Ordonnancement cumulatif en programmation par contraintes : caractérisation énergétique des raisonnements et solutions robustes

Alban Derrien

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

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Alban DERRIEN

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

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 27 novembre 2015
Thèse n° : 2015 EMNA 0230

Ordonnancement cumulatif en programmation par contraintes

Caractérisation énergétique des raisonnements et solutions robustes

JURY

Rapporteurs :	**M. Pierre LOPEZ**, Directeur de Recherche, LAAS - CNRS Toulouse
	M. Claude-Guy QUIMPER, Professeur, Université de Laval, Québec
Examinateurs :	**M. Xavier LORCA**, Maitre-assistant HDR, École des Mines de Nantes
	M. Hadrien CAMBAZARD, Docteur, Université de Grenoble
Directeur de thèse :	**M. Nicolas BELDICEANU**, Professeur, École des Mines de Nantes
Co-directeur de thèse :	**M. Thierry PETIT**, Maitre-assistant HDR, École des Mines de Nantes

Remerciements

Je tiens tout d'abord à remercier les personnes qui m'ont donné envie de faire une thèse. Ces professeurs qui m'ont donné le goût d'enseigner, mais aussi ceux qui m'ont fait croire en moi, en ma capacité à mener à bien ce projet un peu fou.

Je remercie aussi mes encadrants, Thierry et Nicolas, qui m'ont beaucoup appris sur la recherche, de m'avoir compris et écouté même si ce n'était pas chose aisée.

Merci à Hadrien Cambazard et Xavier Lorca d'avoir accepté d'être dans le jury de ma thèse. Ainsi qu'à Pierre Lopez et Claude-Guy Quimper d'avoir accepté d'en être les rapporteurs. Merci pour vos retours sur mon travail et plus particulièrement sur le manuscrit.

Merci à l'ensemble de l'équipe TASC, plus particulièrement Philippe pour tous ces bons moments, Xavier pour ses coups de gueule ! et Charles pour sa bonne humeur. Vous côtoyer au jour le jour fut un plaisir, merci pour ces trois années. Mais non mon petit Jean-Gui, je ne t'oublie pas, toi et ton amour pour ton chat et les chevaux du pré face à notre bureau ! Vous m'avez tous les quatre tellement plus apporté que je ne saurai l'exprimer.

Merci aussi à ma famille, qui m'a supporté durant ces longues années d'études, et comme ce n'est clairement pas le moment j'en profite pour vous faire une bise. Merci aussi à ma nouvelle famille sans qui, clairement, cette thèse ne serait pas ce qu'elle est aujourd'hui. Je ne retire aucun mérite pour mon manuscrit, car il est vôtre.

Enfin, merci à Audrey pour ... tout, tout simplement.

Table des matières

Liste des tableaux

Table des figures

Liste des Algorithmes

1

Introduction

L'informatique a permis au cours des dernières décennies de transformer notre société, car les outils informatiques offrent la possibilité de calculer en quelques millisecondes ce qu'un être humain aurait mis une vie à calculer à la main. Néanmoins, certains problèmes nécessitent pour être résolus une quantité de calculs intrinsèquement exponentielle. Dans cette situation, même en utilisant les ordinateurs les plus puissants, trouver une solution peut prendre des siècles. On qualifie ces problèmes de *difficiles*, au sens de leur *complexité algorithmique*. Il est heureusement possible pour de nombreuses instances réelles de ces problèmes de concevoir des techniques de résolution permettant d'obtenir des solutions dans un temps raisonnable, par exemple en exploitant la spécificité des données pour éviter des calculs inutiles. Parmi les problèmes difficiles, nous pouvons citer des problèmes de nature combinatoire : tournées de véhicules, problèmes d'ordonnancement, problèmes d'affectation, configuration, etc.

Par exemple, la réalisation d'un circuit imprimé peut être un problème difficile, tout comme la création d'un emploi du temps où des salles et des professeurs sont affectés à des cours, ou encore l'affectation de machines virtuelles en « cloud-computing ».

Dans le cas de problèmes d'optimisation, la recherche d'une solution optimale selon un critère – ou, à défaut, de la meilleure solution possible – est traditionnellement privilégiée. Une extension naturelle consiste à considérer plusieurs critères, ce qui permet de couvrir un plus large spectre de problèmes. Il semble cependant qu'au fil du temps et des améliorations scientifiques, ces schémas tendent peu à peu à être remplacés par une vision plus subtile de la notion de qualité des solutions, qui intègre d'autres notions. D'une part pour des problèmes trop durs chercher l'optimalité est hors de portée, d'autre part en pratique la priorité est bien souvent de chercher des solutions satisfaisant des facteurs humains, rarement exprimés de façon formelle par les personnes ayant posé l'énoncé du problème. Ces facteurs humains peuvent parfois être traités sous la forme de critères d'optimisation supplémentaires, parfois sous d'autres formes (méthodes stochastiques, ajout de contraintes ou modification des contraintes du problème d'origine). Dans tous les cas, la prise en compte de ces facteurs peut être réalisée efficacement en élaborant des méthodes théoriques, reposant sur les mathématiques, la logique, l'algorithmique.

Dans un problème d'ordonnancement, par exemple, nous pouvons essayer d'utiliser la puissance de calcul d'un ordinateur pour exhiber une solution robuste aux aléas qui surviendront lors de son exécution, éventuellement non optimale mais la plus proche possible de l'optimal théorique. Cette solution peut s'avérer plus utile en pratique qu'une solution de coût réellement optimal mais qui ne serait plus valide au moindre évènement inattendu survenant lors de sa mise en œuvre.

La programmation par contraintes est un paradigme d'Intelligence Artificielle et de Recherche Opérationnelle, où les problèmes sont modélisés par un ensemble de contraintes, couplées à des algorithmes et portant sur des variables. L'originalité par rapport à d'autres méthodes de résolution génériques est que la programmation par contrainte n'impose pas de restriction concernant la nature des contraintes. Lorsque les domaines des variables sont finis et lorsque le moteur de résolution est basé sur l'inférence d'informations, le principe de la programmation par contrainte consiste à réduire les domaines et à propager chaque évènement de réduction à toutes les contraintes du modèle, afin d'inférer de nouvelles réductions de domaines. La gestion de ce processus est effectuée par un solveur. Dans ce contexte, une des forces de la programmation par contraintes tient dans le fait que chaque contrainte puisse être dotée d'algorithmes puissants mais de coût raisonnable en termes de temps d'exécution/nombre d'opérations, chacun basé sur ses propres règles d'inférence. Certains de ces algorithmes ont été découverts après plusieurs années de travaux de recherche. Une fois produits et prouvés, ils sont à disposition de tous les utilisateurs de la technologie. Il convient de mentionner que la programmation par contraintes n'est pas restreinte à des méthodes de résolution suivant ce schéma, bien qu'il soit l'un des plus courants. Les variables peuvent être continues et simplement bornées [GB06, Cha13] ; le solveur peut être un algorithme de recherche locale [BEG+11]. Cependant, sachant que les différentes techniques peuvent être combinées, par exemple en utilisant une méthode de recherche à voisinage large [Sha98] combinant inférence et recherche locale, tous les solveurs de contraintes proposent de façon native de nombreux algorithmes d'inférence.

Nous nous intéressons à des problèmes d'ordonnancement cumulatifs, usuellement représentés dans les solveurs de contraintes par une unique contrainte globale, modélisant le coeur du problème. Sachant que le problème cumulatif est difficile, aucun algorithme d'inférence exhaustif n'est utilisé, car son coût serait trop élevé. Par conséquent, les algorithmes existants sont basés sur un ensemble de règles sémantiques. La variété de ces règles et leur nature intuitive fait que la programmation par contraintes est un candidat privilégié pour résoudre ce type de problèmes. En effet, il n'existe aucune limite théorique à l'utilisation d'une nouvelle règle d'inférence pour une contrainte, quelle que soit la nature de cette règle. Les algorithmes ont donc été améliorés au fil des années par l'intégration ou le raffinement de certaines règles [Lah82, ELT89, Nui94, Vil11]. Nous pouvons remarquer que toutes ces règles et algorithmes semblent difficiles à qualifier de façon uniforme ou à comparer d'un point de vue théorique, autrement que par leur complexité ou leur efficacité. Cette hétérogénéité apparente induit une difficulté du point de vue de l'utilisateur d'un solveur :

– *quel algorithme choisir ?*
– *pour quel problème ?*
– *pourquoi utiliser plutôt l'un que l'autre ?*

Dans ce manuscrit, nous étudions ces différents mécanismes en essayant autant que possible d'ignorer les intuitions sous-jacentes aux différents principes d'inférence, afin de nous focaliser exclusivement sur leurs propriétés théoriques.

Ce point de vue peut être exploité suivant plusieurs axes, débouchant sur de nouvelles contributions théoriques et pratiques.

① Relativiser l'hétérogénéité des différents algorithmes de l'état de l'art. Nous montrons qu'il est possible de les présenter de façon unifiée et de les comparer selon leurs propriétés.

② Répondre à des questions ouvertes. Un des bénéfices de notre démarche est qu'elle offre des réponses simples à des questions posées depuis plus d'une décennie concernant une méthode d'inférence particulière, le *raisonnement énergétique*.

③ Intégrer des notions de qualité « humaines » à des problèmes résolus en programmation par contraintes. L'idée est de modifier la définition d'un problème de sorte à conserver certaines propriétés théoriques du problème d'origine. En d'autres termes, s'il est possible de transposer une partie de la taxonomie des algorithmes d'inférence classiques à notre nouveau problème, alors il peut être possible de le résoudre avec une efficacité pratique comparable. Nous prouvons le concept avec la notion de robustesse. Nous démontrons que les propriétés théoriques de l'un des algorithmes de référence pour le problème cumulatif classique peuvent être vérifiées dans le cas de problèmes cumulatifs robustes. Nous adaptons cet algorithme au nouveau problème ainsi défini. Nous montrons expérimentalement que cette démarche peut être utile dans un contexte réel.

Le plan de la thèse est le suivant :

- Dans les chapitres 2 et 3 nous présentons les éléments techniques et les notations nécessaires à la compréhension du manuscrit. La partie 2.1 introduit des notions de base de complexité algorithmique. La partie 2.2 introduit la programmation par contraintes. La partie 3.1 décrit un ensemble de notions et de techniques d'ordonnancement de l'état de l'art étudiées pendant cette thèse. La partie 3.2 présente brièvement la notion de robustesse en programmation par contraintes.

- Le chapitre 4 présente les deux premières contributions de cette thèse : une caractérisation des raisonnements pour la contrainte cumulative et une étude du raisonnement énergétique, débouchant sur une énumération précise des intervalles d'intérêt et de nouveaux algorithmes. La partie 4.1 est dédiée aux raisonnements de détection d'incohérence. La partie 4.2 se focalise sur les intervalles d'intérêts pour les différents raisonnements de détection d'incohérence. La partie 4.3 exploite ces propriétés pour démontrer les relations de dominances entre ces raisonnements. La partie 4.4 étend ces travaux aux propagateurs du raisonnement énergétique, ces travaux on été publiés à *CP'14* [DP14].

- Le chapitre 5 porte sur l'ordonnancement robuste. La partie 5.1 introduit une définition de problème d'ordonnancement cumulatif robuste, dans lequel chaque activité prise en compte par l'ordonnancement peut être retardée d'un certain temps sans qu'il soit nécessaire de recalculer la planification. La partie 5.2 présente une nouvelle contrainte cumulative robuste pour traiter ce problème. Nous montrons que les propriétés d'un des algorithmes d'inférence les plus récents du problème cumulatif classique peuvent être maintenues et nous adaptons cet algorithme. Nous appliquons notre technique sur un problème réel dans la partie 5.3,ces travaux on été publiés à *CP'14* [DPZ14a].

- Dans les annexes A.1 nous présentons le *résumé* de travaux publiés à *CP'15* [DFPP15], réalisés en parallèle de ce manuscrit, sur un problème d'ordonnancement dans le cadre d'un projet de résumé video.

- Dans les annexes A.3.2 nous présentons les résultats de la résolution des problèmes de la bibliothèque d'instances *pack*, avec notre implémentation des raisonnements Time-Table, Edge-Finding, Time-Table-Edge-Finding et Raisonnement Energétique.

I

État de l’art

2

Notions de base

Sommaire

La programmation par contraintes est un paradigme permettant de résoudre des problèmes combinatoires. Dans ce chapitre, nous introduisons les notions de complexité algorithmique utiles à la lecture de cette thèse avant d'introduire les principes de base de la programmation par contraintes. Ensuite, nous présentons les principaux algorithmes proposés dans la littérature pour la contrainte Cumulative, qui permet de modéliser un large spectre de problèmes d'ordonnancement sous contraintes de ressources.

2.1 Notions de complexité

2.1.1 La notation de Landau

La complexité d'un algorithme permet de caractériser son temps d'exécution en fonction de la taille des données notée n. Une comparaison asymptotique au regard d'une fonction plus simple permet de comprendre facilement le comportement de l'algorithme et d'évaluer le temps ou la quantité de ressources consommées par l'algorithme. Les notations de Landau (également appelées notations de Knuth) donnent des bornes aux comportements asymptotiques : pour un n arbitrairement grand, une fonction $f(n)$ caractérisée comme appartenant à $\mathcal{O}(g(n))$ ne sera pas plus lente que la fonction $g(n)$. Il s'agit d'une analyse du comportement dans le pire cas.

Notation 2.1 (Landau)**.**

$$f(n) \in \mathcal{O}(g(n)) \Leftrightarrow \exists k > 0, \exists n_0; \forall n > n_0, \qquad |f(n)| \leq |g(n)| \cdot k$$

Les limites pratiques de la notation de Landau

L'algorithme de tri (permettant d'organiser une collection d'objets selon un ordre déterminé) communément admis comme étant le plus rapide est le *quickSort* qui a une complexité dans le pire des cas en $\mathcal{O}(n^2)$. Cela signifie que, si l'on double la taille de l'ensemble à trier, le temps d'exécution sera quadruplé. Pour autant, on peut noter qu'il existe des algorithmes de complexité temporelle, dans le pire cas, optimale : en $\mathcal{O}(n\ log\ n)$. L'algorithme *quickSort* est, malgré cela, considéré comme étant le plus rapide, car la probabilité d'avoir en entrée le pire cas est proche de zéro en pratique. En effet, l'étude du temps d'exécution en moyenne révèle un comportement en $\mathcal{O}(n\ log\ n)$ [Kar14].

L'ordre de grandeur asymptotique ne fait pas apparaître un certain nombre de constantes qui sont négligeables devant n lorsque n est grand, mais qui deviennent importantes lorsque n est petit. Il est d'usage d'utiliser un autre algorithme lorsque la taille des données est petite (le nombre de valeurs à trier est inférieur à 10). Ce qui explique, dans certains cas, pourquoi il existe plusieurs versions d'un même algorithme, en fonction de la taille et du type du problème que l'on va couramment être amené à résoudre. C'est le cas, par exemple, concernant les algorithmes de propagation de la contrainte $\mathrm{Cumulative}$ que nous décrirons dans la section 3.1.3.

2.1.2 Les classes de complexité

Nous présentons ici les classes de complexité qui seront utiles dans ce manuscrit. Chaque *classe de complexité* correspond à un ensemble de fonctions de complexité qu'il est pertinent de regrouper. Le but est de classer les algorithmes en fonction de leur coût d'exécution dans le pire des cas. Le lecteur intéressé par une présentation complète pourra se référer aux ouvrages de Garey et Johnson [GJ79] et de Papadimitriou [Pap94].

On distingue les problèmes de décision des problèmes de recherche de solutions et des problèmes d'optimisation. Pour introduire ces trois classes de problèmes, nous nous appuyons sur trois variantes du problème du voyageur de commerce (TSP, pour *Traveling Salesman Problem* en anglais). Le lecteur intéressé pourra se référer à [Pap94, page 411-422] pour une étude plus poussée de ces trois cas.

Problème de décision

Les problèmes de décision sont des problèmes pour lesquels la réponse attendue est soit "oui" soit "non".

La classe **P** est l'ensemble des problèmes de décision dits polynomiaux : il est possible de les résoudre avec un algorithme dont la complexité est un polynôme de degré k, où k est une constante quelconque. Quel que soit k, ces problèmes sont souvent considérés comme étant simples à résoudre.

Un problème de décision pour lequel il est possible de vérifier la véracité d'une solution en temps polynomial est dans la classe des problèmes **NP**. Nous avons donc **P** $\subseteq$ **NP**, il suffit de résoudre à nouveau le problème pour en vérifier une solution.

Un problème p est dit **NP-difficile** si tout problème p' de **NP** peut se ramener à p via une réduction polynomiale. Cela signifie que l'on peut résoudre p' en le transformant (en temps polynomial) en p et en résolvant p. Le problème p est donc au moins aussi dur que n'importe quel problème de **NP**. Si, de plus, p est dans **NP**, il est alors **NP-complet**.

Exemple 2.1 (TSPDecision)**.** *Étant donné un entier B, un ensemble de villes et une distance entre chaque couple de villes ; existe-t-il un tour passant par toutes les villes exactement une fois tel que la distance parcourue est au plus B ?*

Il a été démontré que TSPDecision est **NP-complet** [GJ79].

Problème de recherche de solutions

Les problèmes de recherche de solutions sont des problèmes pour lesquels il est demandé de donner une solution. Les problèmes de recherche de solutions ne peuvent formellement appartenir à la classe **NP**, puisque la classification précédente ne s'applique qu'à des problèmes de décision.

Exemple 2.2 (TSPRecherche)**.** *Étant donné un entier B, un ensemble de villes et une distance entre chaque couple de villes ; donnez un tour passant par toutes les villes exactement une fois tel que la distance parcourue est au plus B.*

Au vu des liens forts avec les problèmes de décision, Papadimitriou propose dans [Pap94] de créer des classes analogues **FP**, **FNP** et **FNP-complet**.

La lettre F est souvent omise, ce qui permet de confondre les problèmes de décision et de recherche de solutions, sans changer la définition initiale. Cette transgression d'un formalisme mathématique est communément admise puisqu'en pratique l'important est qu'un problème **NP-complet** soit difficile à résoudre, mais qu'il est simple de vérifier qu'une solution est correcte.

Dans ce manuscrit, nous utiliserons la notation **NP-complet** pour les problèmes de recherche de solutions **FNP-complet** dont le problème de décision associé est **NP-complet**, de manière à bien montrer que vérifier la solution se fait en temps polynomial.

Problème d'optimisation

Dans un problème d'optimisation, le but est de minimiser (ou maximiser) la valeur d'un objectif. Prouver qu'une solution est optimale nécessite deux choses : vérifier qu'elle est correcte, mais aussi qu'il n'en existe pas de meilleure, au sens de l'objectif. Dans le cas général, un problème d'optimisation n'est donc pas dans **NP** puisqu'en vérifier une solution ne peut être fait en temps polynomial. Plusieurs classifications ont été proposées pour les problèmes d'optimisation. Nous noterons la classe **OptP** [Kre88], la classe **DP** [Pap94] et la classe **NPOptimisation** [APM+99]. La classe **NPO** (resp. **NPO-complet**) représente alors la classe des problèmes pour lesquels le problème de décision associé est dans **NP** (resp. **NP-complet**)

Exemple 2.3 (TSPOptimisation)**.** *Étant donné un ensemble de villes et une distance entre chaque couple de villes ; donnez un tour passant par toutes les villes exactement une fois tel que la distance parcourue est minimale.*

TSPOptimisation est donc **NPO-complet** puisque TSPDecision est **NP-Complet**. Un raccourci parfois utilisé abusivement est, de façon analogue aux problèmes de recherche de solution, de parler d'un problème d'optimisation **NPO-complet** comme étant **NP-complet**. Mais ici, la notation ne capture pas un point important : vérifier l'optimalité d'une solution ne peut se faire en temps polynomial. Nous noterons un tel problème comme étant **NP-Difficile**. Le lecteur intéressé pourra se référer aux discussions de Puget [Pug13].

2.2 La programmation par contraintes

La programmation par contraintes est un paradigme permettant de résoudre les problèmes de satisfaction de contraintes, notés CSP (pour Constraint Satisfaction Problem en anglais) ainsi que des problèmes d'optimisation, notés COP (pour Constraint Optimisation Problem en anglais). Il s'agit de trouver des solutions à un réseau de contraintes, c'est-à-dire une assignation de valeurs aux variables telle que toutes les contraintes soient satisfaites. Une contrainte spécifie l'ensemble des combinaisons de valeurs (des tuples) que ses variables peuvent prendre.

2.2.1 Les réseaux de contraintes

Un CSP se modélise par un réseau de contraintes : un triplet $\langle variables, domaines, contraintes \rangle$.

Définition 2.1 (Réseau de contraintes [Mon74])**.** *Un réseau de contraintes N est défini par le triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ où :*

- $\mathcal{X} = (x_1, ..., x_n)$ *désigne les n variables ;*
- $\mathcal{D} = (\mathcal{D}(x_1), ..., \mathcal{D}(x_n))$ *désigne les domaines des variables ;*
- $\mathcal{C} = (c_1, ..., c_m)$ *désigne les m contraintes, avec $\mathcal{X}(c_j) \subseteq \mathcal{X}$;*

Dans ce manuscrit, nous considérons que les variables prennent leurs valeurs dans $\mathbb{Z}$. Pour une variable x_i nous notons la plus petite valeur de son domaine $\underline{x_i}$ et la plus grande valeur $\overline{x_i}$. Son domaine est donc inclus dans l'intervalle défini par ces deux valeurs : $\mathcal{D}(x_i) \subseteq [\underline{x_i}, \overline{x_i}]$. Une variable bornée est une variable pour laquelle toutes les valeurs de l'intervalle sont présentes.

Définition 2.2 (Contrainte [Bes06])**.** *Un contrainte c est une relation définie sur une séquence de n variables $\mathcal{X}(c) = (x_1, ..., x_n)$. C'est un sous ensemble de $\mathbb{Z}^n$ qui contient les tuples $\tau \in \mathbb{Z}^n$ qui satisfont c.*

Une contrainte peut être définie en extension : on donne alors la liste de tous les tuples qui satisfont la contrainte, ou bien de façon équivalente la liste des tuples qui ne satisfont pas la contrainte. Ou en intention ; on donne alors une définition de la sémantique de la contrainte.

Une contrainte c définie sur les variables $x_1, x_2, ..., x_k$ sera écrite $c(x_1, x_2, ..., x_k)$. Par souci de lisibilité lorsque les variables peuvent être regroupées, par exemple sémantiquement, chaque ensemble sera directement passé en paramètre.
Par exemple, la contrainte $\mathrm{Cumulative}(s_1, d_1, e_1, h_1, s_2, d_2, e_2, h_2, ..., s_n, d_n, e_n, h_n, capa)$ sera écrite $\mathrm{Cumulative}(activites, capa)$, $activites$ représentant alors un ensemble d'activités, chacune étant représentée par le quadruplet de variables $\langle s_i, d_i, e_i, h_i \rangle$.

Exemple 2.4 (Contrainte)**.** *Soit trois variables x,y et z et leur domaine respectif :* $\{1, 2\}, \{1, 2\}, \{1, 2, 3\}$. *Soit la contrainte* $\mathrm{AllDifferent}(x, y, z)$ *: chaque variable doit prendre une valeur différente. Les deux tuples, solutions de ce réseau (réduis à une unique contrainte) sont :*
$\langle x=1, y=2, z=3 \rangle$ *et* $\langle x=2, y=1, z=3 \rangle$.

Pour résoudre un problème de satisfaction de contraintes, il faut exhiber une solution c'est-à-dire un tuple qui satisfait toutes les contraintes : trouver une affectation de chaque variable à une valeur de son domaine. Pour ce faire, on utilise comme outil un solveur de contraintes. Dans le cadre de cette thèse, les différents algorithmes présentés ont été implémentés dans le solveur de contrainte choco3 [PFL14]. Les solveurs de contraintes reposent sur deux éléments centraux : la recherche et la propagation. La propagation de contraintes permet de couper l'espace de recherche via des algorithmes de détection d'incohérence et de retirer des valeurs incohérentes des domaines des variables à l'aide d'algorithmes de filtrage. L'exploration de l'espace de recherche permet d'énumérer les combinaisons de valeurs admissibles lorsque le raisonnement apporté par la propagation ne fixe pas toutes les variables.

