

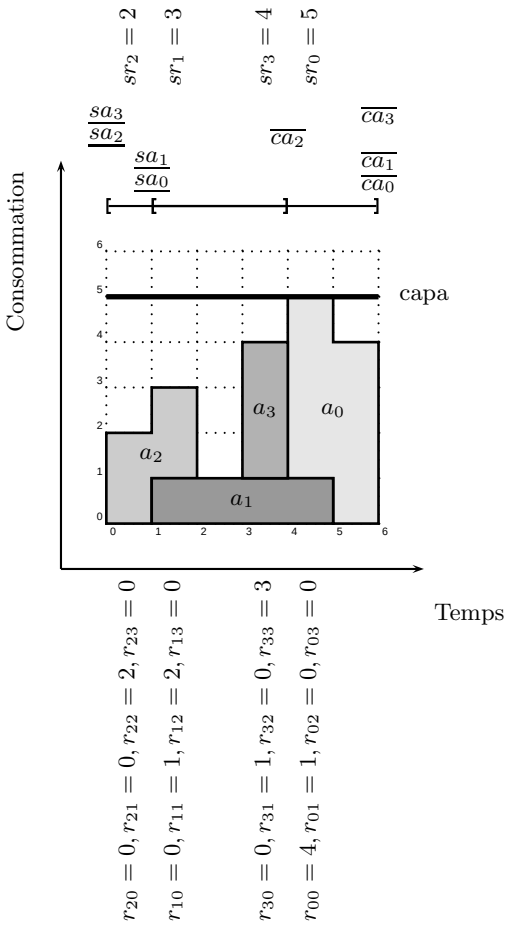


FIGURE 6.3 – Une solution d'un problème cumulatif à 4 activités modélisé par la décomposition activité-ressource de CUMULATIVE. Pour toutes activités $a_i, a_j \in A$, la variable r_{ij} prend la valeur de la hauteur ra_j de l'activité a_j si celle-ci est présente en sa_i, la valeur 0 sinon. La hauteur sr_i du profil à la date de début sa_i de l'activité a_i est égale à $\sum_{j \in [0,2]} r_{ij}$. En chaque début d'activité sa_i, sr_i est inférieure ou égale à la capacité *capa*.

Soit n le nombre d'activités d'un problème cumulatif représenté par la contrainte Cumulative$(A, capa)$. n^2 variables r_{ij} sont créées. Pour toutes activités $a_i, a_j \in A$, r_{ij} prend la valeur de la hauteur de l'activité $a_j \in A$ si elle est présente à la date de début de l'activité a_i, et 0 sinon, comme l'illustre l'exemple 47 dans le cas de la Figure 6.3.

Exemple 47 *Dans le cas instancié représenté par la Figure 6.3, les valeurs affectées aux variables r_{ij} sont résumées dans le tableau suivant, avec les lignes représentant les a_i et les colonnes les a_j. Les cases grisées représentent les activités a_j passant par la date de début sa_i de a_i.*

$a_i \backslash a_j$	a_0	a_1	a_2	a_3
a_0	$ra_0 = 4$	$ra_1 = 1$	0	0
a_1	0	$ra_1 = 1$	$ra_2 = 2$	0
a_2	0	0	$ra_2 = 2$	0
a_3	0	$ra_1 = 1$	0	$ra_3 = 3$

En chaque date de début d'activité sa_i, la somme des valeurs de ces variables donne la hauteur du profil sr_i. Considérer la hauteur du profil aux dates de début des activités suffit. En effet, une augmentation de hauteur du profil cumulatif ne peut survenir que lorsqu'une activité débute. Chaque hauteur sr_i est comparée à la capacité $capa$. La contrainte est respectée si, pour toute activité $a_i \in A$, $sr_i \leq capa$. Elle est violée dans le cas contraire. La valeur des variables sr_i est donné par l'exemple 48 dans le cas de la Figure 6.3.

Exemple 48 *Dans le cas instancié représenté par la Figure 6.3, les valeurs affectées aux variables sr_i,et la capacité capa sont indiqués dans le tableau suivant. En chacun des points de temps considérés $sr_i \leq capa$. La contrainte est donc satisfaite.*

a_i	a_0	a_1	a_2	a_3
sa_i	4	1	0	3
sr_i	5	3	2	4
$capa$	5			

Nous décrivons à présent formellement la décomposition activité-ressource de la contrainte Cumulative.

Décomposition 3 (Décomposition activité-ressource de la contrainte Cumulative) *Soit une ressource avec une capacité limitée par un entier capa et un ensemble $A = \{a_0, a_1, \ldots, a_{n-1}\}$ de n activités. La contrainte Cumulative$(A, capa)$ est équivalente à la décomposition suivante.*

Variables

- *un ensemble de variables entières $\mathcal{R} = \{r_{ij} | i \in [0, n-1], j \in [0, n-1]\}$ tel que, pour tout $a_i, a_j \in A$ (a_i et a_j n'étant pas nécessairement distinctes), r_{ij} représente la contribution de l'activité a_j à la date de début de l'activité a_i, avec $D(r_{ij}) = \{0, ra_j\}$,*

– *un ensemble de variables entières* $\mathcal{SR} = \{sr_0, sr_1, \ldots, sr_{n-1}\}$ *tel que, pour tout* $a_i \in A$, sr_i *représente la hauteur cumulée des activités passant par la date de début de* a_i, *avec* $D(sr_i) = [0, capa]$.

Contraintes

– *Pour toute activité* $a_i \in A$:
  – $sa_i + da_i = ca_i$,
  – $\forall a_j \in A$, $([sa_i \geq sa_j \wedge sa_i < ca_j] \wedge [r_{ij} = ra_j]) \vee ([sa_i < sa_j \wedge sa_i \geq ca_j] \wedge [r_{ij} = 0])$,
  – $sr_i = \sum_{j \in [0,n-1]}(r_{ij})$,
  – $sr_i \leq capa$.

6.3.2 Contribution : décomposition activité-ressource de SoftCumulative

Nous étendons la décomposition quadratique de la contrainte Cumulative, décrite en section 6.3.1, au cas de la contrainte SoftCumulative.

Il y est nécessaire de considérer les dates de changement d'intervalle utilisateur, pour deux raisons :

– la capacité peut changer entre deux intervalles utilisateur, il est donc nécessaire de comparer la hauteur du profil à la capacité en chaque intervalle utilisateur : la date de début de chaque intervalle suffit,
– des activités peuvent passer dans plusieurs intervalles utilisateur. Il ne suffit pas de considérer uniquement l'intervalle utilisateur où ces activités débutent : il est nécessaire de les prendre en compte en chacun des intervalles utilisateur.

Pour présenter cette extension, nous nous appuyons sur l'exemple donné par la Figure 6.4. Elle représente une solution d'un problème cumulatif relaxé à 4 activités, modélisé par la contrainte SoftCumulative. Elle indique les valeurs affectées aux différentes variables de sa décomposition quadratique.

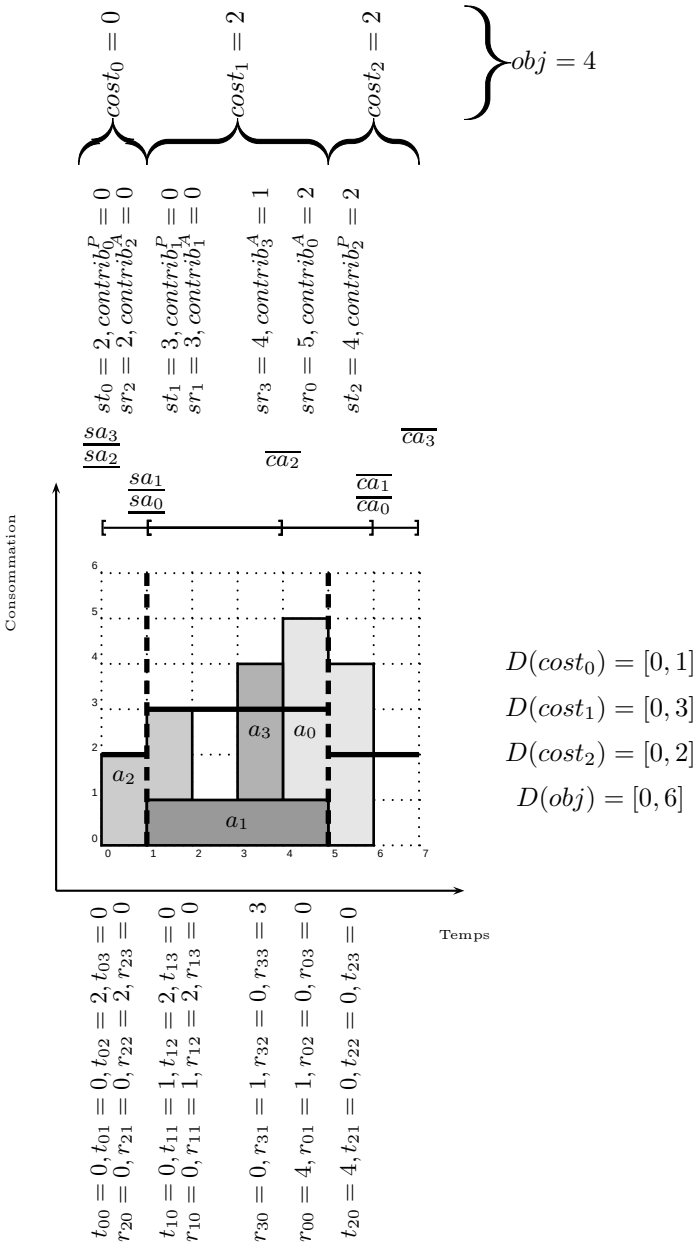


FIGURE 6.4 – Une solution d'un problème cumulatif relaxé à 4 activités et 3 intervalles utilisateur. Celui-ci est posé avec $D(cost_0) = [0, 1]$, $D(cost_1) = [0, 3]$, $D(cost_2) = [0, 2]$ et $D(obj) = [0, 6]$. Pour chaque activité $a_i \in A$, pour toute activité $a_j \in A$, la variable r_{ij} prend la valeur de la hauteur ra_j de l'activité a_j si celle-ci est présente en sa_i, la valeur 0 sinon. Pour chaque intervalle $p_k \in P$, pour toute activité $a_j \in A$, la variable t_{kj} prend la valeur de la hauteur ra_j de l'activité a_j si celle-ci est présente en sp_k, la valeur 0 sinon. La hauteur sr_i du profil à la date de début sa_i de l'activité a_i est égale à $\sum_{j \in [0,3]} r_{ij}$. La hauteur st_k du profil à la date de début sp_k de l'intervalle p_k est égale à $\sum_{j \in [0,3]} t_{kj}$. En chaque date de début d'activité sa_i dans l'intervalle p_k, si sr_i est inférieure ou égale à la capacité lc_k, $contrib_i^A = 0$, sinon, $contrib_i^A = sr_i - lc_k$. En chaque date de début d'intervalle sp_k de l'intervalle p_k, si st_k est inférieure ou égale à la capacité lc_k, $contrib_k^P = 0$, sinon, $contrib_k^P = st_k - lc_k$. Les coûts sont fonctions des variables de type *contrib*, et l'objectif est fonction des coûts.

Soient n le nombre d'activités, et m le nombre de points de temps du problème cumulatif représenté par la contrainte SoftCumulative(A, $capa$, P, Loc, $Cost$, obj,$costC$,$objC$). On considère ici $costC = max$, et $objC = sum$.

n^2 variables r_{ij} sont créées. Pour toutes activités $a_i, a_j \in A$, r_{ij} prend la valeur de la hauteur de l'activité $a_j \in A$, si elle est présente à la date de début de l'activité a_i, et 0 sinon, comme l'illustre l'exemple 49, dans le cas de la Figure 6.4.

Exemple 49 *Dans le cas instancié représenté par la Figure 6.4, les valeurs affectées aux variables r_{ij} sont résumées dans le tableau suivant, avec les lignes représentant les a_i et les colonnes les a_j. Les cases grisées représentent les activités a_j passant par la date de début sa_i de a_i.*

$a_i \backslash a_j$	a_0	a_1	a_2	a_3
a_0	$ra_0 = 4$	$ra_1 = 1$	0	0
a_1	0	$ra_1 = 1$	$ra_2 = 2$	0
a_2	0	0	$ra_2 = 2$	0
a_3	0	$ra_1 = 1$	0	$ra_3 = 3$

Pour prendre en compte les intervalles utilisateur $p_k \in P$, nous introduisons $p \times n$ variables t_{kj}. Pour tout intervalle $p_k \in P$ et toute activité $a_j \in A$, t_{kj} prend la valeur de la hauteur de l'activité a_j, si elle est présente à la date de début d'intervalle sp_k, et 0 sinon, comme l'illustre l'exemple 50, dans le cas de la Figure 6.4.

Exemple 50 *Dans le cas instancié représenté par la Figure 6.4, les valeurs affectées aux variables t_{kj} sont indiquées dans le tableau suivant, avec les lignes représentant les p_k et les colonnes les a_j. Les cases grisées représentent les activités a_j passant par la date de début sp_k de l'intervalle p_k.*

$p_k \backslash a_j$	a_0	a_1	a_2	a_3
p_0	0	0	$ra_2 = 2$	0
p_1	0	$ra_1 = 1$	$ra_2 = 2$	0
p_2	$ra_0 = 4$	0	0	0

Une variable de hauteur sr_i est associée à chaque date de début d'activité sa_i, et une variable de hauteur st_k est associée à chaque date de début d'intervalle sp_k. sr_i et st_k sont la somme des hauteurs des activités passant par le début de l'activité a_i et par le début de l'intervalle p_k, respectivement.

Nous ajoutons au modèle les variables de coût $cost_k \in Cost$ associées aux intervalles utilisateur et la variable objectif obj. Ces variables quantifient les dépassements de capacité. Il est alors nécessaire de différencier les capacités locales $lc_k \in Loc$, selon les intervalles..

Pour toute date de début d'activité $sa_i \in p_k$, la hauteur du profil n'est pas contrainte à être inférieure à la capacité locale lc_k. Cependant, le dépassement à la date de début d'activité sa_i : $contrib_i^A = sr_i - lc_k$ doit être inférieur au coût maximal autorisé $\overline{cost_k}$.

De la même manière, le dépassement à la date de début d'intervalle sp_k : $contrib_k^P = st_k - lc_k$ doit être inférieur au coût maximal autorisé $\overline{cost_k}$. Il est nécessaire de recalculer le dépassement à la date de

début d'intervalle, avec la hauteur de profil courante. En effet, la capacité y est potentiellement différente de celle de l'intervalle précédent, et la valeur du dépassement peut donc y être modifiée.

Le coût associé à un intervalle p_k est le maximum entre la valeur de la variable $contrib_k^P$ et le maximum des valeurs des variables $contrib_i^A$ telles que $sa_i \in p_k$.

Enfin, la valeur de l'objectif *obj* est égale à la somme des valeurs des variables de *Cost*. L'ensemble des valeurs affectées à ces variables dans le cas de la Figure 6.4 sont présentés dans l'exemple 51.

Exemple 51 *Dans le cas instancié représenté par la Figure 6.4, les valeurs affectées aux variables sr_i et st_k sont indiquées dans le tableau suivant. Pour chaque intervalle $p_k \in P$, la capacité lc_k, les variables $contrib_i^A$ associées à chaque début d'activité, les variables $contrib_k^P$ associées à chaque début d'intervalle utilisateur, et les valeurs des variables de coût sont également indiquées dans le tableau suivant. Nous grisons les valeurs de sr_i, st_k, $contrib_i^A$ et $contrib_k^P$ responsables du coût dans chaque intervalle. Nous avons : $D(cost_0) = [0, 1]$, $D(cost_1) = [0, 3]$, $D(cost_2) = [0, 2]$ et $D(obj) = [0, 6]$.En chacun des intervalles $p_k \in P$, $cost_k \leq \overline{cost_k}$. Enfin, la valeur de obj est représentée. Elle est inférieure à $\overline{obj}$. La contrainte est donc satisfaite.*

a_i		a_2		a_1	a_3	a_0	
dates considérées	$sp_0 = 0$	$sa_2 = 0$	$sp_1 = 1$	$sa_1 = 1$	$sa_3 = 3$	$sa_0 = 4$	$sp_2 = 5$
p_k	p_0		p_1				p_2
sr_i		2		3	4	5	
st_k	2		3				4
lc_k	2	3				2	
$contrib_i^A$		0		0	1	2	
$contrib_k^P$	0		0				2
$cost_k$	0	2					2
obj	4						

Nous définissons ensuite formellement la décomposition activité-ressource de la contrainte SoftCumulative, dans le cas où l'on représente la contrainte SoftCumulative(A, *capa*, P, *Loc*, *Cost*, *obj*, *max*, *sum*). Les décompositions de SoftCumulative avec d'autres paramètres *costC* et *objC* sont similaires.

Décomposition 4 (Décomposition activité-ressource de la contrainte SoftCumulative) *Soient un ensemble $A = \{a_0, a_1, \ldots, a_{n-1}\}$ de n activités, un entier capa, $P = \{p_0, p_1, \ldots, p_{p-1}\}$ une partition de l'horizon de temps m, $Loc = \{lc_0, lc_1, \ldots, lc_{p-1}\}$ un ensemble d'entiers représentant les capacités locales des intervalles définis dans P, $Cost = \{cost_0, cost_1, \ldots, cost_{p-1}\}$ un ensemble de variables de coût, obj une variable d'objectif les agrégeant. La contrainte* SoftCumulative*(A, capa, P, Loc, Cost, obj, max, sum) est équivalente à la décomposition suivante.*

Variables

- *Un ensemble de variables entières $\mathcal{R} = \{r_{ij} | i \in [0, n-1], j \in [0, m-1]\}$ tel que, pour tout $a_i, a_j \in A$ (a_i et a_j n'étant pas nécessairement distinctes), r_{ij} représente la contribution de l'activité a_j à la date de début de l'activité a_i, avec $D(r_{ij}) = \{0, ra_j\}$,*
- *un ensemble de variables entières $\mathcal{SR} = \{sr_0, sr_1, \ldots, sr_{n-1}\}$ tel que, pour tout $a_i \in A$, sr_i représente la hauteur cumulée des activités passant par la date de début de a_i, avec $D(sr_i) = [0, \max_{p_k \in P}(lc_k + \overline{cost_k})]$,*

- *un ensemble de variables entières* $\mathcal{T} = \{t_{kj} | k \in [0, p-1], j \in [0, n-1]\}$ *tel que, pour tout* $a_j \in A$ *et* $p_k \in P$, t_{kj} *représente la contribution de l'activité* a_j *à la date de début de l'intervalle* p_k, *avec* $D(t_{kj}) = \{0, ra_j\}$,
- *un ensemble de variables entières* $\mathcal{ST} = \{st_0, st_1, \ldots, st_{p-1}\}$ *tel que, pour tout* $p_k \in P$, st_k *représente la hauteur cumulée des activités passant par la date de début de l'intervalle* p_k, *avec* $D(st_k) = [0, lc_k + \overline{cost_k}]$,
- *un ensemble de variables entières* $contrib^A = \{contrib^A_0, contrib^A_1, \ldots, contrib^A_{n-1}\}$, *tel que, pour tout* $a_i \in A$, $contrib^A_i$ *représente le dépassement de la capacité locale à la date de début de* a_i, *avec* $D(contrib^A_i) = [0, \max_{p_k \in P}(\overline{cost_k})]$,
- *un ensemble de variables entières* $contrib^P = \{contrib^P_0, contrib^P_1, \ldots, contrib^P_{p-1}\}$, *tel que, pour tout* $p_k \in P$, $contrib^P_k$ *représente le dépassement de la capacité* lc_k *à la date de début de* p_k, *avec* $D(contrib^P_k) = [0, \overline{cost_k}]$.

Le nombre de variables utilisées par cette décomposition est donc de $n^2 + n \times p + 2 \times (n + p)$.

Contraintes

- *Pour toute activité* $a_i \in A$:
  - $sa_i + da_i = ca_i$,
  - $\forall a_j \in A$, $([sa_i \geq sa_j \wedge sa_i < ca_j] \wedge [r_{ij} = ra_j]) \vee ([sa_i < sa_j \wedge sa_i \geq ca_j] \wedge [r_{ij} = 0])$,
  - $sr_i = \sum_{j \in [0, n-1]}(r_{ij})$,
  - *pour* k *tel que* $sa_i \in p_k$, $contrib^A_i = \max(0, sr_i - lc_k)$,
- *pour tout intervalle* $p_k \in P$:
  - $\forall a_j \in A$, $([sp_k \geq sa_j \wedge sp_k < ca_j] \wedge [t_{kj} = ra_j]) \vee ([sp_k < sa_j \wedge sp_k \geq ca_j] \wedge [t_{kj} = 0])$,
  - $st_k = \sum_{j \in [0, n-1]}(t_{kj})$,
  - $contrib^P_k = \max(0, st_k - lc_k)$,
- $\forall k \in [0, |Cost| - 1]$, $cost_k = \max(contrib^P_k, \max_{a_i \in A | sa_i \in p_k}(contrib^A_i))$,
- $obj = \sum_{k \in [0, |Cost|-1]}(cost_k)$.

6.4 Une décomposition linéaire en nombre de variables

La décomposition temps-ressource des contraintes Cumulative et SoftCumulative est dépendante du nombre de points de temps de l'horizon, et du nombre d'activités. Les horizons de temps considérés dans ce type de problèmes peuvent être importants, de même que le nombre d'activités considérées. Le nombre de variables peut ainsi être très haut.

La décomposition activité-ressource, quant à elle, un nombre de variables quadratique par rapport au nombre d'activités. Ce n'est pas nécessairement problématique pour des problèmes de petite taille, mais peut l'être pour d'autres problèmes.

Dès lors, introduire une décomposition linéaire en nombre de variables peut être intéressant.

6.4.1 Contribution : décomposition linéaire de Cumulative

Nous introduisons une décomposition de la contrainte Cumulative impliquant un nombre de variables de l'ordre de $O(n)$, où n est le nombre d'activités. A notre connaissance, aucune décomposition linéaire n'avait été jusque là proposée.

Pour illustrer notre description, nous nous appuyons sur la Figure 6.5. Elle représente une solution d'un problème cumulatif à 4 activités et spécifie les valeurs affectées aux différentes variables de sa décomposition linéaire.

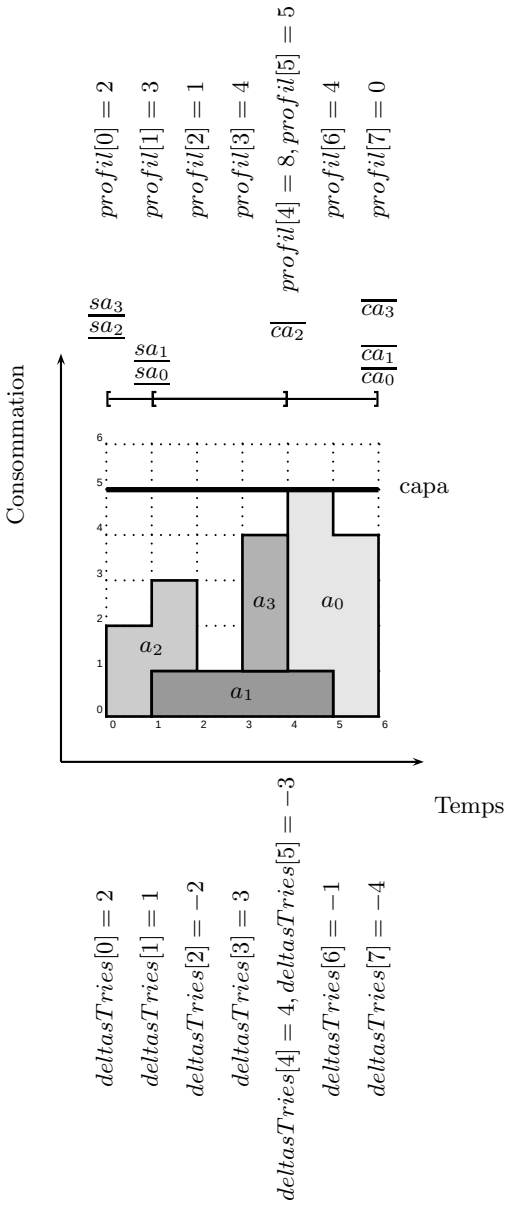


FIGURE 6.5 – Une solution d'un problème cumulatif à 4 activités. Chaque activité a_i engendre deux variables : une variable $deltas[i] = ra_i$ correspondant à la date sa_i, et une variable $deltas[n+i] = -ra_i$ correspondant à ca_i. Ces variables sont ensuite triées par dates croissantes dans les tableaux $datesTriees$ et $deltasTries$. Par souci de clarté, sur le schéma, ces variables sont déjà triées. La variable $profil[0]$ prend la valeur de variable $deltasTries[0]$. A chaque date associée à une variable $deltaTries[j]$ correspond une variable $profil[j] = profil[j-1] + deltasTries[j]$ selon le cas. A chaque changement de date, c'est à dire que $deltasTries[j+1]$ n'est pas associée à la même date que $deltasTries[j]$, $profil[j]$ donne la hauteur du profil. Celle-ci est inférieure à $capa$.

Soit n le nombre d'activités d'un problème cumulatif représenté par la contrainte Cumulative$(A, capa)$. Cette décomposition ne prend en compte que les $2 \times n$ points de temps correspondant au début sa_i ou à la fin ca_i d'une d'activité $a_i \in A$. Ces variables de dates sont consignées dans un tableau *dates*.

A chacune de ces dates est associé une variation du profil : si une activité est fixée, le profil cumulatif prend en compte sa hauteur pendant toute la durée de cette activité. Cette variation est égale à la hauteur ra_i de l'activité a_i à la date sa_i et de $-ra_i$ à la date ca_i. Ces variations sont consignées dans un tableau d'entiers *deltas*, de longueur $2 \times n$, comme l'illustre l'exemple 52 dans le cas de la Figure 6.5.

Exemple 52 *Dans le cas de la Figure 6.5, le tableau dates et le tableau deltas correspondent, dans le cas instancié, aux tableaux suivants.*

dates	$sa_0 = 4$	$sa_1 = 1$	$sa_2 = 0$	$sa_3 = 3$	$ca_0 = 6$	$ca_1 = 5$	$ca_2 = 2$	$ca_3 = 4$
deltas	$ra_0 = 4$	$ra_1 = 1$	$ra_2 = 2$	$ra_3 = 3$	$-ra_0 = -4$	$-ra_1 = -1$	$-ra_2 = -2$	$-ra_3 = -3$

L'idée de l'algorithme de Sweep est ici réutilisée : nous maintenons, de manière incrémentale la hauteur du profil cumulatif au cours du temps. Pour cela, les dates sont triées par ordre croissant, et les variations du profil sont triées en suivant la même permutation *permut*, dans des tableaux $datesTriees$ et $deltasTries$ respectivement, de dimension $2 \times n$. Ainsi, pour tout index id^d de *dates*, $dates[id^d] = datesTriees[permut[id^d]]$. De même, pour tout index id^δ de *deltas*, $deltas[id^\delta] = deltasTries[permut[id^\delta]]$. L'exemple 53 en est une illustration dans le cas de la Figure 6.5.

Exemple 53 *Les tableaux suivants présentent les instanciations des tableaux triés et de la permutation associée, pour le problème de la Figure 6.5.*

$datesTriees$	$sa_2 = 0$	$sa_1 = 1$	$ca_2 = 2$	$sa_3 = 3$	$sa_0 = 4$	$ca_3 = 4$	$ca_1 = 5$	$ca_0 = 6$
permut	4	1	0	3	7	6	2	5
$deltasTries$	$ra_2 = 2$	$ra_1 = 1$	$-ra_2 = -2$	$ra_3 = 3$	$ra_0 = 4$	$-ra_3 = -3$	$-ra_1 = -1$	$-ra_0 = -4$

La hauteur du profil cumulatif varie selon chaque variation de profil contenue dans $deltasTries$, aux dates contenues dans $datesTriees$. Les variations de la hauteur du profil correspondantes à ces dates sont consignées dans un tableau $profil$. La hauteur du profil cumulatif à une date donnée est donnée la par dernière case du tableau $profil$ correspondant à cette date. Si la hauteur du profil cumulatif est inférieure à la capacité *capa*, en tous les points de temps considérés, la contrainte est satisfaite. L'exemple 54 présente le tableau $profil$ dans le cas de la Figure 6.5.

Exemple 54 *Les tableaux suivants sont les instanciations des tableaux $profil$ et $datesTriees$ associé (que nous rappelons) correspondant à la Figure 6.5. Nous indiquons aussi la capacité capa. Les cases de $profil$ correspondant à des hauteurs de profil cumulatif, et les dates associées sont grisées.Elles sont inférieures à la capacité capa. De ce fait, la contrainte est satisfaite.*

Nous définissons à présent formellement notre nouvelle reformulation linéaire.
Pour cela, nous introduisons dans un premier temps la contrainte sortedness.

$datesTriees$	$sa_2 = 0$	$sa_1 = 1$	$ca_2 = 2$	$sa_3 = 3$	$sa_0 = 4$	$ca_3 = 4$	$ca_1 = 5$	$ca_0 = 6$
$profil$	2	3	1	4	8	5	4	0
$capa$	5							

Définition 49 (SORTEDNESS [95, 56]) *Soient trois tableaux de variables entières, de même dimension n : $from$, $permutation$, to. La contrainte* SORTEDNESS$(from, permutation, to)$ *impose :*
- *C1 : pour tout $i \in [0, n-1]$, $from[i] = to[permutation[i]]$*
- *C2 : les variables de to sont triées selon l'ordre croissant.*

Décomposition 5 (Décomposition linéaire de la contrainte CUMULATIVE**)** *Soit une ressource avec une capacité limitée par un entier capa et un ensemble $A = \{a_0, a_1, \ldots, a_{n-1}\}$ de n activités. La contrainte* CUMULATIVE$(A, capa)$ *est équivalente à la décomposition suivante.*

Variables
- *Un tableau de variables dates de dimension $2n$ contenant, pour toute activité $a_i \in A$, sa variable de début sa_i à l'index i, et sa variable de fin ca_i à l'index $n + i$,*
- *un tableau d'entiers deltas de dimension $2n$ contenant, pour toute activité $a_i \in A$, la variation de la hauteur de profil liée à a_i, égale à ra_i à l'index i, et à $-ra_i$ à l'index $n + i$,*
- *un tableau de variables datesTriees de dimension $2n$ contenant l'ensemble des dates du tableau dates, triées par ordre croissant, tel que, pour tout $j \in [0, 2n-1], D(datesTriees[j]) = [0, m-1]$,*
- *un tableau de variables permut, de dimension $2n$, contenant la permutation permettant de passer de dates à datesTriees, tel que, pour tout $j \in [0, 2n-1], D(permut[j]) = [0, 2n-1]$,*
- *un tableau de variables deltasTries de dimension $2n$ contenant l'ensemble des variations de hauteur du tableau deltas, triées selon la même permutation que le tableau datesTriees, tel que, pour tout $j \in [0, 2n-1], D(deltasTries[j]) = [-\max_{a_i \in A}(ra_i), \max_{a_i \in A}(ra_i)]$,*
- *un tableau de variables $profil$ de dimension $2n$ contenant, à l'indice i la somme des variations de hauteur du profil précédant la date $datesTriees[i]$, tel que, pour tout $i \in [0, 2n-1], D(profil[i]) = [0, \sum_{a_i \in A} ra_i]$.*

Contraintes
- *Pour toute activité $a_i \in A$,*
  - *$sa_i + da_i = ca_i$,*
  - *$dates[i] = sa_i$,*
  - *$dates[n + i] = ca_i$,*
  - *$deltas[i] = ra_i$,*
  - *$deltas[n + i] = -ra_i$,*
- SORTEDNESS$(dates, permut, datesTriees)$,
- SORTEDNESS$(deltas, permut, deltasTries)$,
- *$profil[0] = deltasTries[0]$,*
- *pour tout index $j \in [1, 2n-1]$, $profil[j] = profil[j-1] + deltasTries[j]$,*
- *pour tout index $j \in [0, 2n-2]$, $(datesTriees[j] = datesTriees[j+1]) \vee ((datesTriees[j] \neq datesTriees[j+1]) \wedge (profil[j] \leq capa))$.*

6.4.2 Contribution : décomposition linéaire de SOFTCUMULATIVE

Nous étendons notre décomposition linéaire de CUMULATIVE au cas SOFTCUMULATIVE.

Pour décrire cette extension, nous appuyons sur l'exemple de la Figure 6.6. Elle présente un problème cumulatif relaxé à 4 activités, et spécifie les valeurs affectées aux différentes variables de sa décomposition linéaire.

Soit n le nombre d'activités et p le nombre dintervalles utilisateur d'un problème cumulatif relaxé représenté par la contrainte SoftCumulative($A, capa, P, Loc, Cost, obj, costC, objC$). On considère ici $costC = max$, et $objC = sum$.