2.2.2 La recherche

L'exploration de l'espace de recherche est un élément essentiel de la programmation par contraintes. Cette exploration se fait classiquement par "séparation et évaluation" : on effectue une séparation du problème que l'on ne sait pas résoudre comme tel en deux sous-problèmes plus petits jusqu'à obtenir un problème que l'on sait résoudre. L'espace de recherche est alors représenté par un arbre, chaque nœud étant un problème à résoudre et chacun de ses fils un de ses sous-problèmes.

Une première approche est d'évaluer si le réseau est satisfait lorsque qu'une valeur est affectée à chaque variable. L'espace de recherche est alors le produit cartésien des domaines. Classiquement, une méthode de séparation, ou de branchement, consiste à fixer une variable à une valeur pour le premier sous-problème et de retirer cette valeur du domaine de la variable pour le second sous-problème. On va alors, par un parcours en profondeur d'abord, créer un premier tuple, qui pourra être évalué. Si celui ci ne satisfait pas l'ensemble des contraintes du réseau, alors on évaluera le second sous-problème créé lors de la dernière séparation.

Dans l'exemple 2.5, les choix de branchement sont faits en suivant une heuristique statique prenant les variables dans l'ordre lexicographique (x, puis y, puis z) et en affectant la variable à sa plus petite valeur.

Exemple 2.5 (Parcours complet d'un espace de recherche)**.** *Soit un CSP portant sur 3 variables, x, y et z ; leur domaine respectif : {1,2}, {1,2}, {1,2,3}. Soit la contrainte* $\mathrm{AllDifferent}(x, y, z)$ *: toutes les variables doivent prendre des valeurs différentes.*

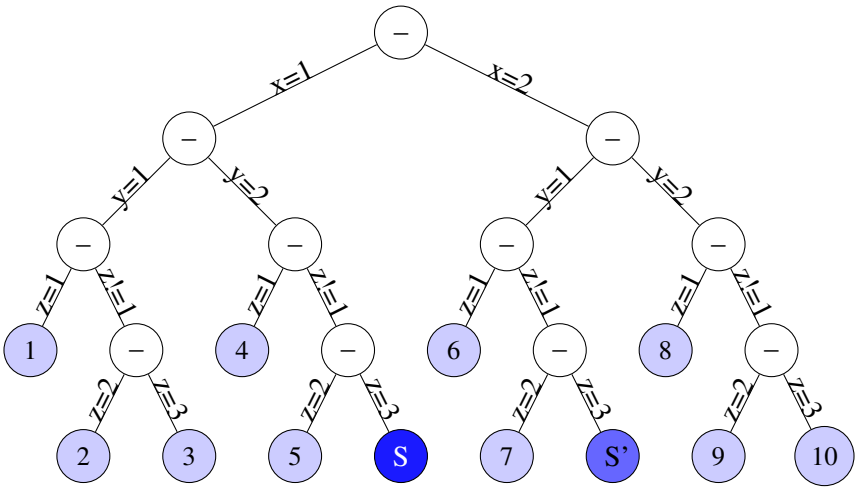


FIGURE 2.1 – Parcours complet de l'espace de recherche.

La figure 2.1 montre un parcours complet de l'espace de recherche. Les instanciations complètes sont représentées par les feuilles de l'arbre, en bleu clair pour les non cohérentes, en bleu foncé pour les deux solutions S et S' du problème.

Sans perte de généralité, on peut considérer que le solveur de contrainte cherche une première solution réalisable. Dans cet exemple, la recherche peut s'arrêter après avoir trouvé la solution S (en bleu foncé). Lorsqu'on résout un problème de satisfaction, ce résultat est suffisant : une solution réalisable a été trouvée. Si le problème est un problème de minimisation, après avoir trouvé une solution réalisable (de coût égal à $cost$), on ajoutera la contrainte imposant que le coût doit être inférieur à $cost$, la recherche de solution pouvant alors continuer, jusqu'à ce qu'aucune solution ne soit réalisable. La solution optimale est alors la dernière solution trouvée.

Le choix effectué à chaque séparation est primordial. On peut s'imaginer faire le bon choix à chaque nœud et par conséquent trouver une solution sans jamais s'être trompé, c'est-à-dire sans jamais avoir à réfuter une solution. Malheureusement, il n'existe pas d'heuristique assurant un

tel comportement dans le cas général. Guider la recherche le plus efficacement possible permet de réduire grandement le nombre de nœuds à parcourir. Il existe de multiples heuristiques de branchement comme par exemple first-fail [HE79], impact-based search [Ref04], dom/wdeg [BHLS04] ou plus récemment, activity-based search [MV12]. L'heuristique utilisée dans l'exemple 2.5 permet d'assurer un ordre de branchement statique : indépendant du domaine courant des variables et de leurs contraintes, et sans choix aléatoire. Nous l'utiliserons donc dans ce manuscrit pour tester nos algorithmes, sans risque de bruit dûs à des éléments extérieur à notre implémentation. La seconde heuristique que nous utiliserons lorsque l'efficacité devra être prise en compte sera first-fail, celle par défaut de choco3 [PFL14].

2.2.3 La propagation

Il n'est pas concevable en pratique de parcourir l'ensemble des affectations complètes lors de la recherche d'une solution, il faut donc doter l'outil de mécanismes permettant de réduire l'arbre de recherche.

En pratique, il est primordial que le solveur soit muni d'un mécanisme de vérification de validité de solution, de manière à ne pas considérer un tuple comme étant cohérent s'il ne l'est pas. Le solveur est donc doté de « checkeur », qui permettent de refuser un tuple s'il n'est pas cohérent. Ces algorithmes sont étendus de manière à pouvoir détecter en cours de recherche qu'aucune solution ne pourra être trouvée dans le sous-arbre courant. On parlera alors d'algorithme de détection d'incohérence.

Exemple 2.6 (Parcours d'un espace de recherche avec un algorithme de détection d'incohérence)**.** *Soit 3 variables, x, y et z ; leur domaine respectif : {1,2}, {1,2}, {1,2,3}. Soit la contrainte* $\mathrm{AllDifferent}(x, y, z)$.

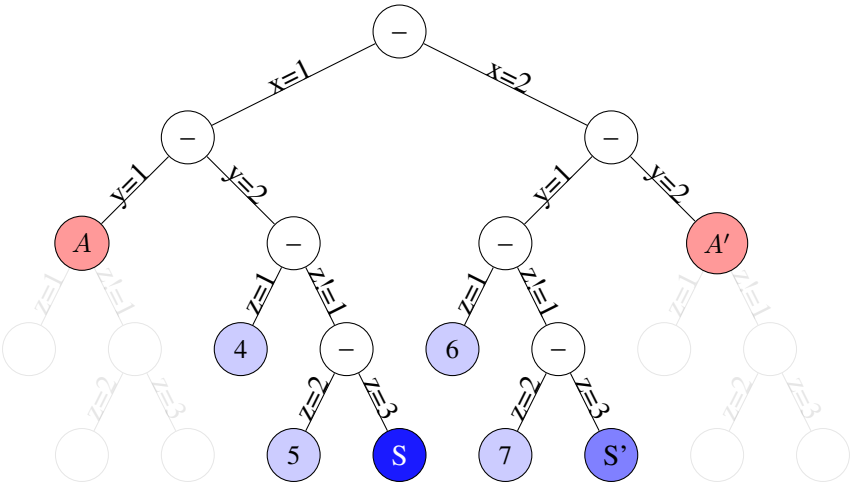


FIGURE 2.2 – Le parcours de l'espace de recherche est coupé grâce à un algorithme de détection d'incohérence de la contrainte AllDifferent.

La figure 2.2 montre un parcours de l'espace de recherche lorsque le solveur est doté d'un algorithme de détection d'incohérence. Au nœud A et A', les sous-arbres sont coupés car aucune solution ne peut y être trouvée. Les nœuds coupés par l'algorithme de détection d"incohérence sont matérialisés en transparent.

Dans l'exemple 2.6, dès les nœuds A et A' (en rouge) il est possible de détecter qu'il n'existe pas de solution dans cette branche, même si z n'est pas instancié, puisque $x = y$. Un algorithme de détection d'incohérence de la contrainte AllDifferent pourra alors, dès ce nœud, réfuter la dernière décision qui a été prise, ce qui évite de parcourir le sous-arbre.

Dans ce manuscrit, nous noterons une *règle de cohérence* pour une contrainte C sous la forme d'une implication : $C \implies N$. Si la condition nécessaire N n'est pas respectée alors la contrainte ne peut être respectée. Un raisonnement de détection d'incohérence cherche alors à prouver que la condition nécessaire ne peut être respectée.

La condition nécessaire utilisée dans l'exemple 2.6 est que si deux variables i, j sont instanciées, alors elle ne peuvent l'être à la même valeur. Cette condition nécessaire peut se formuler ainsi : $|\mathcal{D}(x_i)|>1 \vee |\mathcal{D}(x_j)|>1 \vee \mathcal{D}(x_i)\neq\mathcal{D}(x_j)$.

Pour réduire au maximum l'espace de recherche, on va aussi chercher à supprimer les valeurs qui ne peuvent être étendues à une solution ; on parlera de filtrage, effectué à l'aide d'algorithmes nommés *algorithme de filtrage*.

Exemple 2.7 (Parcours d'un espace de recherche avec un algorithme de filtrage)**.** *Soit 3 variables, x, y et z ; leur domaine respectif : [1,2], [1,2], [1,3]. Soit la contrainte* AllDifferent(x, y, z).

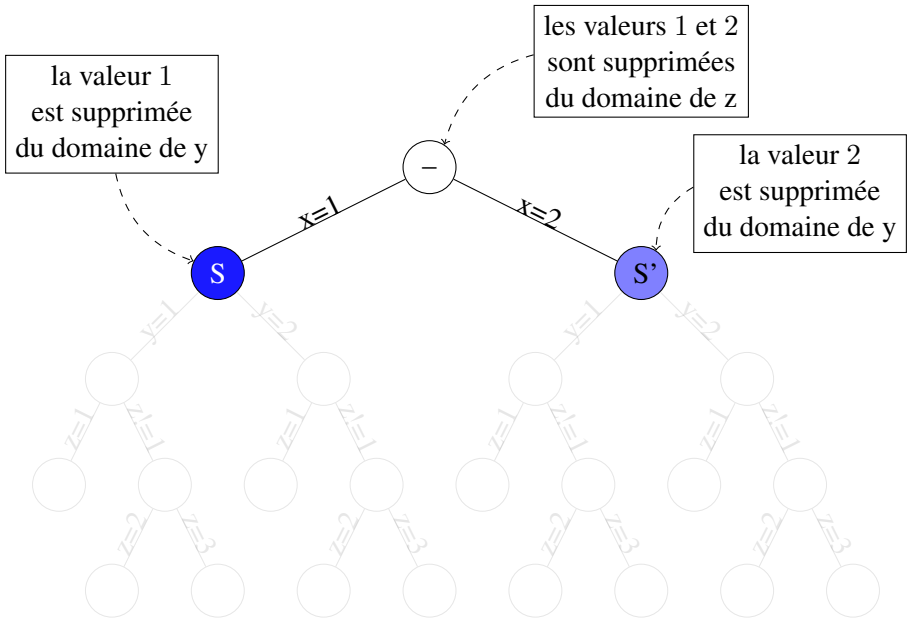


FIGURE 2.3 – Le parcours de l'espace de recherche est réduit grâce à un algorithme de filtrage de la contrainte AllDifferent.

La figure 2.3 montre un parcours de l'espace de recherche lorsque le solveur est doté d'un algorithme de filtrage. Au nœud racine il est possible de détecter que z ne peut prendre comme valeur 1 *ou* 2*, en effet x et y devant chacun prendre comme valeur* 1 *ou* 2*, aucune autre variable ne peut prendre ces valeurs sans violer la contrainte. Elles sont donc supprimées du domaine de z. Dès le premier choix réalisé, une solution est trouvée, car la variable y est aussi filtrée. Chaque variable est alors instanciée et le tuple est cohérent. Les nœuds non parcourus grâce à l'ajout de l'algorithme de filtrage sont matérialisés en transparent.*

La cohérence d'arc généralisée

Dans le cas où toutes les contraintes d'un CSP sont définies sur seulement deux variables, il est possible de représenter schématiquement le CSP par un graphe : chaque nœud étant une variable et chaque arc une contrainte. Réaliser la cohérence d'arc correspond alors à s'assurer que chaque contrainte est cohérente. Cette notion généralisée pour les contraintes de plus de deux variables est la GAC (de l'anglais Generalized Arc Consistency) [VSD95].

Définition 2.3 (GAC : Cohérence d'Arc Généralisé)**.** *Une contrainte c est arc-cohérente si et seulement si, pour toute variable $x_i \in \mathcal{X}(c)$ et toute valeur de son domaine $val \in \mathcal{D}(x_i)$, il existe un tuple $\tau = (v_1 \in \mathcal{D}(x_1), ..., v_i = val, ...v_n \in \mathcal{D}(x_n))$ satisfaisant la contrainte.*

Les cohérences aux bornes

La cohérence d'arc nécessite de stocker en mémoire toutes les valeurs possibles du domaine, ce qui peut s'avérer gourmand en espace mémoire ; de plus, la complexité d'un algorithme qui maintient cette propriété peut s'avérer trop élevée en pratique. Il existe donc d'autres cohérences plus faibles, relaxant la GAC. On peut limiter le raisonnement aux extrémités des domaines des variables : pour limiter le nombre de valeurs à vérifier : on vérifiera les valeurs extrêmes et l'on parlera alors de RC (pour Range Consistency en anglais) ; pour approximer la recherche du support à l'intervalle $[\underline{x}, \overline{x}]$ au lieu du domaine exact ; enfin les deux relaxations en même temps, on parlera alors de BC (Bound Consistency en anglais, ou cohérence aux bornes).

Définition 2.4 (BC : Cohérence aux bornes)**.** *Une contrainte c est borne-cohérente si et seulement si, pour toute variable $x_i \in \mathcal{X}(c)$ et chaque valeur extreme de son domaine $val \in \{\underline{x_i}, \overline{x_i}\}$ il existe un tuple $\tau = (v_1 \in [\underline{x_1}, \overline{x_1}], ..., v_i = val, ...v_n \in [\underline{x_n}, \overline{x_n}])$ qui satisfait la contrainte.*

Dans ce manuscrit, nous nous intéressons à des problèmes d'ordonnancement, nous porterons donc notre intérêt sur des algorithmes qui assurent la cohérence aux bornes. Par conséquent, nous utiliserons des variables bornées ($\mathcal{D}(x_i) = [\underline{x_i}, \overline{x_i}]$).

Les cohérences plus faibles que GAC ou BC

Parfois les problèmes sous-jacents des contraintes que l'on est amené à résoudre sont **NP-complets**. On ne pourra pas assurer retirer toutes les valeurs incohérentes en temps polynomial. En effet, assurer de retirer les valeurs incohérentes signifie que les valeurs restantes sont cohérentes, donc qu'il existe une solution, ce qu'il est impossible de faire en temps polynomial puisque le problème est **NP-complet**.

Parfois, la cohérence d'arc ou la cohérence aux bornes peuvent être réalisées en temps polynomial mais le coût opérationnel de l'algorithme est trop important, en comparaison avec la réduction de l'espace de recherche obtenue grâce au propagateur.

Dans toutes ces situations, il peut être intéressant de concevoir un algorithme plus léger. Lorsqu'un propagateur est défini, il est important de qualifier précisément le filtrage qu'il effectuera.

3

Ordonnancement et Robustesse

Sommaire

Maintenant que le contexte général de travail est posé, nous présentons les différentes parties de l'état de l'art qui seront étudiées en détail dans ce manuscrit. Dans un premier temps nous présenterons la contrainte Cumulative, puis nous étudierons différents raisonnements et algorithmes sur lesquelles nous travaillerons dans la Partie Contributions.

3.1 L'ordonnancement cumulatif

3.1.1 L'ordonnancement en programmation par contraintes

Baker définit dans [Bak74] les problèmes d'ordonnancement comme des problèmes d'allocation de ressources à des activités positionnées dans le temps. On notera la différence entre une activité et un intervalle par le fait qu'une activité consomme une certaine quantité de ressources. On parlera aussi de sa hauteur, en référence à la représentation schématique d'une activité. Dans cette thèse, nous considérons que cette demande est constante dans le temps.

Définition 3.1 (Activité). *Une activité a est représentée par un quadruplet de variables : sa hauteur h_a (h pour height en anglais), sa durée p_a (p pour processing time en anglais), sa date de début s_a (s pour start en anglais) et sa date de fin e_a (e pour end en anglais). Elle respecte sa contrainte d'intégrité : $s_a + d_a = e_a$.*

Exemple 3.1 (Activité). *La figure 3.1 illustre une activité* $a = \langle s_a = 2, p_a = 8, e_a = 10, h_a = 2 \rangle$.

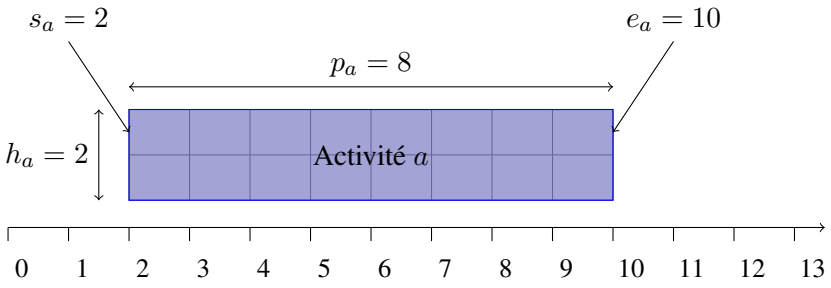


FIGURE 3.1 – Une instance d'activité de durée 8 commençant au point de temps 2.

La résolution d'un problème d'ordonnancement consiste à déterminer les dates de début et fin d'un ensemble d'activités. Il existe un grand nombre de problèmes différents tels que les problèmes disjonctifs ou cumulatifs, avec ou sans préemption, avec ou sans précédence. Dans le cadre de la programmation par contraintes, on peut noter le livre de Baptiste, Le Pape et Nuijten "*Constraint-Based Scheduling*" [BLN01] qui propose, outre l'étude d'un grand nombre de problèmes, des solutions à certains d'entre eux ; c'est le cas du problème d'ordonnancement à ressource cumulative (*CuSP* pour Cumulative Scheduling Problem).

3.1.2 La contrainte Cumulative

La contrainte Cumulative [AB93] décrit un problème d'ordonnancement cumulatif et non préemptif (où les activités doivent s'exécuter sans interruption). Dans ce cadre, le but est d'ordonnancer des activités dans le temps, sous contraintes de ressources. À tout moment où s'exécute une activité, la somme des consommations d'énergie des activités recoupant cet instant ne peut pas dépasser la capacité en ressources.

Définition 3.2 (Contrainte Cumulative(activités, capacité)). *Étant donnés un ensemble* $\mathcal{A}$ *de* n *activités* $\{a_1, \ldots, a_n\}$ *et une limite de consommation de ressource* capa*, la contrainte* Cumulative($\mathcal{A}$, capa) *est satisfaite si et seulement si les conditions d'intégrité des activités (1) et les conditions de capacité (2) sont vérifiées :*

$$\forall a \in \mathcal{A} : s_a + d_a = e_a \tag{1}$$

$$\forall t \in \mathbb{N} : \sum_{\substack{a \in \mathcal{A}, \\ t \in [s_a, e_a[}} h_a \leq \text{capa} \tag{2}$$

Exemple 3.2 (Cumulative). *La figure 3.2 donne la représentation visuelle de trois ordonnancements pour un problème cumulatif avec 4 activités et une capacité limite de 3 :*

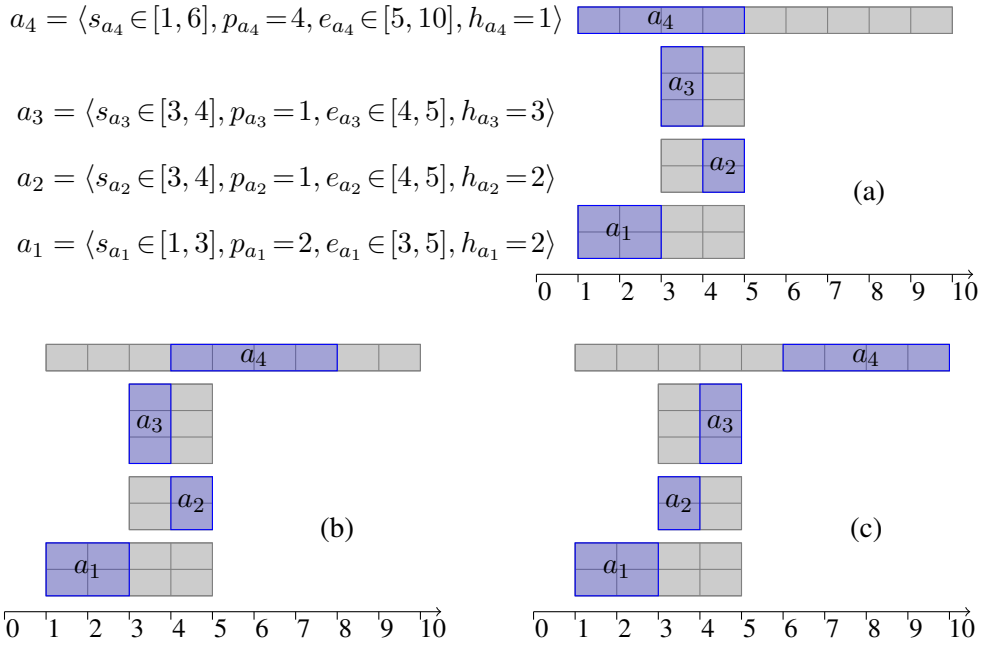


FIGURE 3.2 – Un problème cumulatif à 4 activités

L'ordonnancement illustré en (a) n'est pas valide, car au point de temps 3 la consommation est de 4, ce qui viole la contrainte de capacité. Les deux exemples (b) et (c) illustrent des positionnements valides.

3.1.3 Les filtrages de la contrainte Cumulative

Bien qu'une activité soit définie sur 4 variables : s, d, e, h les principaux raisonnements se placent dans le cas où la consommation et la durée sont fixées et positives et les débuts et fins de consommation sont représentés par des variables bornées. Dans la suite de ce manuscrit, nous nous plaçons dans cette hypothèse. On peut remarquer que, puisque la durée est fixe, s et e sont directement liées par la contrainte (1) d'intégrité d'une activité. Assurer la cohérence de $\underline{s}$ assure la cohérence de $\underline{e}$. De plus, le problème cumulatif est symétrique : l'image d'un problème vu d'un miroir est équivalent au problème lui-même (ou plus formellement en changeant la variable de début s et la variable de fin e par $s' = -e$ et $e' = -s$). Cette symétrie permet, avec un raisonnement pour la cohérence de la borne inférieure, d'assurer aussi la cohérence de la borne supérieure. Le plus souvent, les raisonnements et algorithmes ne présentent donc que le filtrage du début au plus tôt.

L'existence d'une solution d'un problème cumulatif est un problème **NP-complet** [GJ79], dans le cas ou toutes les variables sont bornées. Un algorithme assurant la consistance aux bornes assure qu'il existe une solution avec cette borne. Par conséquent, on ne peut avoir un algorithme polynomial assurant la BC. C'est pourquoi, différents raisonnements relaxant Cumulative ont été introduits. Dans cette section, nous présentons les principaux raisonnements tel qu'ils sont couramment utilisés, par ordre croissant de complexité.

Nous commencerons par présenter en détail le raisonnement Time-Table [Lah82] et l'algorithme de Letort *et al.* [LCB14] que nous allons adapter pour le cas robuste dans le chapitre 5. puis nous présenterons les raisonnements Edge-Finding [Nui94], Time-Table-Edge-Finding [Vil11] et le Raisonnement Energétique [ELT89]. Ces raisonnements utilisent des notions similaires de calcul de consommation pour lesquels nous proposerons une uniformisation dans la section 4. Finalement, nous proposerons une caractérisation précise des intervalles qu'il est intéressant de considérer pour le raisonnement énergétique.

Time-Table

Les premiers algorithmes de filtrage pour la contrainte cumulative sont basés sur la notion de partie obligatoire. Le terme Time-Table se réfère à la structure de données qui explicite la consommation et la disponibilité en ressource dans le temps, Lahrichi [Lah79].

La partie obligatoire représente le profil de consommation de l'activité dans le temps. Elle vaut 0 à tout point de temps si $\underline{e_a} \leq \overline{s_a}$: il n'y a pas de partie obligatoire. Ou bien elle vaut la hauteur h_a de l'activité a durant l'intervalle $[\overline{s_a}, \underline{e_a}[$ et 0 en dehors.

Dans la suite, nous noterons directement la partie obligatoire comme étant sur l'intervalle $[\overline{s_a}, \underline{e_a}[$, considérant que cela définit l'ensemble vide si un tel intervalle n'existe pas.

Exemple 3.3 (Partie obligatoire)**.** *La figure 3.3 montre le placement au plus tôt, le placement au plus tard et la partie obligatoire d'une activité* $a = \langle s_a = [1,5], p_a = 8, e_a = [9,13], h_a = 2 \rangle$.

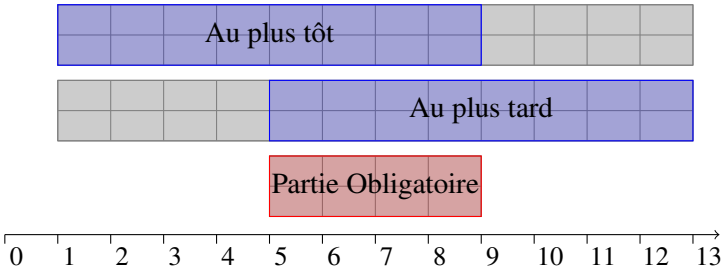


FIGURE 3.3 – Partie obligatoire d'une activité.

En agrégeant les parties obligatoires de toutes les activités on obtient le profil cumulé des parties obligatoires. Il représente la somme des parties obligatoires à tout point de temps t : $\sum_{a \in \mathcal{A}, t \in [\overline{s_a}, \underline{e_a}[} h_a$.

Propriété 3.1 (Règle de cohérence du raisonnement Time-Tabling)**.** *La règle de cohérence Time-Table assure la propriété qu'à tout moment le profil de partie obligatoire ne dépasse pas la capacité :*

$$\text{Cumulative}(\mathcal{A}, \text{capa}) \implies \forall t, \sum_{a \in \mathcal{A},\ t \in [\overline{s_a}, \underline{e_a}[} h_a \leq \text{capa} \tag{3}$$

Exemple 3.4 (Profil des parties obligatoires)**.** *Exemple d'un profil des parties obligatoires pour un problème cumulatif.*

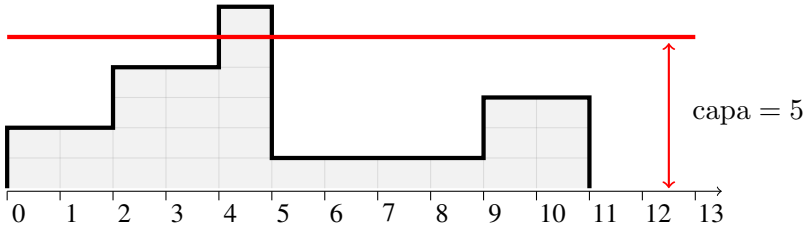


FIGURE 3.4 – Profil des parties obligatoires.

Au point de temps 0 la somme des hauteurs des parties obligatoires est de 2. La contrainte de capacité est alors respectée.
Au point de temps 2 la somme des hauteurs des parties obligatoires est de 4. La contrainte de capacité est alors respectée.
Au point de temps 4 la somme des hauteurs des parties obligatoires est de 6. La contrainte de capacité est alors violée.

Afin de réaliser un filtrage partiel des bornes, il est nécessaire de fournir des règles pour identifier si l'affectation de s_a à ses valeurs extrêmes ne viole pas la contrainte de capacité. La propriété maintenue par l'algorithme de filtrage Time-Table (limité au minimum) est alors :

Propriété 3.2 (Propriété après le filtrage du raisonnement Time-Table [LCB14])**.**

$$\forall a \in \mathcal{A}, \forall t \in [\underline{s_a}, \underline{e_a}[, h_a + \sum_{b \in \mathcal{A} \setminus a, t \in [\overline{s_b}, \underline{e_b}[} h_b \leq \text{capa} \tag{4}$$

Dans [LCB14] Letort *et al.* proposent des algorithmes de balayage pour le raisonnement Time-Table. Le principe sous-jacent de ces algorithmes de balayage est d'imaginer qu'une droite est déplacée sur l'axe temporel, s'arrêtant à quelques évènements temporels spécifiques. Les opérations effectuées sont limitées à des objets coupant la droite de balayage chaque fois qu'elle s'arrête. L'algorithme a effectué l'ensemble des ajustements une fois que la droite est passée sur toutes les activités.

Nous montrons dans la suite de cette section le fonctionnement détaillé de l'algorithme *dynamic sweep* [LCB14, section 3], filtrant le début au plus tôt pour une contrainte Cumulative. Nous en faisons une adaptation dans un cadre robuste dans le chapitre 5.

L'ensemble des évènements va permettre de maintenir les informations sur le profil et sur la liste des activités actives (c'est à dire les activités que l'ont peut encore filtrer). Un évènement est un n-uplet composé de quatre champs : son type, l'activité correspondante, sa date d'effet et un éventuel incrément. Les évènements sont triés par ordre croissant de dates. Dans cet algorithme, il existe 4 types d'évènements :

- $\langle SCP \quad , a, \overline{s_a}, -h_a \rangle$: Le début de partie obligatoire, si elle existe ($\overline{s_a} < \underline{e_a}$),
- $\langle CCP \quad , a, \overline{s_a}, \quad 0 \quad \rangle$: Le début de partie obligatoire, si elle n'existe pas ($\overline{s_a} \geq \underline{e_a}$),
- $\langle ECPD, a, \underline{e_a}, +h_a \rangle$: La fin de partie obligatoire, si elle existe ($\overline{s_a} < \underline{e_a}$),
- $\langle PR \quad , a, \underline{s_a}, \quad 0 \quad \rangle$: Le début au plus tôt de l'activité, si elle n'est pas fixée ($\underline{s_a} < \overline{s_a}$).

Dans le cas où une activité qui avait une partie obligatoire se voit filtrée, sa fin de partie obligatoire est repoussée et les évènements correspondants sont dynamiquement décalés dans le temps. L'évènement ECPD est donc dynamique.
Certains évènements sont dynamiquement créés : une activité qui n'avait pas de partie obligatoire peut se voir filtrée au point de créer une partie obligatoire ; les évènements de début et de fin de partie obligatoire (SCP et ECPD) sont alors créés à la volée.

En parcourant ces évènements, la droite de balayage maintient dynamiquement les informations suivantes :

1) Les points de temps de l'évènement actuel : δ et du suivant : $\delta_{next} > \delta$,
2) Le gap entre la quantité de ressources disponible et consommée pour l'intervalle $[\delta, \delta_{next}[$: $gap = \text{capa} - \sum_{a \in \mathcal{A}, \delta \in [\overline{s_a}, \underline{e_a}[} h_a$
3) Les activités sont dites actives lorsqu'elles peuvent être ordonnancées au point de temps δ. Elles sont gérées par deux structures de données distinctes :
   - L'ensemble des activités qui sont en conflit $h\_conflict$.
   - L'ensemble des activités qui ne sont pas en conflit h_check.

Exemple 3.5 (Illustration des évènement initiaux de l'algorithme SWEEP_MIN [LCB14])**.**
La figure 3.5 montre les évènements associés à trois variables a_1, a_2 et a_3 :

$a_3 = \langle s_{a_3} = [1,3], p_{a_3} = 3, e_{a_3} = [1,6], h_{a_3} = 1 \rangle$

$a_2 = \langle s_{a_2} = [2,2], p_{a_2} = 2, e_{a_2} = [4,4], h_{a_2} = 1 \rangle$

$a_1 = \langle s_{a_1} = [3,5], p_{a_1} = 3, e_{a_1} = [6,8], h_{a_1} = 1 \rangle$

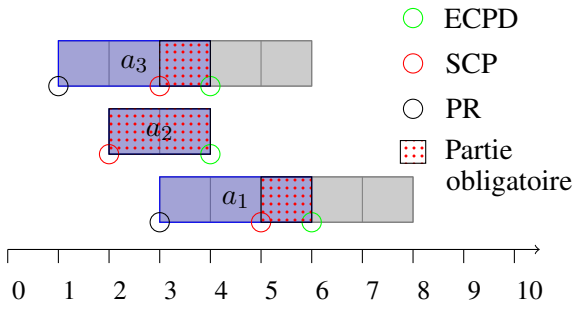


FIGURE 3.5 – Évènements associés à 3 activités

Huit évènements sont initialement créés par l'algorithme :
Trois pour l'activité a_1 :

- $\langle PR,\ a_1,\ \underline{s_{a_1}},\ 0\rangle$, $\langle SCP,\ a_1,\ \overline{s_{a_1}},\ -h_{a_1}\rangle$, $\langle ECPD,\ a_1,\ \underline{e_{a_1}},\ h_{a_1}\rangle$

Deux pour l'activité a_2 :

- $\langle SCP,\ a_2,\ \overline{s_{a_2}},\ -h_{a_2}\rangle$, $\langle ECPD,\ a_2,\ \underline{e_{a_2}},\ h_{a_2}\rangle$

Trois pour l'activité a_3 :

- $\langle PR,\ a_3,\ \underline{s_{a_3}},\ 0\rangle$, $\langle SCP,\ a_3,\ \overline{s_{a_3}},\ -h_{a_3}\rangle$, $\langle ECPD,\ a_3,\ \underline{e_{a_3}},\ h_{a_3}\rangle$

Le gap est dynamiquement modifié aux évènements de début et de fin de partie obligatoire : un évènement SCP reflète un début de partie obligatoire. Le gap est donc diminué de la hauteur de l'activité. Un évènement ECPD reflète la fin de partie obligatoire ; si celle-ci est définitive, alors le gap est augmenté de la hauteur de l'activité.

L'évènement $\langle PR, a, \underline{s_a}, 0\rangle$ montre à l'algorithme que l'activité a doit est active : à partir de ce moment-là, un filtrage peut être détecté. La gestion des évènements PR doit se faire en dernier : le calcul du gap dans l'intervalle à venir doit avoir été effectué. Pour cela une liste $newTasksToPrune$ garde en mémoire les nouvelles activités, qui seront gérées lorsque tous les évènements au temps δ seront effectués. Les activités actives sont placées dans un tas en fonction de leur hauteur : les activités qui ne respectent pas la propriété 3.2 sont dans $h\_conflict$; dès que le gap est assez grand pour que l'activité puisse être placée, l'activité peut rejoindre h_check. Si la droite de balayage dépasse la fin au plus tôt ($\underline{s_a} + p_a \geq \delta$) sans que le gap soit repassé en dessous de la hauteur de l'activité, alors la date au plus tôt est validée et l'activité n'est plus active. Pour mémoriser qu'une activité a ne sera plus modifiée un booléen $evup[a]$ est mis à $vrai$.

L'algorithme proposé par Letort *et al.* s'articule autour d'un algorithme principal : SWEEP_MIN. Celui-ci gère les évènements et organise le balayage. Il délègue le filtrage fait sur un intervalle à l'algorithme FILTER_MIN et la synchronisation des évènements à l'algorithme SYNCHRONIZE.

Algorithm 1 SWEEP_MIN de Letort *et al.* [LCB14]

1: **fonction** SWEEP_MIN() : *bool*
2: `[INITIALISATION]`
3: $h\_events \leftarrow$ évènements pour $\underline{s_a}$, $\overline{s_a}$, $\underline{e_a}$ pour chaque activité a
4: $h\_check, h\_conflict \leftarrow \emptyset$; $newTasksToPrune \leftarrow \emptyset$
5: **pour** $a = 0$ to $n-1$ **faire**
6: $evup_a \leftarrow (\underline{s_a} = \overline{s_a})$; $mins_a \leftarrow \underline{s_a}$
7: **fin pour**
8: $\delta \leftarrow get(h\_events)$; $\delta_{next} \leftarrow \delta$; $gap \leftarrow$ capa
9: `[BOUCLE PRINCIPALE]`
10: **tant que** $\neg vide(h\_events)$ **faire**
11: `[BALAYAGE]`
12: **si** $\delta \neq \delta_{next}$ **alors**
13: **tant que** $\neg vide(newTasksToPrune)$ **faire**
14: extraire une activité a de $newTasksToPrune$
15: **si** $h_a > gap$ **alors** ajouter $\langle h_a, t\rangle$ dans $h\_conflict$
16: **sinon si** $p_a > \delta_{next} - \delta$ **alors** {ajouter $\langle h_a, a\rangle$ dans h_check ; $mins_a \leftarrow \delta$;}
17: **sinon** $evup_a \leftarrow Vrai$
18: **fin tant que**
19: **si** $\neg$FILTER_MIN(δ, δ_{next}) **alors retourne** Faux **fin si**
20: $\delta \leftarrow \delta_{next}$
21: **fin si**
22: `[GESTION DE L'EVENEMENT COURANT]`
23: $\delta \leftarrow$ SYNCHRONIZE(δ)
24: extraire $\langle type, a, \delta, dec\rangle$ de h_events
25: **si** $type = SCP \vee type = ECPD$ **alors** $gap \leftarrow gap + dec$
26: **sinon si** $type = PR$ **alors** $newTasksToPrune \leftarrow newTasksToPrune \cup \{a\}$ **fin si**
27: `[GETTING NEXT EVENT]`
28: **si** $vide(h_{event})$ **alors retourne** FILTER_MIN$(\delta, +\infty)$
29: **sinon** $\delta_{next} \leftarrow$ SYNCHRONIZE(δ)
30: **fin tant que**
31: **retourne** Vrai
32: **fin fonction**

Algorithm 2 FILTER_MIN de Letort *et al.* [LCB14]

1: **fonction** FILTER_MIN(δ, δ_{next}) : boolean
2: `[VERIFICATION DE LA COHERENCE]`
3: **si** $gap < 0$ **alors retourne** Faux
4: `[MISE A JOUR DES ACTIVITES DE` h_check`]`
5: **tant que** $\neg vide(h\_check) \wedge (vide(h\_events) \vee get(h\_check) > gap)$ **faire**
6: extraire $\langle h_a, a \rangle$ de h_check
7: **si** $\delta \geq \overline{s_a} \vee \delta - mins_a \geq p_a \vee vide(h\_events)$ **alors**
8: $adjust\_min\_var(s_a, mins_a); adjust\_min\_var(e_a, mins_a + p_a);$
9: **si** $\neg evup_a$ **alors** {m.à.j. des évén. de partie obligatoire de a ; $evup_a \leftarrow Vrai$;}
10: **sinon**
11: ajout de $\langle h_a, t \rangle$ dans $h\_conflict$
12: `[MISE A JOUR DES ACTIVITES DE` $h\_conflict$`]`
13: **tant que** $\neg vide(h\_conflict) \wedge get(h\_conflict) \leq gap$ **faire**
14: extraire $\langle h_a, a \rangle$ de $h\_conflict$
15: **si** $\delta \geq \overline{s_a}$ **alors**
16: $adjust\_min\_var(s_a, \overline{s_a}); adjust\_min\_var(e_a, \overline{e_a});$
17: **si** $\neg evup_a$ **alors** {m.à.j. des évén. de partie obligatoire de a ; $evup_a \leftarrow Vrai$;}
18: **sinon**
19: **si** $\delta_{next} - \delta \geq p_a$ **alors**
20: $adjust\_min\_var(s_a, \delta); adjust\_min\_var(e_a, \delta + p_a);$
21: **si** $\neg evup_a$ **alors** {m.à.j. des évén. de partie obligatoire de a ; $evup_a \leftarrow Vrai$;}
22: **sinon**
23: add $\langle h_a, a \rangle$ into h_check ; $mins_a \leftarrow \delta$;
24: **retourne** Vrai

Pour éviter des répétitions, Letort *et al.* ont choisi de noter l'instruction "m.à.j. des évén. de partie obligatoire" (lignes 9, 17 et 21 de l'algorithme FILTER_MIN) ; cette instruction met à jour les évènements de partie obligatoire suite à l'ajustement de l'activité.
L'algorithme est découpé en 3 parties :

- `[VERIFICATION DE LA COHERENCE]`

  Si le gap est négatif, alors l'algorithme détecte une incohérence, et renvoie Faux.

- `[MISE A JOUR DES ACTIVITES DE` h_check`]`

  Toutes les activités de h_check dont la hauteur est supérieure au gap sont extraites. Si l'activité peut être placée avant δ ($mins_a + p_a \leq \delta$), alors cette position est validée, et on met à jour son début au plus tôt. Si la droite de balayage a dépassé le début au plus tard de l'activité ($\delta \geq \overline{s_a}$), alors on peut aussi valider la position $mins$. Si la droite de balayage a dépassé tous les évènements, alors on peut aussi valider la position $mins$. Sinon, l'activité est en conflit, et l'on place l'activité dans $h\_conflict$.

- `MISE A JOUR DES ACTIVITES DE` $h\_conflict$`]`

  Toutes les activités de $h\_conflict$ qui ne sont plus en conflit sont extraites. Si la droite de balayage n'est pas située avant le début au plus tard de a ($\delta \geq \overline{s_a}$) alors nous savons que l'activité a n'a pas pu être placée avant son début au plus tard. Sinon on compare la durée de a avec la taille de l'intervalle à venir, et l'on décide si l'on valide la position ou si l'on place a dans h_check.

Algorithm 3 SYNCHRONIZE de Letort *et al.* [LCB14]

```
1:  fonction SYNCHRONIZE(δ) : entier
2:      [MISE A JOUR DU PROCHAIN EVENEMENT]
3:      repeat
4:          si vide(h_events) alors retourne −∞
5:          sync ← Vrai ; ⟨type, t, date, dec⟩ ← consulter premier évènement de h_events ;
6:          [GESTION DE L'EVENEMENT DYNAMIQUE (ECPD)]
7:          si type = ECPD ∧ ¬evup_a alors
8:              si t ∈ h_check alors mise à jour de l'évènement à la date mins_a + p_a
9:              sinon mise à jour de l'évènement à la date s̄_a + p_a
10:             evup_a ← Vrai ; sync ← Faux ;
11:         [GESTION DE L'EVENEMENT CONDITIONNEL (CCP)]
12:         sinon si type = CCP ∧ ¬evup_a ∧ date = δ alors
13:             si t ∈ h_check ∧ mins_a + p_a > δ alors
14:                 ajout de ⟨SCP, t, δ, −h_a⟩ et ⟨ECPD, t, mins_a + p_a, h_a⟩ dans h_events
15:             sinon si t ∈ h_conflict alors
16:                 ajout de ⟨SCP, t, δ, −h_a⟩ et ⟨ECPD, t, ē_a, h_a⟩ dans h_events
17:             evup_a ← Vrai ; sync ← Faux ;
18:     until sync
19:     retourne date
```

La gestion des évènements dynamiques et conditionnels se fait via l'algorithme SYNCHRONIZE. Il met à jour, si besoin, le prochain évènement dans h_events.
L'algorithme est découpé en 3 parties :

- `[MISE A JOUR DU PROCHAIN EVENEMENT]`

  Tant que le premier évènement n'est pas synchronisé (c'est à dire à jour), on doit le mettre à jour.

- `[GESTION DE L'EVENEMENT DYNAMIQUE (ECPD)]`

  Un évènement du type ECPD doit être mis à jour si l'activité a correspondante est dans h_check ou $h\_conflict$. Si l'activité est dans h_check alors on peut valider la date $mins$, sinon l'activité ne peut commencer avant son début au plus tard car la condition de la ligne 15 de l'algorithme FILTER_MIN est valide. L'évènement est alors repoussé pour correspondre à la nouvelle date.

- `[GESTION DE L'EVENEMENT CONDITIONNEL (CCP)]`

  Lorsque l'évènement CCP est balayé, on vérifie si une partie obligatoire a été créée. Si $evup_a$ est à *faux*, alors l'activité est dans h_check ou dans $h\_conflict$. Si l'activité est dans h_check et qu'une partie obligatoire est créée ($mins_a + p_a \geq \delta$), alors les évènements correspondants sont créés (ligne 14). Si l'activité est dans $h\_conflict$, alors le placement au plus tard est validé (ligne 16).

Edge-Finding

Le raisonnement Edge-Finding est un raisonnement sur des ensembles d'activités. Il a été proposé par Pinson pour des problèmes disjonctifs (Job-Shop) [Pin88]. Il a été adapté pour le cas cumulatif, et largement utilisé par Nuijten dans sa thèse [Nui94]. Chaque ensemble d'activités $\mathcal{U} \subseteq \mathcal{A}$ définit un intervalle, appelé *intervalle d'activités* (*task interval* en anglais) sur lequel raisonner. Le raisonnement Edge-Finding se base sur un concept d'énergie. L'énergie d'une activité a représente la charge de travail (workload en anglais) noté w_a. Il se calcule simplement : $w_a = h_a \times p_a$. On compare alors l'énergie disponible et utilisée dans l'intervalle défini par $\mathcal{U}$: $[s_\mathcal{U}, e_\mathcal{U}[$ avec
$s_\mathcal{U}$ le début au plus tôt de l'ensemble d'activités $\mathcal{U}$: $s_\mathcal{U} = \min_{i \in \mathcal{U}}(\underline{s_i})$ et
$e_\mathcal{U}$ la fin au plus tard de l'ensemble d'activités $\mathcal{U}$: $e_\mathcal{U} = \max_{i \in \mathcal{U}}(\overline{e_i})$.
On appelle un intervalle $[s_\mathcal{U}, e_\mathcal{U}[$, défini par l'ensemble d'activités $\mathcal{U}$, un intervalle d'activités (*Task Interval* en anglais).

L'objectif du filtrage Edge-Finding est de trouver des relations de précédence entre activités : c'est-à-dire établir le fait qu'une activité se termine après un ensemble d'autres activités. La notion de précédence dans le cas du raisonnement Edge-Finding est noté $\mathcal{U} \lessdot a$; elle met en relation la fin d'une activité avec la fin d'un ensemble d'activités :

Notation 3.1 (Relation de précédence du raisonnement Edge-Finding : $\lessdot$)**.** *La relation "fini après" notée $\lessdot$ signifie qu'une activité a se termine après toutes les activités d'un ensemble $\mathcal{U}$:*

$$\mathcal{U} \lessdot a \quad \Leftrightarrow \quad \forall i \in \mathcal{U},\ e_i \leq e_a \tag{5}$$

Le filtrage s'effectue en deux temps, il faut dans un premier temps détecter les relations de précédence, pour ensuite effectuer le filtrage leur étant associé.

Propriété 3.3 (Détection de précédences du raisonnement Edge-Finding)**.**

$$\text{capa} \times \left(e_\mathcal{U} - s_{\mathcal{U}\cup\{a\}}\right) < \sum_{i \in \mathcal{U}\cup\{a\}} w_i \implies \mathcal{U} \lessdot a \tag{6}$$

La relation de précédence détectée est entre l'activité a et toutes les activités de $\mathcal{U}$. Elle peut donc, aussi, être appliquée à l'ensemble des sous-ensembles de $\mathcal{U}$: $\mathcal{U} \lessdot a \implies \forall \Omega \subseteq \mathcal{U}, \Omega \lessdot a$.