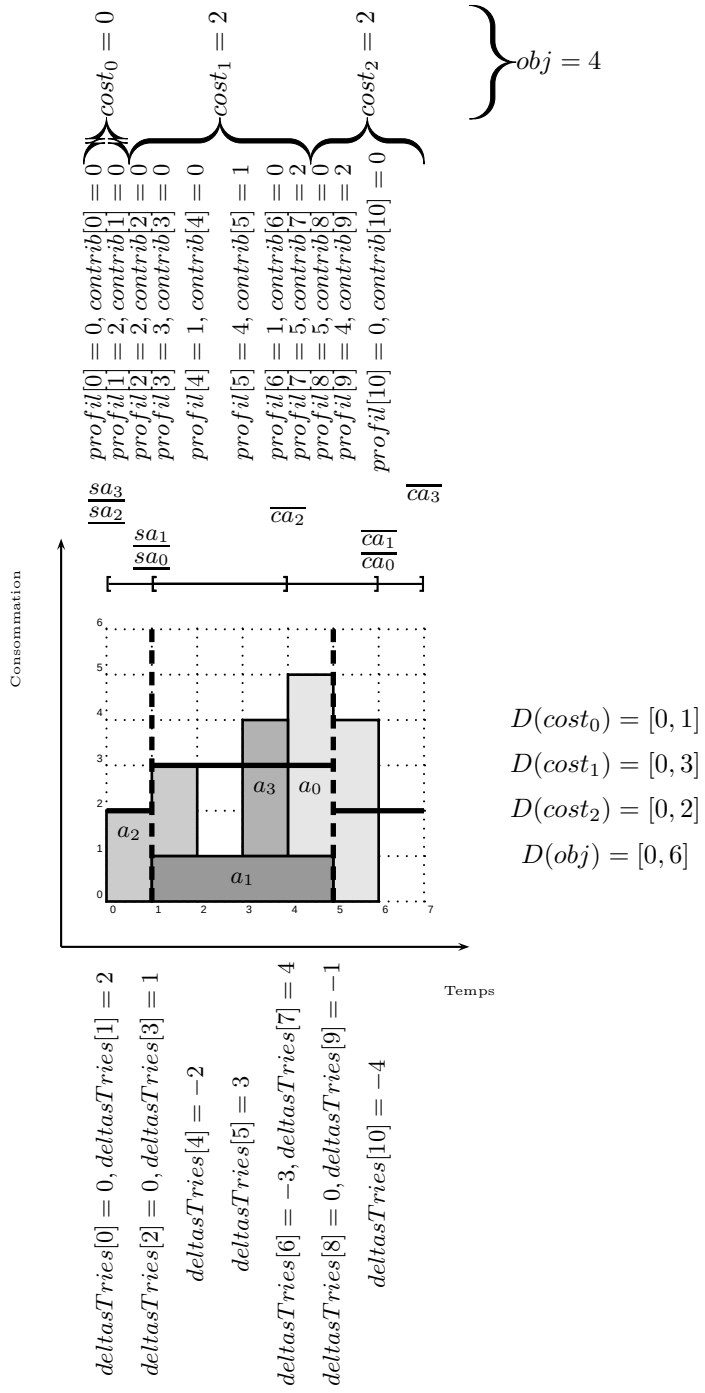


FIGURE 6.6 – Une solution d'un problème cumulatif relaxé à 4 activités et 3 intervalles utilisateur. Celui-ci est posé avec $D(cost_0) = [0, 1]$, $D(cost_1) = [0, 3]$, $D(cost_2) = [0, 2]$ et $D(obj) = [0, 6]$. Chaque activité a_i engendre deux variables : une variable $deltas[i] = ra_i$ correspondant à la date sa_i, et une variable $delta[n + i] = -ra_i$ correspondant à ca_i. Ces variables sont ensuite triées par dates croissantes dans les tableaux $datesTriees$ et $deltasTries$. Par souci de clarté, sur le schéma, ces variables sont déjà triées. La variable $profil[0]$ prend la valeur de la $deltaTries[0]$. A chaque date associée à une variable $deltaTries[j]$ correspond une variable $profil[j] = profil[j - 1] + deltasTries[j]$. A chaque changement de date, c'est à dire que $deltasTries[j + 1]$ n'est pas associée à la même date que le courant, $profil[j]$ donne la hauteur du profil. Les variables $contrib[j]$, telles que $datesTriees[j] \in p_k$ prennent la valeur de $profil[j] - lc_k$ si la prochaine variable $deltasTries[j + 1]$ est associé à une date différente de la date de $deltaTries[j]$. Sinon, $contrib[j] = 0$. Les coûts sont fonctions des variables contrib, et l'objectif est fonction des coûts.

Là encore, et pour les mêmes raisons qu'évoquées pour la décomposition activité-ressource décrite en section 6.3.2, l'adaptation nécessite la prise en compte des dates de début d'intervalle. Nous ajoutons celles-ci au tableau *dates*, de longueur $2 \times n + p$. Notons que *dates* est un tableau de variables. Pour ajouter les dates de début d'intervalle, nous les considérons donc comme des variables dont le domaine est de taille 1. La contribution liée à une date de début d'intervalle est nulle. En conséquence, nous ajoutons la valeur 0 aux dates correspondant aux débuts d'intervalles utilisateur dans le tableau *deltas*, de longueur $2 \times n + p$. L'exemple 55 donne ces tableaux dans le cas de la Figure 6.6.

Exemple 55 *Dans le cas du problème de la Figure 6.6, le tableau dates et le tableau deltas correspondent, dans le cas instancié aux tableaux suivants*

dates	$sa_0 = 4$	$sa_1 = 1$	$sa_2 = 0$	$sa_3 = 3$	$ca_0 = 6$	$ca_1 = 5$	$ca_2 = 2$	$ca_3 = 4$	$sp_0 = 0$	$sp_1 = 1$	$sp_2 = 5$
deltas	$ra_0 = 4$ $= 4$	$ra_1 = 1$ $= 1$	$ra_2 = 2$ $= 2$	$ra_3 = 3$ $= 3$	$-ra_0$ $= -4$	$-ra_1$ $= -1$	$-ra_2$ $= -2$	$-ra_3$ $= -3$	0	0	0

datesTriees est la représentation triée du tableau *dates*. La permutation *permut* permet de passer de *dates* à *datesTriees*. En utilisant la même permutation *permut*, on passe de *deltas* à *deltasTries*, comme l'illustre l'exemple 56 dans le cas de la Figure 6.6.

Exemple 56 *Les tableaux datesTriees, permut et deltasTries instanciés suivants correspondant à la Figure 6.6. Le tableau permut permet de passer de dates à datesTriees et de deltas à deltasTries. Pour tout* $i \in [0, 10]$, $dates[i] = datesTriees[permut[i]]$, *et* $deltas[i] = deltasTries[permut[i]]$.

datesTriees	$sp_0 = 0$	$sa_2 = 0$	$sp_1 = 1$	$sa_1 = 1$	$ca_2 = 2$	$sa_3 = 3$	$sa_0 = 4$	$ca_3 = 4$	$sp_2 = 5$	$ca_1 = 5$	$ca_0 = 6$
permut	6	3	1	5	10	9	4	7	0	2	8
deltasTriees	0	$ra_2 = 2$	0	$ra_1 = 1$	$-ra_2 = -2$	$ra_3 = 3$	$ra_0 = 4$	$-ra_3 = -3$	0	$-ra_1 = -1$	$-ra_0 = -4$

La hauteur du profil cumulatif varie selon chaque variation de profil contenue dans *deltasTries*, aux dates contenues dans *datesTriees*. Les variations de la hauteur du profil correspondantes à ces dates sont consignées dans un tableau $profil$, de longueur $2 \times n + p$. La hauteur du profil cumulatif à une date donnée est donnée par dernière case du tableau $profil$ correspondant à cette date. L'exemple 57 donne les valeurs dans ce tableau dans le cas de la Figure 6.6.

Exemple 57 *Le tableau $profil$ instancié suivant correspond à la Figure 6.6. Nous rappelons également les dates associées aux variations de profil. Les cases de $profil$ correspondant à des hauteurs de profil cumulatif, et les dates associées sont grisées.*

$datesTriees$	sp_0 $=0$	sa_2 $=0$	sp_1 $=1$	sa_1 $=1$	ca_2 $=2$	sa_3 $=3$	sa_0 $=4$	ca_3 $=4$	sp_2 $=5$	ca_1 $=5$	ca_0 $=6$
$profil$	0	2	2	3	1	4	1	5	5	4	0

Le tableau *contrib*, de longueur $2\times n+p$, contient, à chaque case $profil[j]$ du tableau $profil$ définissant une hauteur de profil cumulatif, avec $datesTriees[j] \in p_k$, la valeur du dépassement $h_j - lc_k$. Si $profil[j]$ ne correspond pas à une hauteur de profil cumulatif, contrib[j] $= 0$. Le dépassement $contrib[j] = profil[j] - lc_k$ doit être inférieur au coût maximal autorisé $\overline{cost_k}$. Le coût associé à un intervalle $p_k \in P$ est le maximum des variables $contrib[j]$ telles que $j \in p_k$. Enfin, la valeur de l'objectif *obj* est égale à la somme des valeurs des variables *Cost*. L'exemple 58 donne les valeurs dans ces tableaux dans le cas de la Figure 6.6.

Exemple 58 *Nous présentons les tableaux $profil$, et contrib instanciés correspondant à la Figure 6.6. Nous rappelons également le tableau datesTriees. Les cases de $profil$ correspondant à une hauteur de profil cumulatif, ainsi que les cases des variables de contrib correspondantes sont grisées. Nous indiquons les capacités des différents intervalles utilisateurs, et les coûts associés, calculés en fonction de contrib et des capacités. Enfin, nous indiquons la valeur de l'objectif. Nous avons : $D(cost_0) = [0,1]$, $D(cost_1) = [0,3]$, $D(cost_2) = [0,2]$ et $D(obj) = [0,6]$.En chacun des intervalles $p_k \in P$, $cost_k \leq \overline{cost_k}$. Enfin, la valeur de obj est représentée. Elle est inférieure à $\overline{obj}$. La contrainte est donc satisfaite.*

$datesTriees$	sp_0 $=0$	sa_2 $=0$	sp_1 $=1$	sa_1 $=1$	ca_2 $=2$	sa_3 $=3$	sa_0 $=4$	ca_3 $=4$	sp_2 $=5$	ca_1 $=5$	ca_0 $=6$
p_k	p_0		p_1						p_2		
$profil$	0	2	2	3	1	4	1	5	5	4	0
lc_k	2		3						2		
$contrib$	0	0	0	0	0	1	0	2	0	2	0
$cost_k$	0		2						2		
obj	4										

Nous définissons formellement notre décomposition linéaire de la contrainte SoftCumulative, dans le cas où l'on représente la contrainte SoftCumulative(A, *capa*, P, *Loc*, *Cost*, *obj*, max, *sum*). Les décompositions de SoftCumulative avec d'autres paramètres *costC* et *objC* sont similaires.

Décomposition 6 (Décomposition linéaire de la contrainte SoftCumulative) *Soit une ressource avec une capacité limitée par un entier capa, un ensemble $A = \{a_0, a_1, \ldots, a_{n-1}\}$ de n activités, P une partition de l'horizon de temps m, Loc un ensemble d'entiers représentant les capacités locales des intervalles définis dans P, Cost un ensemble de variables de coût, obj une variable d'objectif les agrégeant, et costC et objC des constantes représentant le mode de calcul des coûts et de l'objectif. On note, de plus, p le nombre d'intervalles utilisateur. On a alors $p = |P| - 1$. La contrainte* SoftCumulative*(A, capa, P, Loc, Cost, obj, costC, objC) est équivalente à la décomposition suivante.*

Variables

- *Un tableau de variables dates de dimension* $2n+p$ *contenant, pour toute activité* $a_i \in A$*, sa variable de début* sa_i *à l'index* i*, et sa variable de fin* ca_i *à l'index* $n+i$*, et pour tout intervalle* $p_k \in P$*, sa date de début* sp_k *à l'index* $2n+k$*,*
- *un tableau d'entiers deltas de dimension* $2n+p$ *contenant, pour toute activité* $a_i \in A$*, la variation de la hauteur de profil liée à* a_i*, égale à* ra_i *à l'index* i*, et à* $-ra_i$ *à l'index* $n+i$*, et pour tout intervalle* $p_k \in P$*, sa contribution* 0 *à l'index* $2n+k$*,*
- *un tableau de variables* $datesTriees$ *de dimension* $2n+p$ *contenant l'ensemble des dates du tableau dates, triées par ordre croissant, tel que, pour tout* $j \in [0, 2n+p-1], D(dates[j]) = [0, m-1]$*,*
- *un tableau de variables permut, de dimension* $2n+p$*, contenant la permutation permettant de passer de dates à* $datesTriees$*, tel que, pour tout* $j \in [0, 2n+p-1], D(permut[j]) = [0, 2n+p-1]$*,*
- *un tableau de variables* $deltasTries$ *de dimension* $2n+p$ *contenant l'ensemble des variations de hauteur du tableau deltas, triées selon la même permutation que le tableau* $datesTriees$*, tel que, pour tout* $j \in [0, 2n+p-1], D(deltasTriees[j]) = [-\max_{a_i \in A} ra_i, \max_{a_i \in A}(ra_i)]$*,*
- *un tableau de variables* $profil$ *de dimension* $2n+p$ *contenant, à l'indice* j *la somme des variations de hauteur du profil précédant la date* $datesTriees[j]$*, tel que, pour tout* $j \in [0, 2n+p-1], D(profil[j]) = [0, \sum_{a_i \in A}(ra_i)]$*.*
- *un tableau de variables contrib de dimension* $2n+p$ *contenant, à l'indice* j *le dépassement de capacité lorsque* $profil[j]$ *correspond à une hauteur de profil cumulatif,* 0 *sinon, tel que, pour tout* $j \in [0, 2n+p-1]$*, avec* $datesTriees[j] \in p_k$*,* $D(contrib[j]) = [0, \max_{p_k \in P}(cost_k)]$*.*

Contraintes

- *Pour toute activité* $a_i \in A$*,*
  - $sa_i + da_i = ca_i$*,*
  - $dates[i] = sa_i$*,*
  - $dates[n+i] = ca_i$*,*
  - $deltas[i] = ra_i$*,*
  - $deltas[n+i] = -ra_i$*,*
- *Pour tout intervalle* $p_k \in P$
  - $dates[n+p+k] = sp_k$*,*
  - $deltas[n+p+k] = 0$*,*
- SORTEDNESS$(dates, permut, datesTriees)$*,*
- SORTEDNESS$(deltas, permut, deltasTries)$*,*
- $profil[0] = deltasTries[0]$*,*
- *pour tout index* $j \in [1, 2n+p-1]$*,* $profil[j] = profil[j-1] + deltasTries[j]$*,*
- *pour tout index* $j \in [0, 2n+p-2]$*, et* p_k *tel que* $datesTriees[j] \in p_k$*,* $(datesTriees[j] = datesTriees[j+1] \wedge contrib[j] = 0) \vee ((datesTriees[j] \neq datesTriees[j+1]) \wedge (contrib[j] = \max(0, profil[j] - lc_k))$*,*
- *pour tout* $k \in [0, |Cost|-1]$*,* $cost_k = \max_{datesTriees[j] \in p_k}(contrib[j])$
- $obj = \sum_{k \in [0, |Cost|-1]}(cost_k)$

Chapitre 7

Expérimentations

Sommaire

7.1 Évaluation de la contrainte globale SoftCumulative

Afin d'évaluer le comportement de notre contrainte globale SoftCumulative, nous identifions différents paramètres susceptibles d'influencer la difficulté de résolution d'une instance :

- le nombre d'activités
- le nombre d'intervalles
- l'introduction d'activités plus grandes (longueur et hauteur)
- le début/fin des activités
- les contraintes additionnelles sur les coûts

En l'absence de jeux d'instances existants, nous en générons aléatoirement, suivant les paramètres précédemment identifiés.

Nous adaptons également au cas sur-contraint des instances existant pour la contrainte Cumulative : les instances de la PSPLib [45].

Nous effectuons nos tests avec la contrainte SoftCumulative paramétrée avec $costC = max$ et $objC = sum$. Nous rappelons que cela signifie que le coût dans chaque intervalle utilisateur est calculé en fonction du dépassement maximal de la capacité dans cet intervalle utilisateur, et que l'objectif global est la somme des variables de coût des intervalles utilisateurs.

Dans un premier temps, nous recherchons une première solution avec différents types d'instances. Nous les présentons et justifions ces tests en section 7.1.1.

Nous effectuons des tests pour chercher une solution optimale, et prouver l'optimalité, pour différents types de problèmes, représentés par la contrainte SoftCumulative, en y ajoutant éventuellement des contraintes additionnelles sur les variables de coût. Ces tests sont détaillés en section 7.1.2.

Enfin, l'adaptation d'instances de la librairie PSPLib [45] au cas cumulatif sur-contraint est présentée dans la section 7.1.3.

L'ensemble des tests est effectué sur un processeur 2,5 GHz Intel Core 2 Duo, avec une mémoire vive de 4Go cadencée à 1067 MHz. Nous effectuons ces tests sous l'environnement *Eclipse* intégré du solveur de contraintes *Choco* de la version 2.1.3.