Exemple 3.6 (Détection de précédences de la règle Edge-Finding)**.**
Soit la contrainte Cumulative$(\{a_1, a_2, a_3, a_4\}, 4)$ *illustrée dans la figure 3.6.*

$a_4 = \langle s_{a_4} \in [1,6], p_{a_4} = 4, e_{a_4} \in [5,10], h_{a_4} = 1 \rangle$

$a_3 = \langle s_{a_3} \in [3,4], p_{a_3} = 1, e_{a_3} \in [4,5], h_{a_3} = 3 \rangle$

$a_2 = \langle s_{a_2} \in [3,4], p_{a_2} = 1, e_{a_2} \in [4,5], h_{a_2} = 2 \rangle$

$a_1 = \langle s_{a_1} \in [1,3], p_{a_1} = 2, e_{a_1} \in [3,5], h_{a_1} = 2 \rangle$

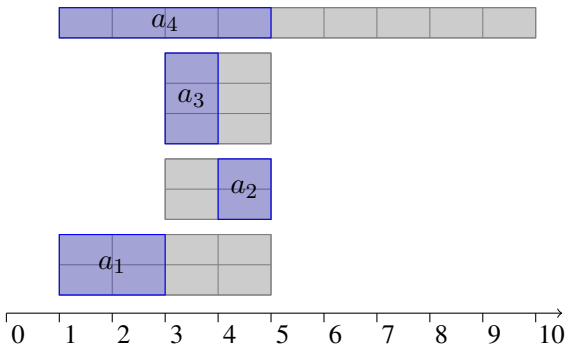


FIGURE 3.6 – Un problème cumulatif à 4 activités.

Le raisonnement Edge-Finding détecte que l'activité a_4 doit finir après les activités de $\mathcal{U} = \{a_1, a_2, a_3\}$.
En effet : $\text{capa} = 3$, $e_\mathcal{U} = 5$, $s_{\mathcal{U}\cup\{a_4\}} = 1$ *et*
$\sum_{i \in \mathcal{U}\cup\{a_4\}} w_i = (2*2) + (2*1) + (3*1) + (1*5) = 13 > 12 = 3*(5-1)$.
Donc, par la propriété 3.3, nous avons $\{a_1, a_2, a_3\} \lessdot a_4$.

De cette relation de précédence une nouvelle borne pour le début au plus tôt de l'activité est calculée. Nous savons que l'activité doit finir après la fin de toutes les activités de l'ensemble. Nuijten propose donc de calculer l'énergie qui sera nécessairement placée avant l'activité et repoussera alors son début au plus tôt. Cette énergie est la différence entre l'énergie consommée dans l'intervalle : $\sum_{i\in\mathcal{U}} w_i$ et celle pouvant être placée en parallèle de l'activité :$(\text{capa} - h_a)\times(e_\mathcal{U} - s_\mathcal{U})$. C'est la notion de Reste pour le raisonnement Edge-Finding :

Définition 3.3 (Reste pour le raisonnement Edge-Finding)**.** *Soit un ensemble d'activités $\mathcal{U} \subset \mathcal{A}$ et une activité $a \notin \mathcal{U}$.*

$$\text{Reste}(\mathcal{U}, a) = \left(\sum_{i\in\mathcal{U}} w_i\right) - (\text{capa} - h_a)\times(e_\mathcal{U} - s_\mathcal{U}) \tag{7}$$

Exemple 3.7 (Notion de Reste pour le raisonnement Edge-Finding)**.** *Soit le problème cumulatif de l'exemple 3.6. La figure 3.7 illustre la notion de* Reste *pour l'ensemble $\mathcal{U} = \{a_1, a_2, a_3\}$ et l'activité a_4.*

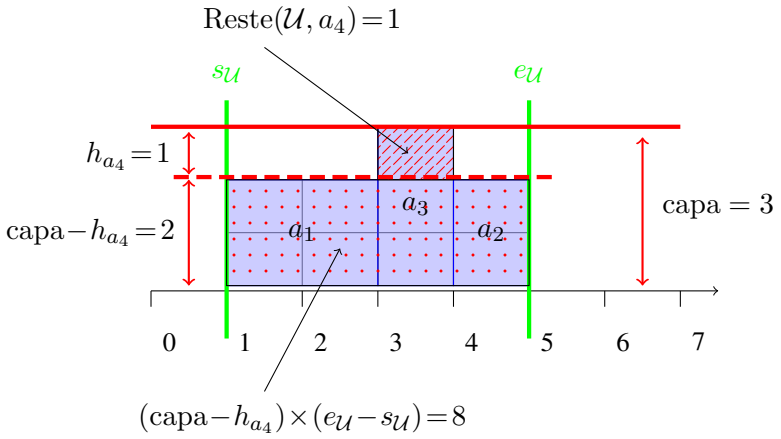


FIGURE 3.7 – La notion de Reste pour le raisonnement Edge-Finding.

L'énergie qui peut être placée en parallèle de l'activité a_4 (la zone en pointillé sur la figure 3.7) est ici $(\text{capa}-h_{a_4})\times(e_\mathcal{U}-s_\mathcal{U}) = (3-1)\times(5-1) = 8$. *Le* Reste*, l'énergie que l'on va forcement devoir placer avant l'activité (la zone hachurée sur la figure 3.7), est donc* $9-8 = 1$.

Le Reste représente donc la quantité d'énergie qui ne peut être placée sans repousser l'activité. Si cette valeur est positive, alors l'activité peut être repoussée, le début au plus tôt de l'activité est alors au moins $s_\Omega + \frac{1}{h_a}\times\text{Reste}(\Omega, a)$.

La règle de filtrage Edge-Finding donnée par Nuijten dans sa thèse est alors :

Propriété 3.1 (Règle de filtrage pour le raisonnement Edge-Finding [Nui94])**.**

$$\mathcal{U} \lessdot a \implies \forall\Omega \subseteq \mathcal{U},\ \text{Reste}(\Omega, a) > 0 \implies s_a \geq s_\Omega + \frac{1}{h_a}\times\text{Reste}(\Omega, a) \tag{8}$$

Exemple 3.8 (Filtrage pour le raisonnement Edge-Finding)**.** *Soit le problème cumulatif de l'exemple 3.6 (page 36). Une relation de précédence a été détectée entre l'ensemble* $\mathcal{U} = \{a_1, a_2, a_3\}$ *et l'activité* a_4 *:* $\mathcal{U} \ll a_4$.
Le filtrage déduit de l'ensemble $\Omega = \mathcal{U}$ *est montré par la figure 3.8.*

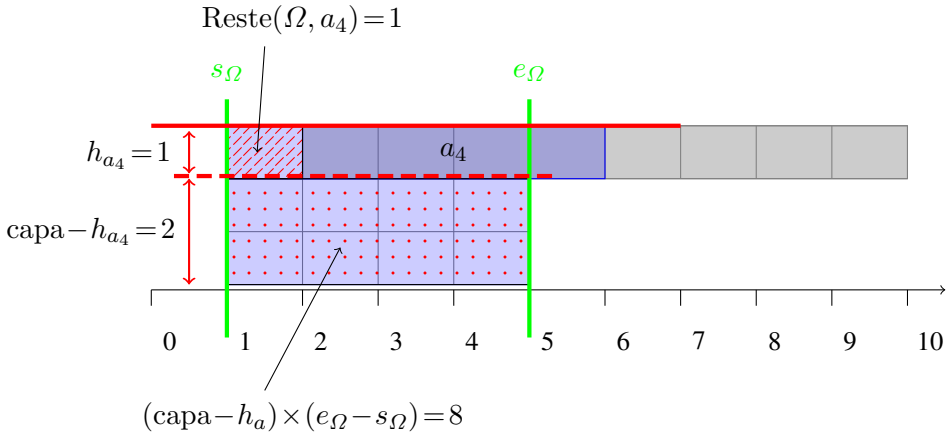


FIGURE 3.8 – Placement au plus tôt d'une activité pour le raisonnement Edge-Finding.

Le Reste *est ici de* 1. *Au mieux nous devons placer une énergie au début de l'intervalle, et repousser d'autant l'activité* a_4. *Le placement au plus tôt de* a_4 *est alors, comme exprimé par le théorème 3.1, de* $s_\Omega + \text{Reste}(\Omega, a_4)/h_{a_4} = 1 + 1/1 = 2$.

Mais la détection de la relation de précédence $\{a_1, a_2, a_3\} \ll a_4$ *signifie que* $\{a_2, a_3\} \ll a_4$. *Le filtrage déduit de l'ensemble* $\Omega_2 = \{a_2, a_3\}$ *est montré par la figure 3.9.*

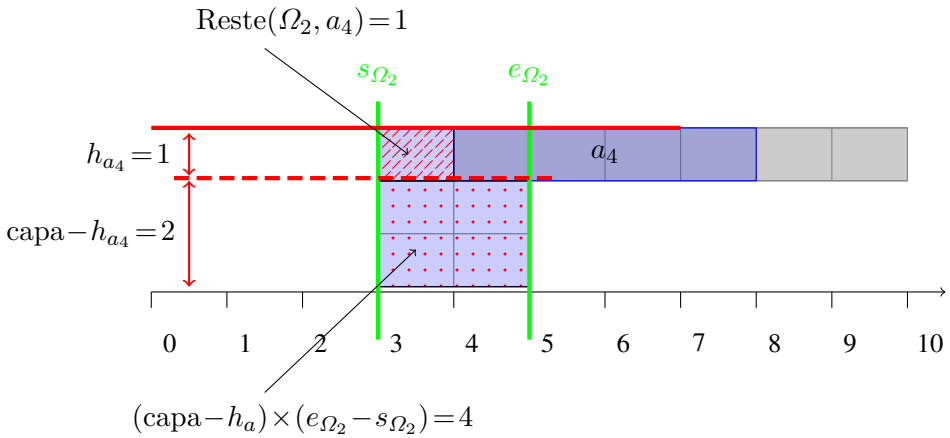


FIGURE 3.9 – Amélioration du placement au plus tôt d'une activité pour le raisonnement Edge-Finding grâce à un sous-ensemble d'activités.

Le placement au plus tôt de a_4 *est alors de* $s_{\Omega_2} + \text{Reste}(\Omega_2, a_4)/h_{a_4} = 3 + 1/1 = 4$. *Celle exemple montre l'importance d'appliquer la règle de filtrage aux sous-ensembles de* $\mathcal{U}$, *lorsqu'une relation de précédence est détectée entre* $\mathcal{U}$ *et une activité.*

On remarquera que dans l'exemple 3.6, bien que le filtrage soit plus grand pour l'ensemble $\{a_2, a_3\}$, le raisonnement Edge-Finding ne peut directement détecter que $\{a_2, a_3\} \lessdot a_4$ sans la détection de la précédence $\{a_1, a_2, a_3\} \lessdot a_4$.

La règle de filtrage Edge-Finding peut être utilisée dès qu'une relation de précédence est détectée. Il est donc possible de renforcer le filtrage en ajoutant d'autre règles de détection de précédences. De nombreuses règles de détection existent, ajoutées dans les différents algorithmes de la littérature. La plus utilisée est la règle de détection Extended-Edge-Finding, introduite par Nuijten [Nui94].

Reprenons l'exemple 3.6 (page 36) dans lequel nous modifions l'activité a_4 :

Exemple 3.9 (Détection de précédences de la règle Extended-Edge-Finding)**.** *La figure 3.10 illustre l'exemple 3.6 auquel seule l'activité a_4 a été modifiée.*

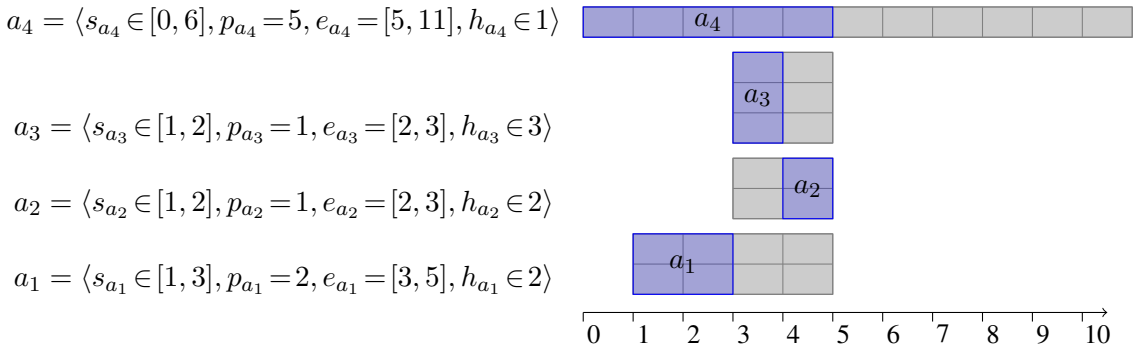


FIGURE 3.10 – Un problème cumulatif à 4 activités.

Alors que le problème exposé dans l'exemple 3.9 est strictement plus contraint que dans l'exemple 3.6, le raisonnement Edge-Finding ne peut détecter que l'activité a_4 doit finir après les activités de $\mathcal{U} = \{a_1, a_2, a_3\}$. En effet, pour le calcul de l'énergie disponible, le raisonnement Edge-Finding agrandit l'intervalle pour l'activité courante. La règle de détection Extended-Edge-Finding répond à ce problème :

Propriété 3.4 (Règle de détection Extended-Edge-Finding [Nui94])**.** *Soit un ensemble d'activités $\mathcal{U} \subset \mathcal{A}$ et une activité $a \notin \mathcal{U}$ telle que $\underline{s_a} \leq s_{\mathcal{U}} < \underline{e_a}$.*

$$\text{capa} \times (e_{\mathcal{U}} - s_{\mathcal{U}}) < h_a \times \left(\underline{e_a} - s_{\mathcal{U}}\right) + \sum_{i \in \mathcal{U}} w_i \implies \mathcal{U} \lessdot a \tag{9}$$

Comparativement à l'Edge-Finding, l'énergie disponible est ici toujours celle de l'intervalle d'activité $[s_{\mathcal{U}}, e_{\mathcal{U}}[$, quelle que soit la date de début de l'activité. C'est pourquoi les activités finissant après l'intervalle ne sont pas prises en compte et que l'énergie consommée par l'activité se limite à $h_a \times \left(\underline{e_a} - s_{\mathcal{U}}\right)$. Ce calcul, plus précis, nécessite un raisonnement plus poussé.

En utilisant la règle Extended-Edge-Finding sur l'exemple 3.9 l'activité a_4 est détectée comme devant finir après les activités de $\mathcal{U} = \{a_1, a_2, a_3\} : 3 \times (5-1) < 1 \times (5-1) + (4+3+2)$. Un filtrage équivalent peut alors être appliqué : $\underline{s_{a_4}} \geq 4$.

Mercier *et al.* [MV08] ont proposé un algorithme en deux phases : une phase de détection Edge-Finding et Extended-Edge-Finding en $\mathcal{O}(n^2)$, puis une phase en $\mathcal{O}(k\ n^2)$ pour le filtrage, k étant le nombre de hauteurs distinctes. Vilím a proposé un algorithme Edge-Finding en $\mathcal{O}(k\ n\ log\ n)$ ($\mathcal{O}(n\ log\ n)$ pour la première phase et $\mathcal{O}(k\ n\ log\ n)$ pour la seconde) [Vil09]. Deux ans plus tard, Kameugne *et al.* ont proposé un algorithme Edge-Finding en $\mathcal{O}(n^2)$. L'algorithme de Vilim est basé sur une structure de données plus complexe que celui de Kameugne. Par conséquent, bien qu'ayant une complexité inférieure lorsque k est petit, il a un comportement

globalement plus lent [KFSNK11]. L'algorithme de Kameugne possède, de plus, l'avantage non négligeable d'être beaucoup plus simple à implémenter dans un solveur de contraintes.

Le raisonnement Edge-Finding est un raisonnement permettant de détecter des relations de précédences et donc d'effectuer un filtrage. Une adaptation de ce raisonnement permet de detecter des incohérences de la contrainte $\mathrm{Cumulative}$. Il est nécessaire, pour que la contrainte soit cohérente, que pour tout intervalle, l'énergie disponible soit supérieure ou égale à l'énergie consommée au sens de l'Edge-Finding :

Propriété 3.5 (Règle de cohérence Edge-Finding (Overload checking))**.**

$$\mathrm{Cumulative}(A, \mathrm{capa}) \implies \forall\, \mathcal{U} \subseteq \mathcal{A},\ \ \mathrm{capa} \times (e_{\mathcal{U}} - s_{\mathcal{U}}) \geq \sum_{i \in \mathcal{U}} w_i \tag{10}$$

Sur l'ensemble intervalles définis par les sous-ensembles $\mathcal{U} \subseteq \mathcal{A}$ de la propriété 3.5, Wolf et Schrader démontrent qu'il faut et qu'il suffit de calculer l'énergie consommée pour n d'entres eux, pour assurer de détecter une incohérence [WS06]. De leur caractérisation des n intervalles d'intérêt, ils proposent un algorithme en $\mathcal{O}(n\ log\ n)$ pour la détection d'incohérence due à la règle Edge-Finding. Fahimi et Quimper proposent un algorithme qui effectue le test d'incohérence Edge-Finding en temps linéaire, après avoir effectué un tri sur les activités [FQ14].

Time-Table-Edge-Finding

Vilím propose d'améliorer le raisonnement Edge-Finding en prenant en compte, dans le calcul de l'énergie consommée dans un intervalle, les parties obligatoires du raisonnement Time-Table [Vil11].

L'algorithme proposé, qui est en $\mathcal{O}(n^2)$, est actuellement l'algorithme le plus performant en pratique pour résoudre des problèmes cumulatifs de la PSPLib [KS96]. Dans un solveur de contrainte basé sur la génération paresseuse de clauses, sa version expliquée a permis d'améliorer les benchmarks typiques de la littérature [SFS13].

L'idée du raisonnement Time-Table-Edge-Finding est donc d'ajouter l'énergie du raisonnement Time-Table au raisonnement Edge-Finding. Pour éviter de comptabiliser deux fois l'énergie consommée dans un intervalle, Vilím propose de séparer en deux la durée d'une activité : la partie fixe noté p^{TT} correspondant à la durée de la partie obligatoire, et la partie libre p^{EF}, le complément, chacune servant au calcul du raisonnement associé :

Définition 3.4 (Partie Fixe et Partie Libre pour le raisonnement Time-Table-Edge-Finding)**.**
Soit a une activité.

$$\textit{La partie fixe de } a \textit{ est} \qquad p_a^{TT} = \max(0, \underline{e_a} - \overline{s_a}) \tag{11}$$

$$\textit{La partie libre de } a \textit{ est} \qquad p_a^{EF} = p_a - p_a^{TT} \tag{12}$$

Définition 3.5 (Activité libre)**.** *Une activité a est dite libre si sa partie libre n'est pas nulle (c'est à dire si elle n'est pas complètement fixée). L'ensemble des activités libres est noté $\mathcal{A}^{EF}$:*

$$\mathcal{A}^{EF} = \{a \in \mathcal{A} \mid \underline{s_a} \neq \overline{s_a}\} \tag{13}$$

Pour calculer la part du profil de partie obligatoire incluse dans un intervalle, Vilím propose de mémoriser la quantité d'énergie qui se situe après chaque date de début et fin possible d'intervalle :

Définition 3.6 (Énergie Time-Table à droite d'un point de temps)**.** *Nous notons $\mathrm{ttApres}(t)$ l'énergie totale après le point de temps t :*

$$\mathrm{ttApres}(t) = \sum_{t' \geq t} \underbrace{\left(\sum_{a \in \mathcal{A}, t' \in [\overline{s_a}, \underline{e_a}[} h_a \right)}_{\substack{\textit{Hauteur du profil des parties} \\ \textit{obligatoire au point de temps t'}}} \tag{14}$$

La surface du profil de parties obligatoire incluse dans un intervalle $[a, b[$ est alors $\mathrm{ttApres}(a) - \mathrm{ttApres}(b)$.

Exemple 3.10 (Énergie consommée par Time-Table dans un intervalle)**.** *La figure 3.11 montre le calcul de l'énergie Time-Table grace à différentes valeurs de* ttApres*, sur le profil des parties obligatoires de l'exemple 3.4 (page 30).*

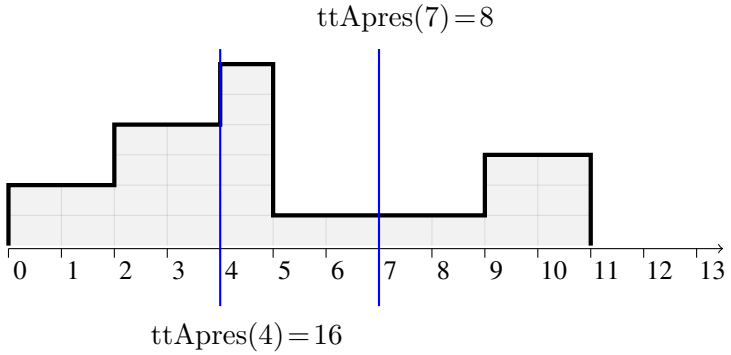


FIGURE 3.11 – Profil de parties obligatoires.

La somme des parties obligatoires après le point de temps 7 est de 8.
La somme des parties obligatoires après le point de temps 4 est de 16.
L'énergie Time-Table dans l'intervalle $[4, 7]$ *est alors de* $16 - 8 = 8$*.*

Soit $\Omega \subseteq \mathcal{A}$ un ensemble d'activités, l'énergie libre des activités de Ω vaut :

$$W_\Omega^{EF} = \sum_{a \in \Omega} p_a^{EF} \times h_a \tag{15}$$

Par conséquent, l'énergie consommée dans l'intervalle $[s_\Omega, e_\Omega[$ est de :

$$W_\Omega^{EF} + \text{ttApres}(s_\Omega) - \text{ttApres}(e_\Omega) \tag{16}$$

et l'énergie disponible dans l'intervalle est notée $\text{Reserve}(\Omega)$:

Définition 3.7 (Notion de réserve pour le raisonnement Time-Table-Edge-Finding [Vil11])**.** *Soit* $\Omega \subseteq \mathcal{A}$ *un ensemble d'activités. L'énergie non consommée dans l'intervalle* Ω *est alors :*

$$\text{Reserve}(\Omega) = \text{capa} \times (e_\Omega - s_\Omega) - (W_\Omega^{EF} + \text{ttApres}(s_\Omega) - \text{ttApres}(e_\Omega)) \tag{17}$$

Si la réserve d'un intervalle d'activités libres est négative, le problème est incohérent :

Propriété 3.6 (Règle de cohérence Time-Table-Edge-Finding [Vil11])**.**

$$\text{Cumulative}(\mathcal{A}, \text{capa}) \implies \forall\, \mathcal{U} \subseteq \mathcal{A}^{EF},\ \ \text{Reserve}(\Omega) \geq 0 \tag{18}$$

Si le fait de positionner une activité a à son placement au plus tôt cause un dépassement de capacité, alors un filtrage de $\underline{s_a}$ peut être effectué. Pour vérifier cet éventuel dépassement, il faut calculer combien d'énergie supplémentaire le placement au plus tôt de l'activité consommera dans l'intervalle. L'énergie de la partie Time-Table est déjà comptabilisée dans l'énergie de l'intervalle ; il reste à calculer l'intersection de la partie libre avec l'intervalle. Cette valeur notée $add(s_\Omega, e_\Omega, a)$, dont le détail du calcul est donné dans [Vil11, table 1], est reportée dans la table 3.1.

position	caractérisation	Consommation additionnelle
dedans	$s_\Omega \leq \underline{s_a} < \underline{s_a} + p_a^{EF} \leq e_\Omega$	e_a^{EF}
à droite	$s_\Omega \leq \underline{s_a} < e_\Omega < \underline{s_a} + p_a^{EF}$	$h_a \times (e_\Omega - \underline{s_a})$
à gauche	$\underline{s_a} < s_\Omega < \underline{s_a} + p_a^{EF} < e_\Omega$	$h_a \times (\underline{s_a} + p_a^{EF} - s_\Omega)$
recouvre	$\underline{s_a} \leq s_\Omega < e_\Omega \leq \underline{s_a} + p_a^{EF}$	$h_a \times (e_\Omega - s_\Omega)$

TABLE 3.1 – $\mathrm{add}(s_\Omega, e_\Omega, a)$: consommation additionnelle de l'activité a dans l'intervalle $[s_\Omega, e_\Omega[$

La détection qu'un filtrage peut être effectué est alors la comparaison entre l'énergie disponible et celle qui sera ajoutée :

$$\mathrm{Reserve}(\Omega) < \mathrm{add}(s_\Omega, e_\Omega, a) \tag{19}$$

Dans ce cas la valeur $\underline{s_a}$ peut être mise à jour.