7.1.1 Recherche de solution

Dans un premier temps, nous recherchons une première solution à des problèmes modélisés par la contrainte SoftCumulative. Ceci consiste à trouver un ordonnancement valide : l'ordonnancement peut occasionner des dépassements, mais ceux ci doivent être dans le domaine des variables de coût et d'objectif. Nous ne cherchons pas pour le moment à minimiser l'objectif global. L'intérêt de cette expérience est d'évaluer si notre approche passe à l'échelle.

Nous rappelons que le problème cumulatif, dans sa variante décisionnelle, est NP-complet au sens fort [34]. Le problème cumulatif relaxé, modélisé par notre contrainte SoftCumulative, dans sa variante décisionnelle, peut se ramener à un problème cumulatif dans certains cas particuliers. En effet, dans le cas où, dans tous les intervalles utilisateurs, la somme de la capacité locale et de la valeur maximale de la variable de coût, est la même, la contrainte SoftCumulative est équivalente à une contrainte Cumulative avec cette somme pour capacité globale. Le problème cumulatif relaxé, que nous représentons par SoftCumulative, est donc un problème NP-complet.

Nous avons présenté des décompositions de la contraintes SoftCumulative dans le chapitre 6. Nous voulons savoir si la taille des problèmes traités par ces décompositions peut être aussi importante que la taille des problèmes traités par la contrainte globale SoftCumulative.

Dans le cas de la recherche de solution, nous débranchons les algorithmes d'Edge-Finding (ceux de Vilím et de Kameugne et al.). Ils induisent en effet une complexité temporelle supplémentaire trop lourde pour la simple recherche de solution. En revanche, nous utilisons l'algorithme de Sweep.

Pour chaque classe d'instances, nous lançons 100 instances *sur-contraintes*, c'est-à- dire qu'il n'existe pas de solution au problème qui n'engendre pas de dépassement. Pour nous en assurer, nous avons généré les problèmes, et tenté de les résoudre en imposant une valeur d'objectif égale à 0. Nous avons conservé les instances prouvées infaisables par ce test, et n'avons pas pris en compte celles qui donnaient une solution.

Nous fixons la limite de temps de résolution à 2 minutes. Cela nous semble raisonnable dans le cas de la recherche d'une première solution.

Nous utilisons la stratégie de recherche décrite dans la section 5.2.2.

7.1.1.1 Variation du nombre d'activités et du nombre d'intervalles

Description Nous testons le passage à l'échelle de notre contrainte, sur des instances à 100, 250, 500 et 1000 activités, et à 8, 16 ou 32 intervalles. En faisant varier la longueur des intervalles, nous nous assurons que l'horizon de temps reste toujours du même ordre, pour des instances avec le même nombre d'activité. Les activités testées ont une longueur entre 1 et 4, et une hauteur entre 1 et 3. Toutes les activités ont leur début au plus tôt en 0 et leur fin au plus tard égale à l'horizon de temps.

Les premiers tests que nous effectuons comparent des instances dont les variables de coût ont toutes le même domaine : $[0, 6]$, et des instances pour lesquelles les variables de coût ont un domaine dont la valeur maximale est choisie aléatoirement dans l'ensemble $\{3, 4, 5, 6\}$ et dont la valeur minimale est 0.

Nous cherchons à connaître le comportement de la contrainte SoftCumulative dans les deux cas.

La Table 7.1 donne les résultats d'expérimentations dans les deux cas.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
100 / 8	16	86	14	0	0.1s	8
100 / 16	8	93	7	0	0.2s	39
100 / 32	4	94	2	4	1.2s	1298
250 / 8	40	91	8	1	1.3s	70
250 / 16	20	98	1	1	1.7s	132
250 / 32	10	96	3	1	2.1s	137
500 / 8	80	97	3	0	16.0s	547
500 / 16	40	91	7	2	12.9s	411
500 / 32	20	98	0	2	10.2s	191
1000 / 8	160	56	9	35	54.6s	11
1000 / 16	80	70	4	26	59.8s	87
1000 / 32	40	79	1	20	59.6s	37

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
100 / 8	20	84	16	0	0.2s	140
100 / 16	10	94	3	3	0.4s	317
100 / 32	5	97	0	3	0.4s	271
250 / 8	48	89	11	0	7.0s	1674
250 / 16	25	97	3	0	7.0s	1679
250 / 32	13	94	0	6	6.2s	1291
500 / 8	92	33	19	48	32.0s	1354
500 / 16	50	48	3	49	47.3s	2830
500 / 32	25	40	0	60	57.8s	191
1000 / 8	175	0	3	97	-	-
1000 / 16	90	0	4	96	-	-
1000 / 32	45	0	3	97	-	-

Table 7.1 – Résultats en satisfaction pour une valeur maximale des domaines des variables de coût égale dans tous les intervalles (premier tableau), puis avec une diversité dans les valeurs maximales des domaines des variables de coût (deuxième tableau). La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonne longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues donne le nombre d'instances pour lesquelles une solution a été trouvée dans le temps imparti, parmi les instances n'ayant pas de solution engendrant un dépassement nul. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues.

Analyse des résultats Les deux tableaux montrent qu'il est plus difficile de résoudre un problème dont les variables de coût ont des domaines de taille différente. En effet, le deuxième tableau montre que, lorsque l'on réduit la borne supérieure de certaines variables de coût, la résolution est plus difficile, car le problème est plus contraint. Ceci est vrai pour toutes les classes d'instance observées, peu importe le nombre d'activités, ou d'intervalles.

Les deux tableaux montrent également que plus la taille du problème est grande, en terme de nombre d'activités, plus la résolution est difficile. Le nombre d'instances non résolues augmente lorsque l'on ajoute des activités, tout comme la moyenne des temps de résolution.

Dans le premier cas, nous arrivons à résoudre des instances jusqu'à 1000 activités, sous deux minutes, avec peu de backtracks. Dans le second, deux minutes ne suffisent pas à trouver une solution pour des instances à 1000 activités.

Nous observons une différence d'influence du nombre d'intervalles utilisateur dans les deux tableaux.

Dans le premier, lorsque les variables de coût ont toutes les même domaine, nous observons que plus le nombre d'activités est grand, moins le nombre d'intervalles utilisateur a d'incidence sur le temps de résolution. Ceci est du à la complexité de l'algorithme de Sweep. En effet, cette complexité est en $O((n+p) \times log(n+p) + n \times (n+p))$. Alors, plus le nombre d'activités n est grand, plus le nombre d'intervalles utilisateur p est négligeable devant n.

Dans le second, lorsque les variables de coût ont des domaines de taille différente, nous constatons que le nombre d'intervalles utilisateur influence fortement le temps de résolution, notamment pour les plus grandes instances. Ceci peut s'expliquer par le fait qu'un nombre important d'intervalles utilisateur implique un nombre important de variables de coût. Celles-ci contraignent plus fortement le problème, si elles ont une valeur maximale petite. Pour cette raison, un plus grand nombre d'intervalles augmente le nombre de variables de coût dont la valeur maximale est faible. Le problème est alors contraint de façon plus importante, et il est donc plus difficile de trouver une solution.

7.1.1.2 Discussion

Les résultats précédents ont montré qu'il était plus difficile de résoudre des problèmes dans lesquels certains intervalles avaient une valeur maximale du domaine de la variable de coût plus petite que dans d'autres intervalles utilisateur. Dans le but de mesurer au mieux l'impact de chacun des paramètres que nous faisons varier dans les problèmes suivants, nous n'introduisons pas cette diversité. Nous nous plaçons dans le cas le plus favorable, de manière à ce que les modifications de performance en fonction des paramètres que nous ferons varier apparaissent de manière plus évidente. Nous utilisons alors, dans la suite, des intervalles utilisateur dont la variable de coût a pour domaine $[0, 6]$.

De plus, nous nous concentrons à présent sur les instances à 250 et 500 activités. Ces deux classes d'instance présentent l'avantage d'avoir un grand nombre de problèmes résolus dans le temps imparti, rendant les résultats plus exploitables, en comparaison aux instances à 1000 activités.

7.1.1.3 Singularités

Description Nous testons l'ajout d'un petit nombre d'activités singulièrement plus "hautes" ou "longues", que les autres. Pour cela, nous générons des instances de même taille que dans la section précédente, mais nous y injectons 10% ou 20% d'activités de même hauteur et de longueur comprise entre 8 et 12, ou de même longueur, et de hauteur comprise entre 6 et 8. Nous ajustons également la longueur maximale des intervalles, de manière à rester dans le même ordre de densité globale.

L'introduction de singularités peut mener à des comportements différents pour la contrainte SoftCumulative.

Par exemple, dans l'algorithme de Sweep, la notion de partie obligatoire est utilisée. Des activités plus longues peuvent engendrer une partie obligatoire plus rapidement, et ainsi modifier le processus de résolution.

Lorsqu'une activité est significativement plus haute que les autres, sa variable de début est sélectionnée en priorité par notre heuristique. De plus, les activités les plus hautes sont les plus susceptibles de mener rapidement à un dépassement dans un intervalle. Cela peut également influencer les performances.

Enfin, il peut être intéressant d'évaluer si l'homogénéité entre les activités peut influencer la résolution. En effet, il semble intuitif que plus les activités sont semblables, plus elles sont interchangeables dans l'ordonnancement.

La Table 7.2 présente les résultats de ces tests.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10% d'activités plus longues						
250 / 8	48	80	20	0	1.5s	98
250 / 16	24	85	14	1	1.7s	90
250 / 32	12	93	4	3	2.7s	239
500 / 8	96	83	16	1	19.3s	704
500 / 16	48	89	9	2	16.5s	538
500 / 32	24	95	4	1	13.8s	320
20% d'activités plus longues						
250 / 8	56	75	24	1	2.6s	306
250 / 16	28	83	12	5	2.4s	209
250 / 32	14	83	12	5	3.1s	272
500 / 8	112	67	20	13	26.7s	1012
500 / 16	56	72	14	14	29.8s	1207
500 / 32	28	83	11	6	21.3s	695

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10% d'activités plus hautes						
250 / 8	48	82	18	0	1.2s	16
250 / 16	24	90	10	0	1.4s	37
250 / 32	12	98	1	1	2.9s	283
500 / 8	96	87	13	0	9.3s	121
500 / 16	48	91	7	2	9.3s	97
500 / 32	24	97	2	1	13.1s	257
20% d'activités plus hautes						
250 / 8	56	84	16	0	1.6s	84
250 / 16	28	92	7	1	1.7s	92
250 / 32	14	95	4	1	2.8s	251
500 / 8	112	85	14	1	13.3s	291
500 / 16	56	93	4	3	15.2s	430
500 / 32	28	94	4	2	18.4s	557

TABLE 7.2 – Résultats en satisfaction pour des classes d'instance avec singularités sur la longueur des activités (premier tableau), puis sur la hauteur des activités (deuxième tableau). La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonne longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues donne le nombre d'instances pour lesquelles une solution a été trouvée dans le temps imparti, parmi les instances n'ayant pas de solution engendrant un dépassement nul. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues.

Analyse des résultats Si l'on compare ces résultats aux résultats de la Table 7.1, nous pouvons constater que l'introduction de singularités ne modifie pas de manière substantielle les résultats. Deux choses semblent s'équilibrer :

– le pouvoir de filtrage supplémentaire introduit par les singularités,
– l'hétérogénéité dans les activités, qui induit moins de solutions équivalentes en terme de placements.

D'un côté, des activités plus longues ou plus hautes, créent des parties obligatoires plus tôt dans l'arbre de recherche, et mènent plus vite à des dépassements, respectivement. Cela pourrait donc accélérer la résolution. De l'autre, comme nous l'avons évoqué dans la présentation du test, l'homogénéité des activités peut introduire des activités interchangeables car de mêmes dimensions. Introduire des activités différentes, en termes de dimensions, réduit cette interchangeabilité, et il est donc moins aisé de trouver une solution.

Nous pensons que ces deux facteurs mènent à un équilibrage du pouvoir de résolution dans le cas de singularités, et ce n'est donc pas un problème d'introduire un petit nombre d'activités de dimensions significativement plus grandes.

7.1.1.4 Diversité des débuts au plus tôt et des fins au plus tard

Description Nous testons des instances de tailles semblables, avec des activités de tailles semblables, mais telles que les débuts au plus tôt et les fins au plus tard ne soient pas les mêmes pour toutes les activités. Nous introduisons deux types d'instance : avec des débuts au plus tôt situés entre 0 et 10%, puis entre 0 et 20% de l'horizon de temps, et des fins au plus tard comprises entre 90 et 100%, puis entre 80 et 100%, respectivement, de l'horizon de temps. Nous ajustons la longueur maximale des intervalles de manière à garder une densité relative du même ordre, dans l'intervalle de temps où peuvent passer toutes les activités.

Ce test est susceptible d'apporter des informations, là encore, quant au comportement de la contrainte SoftCumulative dans le cas d'instances non symétriques, avec des activités non complètement interchangeables.

La Table 7.3 donne les résultats de ces tests.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
débuts entre 0 et 10% de l'horizon, fins entre 90 et 100%						
250 / 8	44	86	10	4	8.7s	1819
250 / 16	22	82	6	12	11.4s	2382
250 / 32	10	70	3	27	21.6s	4815
500 / 8	88	39	7	54	44.2s	2408
500 / 16	44	39	2	59	53.6s	2985
500 / 32	22	16	0	84	62.5s	3173
débuts entre 0 et 20% de l'horizon, fins entre 80 et 100%						
250 / 8	47	88	5	7	11.7s	2361
250 / 16	24	80	4	16	17.5s	3699
250 / 32	12	67	0	33	22.0s	4437
500 / 8	92	27	6	67	63.8s	3071
500 / 16	46	25	2	73	72.9s	3731
500 / 32	24	15	0	85	75.4s	3820

TABLE 7.3 – Résultats en satisfaction pour des classes d'instance avec introduction de diversité sur les dates de début au plus tôt et de fin au plus tard des activités. La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonnes longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues donne le nombre d'instances pour lesquelles une solution a été trouvée dans le temps imparti, parmi les instances n'ayant pas de solution engendrant un dépassement nul. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues.

Analyse des résultats Les résultats de ces tests montrent qu'il est très difficile de résoudre des instances à 500 activités, en introduisant une diversité dans les dates de début et de fin. Cela peut s'expliquer par le fait que cette diversité introduit nécessairement une augmentation de la densité vers le milieu de l'horizon de temps. Nous avons tenté de limiter cette influence en ajustant quelque peu les longueurs d'intervalles utilisateur, mais nous voulions mettre en évidence ce phénomène : plus le nombre d'activités à placer entre deux points de temps est important, plus le problème est contraint par les capacités locales et les valeurs maximales des domaines des variables de coût. Plus ce problème est contraint, plus trouver une solution peut s'avérer difficile, notamment pour des problèmes de grande taille comme nous traitons ici.

7.1.1.5 Bilan des essais en recherche de solution

Notre contrainte prouve son efficacité sur des problèmes de satisfaction de contraintes. Nous constatons que, pour un nombre relativement important d'activités (jusqu'à 1000), et dans un temps restreint (2 minutes), nous trouvons une première solution. Pour montrer l'efficacité de notre contrainte, nous avons

travaillé sur des instances sur-contraintes, pour ne pas biaiser notre approche. Nous remarquons que la contrainte SoftCumulative a un comportement assez sensible aux modifications des domaines des variables impliquées. En particulier, les variables de coût, comme nous pouvions nous y attendre, jouent un rôle déterminant dans la résolution de problèmes : plus la valeur maximale des coûts (et de l'objectif) est petite, plus il est difficile de trouver une solution. Les variables de coût jouent le rôle d'une "seconde capacité", et contraignent donc le problème de manière similaire. De plus, les variables de début et de fin d'activité jouent sur la densité globale du problème, notamment vers le milieu de l'horizon de temps. La résolution y est également très sensible.

Cependant, notre contrainte gère assez bien les singularités, et les activités de taille supérieure, pour les raisons évoquées auparavant. Le passage à l'échelle est donc assuré, et nous verrons dans la section 7.2 que SoftCumulative se comporte mieux que ses décompositions, pour des problèmes de grande taille.

Nous abordons dans la section suivante les problèmes d'optimisation.

7.1.2 Optimisation

Lorsque l'on introduit un problème sur-contraint, le but premier est de minimiser les violations de contraintes. Nous avons introduit la contrainte SoftCumulative en décrivant une sémantique de violation propre au problème qu'elle représente. Dans le cas de nos expérimentations, le dépassement maximal dans chaque intervalle utilisateur donne la valeur des variables de coût locales, et la somme de ces variables de coût donne l'objectif global.