La longueur maximale autorisée dans l'intervalle $[s_\Omega, e_\Omega[$ pour l'activité a est $\mathrm{Reserve}(\Omega)/h_a$. Le calcul de $\mathrm{Reserve}()$ prend en compte l'ensemble des parties obligatoires ; pour calculer le placement au plus tôt de a il faut alors prendre en compte l'intersection de sa partie obligatoire avec l'intervalle, noté $\mathrm{mandatoryIn}(s_\Omega, e_\Omega, a)$:

$$\mathrm{mandatoryIn}(s_\Omega, e_\Omega, a) = \max(0, \min(e_\Omega, \underline{e_a}) - \max(s_\Omega, \overline{s_a}))$$

Le filtrage pouvant être effectué est alors :

$$\underline{s_a} \geq e_\Omega - \mathrm{Reserve}(\Omega)/h_a - \mathrm{mandatoryIn}(s_\Omega, e_\Omega, a)$$

La règle de filtrage est alors :

Propriété 3.7 (Règle de filtrage pour le raisonnement Time-Table-Edge-Finding [Vil11])**.**

$$\forall \Omega \subseteq \mathcal{A}^{EF}, \forall a \in \mathcal{A}^{EF} \setminus \Omega, \mathrm{Reserve}(\Omega) < \mathrm{add}(s_\Omega, e_\Omega, a) \implies \underline{s_a} \geq e_\Omega - \mathrm{Reserve}(\Omega)/h_a - \mathrm{mandatoryIn}(s_\Omega, e_\Omega, a) \tag{20}$$

Cette règle de filtrage proposée par Vilím est une adaptation de celle du raisonnement Edge-Finding. On notera que l'ensemble des activités libres ($\mathcal{A}^{EF}$) est étudié, et non l'ensemble $\mathcal{A}$ de toutes les activités comme c'est le cas pour le raisonnement Edge-Finding.

Dans [KFSN14], Kameugne montre que l'algorithme Time-Table-Edge-Finding proposé par Vilím [Vil11], ne domine pas les raisonnements Edge-Finding et Extended-Edge-Finding, contrairement à ce qui est annoncé dans [Vil11]. Dans les sections 4.3 et 4.4.2 (pages 62 et 68), nous montrons, qu'associé au raisonnement Time-Table, le raisonnement Time-Table-Edge-Finding domine bel et bien les raisonnements Edge-Finding et Extended-Edge-Finding.

Le raisonnement énergétique

Le calcul le plus précis de l'énergie consommée dans un intervalle est défini par le raisonnement énergétique (noté ER), introduit par Lopez [ELT89, Lop91, LEE92]. Le raisonnement énergétique calcule plus précisément l'intersection minimale (noté MI pour *Minimal Intersection* en anglais) de l'activité avec l'intervalle. L'idée étant que l'intersection est minimale lorsque l'activité est placée le plus à gauche ou le plus à droite possible.

Sur un intervalle $[t_1, t_2[$, nous pouvons définir la consommation lorsque l'activité est placée le plus à gauche possible (noté LS pour *Left Shifted* en anglais) :

Définition 3.8 (LS : placement à gauche)**.** *Soit une activité* $a = \langle s_a, p_a, e_a, h_a \rangle$ *et l'intervalle* $[t_1, t_2[$*. La longueur de l'activité qui est comprise dans l'intervalle lorsque celle-ci est placée à son minimum est :*

$$\mathrm{LS}(a, t_1, t_2) = \max(0, \min(\underline{e_a}, t_2) - \max(\underline{s_a}, t_1)) \tag{21}$$

De façon analogue, nous définissons la consommation lorsque l'activité est placée le plus à droite possible :

Définition 3.9 (RS : placement à droite)**.** *Soit une activité* $a = \langle s_a, p_a, e_a, h_a \rangle$ *et un intervalle* $[t_1, t_2[$*. La longueur de l'activité* a *qui est comprise dans l'intervalle lorsque celle-ci est placée à son maximum est :*

$$\mathrm{RS}(a, t_1, t_2) = \max(0, \min(\overline{e_a}, t_2) - \max(\overline{s_a}, t_1)) \tag{22}$$

Nous pouvons alors définir l'intersection minimale d'une activité avec un intervalle comme le minimum entre son intersection à droite et son intersection à gauche :

Définition 3.10 (MI : intersection minimale)**.** *Soit une activité* $a = \langle s_a, p_a, e_a, h_a \rangle$ *et un intervalle* $[t_1, t_2[$*. La longueur minimale de l'activité* a *qui est comprise dans l'intervalle est :*

$$\mathrm{MI}(a, t_1, t_2) = \min\left(\mathrm{LS}(a, t_1, t_2), \mathrm{RS}(a, t_1, t_2)\right) \tag{23}$$

Un second calcul, équivalent, de l'intersection minimale est :

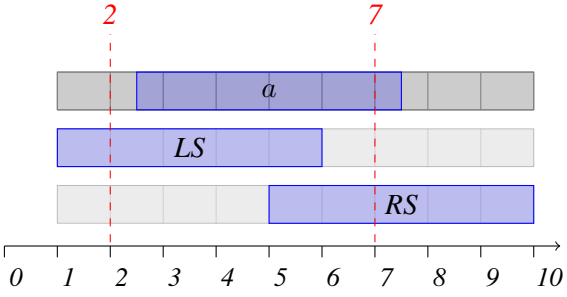
$$\mathrm{MI}(a, t_1, t_2) = \max\left(0,\ \min(t_2 - t_1,\ p_a,\ \underline{e_a} - t_1,\ t_2 - \overline{s_a})\right) \tag{24}$$

Exemple 3.11 (Intersection minimale)**.** *Calcul de l'intersection minimale d'une activité avec un intervalle :*

$a = \langle s_a = [1, 5], p_a = 5, e_a = [6, 10], h_a = 1 \rangle$

$\mathrm{LS}(a, 2, 7) = 4$

$\mathrm{RS}(a, 2, 7) = 2$



L'intersection minimale est alors : $\mathrm{MI}(a, 2, 7) = min(2, 4) = 2$.

Définition 3.11 (Consommation d'un intervalle)**.** *La consommation d'énergie dans un intervalle* $[t_1, t_2[$ *du point de vue du raisonnement énergétique est :*

$$\mathrm{W}(t_1, t_2) = \sum_{i \in \mathcal{A}} h_i \times \mathrm{MI}(i, t_1, t_2) \tag{25}$$

Nous notons le slack d'un intervalle comme étant l'énergie disponible dans celui-ci. Pour le raisonnement énergétique, le slack noté SlackER est alors :

$$\text{SlackER}([t_1, t_2[) = \text{capa} \times (t_2 - t_1) - \text{W}(t_1, t_2) \tag{26}$$

La règle de cohérence du raisonnement énergétique est alors :

Propriété 3.2 (Règle de cohérence du raisonnement énergétique [ELT89])**.**
La contrainte Cumulative *est cohérente seulement s'il n'existe aucun intervalle* $[t_1, t_2[$ *tel que le slack énergétique est négatif.*

$$\text{Cumulative}(\mathcal{A}, \text{capa}) \implies \forall t_1, t_2 \in \mathbb{N}^2, t_1 < t_2, \quad \text{SlackER}([t_1, t_2[) \geq 0 \tag{27}$$

La détection d'une incohérence au vu de cette règle suppose de regarder l'ensemble des intervalles de temps définis sur l'horizon. Baptiste *et al.* ont proposé une caractérisation des intervalles d'intérêts de manière à tester un nombre d'intervalles de l'ordre de $\mathcal{O}(n^2)$, ne dépendant donc pas de la granularité temporelle ou de l'horizon.

Définition 3.12 (Intervalles d'intérêt proposé par Baptiste *et al.* [BLN01, p.68])**.**
Sur l'ensemble des intervalles $[t_1, t_2[\in \mathbb{N}^2$*, il est suffisant de calculer le slack énergétique (*SlackER*) sur les intervalles* $[t_1, t_2[\in O_B = \bigcup_{(i,j)\in\mathcal{A}^2} O_B(i,j)$*, avec :*

$$O_B(i,j) = \begin{cases} (t_1, t_2), t_1 \in O_1(i) < t_2 \in O_2(j), \\ (t_1, t_2), t_1 \in O_1(i) < t_2 \in O_{t_1}(j), \\ (t_1, t_2), t_2 \in O_2(j) > t_1 \in O_{t_2}(i) \end{cases}$$

où

$$O_1(i) = \{\underline{s_i}, \overline{s_i}, \underline{e_i}\}$$
$$O_2(i) = \{\overline{s_i}, \underline{e_i}, \overline{e_i}\}$$
$$O_t(i) = \{\underline{s_i} + \overline{e_i} - t\}$$

La table 3.2 montre les 15 intervalles notés comme étant *d'intérêt* par la caractérisation de Baptiste *et al.* pour chaque paire d'activités (i,j).

t_1 \ t_2	$\overline{e_j}$	$\underline{e_j}$	$\overline{s_j}$	$\underline{s_j}$	$(\underline{s_j}+\overline{e_j})-t_1$
$\underline{s_i}$	X	X	X		Y
$\overline{s_i}$	X	X	X		Y
$\underline{e_i}$	X	X	X		Y
$\overline{e_i}$					
$(\underline{s_i}+\overline{e_i})-t_2$	Z	Z	Z		

TABLE 3.2 – Intervalles d'intérêt pour le raisonnement énergétique de la définition 3.12.
Les cases marquées d'un X représentent les intervalles de O_B définis via O_1 et O_2 (1ère ligne).
Les cases marquées d'un Y représentent les intervalles de O_B définis via O_1 et O_{t_1} (2ème ligne).
Les cases marquées d'un Z représentent les intervalles de O_B définis via O_{t_2} et O_2 (3ème ligne).

Afin de simplifier la présentation, nous regroupons l'ensemble des intervalles d'intérêt par type grâce à la notation suivante :

Notation 3.2. $O_B = \bigcup_{(i,j)} O_B(i,j), \quad O_1 = \bigcup_i O_1(i), \quad O_2 = \bigcup_i O_2(i), \quad O_t = \bigcup_i O_t(i).$

La propriété 3.2 de cohérence du raisonnement énergétique, peut alors se ré-écrire, en limitant les intervalles d'intérêts à O_B :

$$\text{Cumulative}(\mathcal{A}, \text{capa}) \implies \forall t_1, t_2 \in O_B,\ \ \text{SlackER}([t_1, t_2[) \geq 0 \tag{28}$$

La démonstration se base sur l'étude de la fonction de consommation : $t_2 \rightarrow \text{W}(t_1, t_2)$. Ils démontrent que celle-ci est continue et linéaire par morceaux et que les morceaux sont délimités par les 15 intervalles d'intérêt. Par conséquent un minimum global ne peut être trouvé en dehors des intervalles d'intérêt.

Baptiste *et al.* proposent un algorithme de balayage en $O(n^2)$ pour effectuer la détection d'incohérence. L'algorithme se base aussi sur le fait que la fonction est continue et linéaire par morceaux. L'algorithme calcule l'ensemble des intervalles dont la date de début est identique, en maintenant incrémentalement le profil de consommation. En effet, si t_2 n'appartient pas à $O_2 \cup O_{t_1}$ alors la consommation d'énergie n'évolue pas, autrement dit la pente de la fonction $t_2 \rightarrow \text{W}(t_1, t_2)$ est stable autour de t_2 : $pente = \text{W}(t_1, t_2) - \text{W}(t_1, t_2 - 1) = \text{W}(t_1, t_2 + 1) - \text{W}(t_1, t_2)$. Il est donc possible de calculer directement la consommation entre deux évènements. Par contre, à un point de temps t_2 associé à une activité i, caractérisé par la définition 3.12, la consommation peut évoluer. On calcule alors pour cette activité le changement de consommation au point de temps :

$$pente \leftarrow pente - \underbrace{\underbrace{\big(\text{MI}(i, t_1, t_2) - \text{MI}(i, t_1, t_2 - 1)\big)}_{\text{pente pour } i \text{ avant } t_2} + \underbrace{\big(\text{MI}(i, t_1, t_2 + 1) - \text{MI}(i, t_1, t_2)\big)}_{\text{pente pour } i \text{ après } t_2}}_{\text{évolution de la pente due à l'activité } i \text{ en } t_2} \tag{29}$$

L'exemple 3.12 montre ce calcul incrémental de la pente pour une date de début dans le cas d'une contrainte à deux variables :

Exemple 3.12 (Pente pour l'algorithme de détection d'incohérence énergétique [BLN01])**.** *Soit le problème cumulatif décrit par la figure 3.12. Regardons la pente lorsque le début d'intervalle est* $t_1 = 0$ *:*

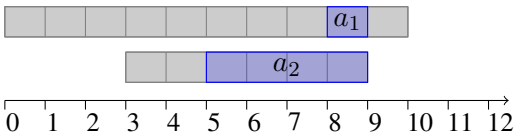


FIGURE 3.12 – Un problème cumulatif à 2 activités.

Pour a_1*, les changements de consommation potentiels se situent aux points de temps définis par* $O_2(a_1) \cup O_{t_1}(a_1) = \{\overline{s_{a_1}} = 1, \underline{e_{a_1}} = 9, \overline{e_{a_1}} = 10\} \cup \{\underline{s_{a_1}} + \overline{e_{a_1}} - t_1 = 1 + 9 - 0\} = \{1, 9, 10\}$.

Pour a_2*, les changements de consommation potentiels se situent aux points de temps définis par* $O_2(a_2) \cup O_{t_1}(a_2) = \{\overline{s_{a_2}} = 5, \underline{e_{a_2}} = 7, \overline{e_{a_2}} = 9\} \cup \{\underline{s_{a_2}} + \overline{e_{a_2}} - t_1 = 3 + 9 - 0\} = \{5, 7, 9, 12\}$.

- *Initialement la pente est* $\text{W}(t_1, t_1 + 1) = 0$
- *Nouvelle valeur de la pente au point de temps 1 par* a_1 *: pente = 0 + (0-0) - (0-0) = 0*
- *Nouvelle valeur de la pente au point de temps 5 par* a_2 *: pente = 0 + (1-0) - (0-0) = 1*
- *Nouvelle valeur de la pente au point de temps 7 par* a_2 *: pente = 1 + (3-2) - (2-1) = 1*
- *Nouvelle valeur de la pente au point de temps 9 par* a_2 *: pente = 1 + (4-4) - (4-3) = 0*
- *Nouvelle valeur de la pente au point de temps 9 par* a_1 *: pente = 0 + (1-0) - (0-0) = 1*
- *Nouvelle valeur de la pente au point de temps 10 par* a_1 *: pente = 1 + (1-1) - (1-0) = 0*
- *Nouvelle valeur de la pente au point de temps 12 par* a_2 *: pente = 1 + (4-4) - (4-4) = 0*

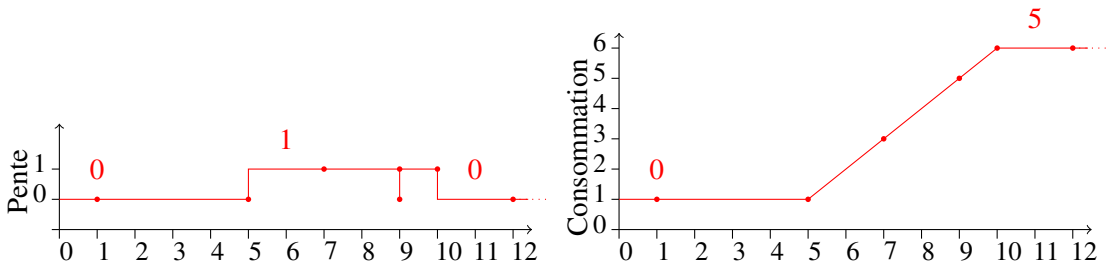


FIGURE 3.13 – Calcul de la consommation via la pente pour le problème de la figure 3.12.

Ici la caractérisation de Baptiste donne 7 points de temps (réduits à 6 points distincts) qu'il est intéressant de regarder, car la pente ne peut changer qu'à ces points. Le nombre de points reste identique, quelle que soit la granularité utilisée, et ne dépend donc que du nombre d'activités. Par exemple, si l'on considère ici des heures, le passage à une granularité à la seconde donne toujours 6 dates d'intérêt, alors que la caractérisation initiale (formule (27)) demanderait d'en regarder plusieurs milliers (l'ensemble des intervalles finissant entre 1 *et* 10×3600*).*

Nous donnons le schéma de l'algorithme proposé par Baptiste [BLN01, chap. 3.3.6.2].

Algorithm 4 Checkeur énergétique de Baptiste *et al.* [BLN01]

1: **procédure** ERCHECKERB
2: **pour tout** $\langle i, t_1 \rangle$ de l'ensemble O_1 **faire**
3: $pente = \sum_j \text{MI}(j, t_1, t_1 + 1)$
4: $\text{SlackER} = 0$, $t_2 old = t_1$
5: **pour tout** $\langle j, t_2 \rangle$ de l'ensemble $O_2 \cup O_{t_1}$ par ordre croissant **faire**
6: $\text{SlackER} = \text{SlackER} + pente * (t_2 - t_2 old)$
7: **si** $\text{SlackER} > \text{capa} \times (t_2 - t_1)$ **alors**
8: Une incohérence a été détecté !
9: **fin si**
10: $pente = pente + \text{MI}(j, t_1, t_2 - 1) - 2 \times \text{MI}(j, t_1, t_2) + \text{MI}(j, t_1, t_2 + 1)$
11: $t_2 old = t_2$
12: **fin pour**
13: **fin pour**
14: **fin procédure**

Par souci de lisibilité, nous présentons le parcours de l'ensemble des dates d'intérêt via le couple ⟨activité, date⟩. Le lecteur intéressé par une implémentation de ces parcours en temps linéaire, permettant d'obtenir un algorithme en temps quadratique, peut se référer à [BLN01, algorithme 9, page 73].

Le calcul incrémental de la pente (ligne 10) s'effectue en suivant la formule (29). On utilise à nouveau ici le fait que la contrainte Cumulative est symétrique : l'algorithme ERCHECKERB calcule seulement les intervalles définis dans la table 3.2 (page 45) par les trois premières lignes ; la version symétrique parcourant la table en colonne : en bouclant en premier sur $t_2 \in O_2$ puis sur l'ensemble des $t_1 \in O_1 \cup O_{t_2}$.

Cette caractérisation permet de réduire grandement le nombre d'intervalles d'intérêt pour l'algorithme de détection d'incohérence énergétique. Une question reste malgré tout ouverte dans [BLN01] : est-il possible d'affiner encore plus cette caractérisation pour réduire le nombre d'intervalles d'intérêt ? Dans le chapitre 4.2.4 nous démontrons qu'il est possible de réduire d'un facteur 7 le nombre d'intervalles d'intérêt, et, à l'aide d'une modification mineure de l'algorithme qui en découle, de réduire d'un tiers son temps d'exécution.

Le filtrage du raisonnement énergétique se base sur la même idée que les raisonnements vus précédemment. Si l'énergie restante dans un intervalle est plus petite que l'énergie demandée par l'activité lorsqu'elle est à son début au plus tôt, alors on peut filtrer cette valeur. Notons $\mathrm{Dispo}(a, t_1, t_2)$ l'énergie disponible dans l'intervalle $[t_1, t_2[$ pour l'activité a, cette valeur se calcule alors :

$$\mathrm{Dispo}(a, t_1, t_2) = \mathrm{capa} \times (t_2 - t_1) - \sum_{i \in \mathcal{A} \setminus \{a\}} h_i \times \mathrm{MI}(i, t_1, t_2) \tag{30}$$

La règle de filtrage du raisonnement énergétique s'écrit alors :

Proposition 3.1 (Règle de filtrage raisonnement énergétique [BLN01, chap. 3.3.6.3])**.** *Si pour une activité a il existe un intervalle $[t_1, t_2[$ tel que $\mathrm{Dispo}(a, t_1, t_2) < h_a \times \mathrm{LS}(a, t_1, t_2)$, on peut alors effectuer un filtrage sur le placement au plus tôt. La quantité d'énergie en surplus dans l'intervalle $[t_1, t_2[$ doit se situer après t_2 : $h_a \times \mathrm{LS}(a, t_1, t_2) - \mathrm{Dispo}(a, t_1, t_2)$. Une borne inférieure pour la fin au plus tôt est alors : $\underline{e_a} = t_2 + h_a \times \mathrm{LS}(a, t_1, t_2) - \mathrm{Dispo}(a, t_1, t_2)$*

La nouvelle borne proposée peut être naturellement améliorée, en se basant sur l'espace disponible dans l'intervalle, et non pas sur la formule qui a amené le filtrage, [DP13, KFF13] : $\underline{s_a} = t_2 - \frac{1}{h_a} \times \mathrm{Dispo}(a, t_1, t_2)$.

Dans [BLN01, chap. 3.3.6.3] Baptiste *et al.* proposent un algorithme de filtrage basé sur ces règles de filtrage :

Algorithm 5 Propagateur énergétique de Baptiste *et al.* [BLN01]

procédure ERPROPAGB
  pour tout $(t_1, t_2) \in O_C(\mathcal{A})$ **faire**
    $W := \sum_{a \in \mathcal{A}} h_a \times \mathrm{MI}(a, t_1, t_2)$
    si $W > \mathrm{capa} \times (t_2 - t_1)$ **alors**
      Une incohérence a été détecté !
    sinon
      pour tout $a \in \mathcal{A}$ **faire**
        $avail := \mathrm{capa} \times (t_2 - t_1) - W + h_a \times \mathrm{MI}(a, t_1, t_2)$
        si $avail < h_a \times \mathrm{LS}(a, t_1, t_2)$ **alors**
          $\underline{s_a} := \max(\underline{s_a},\ t_2 - \frac{1}{h_a} \times avail)$
        fin si
        si $avail < h_a \times \mathrm{RS}(a, t_1, t_2)$ **alors**
          $\overline{e_a} := \min(\overline{e_a},\ t_1 + \frac{1}{h_a} \times avail)$
        fin si
      fin pour
    fin si
  fin pour
fin procédure

3.2 La notion de robustesse

La robustesse d'une solution peut se définir comme sa capacité à garder ses performances suite à une modification des données. Dans de nombreuses applications réelles, la notion de satisfiabilité (et a fortiori d'optimalité) est souvent secondaire, et trouver des solutions offrant des compromis est souvent nécessaire. Par exemple la solution d'un problème d'ordonnancement est souvent assez peu stable : si l'une des activités venait à être retardée l'ensemble des activités pourraient en être impacté ; de plus la performance en est souvent grandement affectée.

Pour répondre à ce genre de problématiques, de nombreuses méthodes ont été proposées. Qu'elles soient réactives ou proactives, la plupart sont stochastiques [Pin12]. Les méthodes réactives, qui permettent de réparer les solutions lorsqu'une perturbation est connue s'opposent aux méthodes proactives, qui proposent des solutions anticipant les perturbations.