Pour évaluer la capacité de notre contrainte à chercher une solution optimale, c'est à dire qui minimise la somme des dépassements, nous testons différents types de problèmes.

La limite de temps est cette fois fixée à 5 minutes.

Nous utilisons cette fois l'adaptation de l'algorithme d'Edge-Finding de Kameugne et al., celle de l'algorithme de Sweep, et les différents calculs de bornes présentés dans le chapitre 5.

Nous choisissons d'utiliser l'adaptation de l'algorithme d'Edge-Finding de Kameugne et al. plutôt que l'adaptation de l'algorithme de Vilím car sa structure, plus légère, donne de meilleurs résultats en terme de temps, dans notre approche.

La stratégie de recherche employée est, pour l'ensemble de ces expérimentations, l'heuristique de choix de variables de Choco *DomOverWDegBranchingNew*, qui implémente l'heuristique **dom/wdeg** présentée en section 1.4 et l'heuristique de choix de valeur *IncreasingDomain* qui implémente l'heuristique **minVal**, en version itérative, décrite dans la section 1.4. Ces heuristiques génériques sont efficaces pour résoudre des problèmes d'optimisation.

Dans un premier temps, nous testons la résolution, à l'optimal, de problèmes "purs", dans la section 7.1.2.1. Ces problèmes n'imposent pas de contraintes entre les variables de coût.

Le chapitre 4 a montré que, dans de nombreux cas pratiques, nous pouvions utiliser la contrainte SoftCumulative, en l'associant avec des contraintes additionnelles sur les variables de coût, pour modéliser des problèmes réels. Nous testons la résolution optimale sur ces modèles en section 7.1.2.2.

7.1.2.1 Variation du nombre d'activités et d'intervalles

Description Une solution *optimale* est souvent recherchée par l'utilisateur qui, face à un problème sur-contraint, souhaite trouver la meilleure solution en pratique, c'est à dire, celle qui se rapproche le plus d'une solution parfaite.

Nous testons le comportement de notre contrainte pour des problèmes d'optimisation, sur des problèmes à 10, 20, 30, 60 et 120 activités, pour 4, 8 et 16 intervalles utilisateur. Nous cherchons à minimiser la valeur de la variable objectif. Les activités ont des longueurs comprises entre 1 et 4 et des hauteurs comprises entre 1 et 3. Les intervalles utilisateur ont une capacité locale aléatoirement choisie entre 3 et

6, et les variables de coût ont un domaine dont la valeur minimale est 0 et la valeur maximale est choisie aléatoirement parmi l'ensemble $\{1, 2, 3, 4, 5, 6\}$.

La Table 7.4 contient les résultats de ces tests.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues à l'optimal ($obj = 0$)	#solution trouvée non optimale	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10 / 4	3	80 (14)	0	20	0	1.3s	38444
20 / 4	6	80 (12)	2	16	2	7.3s	209092
20 / 8	3	85 (28)	4	0	11	9.6s	250931
30 / 4	8	81 (6)	2	13	4	9.0s	111948
30 / 8	4	70 (0)	8	13	9	12.5s	238334
60 / 4	17	66 (0)	5	26	3	4.4s	19337
60 / 8	9	74 (3)	12	3	11	9.0s	36872
60 / 16	5	37 (10)	43	0	20	37.3s	67216
120 / 4	36	87 (3)	1	10	2	2.6s	3695
120 / 8	18	79 (0)	11	6	4	12.1s	14925
120 / 16	9	11 (0)	80	0	9	127.7s	58276

TABLE 7.4 – Résultats en optimisation pour des classes d'instance pures. La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonne longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues à l'optimal ($obj = 0$) donne le nombre d'instances pour lesquelles une solution a été trouvée, et prouvée optimale dans le temps imparti, en mettant entre parenthèses le nombre d'instances dont la valeur optimale de l'objectif est 0. La colonne #solution trouvée non optimale donne le nombre d'instances pour lesquelles une solution a été trouvée, mais elle n'est pas optimale, ou n'a pas été prouvée optimale. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues à l'optimal, avec objectif différent de 0. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues à l'optimal, avec un objectif différent de 0.

Analyse des résultats Nous constatons que notre contrainte est capable de trouver des valeurs optimales (et les prouver) pour la plupart des instances testées, sauf pour les instances dans lesquels sont impliqués un grand nombre d'intervalles (ici, les classes d'instance à 16 intervalles). L'objectif prend la valeur de la somme des coûts dans les intervalles utilisateur. Plus le nombre de variables de coût est grand, plus il est difficile de trouver et prouver la solution optimale.

La taille des instances testées montre que notre contrainte réagit plutôt bien, car trouver un optimal dans les problèmes cumulatifs n'est pas chose aisée. A titre d'exemple, nous pouvons évoquer la PSPLib [45], qui est une des librairies bien connues de problèmes cumulatifs. Dans celle-ci, comme dans la plupart des problèmes cumulatifs classique, l'objectif est de trouver la date de fin au plus tard minimal de l'ordonnancement. Il existe, dans cette librairie, des instances à 60 activités dont l'optimal n'a pas encore été prouvé. Ce problème est certes différent du nôtre, mais nous ne disposons malheureusement pas d'autre élément de comparaison.

7.1.2.2 Optimisation avec contraintes additionnelles

Nous l'avons vu dans le chapitre 4, de nombreux problèmes réels usuels impliquent des contraintes additionnelles sur les variables de coût.

Nous reprenons les cas usuels décrits dans le chapitre 4, et les modèles associés, et réalisons des tests pour chacun d'entre eux.

Répartition équitable des dépassements

Description La section 4.3.2.1 présente un premier modèle pour une situation réelle où les dépassements doivent être répartis de façon équitable.

Nous souhaitons tester le comportement de ce modèle de manière expérimentale.

Nous imposons que, pour chaque séquence de 3 variables de coût consécutives, au moins une variable de coût prenne la valeur 0. Si le nombre p de variables de coût n'est pas un multiple de 3, dans la dernière séquence de variables de coût consécutives, au moins une variable de coût doit prendre la valeur 0 également.

Cette règle contraint fortement le problème et il est alors intéressant d'évaluer l'impact d'une telle contrainte.

Nous cherchons, là encore, à minimiser la valeur de la variable objectif. Pour cela, nous reprenons les instances introduites dans la section 7.1.2.1 pour l'optimisation.

La Table 7.5 donne les résultats de ces tests.

Analyse des résultats Nous observons que le nombre de problèmes résolus à l'optimal est important lorsque l'on impose ces contraintes de répartition.

Nous constatons également que, lorsque l'on impose les contraintes de répartition, un nombre plus important de problèmes sont infaisables (ou du moins, prouvés comme tels), ce qui est relativement intuitif.

La contrainte SoftCumulative réagit bien aux contraintes de répartition : imposer ces contraintes réduit dans de nombreuses classes d'instances le temps moyen de résolution, et le nombre de backtracks moyen par instance.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues à l'optimal ($obj = 0$)	#solution trouvée non optimale	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10 / 4	3	59 (14)	0	41	0	0.3s	5354
20 / 4	6	62 (12)	0	37	1	1.4s	33457
20 / 8	3	86 (28)	3	1	10	9.2s	236949
30 / 4	8	54 (6)	0	43	3	3.8s	56900
30 / 8	4	60 (3)	6	26	8	15.0 s	270851
60 / 4	17	42 (0)	2	55	1	10.2s	56365
60 / 8	9	72 (3)	9	9	10	6.3s	27425
60 / 16	5	50 (10)	29	0	21	54.6s	175153
120 / 4	36	60 (3)	0	40	0	2.1s	2844
120 / 8	18	70 (0)	6	19	5	7.0s	11010
120 / 16	9	17 (0)	71	2	10	83.1s	79013

TABLE 7.5 – Résultats en optimisation pour des classes d'instance avec contraintes de répartition entre les variables de coût. La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonnes longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues à l'optimal ($obj = 0$) donne le nombre d'instances pour lesquelles une solution a été trouvée, et prouvée optimale dans le temps imparti, en mettant entre parenthèses le nombre d'instances dont la valeur optimale de l'objectif est 0. La colonne #solution trouvée non optimale donne le nombre d'instances pour lesquelles une solution a été trouvée, mais elle n'est pas optimale, ou n'a pas été prouvée optimale. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues à l'optimal, avec objectif différent de 0. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues à l'optimal, avec objectif différent de 0.

Lissage des variations de coût

Description Nous l'avons vu dans la section 4.3.2.2, limiter les variations de coût entre deux intervalles consécutifs a un intérêt pratique important.

Pour mesurer les conséquences de l'ajout d'une telle contrainte, nous appliquons des contraintes de différence entre les variables de coût consécutives. Nous reprenons pour cela les mêmes instances que dans la section 7.1.2.1, et nous y appliquons des contraintes de différence telles que pour chaque paire de variables de coût consécutives $cost_i$ et $cost_{i+1}$, nous ayons $|cost_i - cost_{i+1}| \leq 1$.

Nous conservons les mêmes instances de manière à montrer l'incidence de ces contraintes.

La Table 7.6 donne les résultats de ces tests.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues à l'optimal ($obj = 0$)	#solution trouvée non optimale	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10 / 4	3	71 (14)	0	29	0	0.2s	4154
20 / 4	6	73 (12)	1	26	0	7.6s	209580
20 / 8	3	87 (28)	3	1	9	13.6s	349022
30 / 4	8	66 (6)	2	30	2	0.8s	10824
30 / 8	4	51 (3)	6	32	11	7.7s	140925
60 / 4	17	53 (0)	0	43	4	1.9s	13875
60 / 8	9	79 (3)	6	11	4	7.5s	44370
60 / 16	5	69 (10)	16	0	15	37.7s	112673
120 / 4	36	70 (3)	1	29	0	1.5s	2406
120 / 8	18	75 (0)	3	21	1	3.3s	5370
120 / 16	9	64 (0)	20	6	10	111.7s	64486

Table 7.6 – Résultats en optimisation pour des classes d'instance avec contraintes de différence entre les variables de coût. La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonne longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues à l'optimal ($obj = 0$) donne le nombre d'instances pour lesquelles une solution a été trouvée, et prouvée optimale dans le temps imparti, en mettant entre parenthèses le nombre d'instances dont la valeur optimale de l'objectif est 0. La colonne #solution trouvée non optimale donne le nombre d'instances pour lesquelles une solution a été trouvée, mais elle n'est pas optimale, ou n'a pas été prouvée optimale. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues à l'optimal, avec objectif différent de 0. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues à l'optimal, avec objectif différent de 0.

Analyse des résultats Nous pouvons observer que le nombre de problèmes résolus à l'optimal est significativement plus important lorsque l'on impose ces contraintes de différence.

Nous constatons également que, lorsque l'on impose les contraintes de différence, un nombre plus important de problèmes sont infaisables (ou du moins, prouvés comme tels), ce qui est relativement intuitif.

La contrainte SoftCumulative réagit bien aux contraintes de différence : imposer ces contraintes réduit le temps de résolution, et le nombre de backtracks moyen par instance.

Nous rappelons que nous avons utilisé une procédure de filtrage à partir des valeurs minimales dans les domaines des variables de coût, décrite dans la section 5.1.4.

Concentration des coûts

Description La section 4.3.2.3 présente un premier modèle pour une situation réelle où les dépassements doivent être concentrés au maximum sur des séquences de variables de coût consécutives.

Nous utilisons pour cela la contrainte Focus [63], décrite en section 4.3.2.3.

Nous souhaitons tester le comportement de ce modèle de manière expérimentale.

Nous cherchons, là encore, à minimiser la valeur de la variable objectif. Pour cela, nous reprenons les instances introduites dans la section 7.1.2.1 pour l'optimisation :

- pour les instances à 4 intervalles, nous imposons que l'on ait au plus une séquences d'au plus trois variables de coût strictement positifs
- pour les instances à 8 intervalles, nous imposons que l'on ait au plus deux séquences d'au plus trois variables de coût strictement positifs
- pour les instances à 16 intervalles, nous imposons que l'on ait au plus trois séquences d'au plus trois variables de coût strictement positifs

La Table 7.7 donne les résultats de ces tests.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues à l'optimal ($obj = 0$)	#solution trouvée non optimale	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10 / 4	3	72 (14)	0	28	0	0.5s	7564
20 / 4	6	73 (12)	1	23	3	2.8s	72511
20 / 8	3	86 (28)	5	0	9	12.3s	321062
30 / 4	8	73 (6)	1	23	3	8.3s	102872
30 / 8	4	63 (3)	8	24	5	15.3s	275345
60 / 4	17	56 (0)	4	39	1	5.2s	22735
60 / 8	9	72 (3)	11	6	11	9.9s	44960
60 / 16	5	62 (10)	22	0	16	62.1s	232648
120 / 4	36	78 (3)	0	21	1	2.1s	3138
120 / 8	18	76 (0)	7	10	7	6.7s	10289
120 / 16	9	26 (0)	60	7	7	128.0s	126225

Table 7.7 – Résultats en optimisation pour des classes d'instance avec contraintes de concentration sur les variables de coût. La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonne longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues à l'optimal ($obj = 0$) donne le nombre d'instances pour lesquelles une solution a été trouvée, et prouvée optimale dans le temps imparti, en mettant entre parenthèses le nombre d'instances dont la valeur optimale de l'objectif est 0. La colonne #solution trouvée non optimale donne le nombre d'instances pour lesquelles une solution a été trouvée, mais elle n'est pas optimale, ou n'a pas été prouvée optimale. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues à l'optimal, avec objectif différent de 0. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues à l'optimal, avec objectif différent de 0.

Analyse des résultats Nous observons que le nombre de problèmes résolus à l'optimal est moins important que dans les autres cas usuels présentés. Cependant, de nombreuses classes de problèmes donnent tout de même de meilleurs résultats que la recherche d'optimal dans les problèmes purs.

Cependant, notre approche donne tout de même de bon résultats, et de nombreux problèmes sont résolus à l'optimal. Lorsque ce n'est pas le cas, notamment dans les cas les plus contraints : ceux où il y a 16 intervalles, une solution, non prouvée optimale est, dans la plupart des cas, trouvée.

Nous constatons également que, lorsque l'on impose les contraintes de concentration des coût, un nombre plus important de problèmes sont infaisables (ou du moins, prouvés comme tels), ce qui est relativement intuitif.

7.1.2.3 Bilan des essais en optimisation

Notre contrainte prouve son efficacité sur des problèmes d'optimisation avec et sans contraintes additionnelles sur les variables de coût.

Nous observons que notre contrainte réussit à trouver des solutions optimales pour des problèmes purs, jusqu'à 125 activités, en moins de 5 minutes, et pour un nombre d'intervalles allant jusqu'à 16. Ces résultats sont très encourageants.

L'ajout de contraintes additionnelles sur les variables de coût est également plutôt bien géré par notre contrainte, et il est intéressant de constater que cela peut, dans certains cas, accélérer la recherche.

Deux facteurs s'opposent lorsque nous ajoutons des contraintes additionnelles sur les variables de coût :

- cela contraint le problème de manière plus importante, et il est plus difficile de trouver une solution, et de prouver l'optimal,
- une propagation supplémentaire peut être induite par l'ajout de contraintes additionnelles.

Il semble que ce gain en propagation soit intéressant dans certains cas, puisque de nombreux problèmes sont résolus plus rapidement lorsque l'on y ajoute des contraintes additionnelles sur les variables de coût.

Dans d'autres, le fait que les problèmes soient plus contraints induit un temps de résolution plus important.

7.1.3 Adaptation à la PSPLib

Description Le problème cumulatif relaxé, tel que nous le présentons, n'a pas été traité avant notre thèse, il n'existe donc pas de librairie de problèmes qui y soit dédiée.

Cependant, afin d'évaluer le comportement de notre contrainte sur une librairie existante, nous avons choisi d'adapter deux classes d'instances de la PSPLib.

Nous évaluons le comportement de la contrainte SoftCumulative sur une adaptation des instances de taille 30 activités et 60 activités de la PSPLib. Nous décrivons le type d'instances contenues dans la PSPLib :

- les instances de la PSPLib sont des instances cumulatives qui s'exécutent sur plusieurs machines en parallèle,
- chaque activité a une longueur fixe donnée,
- chaque activité a une consommation de ressource propre à chaque machine,
- lorsqu'une activité est exécutée, elle l'est simultanément sur chaque machine, et elle consomme, sur chaque machine la quantité de ressource associée, pendant le temps d'exécution de l'activité,
- chaque machine a sa capacité propre,
- des relations de précédence de type "commence après la fin" sont imposées entre certaines activités,
- une date de fin au plus tard de l'ordonnancement est donnée.

Ces problèmes peuvent être représentés par une conjonction de contraintes Cumulative (une Cumulative par machine), et de contraintes type "inférieur ou égal" pour modéliser les contraintes de précédence.