Alors que les modèles stochastiques permettent d'envisager des variables aléatoires sur lesquelles la théorie des probabilités peut être appliquée, dans de nombreux cas pratiques, soit l'utilisateur final ne dispose pas de données adéquates avec lesquelles travailler, soit les modèles probabilistes ne sont tout simplement pas pertinents. Par conséquent, les modèles d'optimisation déterministes qui tiennent compte de toutes les incertitudes possibles ont été largement étudiés dans les quinze dernières années, principalement à travers le concept de solution robuste, par exemple, [DJB01, BN02, BS03, BS04, BMS08, WBB09, ZVS+13, DPZ14a].

Nous décrivons ici les approches robustes qui sont directement liées à la contribution présentée dans le chapitre 5 : une approche qui soit à la fois proactive, déterministe et qui, bien que dédiée aux problèmes d'ordonnancements, reste modulaire, de manière à couvrir un ensemble de problèmes le plus large possible.

3.2.1 La robustesse en programmation par contraintes

Hebrard *et al.* ont proposé une approche générique pour la robustesse dans les réseaux de contraintes, basée sur la notion de super-solution [Heb04, HHW04]. Dans une (a, b)-super-solution, le changement de valeur d'au plus a variables peut être réparé en changeant la valeur d'au plus b variables.

Définition 3.13 (super-solution [HHW04])**.** *Une solution d'un réseau de contraintes est une (a, b)-super-solution si, et seulement si, la perte de valeur pour au plus a variables peut être réparée en affectant une nouvelle valeur à ces variables et au plus b autres.*

Plus la valeur de a est grande, plus la solution est robuste. Il existe alors une réparation suite au changement d'un plus grand nombre de variables. De même plus la valeur de b est petite, plus la solution est stable puisqu'il est possible de réparer en affectant moins de variables.

Exemple 3.13 ((a, b)-super-solution [HHW04])**.** *Soit un réseau de contraintes composé de 3 variables X, Y et Z dont le domaine est $\{1, 2, 3\}$ et de deux contraintes : $X \leq Y$ et $Y \leq Z$.*

Sur les dix solutions possibles attardons-nous sur $\langle 1, 2, 3\rangle$ et $\langle 1, 1, 1\rangle$ pour comparer leur niveau de robustesse.

La solution $\langle 1, 1, 1\rangle$ n'est pas une $(1, 0)$-super-solution. En effet, s'il n'est pas possible d'affecter la valeur 1 à X, alors il n'existe pas de solution (car X doit être inférieure à Y). Elle n'est pas non plus $(1, 1)$-super-solution, en effet, même en réparant Y, c'est la contrainte $Y \leq Z$ qui n'est plus respectée.

La solution $\langle 1, 2, 3\rangle$ est, elle, une $(2, 0)$-super-solution. En effet, il suffit d'affecter aux deux variables la valeur de la troisième.

Nous avons donc un formalisme permettant de préférer la solution $\langle 1, 2, 3\rangle$ à la solution $\langle 1, 1, 1\rangle$, la première étant plus robuste.

Cette approche, parce qu'elle est générique, est telle que toutes les variables ont le même statut. La sémantique du problème n'est pas prise en compte. En ordonnancement, elle a été appliquée sur des benchmarks de job-shop [HHW04]. La propriété intéressante de ce paradigme est sa déclarativité : le facteur de robustesse des solutions est formellement quantifié, en amont de la résolution.

3.2.2 La robustesse en ordonnancement

Une alternative plus spécifique à l'ordonnancement consiste à augmenter la durée des activités. L'ajout pourra être utilisé pour répondre à un évènement inattendu. Elle a été améliorée en incorporant dans les algorithmes de résolution, en cours de recherche, un raisonnement sur l'incertitude [DJB01].

D'autres approches existent en Recherche Opérationnelle pour les problèmes de job-shop et d'open-shop, ainsi que des variantes plus spécifiques à des familles d'applications [BMS08, WBB09]. Enfin, il est intéressant de noter que des approches similaires en programmation linéaire ont été proposées pour traiter des problèmes robustes cumulatifs avec précédences [ALT11].

Contributions

4

Caractérisations des raisonnements pour la contrainte Cumulative

Sommaire

Nous avons décrit dans le chapitre précédent plusieurs raisonnements pour la contrainte Cumulative. Dans la littérature chacun de ces raisonnements est présenté de façon différente : le raisonnement Time-Table porte sur la notion de partie obligatoire, les raisonnements Edge-Finding sur une notion d'ensemble d'activités, le raisonnement énergétique sur une notion d'intervalles et le raisonnement Time-Table-Edge-Finding améliore Edge-Finding en étant dépendant de Time-Table.

Dans ce chapitre, nous proposons de réécrire les formules de chaque raisonnement de façon unifié, pour faciliter la démonstration des relations de dominance entre raisonnements. Ensuite, nous montrerons dans les sections 4.2.4 et 4.4.1 qu'à partir de cette étude sur la mise en perspective de ces raisonnements, il est possible d'améliorer l'algorithme de détection d'incohérence et le propagateur énergétique.

4.1 Détection d'incohérence

Nous montrons dans cette section que chacun des raisonnements permettant de détecter une incohérence peut être vu comme un raisonnement par intervalles : pour tout intervalle de temps, la consommation ne peut excéder la production. C'est alors uniquement le calcul de la consommation qui diffère entre chacun des raisonnements. Nous pouvons écrire de manière générique la règle de cohérence suivante :

Propriété 4.1 (Règle de cohérence générique pour la contrainte cumulative)**.** *Soit $\mathcal{A}$ un ensemble d'activités et* capa *un entier. La contrainte* $\text{Cumulative}(\mathcal{A}, \text{capa})$ *est incohérente s'il existe un intervalle* $[t_1, t_2[$ *dans lequel la consommation est supérieure à la capacité.*

$$\text{Cumulative}(\mathcal{A}, \text{capa}) \implies \forall t_1, t_2 \in \mathbb{N}^2, t_1 < t_2, \quad \text{Conso}(\mathcal{A}, [t_1, t_2[) \leq \text{capa} \times (t_2 - t_1) \quad (31)$$

Dans la suite de cette section, Nous présentons les déclinaisons de la fonction Conso pour chacun des raisonnements : ConsoER pour le raisonnement énergétique, ConsoEF pour le raisonnement Edge-Finding, ConsoTT pour le raisonnement Time-Table et enfin ConsoTTEF pour le raisonnement Time-Table-Edge-Finding.

4.1.1 Raisonnement énergétique

Le raisonnement énergétique est présenté dans l'état de l'art comme étant un raisonnement par intervalle. C'est naturellement que celui-ci semble le plus proche de la fonction de consommation définie en 3.11 (page 44) :

$$\text{ConsoER}(\mathcal{A}, [t_1, t_2[) = \sum_{i \in \mathcal{A}} h_i \times \text{MI}(i, t_1, t_2) \quad (32)$$

où, pour rappel, $\text{MI}(i, t_1, t_2)$ correspond à l'intersection minimale de l'activité i avec l'intervalle $[t_1, t_2[$ (*cf.* définition 3.10).

4.1.2 Edge-Finding

Le raisonnement Edge-Finding est un raisonnement basé sur un ensemble d'activités, définissant des intervalles. Caseau et Laburthe montrent l'équivalence entre raisonnements par intervalles et par ensemble d'activités pour le raisonnement Edge-Finding [CL94]. La consommation se calcule de la façon suivante :

$$\text{ConsoEF}(\mathcal{A}, [t_1, t_2[) = \sum_{\substack{i \in \mathcal{A}, \\ t_1 \leq \underline{s_i} \wedge \overline{e_i} \leq t_2}} h_i \times p_i \quad (33)$$

4.1.3 Time-Table

Le raisonnement Time-Table est un raisonnement basé sur les parties obligatoires. L'énergie consommée dans un intervalle correspond donc à l'intersection de la partie obligatoire $[\overline{s_i}, \underline{e_i}[$ et de l'intervalle $[t_1, t_2[$: $[\max(\overline{s_i}, t_1), \min(\underline{e_i}, t_2)[$. La consommation se calcule comme suit :

$$\text{ConsoTT}(\mathcal{A}, [t_1, t_2[) = \sum_{i \in \mathcal{A}} h_i \times \max(0, \min(\underline{e_i}, t_2) - \max(\overline{s_i}, t_1)) \quad (34)$$

La formule (34) peut se réécrire en décomposant chaque point de temps de l'intervalle $[t_1, t_2[$:

$$\text{ConsoTT}(\mathcal{A}, [t_1, t_2[) = \sum_{t \in [t_1, t_2[} \left(\sum_{i \in \mathcal{A}} h_i \times \underbrace{\max(0, \min(\underline{e_i}, t+1) - \max(\overline{s_i}, t))}_{= 1 \text{ si } t \text{ est compris dans la partie obligatoire, 0 sinon}} \right) \quad (35)$$

d'où :

$$\mathrm{ConsoTT}(\mathcal{A}, [t_1, t_2[) = \sum_{t \in [t_1, t_2[} \Big(\sum_{\substack{i \in \mathcal{A}, \\ t \in [\overline{s_i}, \underline{e_i}[}} h_i \Big) \tag{36}$$

Pour les intervalles de taille 1, la première somme ($\sum_{t \in [t_1, t_2[}$) se fait sur $t = t_1$ uniquement. Les calculs sont donc équivalents à ceux de la condition (3) de la propriété 3.1 (page 30). Pour les intervalles de taille supérieure, il s'agit de la relaxation surrogate [Glo65] des conditions pour les intervalles de taille 1. Par conséquent, la propriété 4.1 de détection d'incohérence basé sur ConsoTT est bien équivalente à celle de la propriété 3.1.

4.1.4 Time-Table-Edge-Finding

Le raisonnement Time-Table-Edge-Finding allie le calcul des deux raisonnements Time-Table et Edge-Finding dans le calcul de l'énergie d'un intervalle. L'énergie complète du raisonnement Time-Table (c'est-à-dire l'intersection des parties obligatoires avec l'intervalle) et la partie libre des activités entièrement inclues dans l'intervalle sont prise en compte. Pour cela, la longueur d'une activité est découpée en deux : la partie obligatoire (dite *partie fixe* dans le cadre du raisonnement Time-Table-Edge-Finding) et la partie libre. La partie obligatoire d'une activité a est définie par $p_a^{TT} = \max(0, \underline{e_a} - \overline{s_a})$. La partie libre d'une activité est définie par $p_a^{EF} = p_a - p_a^{TT}$ (définition 3.4 page 41). La consommation peut alors être réécrite comme suit :

$$\begin{aligned} \mathrm{ConsoTTEF}(\mathcal{A}, [t_1, t_2[) = & \mathrm{ConsoTT}(\mathcal{A}, [t_1, t_2[) + \\ & \big(\mathrm{ConsoEF}(\Omega, [t_1, t_2[) - \mathrm{ConsoTT}(\Omega, [t_1, t_2[)\big) \end{aligned} \tag{37}$$

Avec Ω l'ensemble des activités incluses dans l'intervalle $[t_1, t_2[$.

Nous avons déjà vu que $\mathrm{ConsoTT}(\mathcal{A}, [t_1, t_2[)$ correspond à la consommation du raisonnement Time-Table dans l'intervalle $[t_1, t_2[$. Chaque activité de l'ensemble Ω étant entièrement incluse dans l'intervalle, la différence $\big(\mathrm{ConsoEF}(\Omega, [t_1, t_2[) - \mathrm{ConsoTT}(\Omega, [t_1, t_2[)\big)$ vaut la somme des énergies libres des activités d'Ω (*c.f* formule (15), page 42). Le calcul d'énergie est donc équivalent à celui de la formule (16) (page 42). Par conséquent, la propriété 4.1 de détection d'incohérence basé sur ConsoTTEF est bien équivalente à celle de la propriété 3.6 (page 42).

Une écriture équivalente, mais inversée, est de prendre en compte le calcul complet de l'énergie du raisonnement Edge-Finding et l'ajout du calcul de l'énergie Time-Table qui n'a pas été prise en compte dans le raisonnement Edge-Finding.

$$\mathrm{ConsoTTEF}(\mathcal{A}, [t_1, t_2[) = \mathrm{ConsoEF}(\mathcal{A}, [t_1, t_2[) + \mathrm{ConsoTT}(\mathcal{A} \setminus \Omega, [t_1, t_2[) \tag{38}$$

Nous pouvons considérer, pour le moment, que le raisonnement Time-Table-Edge-Finding se fait sur l'ensemble des intervalles $[t_1, t_2[,\ t_1, t_2 \in \mathcal{N}$, et que seul l'algorithme se limite aux intervalles d'activités libres : $\Omega \subset \mathcal{A}^{EF}$. Dans la section 4.2.3 nous étudirons en détail les intervalles d'intérêts du raisonnement Time-Table-Edge-Finding.

4.2 Intervalles d'intérêt pour la détection d'incohérence

Dans cette section nous proposons de caractériser le plus finement possible les intervalles qu'il est nécessaire de vérifier pour assurer la détection d'une incohérence. En effet les différentes caractérisations que nous avons vues dans la section précédente ont toutes été définies comme étant un raisonnement sur l'ensemble des $\mathcal{N}^2$ intervalles. Or, il est inutile de considérer un intervalle qui est en dehors de l'horizon de toute activité, car la consommation sur cet intervalle sera nulle. On peut donc limiter les raisonnements aux intervalles inclus dans $[\min_{a\in\mathcal{A}} \underline{s_a}, \max_{a\in\mathcal{A}} \overline{e_a}[$. Pour chacun des raisonnements, il est possible de réduire encore l'ensemble des intervalles d'intérêts. Nous proposons d'étudier au cas par cas chacun des raisonnements et d'en donner une caractérisation la plus fine possible, c'est-à-dire caractériser le moins possible d'intervalles comme étant d'intérêt.

4.2.1 Time-Table

Dans un souci d'unification, la formule (34) de consommation du raisonnement Time-Table prend en considération tous les intervalles de la forme $[t_1, t_2[, t_1, t_2 \in \mathbb{N}^2$. Pour le raisonnement Time-Table il est possible de ne considérer qu'un nombre d'intervalles correspondant au nombre d'activités ayant une partie obligatoire. En effet, seuls les intervalles de la forme $[\overline{s_a},\ \overline{s_a}+1[$, c'est-à-dire le début de partie obligatoire, pour chaque activité a ayant une partie obligatoire sont d'intérêts. La formule devient alors :

$$\text{Cumulative}(\mathcal{A}, \text{capa}) \implies \forall a \in \mathcal{A}, \overline{s_a} < \underline{e_a},\ \text{ConsoTT}(\mathcal{A}, [\overline{s_a}, \overline{s_a}+1[) \leq \text{capa} \tag{39}$$

Démonstration. Dans un premier temps nous montrons que les intervalles de taille supérieure ou égale à 2 sont dominés par les intervalles de taille 1. Nous montrons, dans un second temps, que parmi les intervalles de taille 1, seuls les intervalles de la forme $[\overline{s_a},\ \overline{s_a}+1[$ sont d'intérêts.

Nous avons déjà remarqué que, pour le raisonnement Time-Table, la formule de la consommation pour les intervalles de taille supérieure à 1 est la relaxation surrogate de la formule pour ceux de taille 1. Les intervalles de taille supérieure à 2 sont donc dominés, et nous pouvons nous limiter aux intervalles de taille 1.

Considérons maintenant les intervalles de taille 1. Si l'intervalle n'intersecte aucune partie obligatoire, la consommation est nulle et l'intervalle ne mènera pas à un échec, il n'est donc pas d'intérêt. S'il intersecte une partie obligatoire sans être au début d'une partie obligatoire, alors l'intervalle précédent a exactement la même consommation. Les intervalles de taille 1 qui ne commencent pas en début de partie obligatoire sont donc dominés par ceux commençant à une partie obligatoire. □

Il est intéressant de remarquer que le raisonnement Time-Table est équivalent au raisonnement énergétique limité aux intervalles de taille 1. En effet pour les intervalles de taille 1 $\text{MI}(i, t, t+1)$ vaut 1 si t intersecte la partie obligatoire de i, 0 sinon.

4.2.2 Edge-Finding

Wolf et Schrader ont démontré qu'il suffit de calculer la consommation pour un nombre linéaire d'intervalles, en utilisant une structure de donnée adaptée. Leur caractérisation des intervalles d'intérêt mène à un algorithme de détection d'incohérence Edge-Finding en $\mathcal{O}(n\ log\ n)$ [WS06].

4.2.3 Time-Table-Edge-Finding

Nous avons défini le raisonnement Time-Table-Edge-Finding comme calculant l'ensemble des intervalles de la forme $[t_1, t_2[,\ t_1, t_2 \in \mathcal{N}$. Dans [Vil11], l'algorithme de détection d'incohérence ne parcourt que l'ensemble des activités libres $\mathcal{A}^{EF}$. Par conséquent, il se limite à vérifier la

cohérence sur les intervalles que nous nommons les *intervalles d'activités libres*, ou intervalles définis par $\mathcal{A}^{EF}$: $\{[t_1, t_2[\mid \exists \Omega \subseteq \mathcal{A}^{EF}, t_1 = \min_{a\in\Omega}(\underline{s_a}), t_2 = \max_{a\in\Omega}(\overline{e_a})\}$ (en référence aux *intervalles d'activités* du raisonnement Edge-Finding, définis par l'ensemble $\mathcal{A}$).

Le but du raisonnement Time-Table-Edge-Finding est d'améliorer le raisonnement Edge-Finding en y ajoutant l'énergie du raisonnement Time-Table. Lorsque Vilím a proposé son algorithme il a supposé que, dans un solveur de contraintes, les algorithmes du raisonnement Time-Table sont forcement appliqués. Il a donc proposé un algorithme qui ne ré-effectue pas les calculs déjà réalisés par Time-Table[1].

Pour effectuer un raisonnement Time-Table-Edge-Finding complet on ne peut pas se limiter aux intervalles d'activités libres (les intervalles définis par les ensembles $\Omega \subseteq \mathcal{A}^{EF}$). En effet, lorsque toutes les activités sont fixées, l'algorithme ne fait rien, et ne peut donc détecter aucune incohérence, alors que le raisonnement Edge-Finding sera quant à lui capable de détecter des incohérences.

En supposant que le raisonnement Time-Table a préalablement été réalisé et donc que la propriété 3.2 (page 31) est vérifiée, on peut démontrer que l'algorithme de Vilím est correct et qu'il domine le raisonnement Edge-Finding.

Nous démontrons que les intervalles qui ne sont pas des intervalles définis par un ensemble $\Omega \subset \mathcal{A}^{EF}$ sont dominés par un intervalle de taille plus petite.

Preuve de dominance des intervalles Time-Table-Edge-Finding. Soit un intervalle $[t_1, t_2[$ tel que t_2 n'est pas une date d'intérêt pour le raisonnement Time-Table-Edge-Finding : $\nexists a \in \mathcal{A}^{EF}$, $\overline{e_a} = t_2$. Pour détecter une incohérence on peut alors effectuer le test sur l'intervalle $[t_1, t_2-1[$:

$$\begin{array}{ll} \text{ConsoTTEF}(\mathcal{A}, [t_1, t_2[) & \leq \text{capa} \times (t_2 - t_1) \\ \text{ConsoTTEF}(\mathcal{A}, [t_1, t_2-1[) & \leq \text{capa} \times ((t_2-1) - t_1) \end{array} \quad \Longrightarrow$$

Cela est vrai si :

$$\text{ConsoTTEF}(\mathcal{A}, [t_1, t_2[) - \text{ConsoTTEF}(\mathcal{A}, [t_1, t_2-1[) \leq \text{capa}$$

Décomposons la consommation telle qu'exprimée par l'équation (38) (page 55). L'instant t_2 n'étant n'est pas une date d'intérêt pour le raisonnement Edge-Finding, l'ensemble Ω des activités comptabilisées pour le compte du raisonnement Edge-Finding est le même pour les intervalles $[t_1, t_2[$ et $[t_1, t_2-1[$ donc $\text{ConsoEF}(\mathcal{A}, [t_1, t_2[) = \text{ConsoEF}(\mathcal{A}, [t_1, t_2-1[)$. Il reste donc le calcul de l'énergie due à Time-Table.

$$\text{ConsoTT}(\mathcal{A} \setminus \Omega, [t_1, t_2[) - \text{ConsoTT}(\mathcal{A} \setminus \Omega, [t_1, t_2-1[) \leq \text{capa}$$

En reprenant la formulation de la consommation Time-Table vue en (36) (page 55), on obtient :

$$\sum_{t\in[t_1,t_2[} \Big(\sum_{\substack{i\in\mathcal{A}\setminus\Omega,\\ t\in[\overline{s_i},\underline{e_i}[}} h_i \Big) \quad - \quad \sum_{t\in[t_1,t_2-1[} \Big(\sum_{\substack{i\in\mathcal{A}\setminus\Omega,\\ t\in[\overline{s_i},\underline{e_i}[}} h_i \Big) \quad \leq \text{capa}$$

Il ne reste finalement que le point de temps t_2, soit :

$$\sum_{i\in\mathcal{A}\setminus\Omega, t_2\in[\overline{s_i},\underline{e_i}[} h_i \leq \text{capa}$$

Le raisonnement Time-Table ayant déjà été appliqué, cette condition est respectée. La démonstration dans le cas où t_1 n'est pas une date d'intérêt pour le raisonnement Time-Table-Edge-Finding est strictement équivalente.

Par conséquent si un intervalle n'est pas caractérisé comme d'intérêt par le raisonnement Time-Table-Edge-Finding, il est dominé. Il est donc suffisant d'effectuer le test d'incohérence sur les intervalles d'activités libres. □

[1] Communication privée.

4.2.4 Raisonnement énergétique

Nous avons rappelé dans l'état de l'art (tableau 3.2, page 45) que la caractérisation de Baptiste *et al.* permet de réduire le nombre d'intervalles d'intérêt à 15 par couple d'activités [BLN01]. Leur étude montre qu'en dehors de ces 15 points, la fonction représentant l'évolution de la consommation n'évolue pas : sa dérivée est constante.

Dans cette section, nous proposons d'affiner cette caractérisation de manière à réduire de 15 à 2 le nombre d'intervalles d'intérêt pour chaque couple d'activités. Enfin cette caractérisation sera réutilisée dans la section 4.4.1 pour améliorer la caractérisation du propagateur.

Etude de l'intersection minimale

Baptiste *et al.* ont démontré que la fonction $f_1 : (t_1, t_2) \rightarrow \text{capa} \times (t_2 - t_1) - \sum_{a \in \mathcal{A}} h_a \times \text{MI}(a, t_1, t_2)$, représentant l'écart entre capacité et consommation dans un intervalle, est linéaire et continue par morceaux, et que chaque partie est bornée par des points de la caractérisation. Puisqu'un extremum ne peut être trouvé qu'aux bornes d'une partie, leur caractérisation est suffisante pour effectuer l'ensemble des deductions [BLN01].

Nous proposons d'étudier plus finement les variations de la fonction pour améliorer la caractérisation. Tous les changements de variation ne peuvent impliquer un minimum local ; il y a minimum local seulement si la dérivée à droite est négative, et la dérivée à gauche est positive, et l'une d'elles est non nulle. Cette propriété montré par *test de la dérivée seconde*, est illustrée par la figure 4.1 :

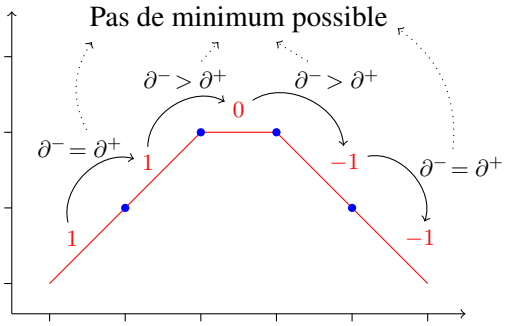


FIGURE 4.1 – Mise en lumière du test de la dérivée seconde : aucun minimum local ne peut être trouvé si la dérivée à gauche est supérieure ou égale à la dérivée à droite.