Ces instances sont dédiées à l'optimisation : le but est de minimiser la date de fin de la dernière activité (le makespan) :

- pour les instances à 30 activités, le makespan optimal est connu,
- pour les instances à 60 activités, quand le makespan optimal n'est pas connu, une borne inférieure et une borne supérieure sont fournies.

Nous proposons d'adapter ces instances au cas SoftCumulative. Pour cela :

- nous fixons l'horizon de temps à la valeur optimale du makespan ou, à défaut, à la borne supérieure fournie,
- nous gardons les contraintes de précédence,
- nous réduisons de 20% la capacité de chaque machine,
- pour chaque machine, nous posons une contrainte SoftCumulative prenant en paramètres l'ensemble des activités, avec la hauteur correspondante pour cette machine,
- nous posons $costC = max$ et $objC = sum$ pour chaque contrainte SoftCumulative,
- chaque contrainte SoftCumulative a un intervalle utilisateur, et une unique variable de coût dont le domaine est entre 0 à 20% de la capacité initiale de la machine,
- nous posons une variable objectif globale qui prend la valeur maximale des variables objectif des contrainte SoftCumulative posées,
- nous minimisons cette variable.

Ce test permet d'évaluer plusieurs choses. Il permet dans un premier temps d'évaluer le comportement d'un problème avec conjonctions de contraintes SoftCumulative. Il permet également d'obtenir des informations sur les instances de la PSPLib. Enfin, nous évaluons la conjonction des contraintes SoftCumulative avec des contraintes additionnelles de précédence. Jusqu'ici, les contraintes additionnelles posées sur la SoftCumulative l'étaient sur les variables de coût. Ici, elles sont cette fois également imposées sur les activités.

Signalons que nous n'avons pas défini d'heuristique particulière basée sur les précédences par exemple, car nous voulions évaluer notre adaptation dans un cadre générique. Nous utilisons donc de nouveau l'heuristique de choix de variables de Choco *DomOverWDegBranchingNew* (plus petit domaine divisé par le degré pondéré de la variable) et l'heuristique de choix de valeur *IncreasingDomain* (de la plus petite valeur à la plus grande).

Les classes d'instances 30 et 60 de la PSPLib sont constitués chacune de 480 instances, réparties en 48 classes de 10 instances. Nous fixons la limite de temps de résolution à 5 minutes.

La Table 7.8 présente les résultats de ces tests.

# activités	# résolues à l'optimal ($obj = 0$)	#solution trouvée non optimale	# prouvées infaisables	# time out	temps moyen ($obj = 0$)	# backtracks moyen ($obj = 0$)
30	359(70)	22	0	99	12.6s(4.4s)	7101(1728)
60	253 (140)	80	0	147	13.6s (1.7s)	1773(110)

TABLE 7.8 – Résultats en pour les problèmes de la PSPLib de taille 30 et 60, adaptés à la contrainte SOFTCUMULATIVE. La colonne # activités donnele nombre d'activités de la classe de problèmes. La colonne # résolues à l'optimal ($obj = 0$) donne le nombre d'instances résolues à l'optimal dans le temps imparti, et, parmi elles, le nombre d'instances dont l'objectif optimal est égal à 0. La colonne # solution trouvée non optimale donne le nombre d'instance pour lesquelles une solution a été trouvée dans le temps imparti, mais qui n'est pas optimale, ou qui n'a pas été prouvée comme tel. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen ($obj = 0$) donne le temps moyen de résolution parmi les instances résolues à l'optimal, et distingue de celui-ci le temps moyen pour les instances résolues à l'optimal quand l'objectif est égal à 0. La colonne # backtracks moyen ($obj = 0$) donne le nombre moyen de retours arrière dans la recherche pour trouver la solution optimale, pour les instances résolues à l'optimal et distingue ce nombre moyen pour les instances résolues à l'optimal dont l'objectif est égal à 0.

Analyse des résultats Les résultats des tests sur la PSPLib semblent démontrer que la conjonction de contraintes SoftCumulative fonctionne en optimisation.

Le temps de résolution est assez bas.

Ces résultats doivent être nuancés par le fait que la contrainte SoftCumulative est ici utilisée dans le cas particulier où elle n'utilise qu'un intervalle. Cependant, ce type d'utilisation sur plusieurs machines peut être une perspective intéressante de travail.

Il est intéressant de noter que de nombreuses instances ont un optimal à 0. Pour ces instances, ceci signifie que :

- le makespan optimal (ou sa borne supérieure, le cas échéant) resterait le même en baissant de 20% la capacité de chaque machine , avec les mêmes activités et les mêmes relations de précédence,
- les instances en question peuvent être considérées comme "lâches" : les capacités des machines contraignent moins ces instances que les relations de précédence.

Pour les autres instances résolues à l'optimal, les capacités peuvent également être abaissées en fonction de l'objectif optimal.

Le lecteur intéressé par les différentes approches possibles de l'ordonnancement de projet pourra se référer par exemple à [2].

7.2 Évaluation des décompositions

Lorsque l'on traite des problèmes d'ordonnancement exclusivement par des techniques de résolution exactes à base de contraintes, les décompositions de contraintes globales rivalisent en général assez peu avec les algorithmes dédiés, du moins pour des instances difficiles.

En revanche, ces décompositions peuvent s'avérer très utiles pour rechercher une première solution, voire une "bonne solution" selon les instances.

Dans ce contexte, l'objectif principal de nos expérimentations est de déterminer s'il est possible d'utiliser des décompositions de la contrainte SoftCumulative pour résoudre des problèmes concrets de combinatoire moyenne, comme cela peut être le cas avec les décompositions de la contrainte Cumulative. En d'autres termes, la question principale était de déterminer si l'adaptation au cas de la contrainte SoftCumulative engendrait ou non une perte d'efficacité dans le processus de résolution.

Nous avons également évalué le comportement des décomposition dans le cas de problèmes d'optimisation, et proposé, suite à l'analyse des résultats, quelques perspectives à notre travail.

Les tests sont faits sur des instances à 100 et à 250 activités, et à 8, 16 ou 32 intervalles utilisateurs pour la recherche d'une solution, et sur des problèmes de petite taille pour la recherche d'une solution optimale.

7.2.1 Evaluation relative des décompositions

Nous comparons le comportement des décompositions de la contrainte SoftCumulative avec celui des décompositions équivalentes pour la contrainte Cumulative. Les instances de problèmes sont générées aléatoirement en fonction d'un nombre d'activités et du nombre d'intervalle utilisateur considérés.

Pour chaque classe (nombre d'activités dans le cas de Cumulative et nombre d'activités / nombre d'intervalles dans le cas de SoftCumulative), 100 instances aléatoires ont été testées, et les résultats fournissent la moyenne des temps de résolution et du nombre de backtracks.

La limite de temps de résolution a été fixée à deux minutes : au delà, si aucune solution n'a été trouvée alors on considère que l'instance est non résolue.

La stratégie de recherche employée est, pour l'ensemble des expérimentations sur les décompositions, l'heuristique de choix de variables de Choco *DomOverWDegBranchingNew* (plus petit domaine divisé par

le degré pondéré de la variable) et l'heuristique de choix de valeur *IncreasingDomain* (de la plus petite valeur à la plus grande). Pour comparer aux algorithmes ad-hoc de la contrainte globale SoftCumulative, nous avons utilisé sur celle-ci, en satisfaction, la stratégie de recherche définie dans le chapitre 5.

Les durées des activités varient entre 1 et 4.

Dans le cas des décompositions de la contrainte SoftCumulative, l'horizon de temps est fixé. Il est égal à la somme des longueurs des intervalles dont la valeur est choisie aléatoirement entre 1 et la valeur maximale de la longueur des intervalles.

Pour chaque intervalle utilisateur, la capacité locale que l'on peut dépasser est fixée aléatoirement entre 3 et 6. La valeur maximale du dépassement est fixée à 6. Nous voulons en effet mettre en évidence l'incidence directe de la décomposition, et choisissons donc un cas favorable. Autoriser un dépassement important est un cas favorable. Ces valeurs contraignent le problème, y compris dans le cas où l'on cherche uniquement une première solution satisfaisant les contraintes, sans optimisation.

Notons que les instances générées ne sont pas exactement les mêmes que dans la section 7.1.1, ce qui n'impacte en rien les résultats.

7.2.1.1 Décomposition temps-ressource

Description Nous comparons dans cette section l'efficacité de la décomposition temps-ressource dans le cas de la contrainte Cumulative (Table 7.9) et dans le cas de la contrainte SoftCumulative (Table 7.10).

# activités	horizon minimal	horizon maximal	# résolues	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
100	50	80	98	0	2	0.4s	23
250	130	160	96	0	4	4.6s	111

Table 7.9 – Résultats en satisfaction de la décomposition temps-ressource de Cumulative. La colonne # activités donne le nombre d'activités de la classe de problèmes. Les colonnes horizon minimal et horizon maximal donnent respectivement la valeur minimale, puis maximale de l'horizon de temps choisi aléatoirement pour l'ordonnancement. La colonne # résolues donne le nombre d'instances pour lesquelles une solution a été trouvée dans le temps imparti. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10/4	4	90	10	0	0.0s	14
20/4	8	77	8	15	0.0s	9
30/4	12	76	2	22	0.3s	6594
100/8	16	85	0	15	0.7s	106
100/16	8	93	0	7	0.6s	51
100/32	4	97	0	3	0.7s	45
250/8	40	90	0	10	8.3s	134
250/16	20	97	0	3	7.5s	136
250/32	8	96	0	4	8.0s	118

TABLE 7.10 – Résultats en satisfaction de la décomposition temps-ressource de SOFTCUMULATIVE. La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonne longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues donne le nombre d'instances pour lesquelles une solution a été trouvée dans le temps imparti, parmi les instances n'ayant pas de solution engendrant un dépassement nul. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues.

Analyse des résultats Nous pouvons observer que certaines instances n'admettent pas de solution, même lorsqu'on tolère des dépassements de capacité. Leur nombre est indiqué dans la colonne "Infaisables" de la table 7.10. Nous remarquons aussi que le nombre d'instances non résolues n'augmente pas proportionnellement à la taille des instances, ce qui montre que trouver une première solution, en accord avec les valeurs maximales autorisées pour les variables de coûts, n'est pas plus difficile dans les cas d'instances de 100 activités ou plus.

Les résultats démontrent que l'ordre de grandeur des instances pour la recherche d'une première solution reste le même avec notre adaptation de la décomposition temps-ressource de la contrainte Cumulative au cas de la contrainte SoftCumulative.

7.2.1.2 Décomposition activité-ressource

Description Nous comparons dans cette section l'efficacité de la décomposition activité-ressource dans le cas de la contrainte Cumulative (Table 7.11) et dans le cas de la contrainte SoftCumulative (Table 7.12).

Contrairement au cas de la décomposition temps-ressource, nous ne donnons pas de résultats d'instances ayant 250 activités car l'espace mémoire requis par cette décomposition, aussi bien dans le cas classique que dans le cas de problèmes cumulatifs avec dépassements, s'est avéré trop grand pour les capacités de la machine. Nous n'avons pas pu lancer la résolution.

# activités	horizon minimal	horizon maximal	# résolues	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
100	50	80	97	0	3	1.2s	22

Table 7.11 – Les résultats en satisfaction de la décomposition activité-ressource de Cumulative. La colonne # activités donne le nombre d'activités de la classe de problèmes. Les colonnes horizon minimal et horizon maximal donnent respectivement la valeur minimale, puis maximale de l'horizon de temps choisi aléatoirement pour l'ordonnancement. La colonne # résolues donne le nombre d'instances pour lesquelles une solution a été trouvée dans le temps imparti. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10/4	4	90	10	0	0.0s	11
20/4	8	77	6	17	0.2s	142
30/4	12	9	0	1	0.1s	14
100/8	16	83	0	17	6.9s	42
100/16	8	93	0	7	12.1s	47
100/32	4	95	0	5	22.5s	40

TABLE 7.12 – Résultats en satisfaction de la décomposition activité-ressource de SOFTCUMULATIVE. La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonne longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues donne le nombre d'instances pour lesquelles une solution a été trouvée dans le temps imparti, parmi les instances n'ayant pas de solution engendrant un dépassement nul. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues.

Analyse des résultats Les résultats démontrent à nouveau que l'ordre de grandeur des instances pour la recherche d'une première solution reste le même avec notre adaptation de la décomposition activité-ressource de la contrainte Cumulative au cas de la contrainte SoftCumulative.

7.2.1.3 Décomposition linéaire

Description Nous fournissons pour la décomposition linéaire uniquement les résultats pour le cas de la décomposition de SoftCumulative. Les résultats obtenus dans le cas de la décomposition de Cumulative sont équivalents.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10/4	4	18	0	82	26.8s	196179
20/4	8	0	0	100	-	-

Table 7.13 – Résultats en satisfaction de la décomposition linéaire de SoftCumulative. La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonne longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues donne le nombre d'instances pour lesquelles une solution a été trouvée dans le temps imparti, parmi les instances n'ayant pas de solution engendrant un dépassement nul. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues.

Analyse des résultats Les résultats présentés dans la table 7.13 montrent qu'avec la décomposition linéaire nous ne parvenons qu'à résoudre des instances de taille 10. Ce résultat est décevant et notre analyse de la trace semble indiquer que la perte de propagation est due à la contrainte de tri sortedness, pour laquelle il n'existe pas, dans Choco, d'algorithme complet, mais seulement une décomposition effectuant un filtrage partiel aux bornes. De plus, le solveur branche parfois sur les variables représentant les indices de la permutation, ce qui ne semble pas très opportun.

Perspectives L'amélioration de l'algorithme de filtrage de la contrainte sortedness ainsi que l'élaboration d'une stratégie de recherche dédiée semblent indispensables pour utiliser notre décomposition linéaire. Cependant, cela serait contraire à l'idée selon laquelle notre décomposition est générique et adaptable sur n'importe quel solveur de contraintes. Pour cela, nous n'avons pas creusé cette piste plus en détail. Ce résultat montre les limites des travaux théoriques sur les décompositions. Notamment, s'il est nécessaire d'implémenter une stratégie de recherche dédiée à la décomposition, alors la caractéristique de "simplicité d'implémentation" de la décomposition semble perdue. Ces questions constituent des perspectives intéressantes à notre travail.

7.2.2 Comparaison aux algorithmes ad-hoc

Description Nous évaluons ici l'utilisation des décompositions pour la recherche de solutions optimales.

La génération aléatoire des instances est réalisée avec les mêmes paramètres que dans la section précédente. En revanche, la limite du temps d'exécution est augmentée à cinq minutes (au lieu de deux minutes).

Dans le cas de la contrainte Cumulative, sans une stratégie de recherche dédiée il n'est pas possible de prouver en moins de cinq minutes l'optimalité des solutions pour la plupart des instances générées aléatoirement, à l'aide des décompositions temps-ressource et activité-ressource. Nous présentons ici des tests similaires dans le cas des décompositions de SoftCumulative.

Nous comparons les résultats des décompositions (Tables 7.16 et 7.15) à ceux de notre contrainte globale SoftCumulative, avec les algorithmes de Sweep et d'Edge-finding de Kameugne et al. (Table 7.14).

Notons que les instances générées ne sont pas exactement les mêmes que dans la section 7.1.2, ce qui n'impacte en rien les résultats.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues à l'optimal ($obj = 0$)	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10/4	4	80(14)	20	0	2.0s	38646
20/4	8	79(12)	16	5	6.4s	136525

Table 7.14 – Résultats en optimisation de la contrainte SoftCumulative. La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonnes longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues à l'optimal ($obj = 0$) donne le nombre d'instances pour lesquelles une solution a été trouvée, et prouvée optimale dans le temps imparti, en mettant entre parenthèses le nombre d'instances dont la valeur optimale de l'objectif est 0. La colonne #solution trouvée non optimale donne le nombre d'instances pour lesquelles une solution a été trouvée, mais elle n'est pas optimale, ou n'a pas été prouvée optimale. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues à l'optimal, avec objectif différent de 0. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues à l'optimal, avec objectif différent de 0.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues à l'optimal ($obj = 0$)	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10/4	4	67(13)	13	20	55.7s	1025379
20/4	8	3(3)	1	96	-	-

TABLE 7.15 – Résultats en optimisation de la décomposition temps-ressource de SOFTCUMULATIVE. La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonne longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues à l'optimal ($obj = 0$) donne le nombre d'instances pour lesquelles une solution a été trouvée, et prouvée optimale dans le temps imparti, en mettant entre parenthèses le nombre d'instances dont la valeur optimale de l'objectif est 0. La colonne #solution trouvée non optimale donne le nombre d'instances pour lesquelles une solution a été trouvée, mais elle n'est pas optimale, ou n'a pas été prouvée optimale. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues à l'optimal, avec objectif différent de 0. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues à l'optimal, avec objectif différent de 0.