Pour faciliter la lecture, nous notons les variables muettes pour les activités i ou j en fonction de leur rôle dans la caractérisation : i représente l'activité induisant la date de début de l'intervalle d'intérêt et j représente l'activité induisant la date de fin de l'intervalle d'intérêt. Nous utilisons les notations usuelles en analyses pour les dérivées à gauche (∂^-) et à droite (∂^+).

Lemme 4.1 (conditions nécessaires à la minimalité de f_1)**.**
f_1 est localement minimale seulement s'il existe deux activités i et j telles que les conditions ci-dessous sont satisfaites.

$$\frac{\partial^- \text{MI}(i, t_1, t_2)}{\partial t_1} > \frac{\partial^+ \text{MI}(i, t_1, t_2)}{\partial t_1} \tag{40}$$

$$\frac{\partial^- \text{MI}(j, t_1, t_2)}{\partial t_2} > \frac{\partial^+ \text{MI}(j, t_1, t_2)}{\partial t_2} \tag{41}$$

Démonstration. Supposons que pour aucune activité de $\mathcal{A}$, la condition (40) ne soit pas satisfaite. Alors $\sum_{a \in \mathcal{A}} h_a \times \text{MI}(a, t_1, t_2)$ a sa dérivée à gauche plus petite que sa dérivée à droite. Par conséquent, f_1 a sa dérivé à gauche plus grande que sa dérivé à droite. Par le test de la seconde dérivée, un minimum local d'une fonction ne peut exister que si la dérivée à gauche est plus petite que la dérivée à droite. (t_1, t_2) ne peut donc pas être un minimum local. La preuve pour la condition (41) est identique. Ce qui démontre le lemme. □

L'ensemble d'intervalles O_B caractérise 15 intervalles pour tout couple d'activité (i, j). Ce nombre pour être réduit grâce au lemme 4.1 : Nous pouvons en déduire les conditions nécessaires pour déterminer un sous-ensemble d'intervalles d'intérêt. Nous commençons par caractériser les conditions pour lesquelles une fin d'intervalle est d'intérêt.

Lemme 4.2 (Unicité de la date de fin d'intervalle)**.** *Pour chaque activité j et pour tout début d'intervalle t_1 il existe au plus un intervalle $[t_1, t_2[$ tel que $\frac{\partial^- \mathrm{MI}(j,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{MI}(j,t_1,t_2)}{\partial t_2}$:*

① *si* $t_1 \leq \underline{s_j}$ *alors seulement* $[t_1, \overline{e_j}[$ *a besoin d'être considéré.*

② *si* $t_1 > \underline{s_j} \wedge t_1 \geq \underline{e_j}$ *alors aucun intervalle n'a à être considéré.*

③ *si* $t_1 > \underline{s_j} \wedge t_1 < \underline{e_j} \wedge t_1 < \overline{s_j}$ *alors seulement* $[t_1, \underline{s_j} + \overline{e_j} - t_1[$ *a besoin d'être considéré.*

④ *si* $t_1 > \underline{s_j} \wedge t_1 < \underline{e_j} \wedge t_1 \geq \overline{s_j}$ *alors seulement* $[t_1, \underline{e_j}[$ *a besoin d'être considéré.*

Démonstration. Étudions les variations de $f_2^j : t_2 \to \mathrm{MI}(j, t_1, t_2)$ en fonction de t_2. La figure 4.2 permet d'expliquer la condition ①. Elle représente l'évolution de l'intersection minimale d'une activité $j = \{s_j \in [2, 4],\ p_j = 4,\ e_j \in [6, 8], h_j\}$. Nous pouvons distinguer trois cas.

- Si $t_2 \leq \overline{s_j}$ alors $\mathrm{MI}(j, t_1, t_2) = 0$.
- Si $\overline{s_j} \leq t_2 \leq \overline{e_j}$ alors $\mathrm{MI}(j, t_1, t_2) = t_2 - \overline{s_j}$.
- Si $\overline{e_j} \leq t_2$ alors $\mathrm{MI}(j, t_1, t_2) = p_j$.

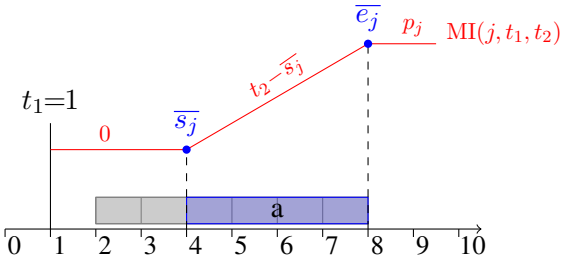


FIGURE 4.2 – Évolution de $\mathrm{MI}(j, t_1, t_2)$ en fonction de la fin de l'intervalle, t_2.

Le seul intervalle pour lequel $\frac{\partial^- \mathrm{MI}(j,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{MI}(j,t_1,t_2)}{\partial t_2}$ est $[t_1, \overline{e_j}[$ ($[1, 8[$ dans cet exemple). Une étude par cas similaire s'applique aux conditions ②, ③ et ④ [DP13]. □

Nous venons de démontrer qu'il existe trois dates potentielles pour que t_2 soit d'intérêt. Par symétrie nous pouvons en déduire qu'il en existe également trois pour t_1. Pour chaque couple, cela créée 9 dates potentielles d'intérêt. Une des neuf ayant été montrée comme n'étant pas d'intérêt (*cf.* propriété 3.12). Les 8 dates d'intérêt sont caractérisées par le lemme 4.3.

Lemme 4.3 (Caractérisation précise des intervalles d'intérêt pour le raisonnement énergétique)**.** *f_1 est un minimum local en (t_1, t_2) seulement s'il existe deux activités i et j tel que $(t_1, t_2) \in O_C(i, j)$ avec*

$$O_C(i,j) = \begin{cases} [\underline{s_i},\ \overline{e_j}[& si & \underline{s_i} \leq \underline{s_j} \ \wedge\ \overline{e_j} \geq \overline{e_i} \\ [\underline{s_i},\ \underline{s_j}+\overline{e_j}-\underline{s_i}[& si & \underline{s_i} > \underline{s_j} \ \wedge\ \underline{s_i} < \underline{e_j} \ \wedge\ \underline{s_i} < \overline{s_j} \ \wedge\ \underline{s_j}+\overline{e_j}-\underline{s_i} \geq \overline{e_i} \\ [\underline{s_i},\ \underline{e_j}] & si & \underline{s_i} > \underline{s_j} \ \wedge\ \underline{s_i} < \underline{e_j} \ \wedge\ \underline{s_i} \geq \overline{s_j} \ \wedge\ \underline{e_j} \geq \overline{e_i} \\ [\overline{s_i}, \overline{e_j}] & si & \overline{s_i} \leq \underline{s_j} \ \wedge\ \overline{e_j} < \overline{e_i} \ \wedge\ \overline{e_j} > \overline{s_i} \ \wedge\ \overline{e_j} \leq \underline{e_j} \\ [\overline{s_i},\ \underline{s_j}+\overline{e_j}-\overline{s_i}] & si & \overline{s_i} > \underline{s_j} \ \wedge\ \overline{s_i} < \underline{e_j} \ \wedge\ \overline{s_i} < \overline{s_j} \ \wedge \\ & & \overline{s_i} < \underline{s_j}+\overline{e_j}-\overline{s_i} \leq \underline{e_i} \ \wedge\ \underline{s_j}+\overline{e_j}-\overline{s_i} < \overline{e_i} \\ [\overline{s_i},\ \underline{e_j}] & si & \overline{s_i} > \underline{s_j} \ \wedge\ \overline{s_i} < \underline{e_j} \ \wedge\ \overline{s_i} \geq \overline{s_j} \ \wedge \\ & & \underline{e_j} < \overline{e_i} \ \wedge\ \underline{e_j} > \overline{s_i} \ \wedge\ \underline{e_j} \leq \underline{e_i} \\ [\underline{s_i}+\overline{e_i}-\overline{e_j},\ \overline{e_j}] & si & \overline{e_j} < \overline{e_i} \ \wedge\ \overline{e_j} > \overline{s_i} \ \wedge\ \overline{e_j} > \underline{e_i} \ \wedge\ \underline{s_i}+\overline{e_i}-\overline{e_j} \leq \overline{s_j} \\ [\underline{s_i}+\overline{e_i}-\underline{e_j},\ \underline{e_j}] & si & \underline{e_j} < \overline{e_i} \ \wedge\ \underline{e_j} > \overline{s_i} \ \wedge\ \underline{e_j} > \underline{e_i} \ \wedge \\ & & \overline{s_j} \leq \underline{s_i}+\overline{e_i}-\underline{e_j} < \underline{e_j} \ \wedge\ \underline{s_j} < \underline{s_i}+\overline{e_i}-\underline{e_j} \end{cases}$$

Démonstration. Supposons $\nexists (i,j)$ tel que $(t_1, t_2) \in O_C(i,j)$. Alors par le lemme 4.2 et son symétrique, les deux conditions $\frac{\partial^- \mathrm{MI}(j,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{MI}(j,t_1,t_2)}{\partial t_2}$ et $\frac{\partial^- \mathrm{MI}(j,t_1,t_2)}{\partial t_1} > \frac{\partial^+ \mathrm{MI}(j,t_1,t_2)}{\partial t_1}$ ne peuvent pas être satisfaites. Par le lemme 4.1, f_1 ne peut pas être minimale. □

La table 4.1 montre les 8 intervalles d'intérêt pour la détection d'incohérence du raisonnement énergétique caractérisés par le lemme 4.3. Les intervalles qui restent d'intérêt sont notés, en reprenant les notations de la table 3.2, par une lettre (« X », « Y » ou « Z »). Les intervalles que le lemme 4.3 montre comme n'étant pas d'intérêt sont notés « . ».

t_1 \ t_2	$\overline{e_j}$	$\underline{e_j}$	$\overline{s_j}$	$\underline{s_j}$	$(\underline{s_j}+\overline{e_j})-t_1$
$\underline{s_i}$	X	X	.		Y
$\overline{s_i}$	X	X	.		Y
$\underline{e_i}$	.	.	.		.
$\overline{e_i}$					
$(\underline{s_i}+\overline{e_i})-t_2$	Z	Z	.		

TABLE 4.1 – Les 8 intervalles d'intérêt pour la détection d'incohérence du raisonnement énergétique du lemme 4.3. La notation utilisée reprend celle de la table 3.2 ; les intervalles démontrés comme n'étant pas d'intérêt pour la détection d'incohérence (par le lemme 4.3) sont marqués d'un « . ».

Théorème 4.1 (Suffisance de l'ensemble O_C)**.** *Pour assurer la détection d'incohérence énergétique (propriété 3.2, page 45) il est suffisant de vérifier les intervalles de la forme $O_C(\mathcal{A}) = \bigcup_{(i,j)\in\mathcal{A}^2} O_C(i,j)$.*

Démonstration. Supposons qu'il existe un intervalle $[t_1, t_2[$ tel qu'une incohérence y soit détectable : $\sum_{i\in\mathcal{A}} h_i \times \mathrm{MI}(i,t_1,t_2) - C \times (t_2 - t_1) < 0$. Par le lemme 4.3, $\exists [t_1^*, t_2^*[\in O_C(\mathcal{A})$ tel que $\sum_{i\in\mathcal{A}} h_i \times \mathrm{MI}(i,t_1^*,t_2^*) - C \times (t_2^* - t_1^*) \leq \sum_{i\in\mathcal{A}} h_i \times \mathrm{MI}(i,t_1,t_2) - C \times (t_2 - t_1)$. Alors vérifier $[t_1^*, t_2^*[$ mène à une réfutation. La caractérisation est suffisante. □

Cette caractérisation, plus précise, réduit le nombre d'intervalles d'intérêt pour chaque paire d'activités. Notre caractérisation mène à deux intervalles d'intérêt pour chaque paire d'activité, puisque au plus deux conditions peuvent être simultanément valides. Nous avons donc réduit le nombre d'intervalles d'intérêt par un facteur 7 en comparaison de la caractérisation de Baptiste *et al.* De plus, nous avons montré qu'aucun intervalle d'intérêt ne commence en $\underline{e_i}$ ou ne finit en $\overline{s_j}$.

Applications et experimentations

Baptiste *et al.* ont proposé un algorithme de détection d'incohérence en $O(n^2)$ basé sur leur caractérisation. Nous l'avons présenté dans l'état l'art (algorithme 4 page 47). Cet Algorithme boucle sur $O_1 = \bigcup_{a \in \mathcal{A}} \{\underline{s_a}, \overline{s_a}, \underline{e_a}\}$ pour calculer tous les intervalles d'intérêt commençant par une valeur de O_1.

Nous avons montré que $\underline{e_a}$ n'est pas une valeur de début d'intervalle qui peut mener à un échec. Nous proposons alors une adaptation de leur algorithme en replaçant les ensembles O_1 par $O'_1 = \bigcup_{a \in \mathcal{A}} \{\underline{s_a}, \overline{s_a}\}$ et (symétriquement) O_2 par $O'_2 = \bigcup_{a \in \mathcal{A}} \{\underline{e_i}, \overline{e_i}\}$.

Algorithm 6 Algorithme de détection d'incohérence énergétique

1: **procédure** ERCHECKER
2: **pour tout** $\langle i, t_1 \rangle$ de l'ensemble O'_1 **faire**
3: pente = $\sum_j \mathrm{MI}(j, t_1, t_1 + 1)$
4: SlackER= 0, $t_2old = t_1$
5: **pour tout** $\langle j, t_2 \rangle$ de l'ensemble $O'_2 \cup O_{t_1}$ par ordre croissant **faire**
6: SlackER = SlackER + $pente * (t_2 - t_2old)$
7: **si** SlackER $>$ capa $\times$ $(t_2 - t_1)$ **alors**
8: Une incohérence a été détecté !
9: **fin si**
10: $pente = pente + \mathrm{MI}(j, t_1, t_2 - 1) - 2 \times \mathrm{MI}(j, t_1, t_2) + \mathrm{MI}(j, t_1, t_2 + 1)$
11: $t_2old = t_2$
12: **fin pour**
13: **fin pour**
14: **fin procédure**

Nous avons effectué les expérimentations sur un ordinateur équipé d'un processeur 2.9 GHz Intel Core i7, avec Choco 3 [PFL14] (version 13.03). Pour vérifier expérimentalement le gain de notre caractérisation nous avons considéré 100 instances aléatoires et des instances de la librairie PSPLIB [KS96]. Les instances aléatoires ont 10 ou 20 activités, d'une durée choisie entre $[1, 10]$ et de hauteur entre $[1, 5]$. Nous avons utilisé l'heuristique de recherche *first fail* [HE79] (l'heuristique par défaut de Choco) et comparé notre algorithme ERCHECKER à l'algorithme ERCHECKERB :

Instances	ERCHECKERB (μs/nœud)	ERCHECKER (μs/nœud)	Gain en %
Random10	25	16	36
Random20	56	44	21
PspLib 30	619	451	27
PspLib 120	1 683	1 339	20

TABLE 4.2 – Comparaison en moyenne des algorithmes de détection d'incohérence énergétique.

Ces travaux ont fait l'objet d'une présentation au *doctoral program* de la conférence *CP'13* [DP13].

4.3 Relations de dominance des détections d'incohérence

Dans cette section, nous proposons une comparaison des relations de dominance entre les algorithmes de détection d'incohérence, pour cela nous considérons la caractérisation proposée dans la section 4.1.

Pour évaluer les dominances entre raisonnements, il suffit de comparer la fonction de consommation. Si la consommation d'un raisonnement est toujours inférieure, alors celui-ci ne peut détecter d'échec sans que le premier ne le détecte aussi ; il est dominé.

Proposition 4.1 (Time-Table-Edge-Finding $\leq$ Raisonnement Energétique)**.**

Démonstration. Comparons les fonctions $\mathrm{ConsoER}$ (formule (32), page 54) et $\mathrm{ConsoTTEF}$ (formule (38), page 55) en comparant la consommation d'une activité a en fonction de son inclusion dans un intervalle quelconque $[t_1, t_2[$.

Dans la formule de consommation Time-Table-Edge-Finding, Ω représente l'ensemble des activités entièrement incluses dans $[t_1, t_2[$. La consommation de a est donc comptabilisée soit au titre de $\mathrm{ConsoEF}$, soit au titre de $\mathrm{ConsoTT}$.

Si l'activité a appartient complètement à l'intervalle $[t_1, t_2[$, elle est comptabilisée dans $\mathrm{ConsoEF}$ et la consommation est égale à celle du raisonnement énergétique.

Si l'activité n'est pas entièrement incluse, alors elle est comptabilisée dans $\mathrm{ConsoTT}$, qui est inférieure ou égale à la consommation du raisonnement énergétique.

La consommation $\mathrm{ConsoTTEF}$ du raisonnement Time-Table-Edge-Finding est donc inférieure ou égale à la consommation $\mathrm{ConsoER}$ du raisonnement énergétique ; donc le raisonnement Time-Table-Edge-Finding est dominé par le raisonnement énergétique. □

Proposition 4.2 (Time-Table $\leq$ Time-Table-Edge-Finding)**.**

Démonstration. Exprimée comme dans l'equation (37) (page 55), la consommation Time-Table-Edge-Finding est trivialement supérieure ou égale à la consommation Time-Table. Par conséquent le raisonnement Time-Table-Edge-Finding domine le raisonnement Time-Table. □

Rappelons que Vilím a considéré, en introduisant l'algorithme Time-Table-Edge-Finding, qu'un algorithme Time-Table est préalablement appliqué. Par conséquent bien que le raisonnement Time-Table-Edge-Finding domine le raisonnement Time-Table, l'algorithme Time-Table-Edge-Finding ne cherche pas à ré-effectuer ce qui est deja effectué par un algorithme Time-Table ; il ne domine pas en ce sens les algorithmes Time-Table.

Proposition 4.3 (Edge-Finding $\leq$ Time-Table-Edge-Finding)**.**

Démonstration. Exprimée comme dans l'equation (38) (page 55), la consommation Time-Table-Edge-Finding est trivialement supérieure ou égale à la consommation Edge-Finding. Par conséquent le raisonnement Time-Table-Edge-Finding domine le raisonnement Edge-Finding. □

Il est important de noter que l'algorithme Time-Table-Edge-Finding tel qu'il est présenté par Vilím, ne domine pas le raisonnement Edge-Finding dans le cas général. C'est le cas lorsqu'un algorithme Time-Table est aussi appliqué, comme nous l'avons démontré en section 4.2.3 (page 56).

4.4 Extension de la caractérisation aux propagateurs

4.4.1 Intervalles d'intérêt pour le propagateur énergétique

Dans cette section nous nous intéressons à caractériser le plus finement possible les intervalles qu'il est réellement intéressant d'ajouter au calcul pour détecter les filtrages.

Baptiste *et al.* laissent ouverte la complétude de la caractérisation des intervalles d'intérêt que nous avons étudiée précédemment [BLN01]. Nous en faisons la démonstration dans cette section :

Théorème 4.2 (complétude de la caractérisation de Baptiste *et al.* des intervalles d'intérêt pour le propagateur énergétique [BLN01])**.** *Il est suffisant d'effectuer le test de détection de filtrage sur les intervalles de la forme O_B pour effectuer l'ensemble des déductions du raisonnement énergétique.*

Notre démonstration de la complétude de la caractérisation se base sur l'étude des conditions amenant à un filtrage (proposition 3.1, page 48). Rappelons que la proposition indique que si $\mathrm{Dispo}(a,t_1,t_2) - h_a \times \mathrm{LS}(a,t_1,t_2) < 0$, alors un filtrage peut être effectué. Nous cherchons alors à trouver tous les intervalles $[t_1,t_2]$ tels que la fonction induite $f_3^a : (t_1,t_2) \to \mathrm{Dispo}(a,t_1,t_2) - h_a \times \mathrm{LS}(a,t_1,t_2)$ est négative.

Étude de la fonction de consommation d'énergie

Comme pour la caractérisation des intervalles d'intérêt pour l'algorithme de détection d'incohérence, nous cherchons les points pour lesquels la fonction peut être minimale. Nous étudions alors les variations de f_3^a :

Lemme 4.4. *f_3^a est localement minimale en (t_1,t_2) seulement si une des quatre conditions est satisfaite :*

$$\exists (i,j), \frac{\partial^- \mathrm{MI}(i,t_1,t_2)}{\partial t_1} > \frac{\partial^+ \mathrm{MI}(i,t_1,t_2)}{\partial t_1} \wedge \frac{\partial^- \mathrm{MI}(j,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{MI}(j,t_1,t_2)}{\partial t_2} \tag{42}$$

$$\exists i, \frac{\partial^- \mathrm{MI}(i,t_1,t_2)}{\partial t_1} > \frac{\partial^+ \mathrm{MI}(i,t_1,t_2)}{\partial t_1} \wedge \frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_2} \tag{43}$$

$$\exists j, \frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_1} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_1} \wedge \frac{\partial^- \mathrm{MI}(j,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{MI}(j,t_1,t_2)}{\partial t_2} \tag{44}$$

$$\frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_1} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_1} \wedge \frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_2} \tag{45}$$

Démonstration. La démonstration est similaire à celle du lemme 4.1 : par le test de la dérivée seconde nous savons qu'une fonction ne peut être minimale que si la dérivée à gauche est plus petite que la dérivée à droite. Cette propriété doit être simultanément vérifiée pour les deux variables. La fonction f_3^a étant une somme, il faut qu'au moins un membre de la somme respecte cette propriété : il faut donc que $\frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_1} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_1}$ ou $\frac{\partial^- \mathrm{MI}(i,t_1,t_2)}{\partial t_1} > \frac{\partial^+ \mathrm{MI}(i,t_1,t_2)}{\partial t_1}$ et qu'au même point $\frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_2}$ ou $\frac{\partial^- \mathrm{MI}(j,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{MI}(j,t_1,t_2)}{\partial t_2}$. Ce qui démontre le lemme. □

Pour une activité a, nommons L^a l'ensemble des intervalles pouvant satisfaire l'une des quatre conditions du lemme 4.4. Nous pouvons décomposer L^a en quatre ensembles, représentant les intervalles satisfaisant chacune des conditions.

Les intervalles satisfaisant la condition (42) ont déjà été définis.
Ce sont les intervalles d'intérêt pour l'algorithme de détection d'incohérence : $O_C(\mathcal{A}\backslash\{a\})$.
Nommons $L_1^a(i)$ les intervalles pour lesquels i satisfait la condition (43).
L'ensemble des intervalles qui satisfont (43) est alors $\bigcup_{i\in\mathcal{A}\backslash a} L_1^a(i)$
De même, nommons $L_2^a(j)$ les intervalles pour lesquels j satisfait la condition (44).
L'ensemble des intervalles qui satisfont (44) est alors $\bigcup_{j\in\mathcal{A}\backslash a} L_2^a(j)$
Enfin, nommons L_3^a les intervalles satisfaisant la condition (45).

Pour caractériser chacun de ces intervalles, nous cherchons les points pour lesquels la dérivée à gauche est plus petite que la dérivée à droite pour les fonctions MI et LS. Puisque la fonction MI a déjà été étudiée dans la section 4.2.4, nous étudions ensuite les variations de la fonction $f_4^a : (t_1, t_2) \to \mathrm{LS}(a, t_1, t_2)$.

Lemme 4.5. *Pour toute activité a et tout point de temps t_1, il existe au plus un point de temps t_2 tel que* $\frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_2}$ *:*

- *Si* $t_1 < \underline{e_a}$ *alors seulement* $[t_1, \underline{e_a}[$ *est d'intérêt.*
- *Si* $t_1 \geq \underline{e_a}$ *alors aucun intervalle n'a à être considéré.*

Démonstration. Soit les trois cas suivants :

① $t_1 < \underline{s_a}$ (illustré par la figure 4.3) :
Alors $\mathrm{LS}(a, t_1, t_2) = \max(0, \min(\underline{e_a}, t_2) - \underline{s_a})$. Étudions ses valeurs en fonction de celles de t_2 :

(a) Si $t_2 \leq \underline{s_a}$ alors $\mathrm{LS}(a, t_1, t_2) = 0$.
(b) Si $\underline{s_a} \leq t_2 \leq \underline{e_a}$ alors $\mathrm{LS}(a, t_1, t_2) = t_2 - \underline{s_a}$.
(c) Si $\underline{e_a} \leq t_2$ alors $\mathrm{LS}(a, t_1, t_2) = p_a$.

le seul intervalle pour lequel $\frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_2}$ est alors $[t_1, \underline{e_a}[$.