# activités / # intervalles utilisateur	longueur max. des intervalles utilisateur	# résolues à l'optimal ($obj = 0$)	# prouvées infaisables	# time out	temps moyen	# backtracks moyen
10/4	4	47 (13)	20	33	77.8s	158662
20/4	8	3 (3)	0	97	-	-

TABLE 7.16 – Résultats en optimisation de la décomposition activité-ressource de SOFTCUMULATIVE. La colonne # activités / # intervalles utilisateur donne respectivement le nombre d'activités de la classe de problèmes, et le nombre d'intervalles utilisateur. La colonne longueur max. des intervalles utilisateur donne la longueur maximale des intervalles utilisateur dans le problème. La colonne # résolues à l'optimal ($obj = 0$) donne le nombre d'instances pour lesquelles une solution a été trouvée, et prouvée optimale dans le temps imparti, en mettant entre parenthèses le nombre d'instances dont la valeur optimale de l'objectif est 0. La colonne #solution trouvée non optimale donne le nombre d'instances pour lesquelles une solution a été trouvée, mais elle n'est pas optimale, ou n'a pas été prouvée optimale. La colonne # prouvées infaisables donne le nombre de problèmes pour lesquels il a été prouvé, dans le temps imparti, qu'ils n'avaient pas de solution. La colonne # time out représente le nombre d'instances pour lesquelles aucune solution, ou preuve d'infaisabilité n'a été donnée dans le temps imparti. La colonne temps moyen donne le temps moyen de résolutions parmi les instances résolues à l'optimal, avec objectif différent de 0. La colonne # backtracks moyen donne le nombre moyen de retours arrière dans la recherche pour trouver une solution, pour les instances résolues à l'optimal, avec objectif différent de 0.

Analyse des résultats Les résultats montrent que les décompositions de la contrainte SoftCumulative ne se montrent pas plus performantes que celles de la contrainte Cumulative pour prouver l'optimalité des solutions. Cela s'explique aisément par le fait qu'une conjonction avec des contraintes primitives n'est pas équivalente, en terme de puissance de filtrage, aux algorithmes dédiés.

Néanmoins, il est intéressant, en pratique, d'avoir une idée de la distance à laquelle on se situe, au bout de cinq minutes, par rapport à la solution optimale. Pour illustrer ce point, nous présentons trente instances à 20 activités et 4 intervalles, générées avec des "seeds" successifs. Pour chaque instance, la solution optimale est fournie par la contrainte SoftCumulative. La table 7.17 indique les différentes valeurs d'objectif trouvées, selon le type de décomposition. Nous indiquons le fait qu'une décomposition n'ait pas été en mesure de fournir une solution par le symbole '-'.

Instances	T-R	A-R	SOFT
instance1	12	12	9
instance2	7	4	4
instance3	-	-	13
instance4	0	0	0
instance5	8	7	7
instance6	7	3	2
instance7	2	1	0
instance8	5	5	3
instance9	12	11	11
instance10	8	7	-
instance11	9	7	7
instance12	11	12	7
instance13	9	8	6
instance14	9	8	8
instance15	4	3	2
instance16	5	3	3
instance17	5	4	3
instance18	10	10	2
instance19	9	5	2
instance20	10	8	7
instance21	4	3	2
instance22	6	4	1
instance23	10	7	4
instance24	14	13	11
instance25	12	10	9
instance26	8	7	7
instance27	12	11	9
instance28	10	9	8
instance29	2	1	0
instance30	5	3	1

TABLE 7.17 – Les valeurs de l'objectif global obtenues pour la résolution de l'instance 20/4 des décompositions temps-ressource et activité-ressource et de la contrainte SOFTCUMULATIVE. La colonne "Instances" indique des instances de la classe 20/4. La colonne T-R indique les résultats obtenus par la décomposition temps-ressource de la contrainte SOFTCUMULATIVE. La colonne A-R indique les résultats obtenus par la décomposition activité-ressource de la contrainte SOFTCUMULATIVE. La colonne SOFT indique les résultats optimaux obtenus par la contrainte SOFTCUMULATIVE.

Perspectives Une amélioration du processus de résolution pourrait être obtenue en ajoutant des contraintes *implicites* aux décompositions. Une contrainte implicite est une contrainte qui est redondante du point de vue de la modélisation (le problème peut être modélisé sans cette contrainte), mais dont la présence au sein d'un modèle apporte un filtrage supplémentaire. Dans ce cadre, une piste peut être d'étudier dans quelle mesure il serait possible de concevoir une décomposition qui hybride les décompositions temps-ressource et temps-activité.

Chapitre 8

Bilan et perspectives

Dans de nombreux cas pratiques, en ordonnancement cumulatif, la date de fin du projet est fixée, et des dépassements peuvent intervenir. La contrainte Cumulative classique ne permet pas de prendre en compte ces dépassements.

Nous avons défini une manière de relaxer cette contrainte en introduisant la contrainte globale SoftCumulative. Nous l'avons définie de manière à ce qu'elle ait un pouvoir expressif important.

Pour cela, nous nous sommes placés dans le cas où l'horizon de temps est découpé en plusieurs intervalles (les *intervalles utilisateur*) de capacités potentiellement différentes. Nous avons autorisé dans une certaine mesure des dépassements de ces capacités. Nous avons alors introduit des variables de coût et défini différentes mesures de violation, de manière à quantifier les dépassements de capacité.

Nous avons mis en lumière trois cas pratiques dans lesquels des contraintes sur les dépassements pouvaient intervenir. Nous avons décrit une application dans laquelle une répartition équitable des valeurs des variables de coûts était nécessaire entre des intervalles utilisateur consécutifs. Nous avons ensuite montré que des applications pouvaient imposer un lissage des valeurs des variables de coût. Une application peut en effet imposer qu'il n'y ait pas ou peu de grandes variations de valeur entre deux variables de coût consécutives. Enfin, nous avons présenté un cas pratique dans lequel il était nécessaire de concentrer les coûts sur un certain nombre d'intervalles utilisateur successifs.

Pour modéliser ces cas, nous avons couplé la contrainte SoftCumulative à d'autres contraintes. Ces contraintes portent sur les variables de coût de la contrainte.

Plusieurs méthodes de filtrage, adaptées des techniques parmi les plus puissantes de l'état de l'art en ordonnancement cumulatif classique ont été introduites. Nous avons notamment adapté les algorithmes de Sweep et d'Edge-Finding, qui sont les algorithmes les plus répandus pour la contrainte Cumulative.

Nous avons également proposé plusieurs règles et algorithmes de filtrage originaux, propres à notre problème. Nous avons notamment introduit un algorithme permettant de filtrer à partir des valeurs minimales dans les domaines des variables de coût. Ce type de filtrage est inhabituel et nous avons motivé son emploi par une situation pratique. Nous avons décrit une stratégie de recherche adaptée à notre problème.

Enfin, l'approche par décomposition nous semblait intéressante à étudier. Pour cela, nous avons adapté des décompositions de la contrainte Cumulative de l'état de l'art à la contrainte SoftCumulative. Nous avons également proposé une nouvelle décomposition, à la fois pour la contrainte Cumulative et pour la contrainte SoftCumulative. Cette décomposition a un nombre de variables linéaire par rapport au

nombre d'activités impliquées dans le problème.

Nous avons ensuite validé notre approche par contrainte globale par des expérimentations. Nous avons testé la recherche de solutions à des problèmes n'admettant pas de solution sans dépassement. Dans ces problèmes, nous avons essayé plusieurs paramétrages et types d'activités et d'instances.

Nous avons testé des problèmes d'optimisation sans, puis avec des contraintes additionnelles, afin d'illustrer les modèles proposés pour des situations pratiques.

Enfin, nous avons adapté la librairie de problèmes cumulatifs PSPLib au cas sur-contraint. Les résultats sur ces tests ont montré que la contrainte SoftCumulative peut résoudre efficacement certaines classes d'instances.

Comme nous pouvions nous y attendre, les décompositions de la contrainte SoftCumulative ont donné de moins bons résultats, même si nous avons montré qu'elles pouvaient être intéressantes pour trouver une première solution. Malheureusement, les résultats obtenus sur notre nouvelle décomposition linéaire par rapport au nombre d'activités ont été décevants, mais il reste de nombreuses perspectives d'amélioration. Cette décomposition linéaire utilise notamment une contrainte de tri qu'il serait intéressant d'améliorer.

Notre travail constitue une nouvelle approche du problème cumulatif avec dépassements de capacité en programmation par contraintes. De nombreuses pistes pourraient encore être explorées. Nous en donnons quelques unes.

Adaptation d'autres algorithmes de filtrage Nous avons adapté les algorithmes de filtrage de l'état de l'art qui nous semblaient les plus prometteurs quant aux performances sur notre problème. Cependant, nous avons vu dans le chapitre 2 que de nombreux autres algorithmes de filtrage existaient pour la contrainte Cumulative. Dès lors, il est raisonnable de penser qu'il existe certains cas dans lesquels ces autre algorithmes pourraient également fournir un filtrage intéressant dans le cadre de notre contrainte SoftCumulative. De plus, un nouvel algorithme de Sweep pour Cumulative [52], plus performant, sera présenté à la conférence CP'12. Il pourrait donc être intéressant de tenter de l'adapter au cas de la contrainte SoftCumulative.

De nouveaux types de coût Nous avons introduit différents types de coût pour modéliser les dépassements. Nous avons vu dans le chapitre 4 qu'ils peuvent modéliser de nombreux cas pratiques. Il serait cependant possible d'introduire de nouvelles manières de calculer les coûts, peut-être plus adaptées à d'autres applications. Dans ce cas, une réadaptation des différentes algorithmes et règles de filtrage présentées dans cette thèse serait évidemment nécessaire.

Intégration au filtrage des contraintes additionnelles sur les coûts Les modèles intégrant la contrainte SoftCumulative et d'autres contraintes sur les coûts peuvent parfois profiter d'un gain de propagation dû à l'ajout de contraintes. Afin d'améliorer le processus de résolution, une perspective intéressante de notre travail pourrait être de considérer, dans le filtrage, à la fois la contrainte SoftCumulative et les contraintes additionnelles. Pour cela, de nouvelles contraintes globales étendant la SoftCumulative pourraient être introduites. Le pouvoir de déduction en considérant à la fois SoftCumulative et les contraintes additionnelles, pourrait alors en être décuplé.

Relaxation du graphe de précédence Dans de nombreux problèmes cumulatifs, comme nous avons pu le voir avec la PSPLib, des relations de précédence entre les activités sont imposées. Celles-ci sont diffé-

rentes des précédences implicites évoquées dans les algorithmes d'Edge-Finding. Il serait alors intéressant de les intégrer au problème cumulatif relaxé. De là, deux pistes s'ouvriraient à nous. La première serait, là encore, d'intégrer au filtrage ces contraintes sur les activités, afin d'obtenir un pouvoir de déduction plus important. La seconde piste serait de considérer une sémantique de relaxation portant sur ces contraintes de précédence, et non seulement sur les dépassements de capacité.

Passage en mode multi-machines Beldiceanu et Carlsson [8] ont introduit la contrainte Cumulatives qui considère une contrainte cumulative multi-ressources, c'est à dire que plusieurs ressources sont considérées à la fois et qu'une activité donnée peut consommer de la ressource sur l'une ou l'autre des machines. La librairie PSPLib que nous avons adapté dans les expérimentations présente des activités pouvant consommer simultanément de la ressource sur plusieurs machines distinctes. Nous pourrions donc considérer une seule contrainte SoftCumulative sur plusieurs machines, et intégrer des filtrages dans celles-ci, plutôt que de poser une conjonction de contraintes SoftCumulative afin de modéliser les problèmes, comme nous l'avons fait dans les expérimentations.

L'ensemble des perspectives évoquées précédemment suggère que l'axe utilisé pour l'étude pourrait encore être suivi pour modéliser, et résoudre de nouveaux problèmes pratiques en ordonnancement cumulatif sur-contraint.

Bibliographie

[1] Abdelkader Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7) :57–73, 1993. pages 1, 2, 25, 122

[2] Christian Artigues, Sophie Demassey, and Emmanuel Neron. *Resource-Constrained Project Scheduling : Models, Algorithms, Extensions and Applications*. ISTE/Wiley, 2008. pages 163

[3] Philippe Baptiste, Claude Le Pape, and Laurent Péridy. Global constraints for partial csps : A case-study of resource and due date constraints. In *CP '98 : Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, pages 87–101, London, UK, 1998. Springer-Verlag. pages 57, 66

[4] Philippe Baptiste, Claude Le Pape, and Nuijten Wim. *Constraint-based scheduling, Applying constraint programming to scheduling problems*. Kluwer Academic, 2001. pages 1, 2, 26, 36, 38, 51

[5] Philippe Baptiste and Wim Nuijten. Satisfiability tests and timebound adjustments for cumulative scheduling problems. *Annals of Operations Research*, 1997. pages 25

[6] Nicolas Beldiceanu and Mats Carlsson. Revisiting the cardinality operator and introducing the cardinality-path constraint family. *Proc. ICLP*, 2237 :59–73, 2001. pages 70

[7] Nicolas Beldiceanu and Mats Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In Toby Walsh, editor, *Principles and Practice of Constraint Programming — CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 377–391. Springer Berlin / Heidelberg, 2001. pages 2, 4, 26, 28

[8] Nicolas Beldiceanu and Mats Carlsson. A new multi-resource *cumulatives* constraint with negative heights. pages 63–79, 2002. pages 2, 4, 26, 177

[9] Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. Research Report T2012-02, Swedish Institute of Computer Science, 2012. pages 129

[10] Nicolas Beldiceanu and Thierry Petit. Cost evaluation of soft global constraints. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, volume 3011 of LNCS*, pages 80–95. Springer-Verlag, 2004. pages 58

[11] Thierry Benoist, Antoine Jeanjean, Guillaume Rochart, Hadrien Cambazard, Emilie Grellier, and Narendra Jussien. Subcontractors scheduling on residential buildings construction sites. In *International Scheduling Symposium (ISS'06)*, pages 32–37, Arcadia Ichigaya, Tokyo, Japan, July 2006. Japan Society of Mechanical Engineers. pages 66

[12] Christian Bessiere, George Katsirelos, Nina Narodytska, and Toby Walsh. Circuit complexity and decompositions of global constraints. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 412–418, 2009. pages 21

[13] Christian Bessière and Pascal Van Hentenryck. To be or not to be ... a global constraint. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 789–794. Springer Berlin / Heidelberg, 2003. pages 20

[14] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Constraint solving over semirings. In *In Proc. IJCAI95*, pages 624–630. Morgan, 1995. pages 2, 4, 56

[15] Stefano Bistarelli, Ugo Montanari, Francesca Rossi, Thomas Schiex, Gérard Verfaillie, and Hélène Fargier. Semiring-based csps and valued csps : Frameworks, properties, and comparison. *Constraints*, 4(3) :199–240, 1999. pages 56

[16] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In Ramon López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 146–150. IOS Press, 2004. pages 19, 20

[17] Hadrien Cambazard. *Résolution de problèmes combinatoires par des approches fondées sur la notion d'explication*. PhD thesis, École des Mines de Nantes, 2006. pages 54

[18] Jacques Carlier and Eric Pinson. An algorithm for solving the job-shop problem. *Manage. Sci.*, 35(2) :164–176, February 1989. pages 1

[19] Jacques Carlier and Eric Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78(2) :146–161, 1994. pages 1, 2, 36

[20] Mats Carlsson and et al. *SICStus Prolog User's Manual*. SICS, 4.2.1 edition, 2012. pages 2

[21] Yves Caseau and François Laburthe. Improved clp scheduling with task intervals. In *ICLP*, pages 369–383, 1994. pages 38, 47, 51

[22] Yves Caseau and François Laburthe. Cumulative scheduling with task intervals. *Proc. JICSLP (Joint International Conference and Symposium on Logic Programming)*, pages 363–377, 1996. pages 47, 51

[23] Martin C. Cooper and Thomas Schiex. Arc consistency for soft constraints. *CoRR*, cs.AI/0111038, 2001. pages 59

[24] Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier. Correctness of constraint retraction algorithms. In *16th International Florida Artificial Intelligence Research Society Conference (FLAIRS'03)*, pages 172–176, St. Augustine, Florida, USA, May 2003. AAAI press. ISBN 978-1-57735-177-1. pages 2, 4, 54

[25] Didier Dubois, Hélène Fargier, and Henri Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Fuzzy Systems, 1993., Second IEEE International Conference on*, pages 1131 –1136 vol.2, 1993. pages 56

[26] Amin El Kholy. *Resource Feasibility in planning*. PhD thesis, Imperial College, University of London, 1996. pages 2, 129

[27] Abdallah Elkhyari, Christelle Guéret, and Narendra Jussien. Solving dynamic resource constraint project scheduling problems using new constraint programming tools. In Edmund Burke and Patrick De Causmaecker, editors, *Practice and Theory of Automated Timetabling IV*, volume 2740 of *Lecture Notes in Computer Science*, pages 39–59. Springer Berlin / Heidelberg, 2003. pages 54

[28] Abdallah Elkhyari, Christelle Guéret, and Narendra Jussien. Ordonnancement dynamique de projet à contraintes de ressources. In Jean-Charles Billaut, Aziz Moukrim, and Éric Sanlaville, editors, *Flexibilité et Robustesse en Ordonnancement*. Hermès, March 2005. ISBN 2746210282. pages 54

[29] Jacques Erschler, Pierre Lopez, and Catherine Thuriot. Raisonnement temporel sous contraintes de ressources et problèmes d'ordonnancement. *Revue d'Intelligence Artificielle*, 5(3) :7–36, 1991. pages 25

[30] Hélène Fargier, Jérôme Lang, and Thomas Schiex. Selecting preferred solutions in fuzzy constraint satisfaction problems. In *Fuzzy and Intelligent Technologies, 1993., First European Congress on*, 1993. pages 56

[31] Eugene C. Freuder. Partial constraint satisfaction. In *IJCAI*, pages 278–283, 1989. pages 54, 59

[32] Eugene C. Freuder and Alan K. Mackworth. Chapter 2 constraint satisfaction : An emerging paradigm. In Peter van Beek Francesca Rossi and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 13 – 27. Elsevier, 2006. pages 18

[33] Eugene C. Freuder and Richard Wallace. Partial constraint satisfaction. *JAI*, 58 :21–70, 1992. pages 54, 56, 59

[34] Michael R. Garey and David S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. pages 10, 25, 146

[35] John Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *Proceedings of the 5th international joint conference on Artificial intelligence - Volume 1*, pages 457–457, San Francisco, CA, USA, 1977. Morgan Kaufmann Publishers Inc. pages 18

[36] John Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence,*, pages 268–277, Toronto, Ont., 1978. pages 18

[37] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *J. ACM*, 12 :516–524, October 1965. pages 18

[38] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th international joint conference on Artificial intelligence - Volume 1*, pages 356–364, San Francisco, CA, USA, 1979. Morgan Kaufmann Publishers Inc. pages 20

[39] Emmanuel Hebrard, Dániel Marx, Barry O'Sullivan, and Igor Razgon. Soft constraints of difference and equality. *Journal of Artificial Intelligence Research*, 41 :97–130, 2011. pages 57

[40] Emmanuel Hebrard, Barry O'Sullivan, and Igor Razgon. A soft constraint of equality : Complexity and approximability. *Proc. CP*, 5202 :358–371, 2008. pages 72

[41] Narendra Jussien. *The versatility of using explanations within constraint programming*. PhD thesis, Université de Nantes, 2003. Habilitation Thesis. pages 2, 4, 54

[42] Narendra Jussien and Patrice Boizumault. Best-first search for property maintenance in reactive constraints systems. In *Proceedings of the 1997 international symposium on Logic programming*, ILPS '97, pages 339–353, Cambridge, MA, USA, 1997. MIT Press. pages 2, 4, 54

[43] Roger Kameugne, Laure Pauline Fotso, Joseph Scott, and Youcheu Ngo-Kateu. A quadratic edge-finding filtering algorithm for cumulative resource constraints. In J. Lee, editor, *Principles and Practice of Constraint Programming – CP 2011*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011. pages 3, 2, 4, 36, 47, 82, 88

[44] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8 :18–24, April 1976. pages 10

[45] Rainer Kolisch and Arno Sprecher. Psplib – a project scheduling problem library. *European Journal of Operational Research*, 96 :205–216, 1996. pages 3, 5, 122, 145, 146, 154

[46] Abdelkader Lahrichi. The notions of Hump, Compulsory Part and their use in Cumulative Problems. *C.R. Acad. sc.*, t. 294 :204–211, 1982. pages 26

[47] Javier Larrosa and Pedro Meseguer. Exploiting the use of dac in max-csp. In Eugene Freuder, editor, *Principles and Practice of Constraint Programming — CP96*, volume 1118 of *Lecture Notes in Computer Science*, pages 308–322. Springer Berlin / Heidelberg, 1996. pages 55, 59

[48] Javier Larrosa, Pedro Meseguer, and Thomas Schiex. Maintaining reversible dac for max-csp. *Artif. Intell.*, 107 :149–163, January 1999. pages 55, 59

[49] Javier Larrosa and Thomas Schiex. Solving weighted csp by maintaining arc consistency. *Artif. Intell.*, 159(1-2) :1–26, 2004. pages 59

[50] Jimmy Ho-Man Lee and Ka Lun Leung. Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 559–565, 2009. pages 59

[51] Jimmy Ho-Man Lee and Ka Lun Leung. A stronger consistency for soft global constraints in weighted constraint satisfaction. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010. pages 57, 59

[52] Arnaud Letort, Nicolas Beldiceanu, and Mats Carlsson. A scalable sweep algorithm for the cumulative constraint. In *To appear, 18th International Conference on Principles and Practice of Constraint Programming (CP'12)*, Lecture Notes in Computer Science, Quebec City, Canada, 2012. Springer-Verlag. pages 31, 32, 176

[53] Pierre Lopez. *Approche énergétique pour l'ordonnancement de tâches sous contraintes de temps et de ressources*. Thèse de Doctorat, Toulouse, 1991. pages 1, 25

[54] Pierre Lopez, Jacques Erschler, and Patrick Esquirol. Ordonnancement de tâches sous contraintes : une approche énergétique. *Automatique, Productique, Informatique Industrielle*, 26 :453–481, 1992. pages 25

[55] Pierre Lopez and Patrick Esquirol. Consistency enforcing in scheduling : A general formulation based on energetic reasoning. In *Proc. of the 5^{th} International Workshop on Project Management and Scheduling*, pages 155–158, Poznan, Poland, 1996. pages 25

[56] Kurt Mehlhorn and Sven Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In Rina Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 306–319. Springer, 2000. pages 139

[57] Luc Mercier and Pascal Van Hentenryck. Edge finding for cumulative scheduling. *INFORMS Journal on Computing*, 20(1) :143–153, 2008. pages 36

[58] Jean-Philippe Métivier, Patrice Boizumault, and Samir Loudni. Softening gcc and regular with preferences. In *SAC*, pages 1392–1396, 2009. pages 57

[59] Wim Nuijten. *Time and resource Constrained Scheduling : A Constraint Satisfaction Approach*. PhD thesis, Eindhoven University of Technology, 1994. pages 2, 25, 36, 38

[60] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. pages 10

[61] Gilles Pesant and Jean-Charles Régin. Spread : A balancing constraint based on statistics. *Proc. CP*, pages 460–474, 2005. pages 70

[62] Thierry Petit. *Modélisation et algorithmes de résolution des problèmes sur-contraints*. PhD thesis, Ph. D. thesis, LIRMM-Université de Montpellier II, 2002. pages 2, 4, 56, 57, 59

[63] Thierry Petit. Focus : A constraint for concentrating high costs. In *To appear, 18th International Conference on Principles and Practice of Constraint Programming (CP'12)*, Lecture Notes in Computer Science, Quebec City, Canada, 2012. Springer-Verlag. pages 72, 158

[64] Thierry Petit, Nicolas Beldiceanu, and Xavier Lorca. A generalized arc-consistency algorithm for a class of counting constraints : Revised edition that incorporates one correction. *CoRR (revised IJCAI'11 paper)*, abs/1110.4719, 2011. pages 70

[65] Thierry Petit and Emmanuel Poder. Global propagation of side constraints for solving over-constrained problems. *Annals of Operations Research*, 184 :295–314, 2011. pages 66, 67, 68, 69, 72

[66] Thierry Petit and Jean-Charles Régin. The ordered distribute constraint. In *ICTAI*, pages 431–438, 2010. pages 70

[67] Thierry Petit, Jean-Charles Régin, and Christian Bessière. Meta constraints on violations for over constrained problems. *Proc. IEEE-ICTAI*, pages 358–365, 2000. pages 2, 4, 57, 70

[68] Thierry Petit, Jean-Charles Régin, and Christian Bessière. Specific filtering algorithms for over-constrained problems. In Toby Walsh, editor, *Principles and Practice of Constraint Programming — CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 451–463. Springer Berlin / Heidelberg, 2001. pages 57, 58

[69] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, pages 362–367, 1994. pages 21

[70] Jean-Charles Régin, Thierry Petit, Christian Bessière, and Jean-François Puget. An original constraint based approach for solving over constrained problems. In Rina Dechter, editor, *Principles and Practice of Constraint Programming – CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 543–548. Springer Berlin / Heidelberg, 2000. pages 55, 59

[71] Martí Sánchez, Simon de Givry, and Thomas Schiex. Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints*, 13(1-2) :130–154, 2008. pages 59

[72] Pierre Schaus, Yves Deville, Pierre Dupont, and Jean-Charles Régin. The deviation constraint. *Proc. CPAIOR*, 4510 :260–274, 2007. pages 70

[73] Thomas Schiex. Possibilistic constraint satisfaction problems or "how to handle soft constraints ?". In *Proceedings of the eighth conference on Uncertainty in Artificial Intelligence*, UAI '92, pages 268–275, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. pages 56

[74] Thomas Schiex. Arc consistency for soft constraints. In Rina Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 411–424. Springer, 2000. pages 59

[75] Thomas Schiex, Hélène Fargier, and Gérard Verfaillie. Valued constraint satisfaction problems : Hard and easy problems. In *IJCAI (1)*, pages 631–639, 1995. pages 2, 4, 56

[76] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling. In Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2012. Corresponds to Gecode 3.7.3. pages 2

[77] Andreas Schutt, Thibaut Feydy, Peter Stuckey, and Mark Wallace. Why cumulative decomposition is not as bad as it sounds. In *CP'09*, pages 746–761, 2009. pages 2, 121, 122, 129

[78] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Explaining the cumulative propagator. *Constraints*, 16(3) :250–282, July 2011. pages 54

[79] Andreas Schutt and Armin Wolf. A new $o(n^2log(n))$ not-first/not-last pruning algorithm for cumulative resource constraints. In David Cohen, editor, *Principles and Practice of Constraint Programming – CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 445–459. Springer Berlin / Heidelberg, 2010. pages 51

[80] Andreas Schutt, Armin Wolf, and Gunnar Schrader. Not-first and not-last detection for cumulative scheduling in o(n3log n). In Masanobu Umeda, Armin Wolf, Oskar Bartenstein, Ulrich Geske, Dietmar Seipel, and Osamu Takata, editors, *Declarative Programming for Knowledge Management*, volume 4369 of *Lecture Notes in Computer Science*, pages 66–80. Springer Berlin / Heidelberg, 2006. pages 51

[81] Philippe Torres and Pierre Lopez. On not-first/not-last conditions in disjunctive scheduling. *European Journal of Operational Research*, 127(2) :332–343, 2000. pages 51

[82] Pascal Van Hentenryck and Laurent Michel. Control abstractions for local search. *9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, pages 66–80, 2003. pages 58

[83] Willem J. van Hoeve. The alldifferent constraint : A survey. *CoRR*, cs.PL/0105015, 2001. pages 21

[84] Willem J. van Hoeve. *Operations Research Techniques in Constraint Programming*. PhD thesis, Ph D Thesis, University of Amsterdam, 2005. pages 17

[85] Willem J. van Hoeve, Gilles Pesant, and Louis-Martin Rousseau. On global warming : Flow-based soft global constraints. *Journal of Heuristics*, 12 :347–373, September 2006. pages 57, 58

[86] Gérard Verfaillie, Michel Lemaître, and Thomas Schiex. Russian doll search for solving constraint optimization problems. In William J. Clancey and Daniel S. Weld, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 1*, pages 181–187. AAAI Press / The MIT Press, 1996. pages 61

[87] Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments : A survey. *Constraints*, 10 :253–281, 2005. pages 54

[88] Petr Vilím. $O(n \log n)$ filtering algorithms for unary resource constraint. In Jean-Charles Régin and Michel Rueher, editors, *Proceedings of CP-AI-OR 2004*, volume 3011 of *Lecture Notes in Computer Science*, pages 335–347, Nice, France, April 2004. Springer-Verlag. pages 51

[89] Petr Vilím. *Global Constraints in Scheduling*. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, KTIML MFF, Universita Karlova, Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic, August 2007. pages 2, 36, 38, 51

[90] Petr Vilím. Edge finding filtering algorithm for discrete cumulative resources in o(kn log n). In Ian Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 802–816. Springer Berlin / Heidelberg, 2009. pages 3, 2, 4, 36, 82, 86

[91] Petr Vilím. Max energy filtering algorithm for discrete cumulative resources. In Willem-Jan van Hoeve and John Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5547 of *Lecture Notes in Computer Science*, pages 294–308. Springer Berlin / Heidelberg, 2009. pages 37, 38

[92] Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In Tobias Achterberg and J. Beck, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6697 of *Lecture Notes in Computer Science*, pages 230–245. Springer Berlin / Heidelberg, 2011. pages 51

[93] Richard J. Wallace. Directed arc consistency preprocessing. In Manfred Meyer, editor, *Constraint Processing, Selected Papers*, volume 923 of *Lecture Notes in Computer Science*, pages 121–137. Springer, 1995. pages 55, 59

[94] Armin Wolf and Gunnar Schrader. O(n log n) overload checking for the cumulative constraint and its application. In Masanobu Umeda, Armin Wolf, Oskar Bartenstein, Ulrich Geske, Dietmar Seipel, and Osamu Takata, editors, *Declarative Programming for Knowledge Management*, volume 4369 of *Lecture Notes in Computer Science*, pages 88–101. Springer Berlin / Heidelberg, 2006. pages 51

[95] Jianyang Zhou. A permutation-based approach for solving the job-shopproblem. *Constraints*, 2(2) :185–213, October 1997. pages 139

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

ECOLE DES MINES DE NANTES

Thèse de Doctorat

Alexis De Clercq

Ordonnancement cumulatif avec dépassements de capacité

Contrainte globale et décompositions

Cumulative scheduling with overloads of resource

Global constraint and decompositions

Résumé

La programmation par contraintes est une approche intéressante pour traiter des problèmes d'ordonnancement. En ordonnancement cumulatif, les activités sont définies par leur date de début, leur durée et la quantité de ressource nécessaire à leur exécution. La ressource totale disponible (la capacité) en chaque instant est fixe. La contrainte globale Cumulative modélise ce problème en programmation par contraintes. Dans de nombreux cas pratiques, la date limite de fin d'un projet est fixée et ne peut être retardée. Dans ce cas, il n'est pas toujours possible de trouver un ordonnancement des activités qui n'engendre aucun dépassement de la capacité en ressource. On peut alors tolérer de relâcher la contrainte de capacité, dans une limite raisonnable, pour obtenir une solution. Nous proposons une nouvelle contrainte globale : la contrainte SoftCumulative qui étend la contrainte Cumulative pour prendre en compte ces dépassements de capacité. Nous illustrons son pouvoir de modélisation sur plusieurs problèmes pratiques, et présentons différents algorithmes de filtrage. Nous adaptons notamment les algorithmes de balayage et d'Edge-Finding à la contrainte SoftCumulative. Nous montrons également que certains problèmes pratiques nécessitent d'imposer des violations de capacité pour satisfaire des règles métiers, modélisées par des contraintes additionnelles. Nous présentons une procédure de filtrage originale pour traiter ces dépassements imposés. Nous complétons notre étude par une approche par décomposition. Enfin, nous testons et validons nos différentes techniques de résolution par une série d'expériences.

Mots clés

Programmation par contraintes, Ordonnancement, Problèmes sur-contraints, Contraintes globales.

Abstract

Constraint programming is an interesting approach to solve scheduling problems. In cumulative scheduling, activities are defined by their starting date, their duration and the amount of resource necessary for their execution. The total available resource at each point in time (the capacity) is fixed. In constraint programming, the Cumulative global constraint models this problem.
In several practical cases, the deadline of the project is fixed and can not be delayed. In this case, it is not always possible to find a schedule that does not lead to an overload of the resource capacity. It can be tolerated to relax the capacity constraint, in a reasonable limit, to obtain a solution.
We propose a new global constraint : the SoftCumulative constraint that extends the Cumulative constraint to handle these overloads. We illustrate its modeling power on several practical problems, and we present various filtering algorithms. In particular, we adapt the sweep and Edge-Finding algorithms to the SoftCumulative constraint. We also show that some practical problems require to impose overloads to satisfy business rules, modelled by additional constraints. We present an original filtering procedure to deal with these imposed overloads. We complete our study by an approach by decomposition. At last, we test and validate our different resolution techniques through a series of experiments.

Key Words

Constraint programming, Scheduling, Over-constrained problems, Global constraints.