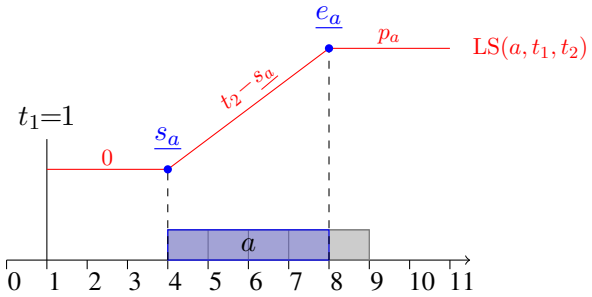


FIGURE 4.3 – Evolution de $\mathrm{LS}(a, t_1, t_2)$ en fonction de la fin de l'intervalle, t_2.

② $\underline{s_a} \leq t_1 < \underline{e_a}$ (illustré par la figure 4.4) :
Alors $\mathrm{LS}(a, t_1, t_2) = \max(0, \min(\underline{e_a}, t_2) - t_1)$. Étudions ses valeurs en fonction de celles de t_2 :

(a) Si $t_2 \leq \underline{e_a}$ alors $\mathrm{LS}(a, t_1, t_2) = t_2 - t_1$.

(b) Si $\underline{e_a} \leq t_2$ alors $\mathrm{LS}(a, t_1, t_2) = \underline{e_a} - t_1$.

Le seul intervalle pour lequel $\frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_2}$ est alors $[t_1, \underline{e_a}[$.

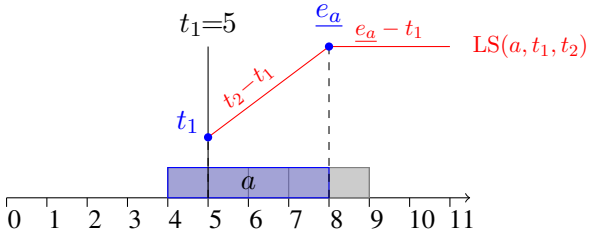


FIGURE 4.4 – Evolution de $\mathrm{LS}(a, t_1, t_2)$ en fonction de la fin de l'intervalle, t_2.

③ $\underline{e_a} \leq t_1$ (illustré par la figure 4.5) :
Alors $\mathrm{LS}(a, t_1, t_2) = 0$ et aucun intervalle ne satisfait la condition.

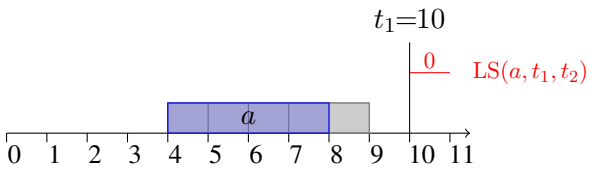


FIGURE 4.5 – Evolution de $\mathrm{LS}(a, t_1, t_2)$ en fonction de la fin de l'intervalle, t_2.

La combinaison des cas ①, ② et ③ démontre le lemme. □

Nous pouvons alors définir les intervalles $L_1^a(i)$, $L_2^a(j)$ et L_3^a en utilisant les caractérisations des lemmes 4.5, 4.2 et de leur symétrique.
Grace au lemme 4.5 et au symétrique du lemme 4.2, nous pouvons caractériser, pour chaque activité i, l'intervalle qui satisfait la condition (43).

$$L_1^a(i) = \begin{cases} [\underline{s_i},\ \underline{e_a}[& \text{si} \quad \underline{s_i} < \overline{s_a} \ \wedge\ \overline{e_i} < \underline{e_a} \\ [\underline{s_i} + \overline{e_i} - \underline{e_a},\ \underline{e_a}[& \text{si} \quad \underline{s_i} + \overline{e_i} - \underline{e_a} < \underline{e_a} \ \wedge\ \underline{s_i} + \overline{e_i} - \underline{e_a} < \overline{s_a} \ \wedge \\ & \qquad\quad \underline{e_a} < \overline{e_i} \ \wedge\ \underline{e_a} > \overline{s_i} \ \wedge\ \underline{e_a} > \underline{e_i} \\ [\overline{s_i},\ \underline{e_a}[& \text{si} \quad \overline{s_i} < \overline{s_a} \ \wedge\ \underline{e_a} < \overline{e_i} \ \wedge\ \underline{e_a} < \overline{s_i} \ \wedge\ \underline{e_a} \leq \underline{e_i} \end{cases}$$

Grâce au symétrique du lemme 4.5 et au lemme 4.2, nous pouvons caractériser, pour chaque activité j, l'intervalle qui satisfait la condition (44).

$$L_2^a(j) = \begin{cases} [\underline{s_a},\ \overline{e_j}[& \text{si} \quad \underline{s_a} \leq \underline{s_j} \ \wedge\ \overline{e_j} < \overline{e_a} \\ [\underline{s_a},\ \underline{s_j} + \overline{e_j} - \underline{s_a}[& \text{si} \quad \underline{s_a} > \underline{s_j} \ \wedge\ \underline{s_a} < \underline{e_j} \ \wedge\ \underline{s_a} < \overline{s_j} \ \wedge \\ & \qquad\quad \underline{s_j} + \overline{e_j} - \underline{e_a} > \underline{s_a} \ \wedge\ \underline{s_j} + \overline{e_j} - \underline{e_a} < \overline{e_a} \\ [\underline{s_a},\ \underline{e_j}[& \text{si} \quad \underline{s_a} > \underline{s_j} \ \wedge\ \underline{s_a} < \underline{e_j} \ \wedge\ \underline{s_a} \geq \overline{s_j} \ \wedge\ \overline{e_j} < \overline{e_a} \end{cases}$$

Grâce au lemme 4.5 et son symétrique, nous pouvons caractériser l'intervalle qui satisfait la condition (45).

$$L_3^a = \begin{cases} [\underline{s_a},\ \underline{e_a}[\end{cases}$$

Lemme 4.6. *f_3^a ne peut être localement minimale qu'en $(t_1, t_2) \in O_L^a$, avec $O_L^a = O_C(\mathcal{A}\backslash a) \cup L^a$.*

Démonstration. Supposons $\nexists (i,j)$ tel que $(t_1, t_2) \in O_L^a$. Alors par les lemmes 4.2 et 4.5 et leur symétrique, aucune des quatre conditions ne peut être satisfaite :

$$\exists (i,j), \frac{\partial^- \mathrm{MI}(i,t_1,t_2)}{\partial t_1} > \frac{\partial^+ \mathrm{MI}(i,t_1,t_2)}{\partial t_1} \wedge \frac{\partial^- \mathrm{MI}(j,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{MI}(j,t_1,t_2)}{\partial t_2}$$
$$\exists i, \frac{\partial^- \mathrm{MI}(i,t_1,t_2)}{\partial t_1} > \frac{\partial^+ \mathrm{MI}(i,t_1,t_2)}{\partial t_1} \wedge \frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_2}$$
$$\exists j, \frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_1} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_1} \wedge \frac{\partial^- \mathrm{MI}(j,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{MI}(j,t_1,t_2)}{\partial t_2}$$
$$\frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_1} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_1} \wedge \frac{\partial^- \mathrm{LS}(a,t_1,t_2)}{\partial t_2} > \frac{\partial^+ \mathrm{LS}(a,t_1,t_2)}{\partial t_2}$$

Donc, par le lemme 4.4, f_3^a ne peut pas être minimale. □

Un raisonnement identique amène à la caractérisation des intervalles d'intérêt pour le placement à droite, noté $R^a = \bigcup_{j \in \mathcal{A}\backslash a} R_1^a(j) \bigcup_{i \in \mathcal{A}\backslash a} R_2^a(i) \bigcup R_3^a$.

$$R_1^a(i) = \begin{cases} [\underline{s_j},\ \overline{e_i}[& \text{if} \quad \underline{s_j} > \underline{s_i} \ \wedge\ \overline{e_i} > \overline{e_j} \\ [\underline{s_j} + \overline{e_j} - \overline{e_i},\ \overline{e_i}[& \text{if} \quad \underline{s_j} + \overline{e_j} - \overline{e_i} < \overline{e_i} \ \wedge\ \underline{s_j} + \overline{e_j} - \overline{e_i} > \underline{s_i} \ \wedge \\ & \qquad\qquad \overline{e_i} < \underline{e_j} \ \wedge\ \overline{e_i} > \underline{e_j} \\ [\overline{s_j},\ \overline{e_i}[& \text{if} \quad \overline{s_j} > \underline{s_i} \ \wedge\ \overline{e_i} < \overline{e_j} \ \wedge\ \overline{e_i} > \overline{s_j} \ \wedge\ \overline{e_i} \leq \underline{e_j} \end{cases}$$

$$R_2^a(i) = \begin{cases} [\overline{s_j},\ \overline{e_i}[& \text{if} \quad \underline{s_j} \leq \underline{s_i} \ \wedge\ \overline{e_i} > \underline{e_j} \\ [\overline{s_j},\ \underline{s_i} + \overline{e_i} - \overline{s_j}[& \text{if} \quad \overline{s_j} > \underline{s_i} \ \wedge\ \overline{s_j} < \underline{e_i} \ \wedge\ \overline{s_j} < \overline{s_i} \ \wedge \\ & \qquad\qquad \underline{s_i} + \overline{e_i} - \overline{s_j} < \overline{s_j} \ \wedge\ \underline{s_i} + \overline{e_i} - \overline{s_j} < \underline{e_j} \\ [\overline{s_j},\ \underline{e_i}[& \text{if} \quad \overline{s_j} > \underline{s_i} \ \wedge\ \overline{s_j} < \underline{e_i} \ \wedge\ \overline{s_j} \geq \overline{s_i} \ \wedge\ \underline{e_i} > \underline{e_j} \end{cases}$$

$$R_3^a = \{[\overline{s_i},\ \overline{e_i}[\}$$

Le nombre d'intervalles d'intérêt pour une activité a est alors $|O_C(\mathcal{A} \setminus a) \cup L^a \cup R^a|$. Par construction $|O_C(\mathcal{A}\backslash a)| = 2(n-1)^2$ et $|L^a| = |R^a| = 2.n + 1$. En comparaison la caractérisation de Baptiste *et al.* dénombre par : $15.n$ intervalles d'intérêt (*cf.* table 3.2, page 45). Notre caractérisation réduit donc par un facteur 7 le nombre d'intervalles d'intérêt pour le filtrage d'une activité a.

De cette caractérisation des intervalles d'intérêt pour le filtrage d'une activité, nous pouvons en déduire que l'ensemble des intervalles d'intérêt pour le raisonnement énergétique est : $O_P = O_C(\mathcal{A}) \bigcup_{a \in \mathcal{A}} L^a \bigcup_{a \in \mathcal{A}} R^a$.

Théorème 4.3. *Pour détecter l'ensemble des propagations lié au raisonnement énergétique, il est suffisant d'analyser les intervalles de O_P.*

Démonstration. Supposons qu'il existe $[t_1, t_2[$ tel que $\mathrm{Dispo}(a, t_1, t_2) - h_a \times \mathrm{LS}(a, t_1, t_2) < 0$ (f_3^a est négative). Par le lemme 4.6, $\exists [t_1^*, t_2^*[\in O_L^a$ tel que $f_3^a(t_1^*, t_2^*) \leq f_3^a(t_1, t_2)$. f_3^a est aussi négative en (t_1^*, t_2^*), alors étudier $[t_1^*, t_2^*[$ mène à un filtrage. La caractérisation est suffisante. □

Cela répond à la seconde question laissée ouverte dans [BLN01] :

Théorème 4.4. *La caractérisation des intervalles d'intérêt O_B donnée par Baptiste* et al. *est suffisante pour assurer une propagation complète du raisonnement énergétique.*

Démonstration. Par le théorème 4.3, O_P est suffisant et $O_P \subseteq O_B$. □

Applications et expérimentations

Nous venons de caractériser finement les intervalles d'intérêt pour le propagateur énergétique, en réduisant par un facteur 7 le nombre théorique d'intervalles pour chaque activité. Pour rappel les intervalles d'intérêt pour l'ensemble des activités sont de la forme $O_P = O_C(\mathcal{A}) \bigcup_{a\in\mathcal{A}} L^a \bigcup_{a\in\mathcal{A}} R^a$. Nous proposons donc une adaptation en deux parties de l'algorithme ERPROPAGB. La première pour les intervalles de $O_C(\mathcal{A})$, la seconde pour les intervalles de $\bigcup_{a\in\mathcal{A}} L^a$ et $\bigcup_{a\in\mathcal{A}} R^a$.

L'algorithme ERPROPAGB parcourt l'ensemble $O_B(\mathcal{A})$ pour effectuer toutes les déductions. La première partie de notre algorithme est une adaptation directe où l'on remplace la boucle principale par une boucle sur $O_C(\mathcal{A})$. (lignes 2 à 17 de l'algorithme ERPROPAGATEUR)

La seconde partie effectue la détection de filtrage sur les intervalles d'intérêt de chaque activité : L^a (lignes 19 à 26) et R^a (lignes 27 à 34).

Algorithm 7 Propagateur énergétique adapté de l'algorithme ERPROPAGB

1: **procédure** ERPROPAGATEUR
2: **pour tout** $(t_1, t_2) \in O_C(\mathcal{A})$ **faire** ▷ Partie 1
3: $W := \sum_{a\in\mathcal{A}} h_a \times \mathrm{MI}(a, t_1, t_2)$
4: **si** $W > \mathrm{capa} \times (t_2 - t_1)$ **alors**
5: Une incohérence a été détecté !
6: **sinon**
7: **pour tout** $a \in \mathcal{A}$ **faire**
8: $avail := C \times (t_2 - t_1) - W + h_a \times \mathrm{MI}(a, t_1, t_2)$
9: **si** $avail < h_a \times \mathrm{LS}(a, t_1, t_2)$ **alors**
10: $\underline{s_a} := \max(\underline{s_a},\ t_2 - \frac{1}{h_a} \times avail)$
11: **fin si**
12: **si** $avail < h_a \times \mathrm{RS}(a, t_1, t_2)$ **alors**
13: $\overline{e_a} := \min(\overline{e_a},\ t_1 + \frac{1}{h_a} \times avail)$
14: **fin si**
15: **fin pour**
16: **fin si**
17: **fin pour**
18: **pour tout** $a \in \mathcal{A}$ **faire** ▷ Partie 2
19: **pour tout** $(t_1, t_2) \in L^a$ **faire**
20: $avail := \mathrm{capa} \times (t_2 - t_1) - \sum_{i\in\mathcal{A}\setminus a} h_a \times \mathrm{MI}(i, t_1, t_2)$
21: **si** $avail < h_a \times \mathrm{MI}(a, t_1, t_2)$ **alors**
22: Une incohérence a été détecté !
23: **sinon si** $avail < h_a \times \mathrm{LS}(a, t_1, t_2)$ **alors**
24: $\underline{s_a} := \max(\underline{s_a},\ t_2 - \frac{1}{h_a} \times avail)$
25: **fin si**
26: **fin pour**
27: **pour tout** $(t_1, t_2) \in R^a$ **faire**
28: $avail := \mathrm{capa} \times (t_2 - t_1) - \sum_{i\in\mathcal{A}\setminus a} h_a \times \mathrm{MI}(i, t_1, t_2)$
29: **si** $avail < h_a \times \mathrm{MI}(a, t_1, t_2)$ **alors**
30: Une incohérence a été détecté !
31: **sinon si** $avail < h_a \times \mathrm{RS}(a, t_1, t_2)$ **alors**
32: $\overline{e_a} := \min(\overline{e_a},\ t_1 + \frac{1}{h_a} \times avail)$
33: **fin si**
34: **fin pour**
35: **fin pour**
36: **fin procédure**

Nous avons effectué une série de tests en pratique pour comparer les deux algorithmes. Nous avons effectué les expérimentations sur un ordinateur équipé d'un processeur 2.9 GHz Intel Core i7, avec Choco 3 [PFL14] (version 13.03). Nous avons considéré 100 instances aléatoires et des instances de la PSPLIB [KS96]. Les instances aléatoires ont 10 ou 20 activités, d'une durée choisie entre $[1, 10]$ et de hauteur entre $[1, 5]$. Nous avons utilisé l'heuristique de recherche *first fail* [HE79] (l'heuristique par défaut de choco) et comparé notre algorithme ERPROPAGATEUR à l'algorithme état de l'art ERPROPAGB.

Instances	ERPROPAGB (μs/nœud)	ERPROPAGATEUR (μs/nœud)	Gain in %
Random10	244	91	62
Random20	641	327	49
PspLib 30	8 809	4 372	50
PspLib 120	151 390	41 418	72

TABLE 4.3 – Comparaison du temps moyen par nœud des propagateurs énergétiques.

La table 4.3 montre un gain entre 49 et 72% en faveur de notre algorithme. Le gain substantiel de la caractérisation théorique permet de diviser entre 2 et 4 le temps d'éxécution en pratique sur les instances typiques de la littérature.

Nous avons aussi chercher à comparer le nouvel algorithme, sur des preuves d'optimalité, au raisonnement Time-Table-Edge-Finding qui est l'état de l'art des algorithmes de propagation pour la contrainte Cumulative.

Nous avons donc combiné notre implémentation des différents algorithmes :

- Time-Table et Time-Table-Edge-Finding
- Time-Table et Raisonnement énergétique.

Sur les instances Random20 utilisées précédemment, la combinaison Time-Table / Raisonnement énergétique à été capable de prouver l'optimalité de 63 des 100 instances. La combinaison Time-Table / Time-Table-Edge-Finding a été capable de prouver l'optimalité pour seulement 8 instances.

Ce résultat démontre l'intérêt que peut avoir un propagateur énergétique dans un solveur de contraintes. Il est cependant mitigé car il ne concerne que la preuve d'optimalité et il est principalement dû à l'absence d'un algorithme de détection d'incohérence énergétique dans la combinaison Time-Table / Time-Table-Edge-Finding.

4.4.2 Intervalles d'intérêt pour le propagateur Time-Table-Edge-Finding

Kameugne a démontré que le raisonnement Time-Table-Edge-Finding ne domine pas les raisonnements Edge-Finding et Extended-Edge-Finding [KFSN14]. Il fournit en effet un contre-exemple à la démonstration de Vilím :

Exemple 4.1. *La figure 4.6 illustre un contre-exemple de la dominance de Time-Table-Edge-Finding sur Edge-Finding.*

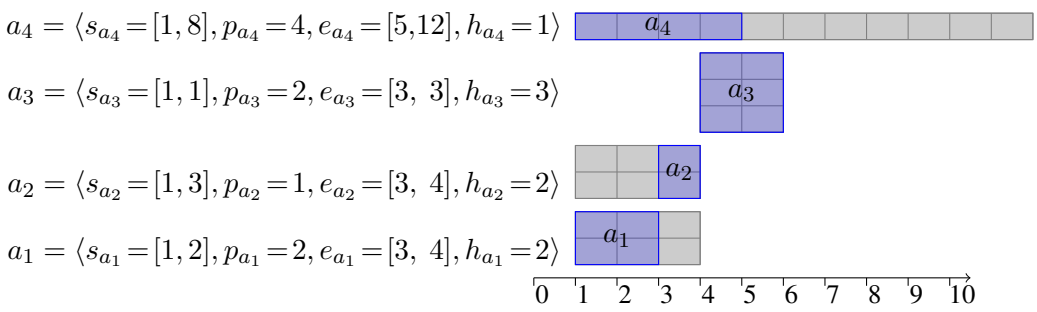


FIGURE 4.6 – Contre-exemple de la dominance de Time-Table-Edge-Finding sur Edge-Finding [KFSN14]

La démonstration de la dominance de Time-Table-Edge-Finding sur les raisonnements Edge-Finding initialement proposée se base uniquement sur les calculs de l'énergie [Vil11]. Et en effet, comme nous l'avons démontré, le calcul de l'énergie du raisonnement Time-Table-Edge-Finding domine ceux du raisonnement Edge-Finding. Le contre-exemple qu'expose Kameugne tient dans la caractérisation des intervalles d'intérêt. En effet, en limitant le test de propagation aux ensembles d'activités ayant une partie libre, l'algorithme ne considère pas les activités fixées, qui peuvent également apporter un filtrage. Dans l'exemple 4.1, l'algorithme Time-Table-Edge-Finding ignore l'ensemble $\Omega = \{a_1, a_2, a_3\}$ car l'activité a_3 est fixée. Nous pourrions en déduire que Time-Table-Edge-Finding et Edge-Finding ne sont pas comparables.

Dans un solveur de contraintes, il est commun de cumuler plusieurs raisonnements pour améliorer le filtrage. L'ajout du propagateur Time-Table est souvent réalisé, car il est rapide à exécuter. Nous allons maintenant démontrer que, comme pour l'algorithme de détection d'incohérence, si l'on considère que le raisonnement Time-Table a été appliqué, alors le raisonnement Time-Table-Edge-Finding domine le raisonnement Edge-Finding.

Nous avons deja montré que le calcul de l'énergie du raisonnement Time-Table-Edge-Finding est plus grand que celui du raisonnement Edge-Finding. Il nous reste donc à montrer que les intervalles non caractérisés par l'algorithme sont dominés.

Preuve de dominance des intervalles du propagateur Time-Table-Edge-Finding. Démontrons que les intervalles qui ne sont pas des intervalles définis par un ensemble $\Omega \subset \mathcal{A}^{EF}$ sont dominés par un intervalle de plus petite taille.

Soit un intervalle $[t_1, t_2[$ tel que t_2 ne soit pas une date d'intérêt pour le raisonnement Time-Table-Edge-Finding : $\nexists a \in \mathcal{A}^{EF}, \overline{e_a} = t_2$. Nous cherchons à montrer que, si l'intervalle $[t_1, t_2[$ détecte qu'un filtrage peut être effectué, alors $[t_1, t_2 - 1[$ le détecte aussi. La détection (formule (19), page 43) qu'un filtrage peut être effectué peut s'écrire, également :

$$\text{ConsoTTEF}(\mathcal{A}, [t_2, t_1]) + \text{add}(t_1, t_2, a) > \text{capa} \times (t_2 - t_1)$$

Nous cherchons donc à montrer que :

$$\begin{array}{lll} \text{ConsoTTEF}(\mathcal{A}, [t_2, t_1]) & + \text{add}(t_1, t_2, a) & > \text{capa} \times (t_2 - t_1) \\ \text{ConsoTTEF}(\mathcal{A}, [t_2 - 1, t_1]) + \text{add}(t_1, t_2 - 1, a) & & > \text{capa} \times ((t_2 - 1) - t_1) \end{array} \quad \Longrightarrow$$

Cela est vrai si :

$$\begin{array}{l} \text{ConsoTTEF}(\mathcal{A}, [t_1, t_2[) - \text{ConsoTTEF}(\mathcal{A}, [t_1, t_2 - 1[) + \\ \qquad\qquad \text{add}(t_1, t_2, a) - \text{add}(t_1, t_2 - 1, a) \leq \text{capa} \end{array}$$

Décomposons la consommation telle qu'exprimée par la formule (38). t_2 n'étant n'est pas une date d'intérêt pour le raisonnement Edge-Finding, l'ensemble Ω des activités comptabilisées pour le compte du raisonnement Edge-Finding est le même pour les deux intervalles donc $\text{ConsoEF}(\mathcal{A}, [t_1, t_2[) = \text{ConsoEF}(\mathcal{A}, [t_1, t_2 - 1[)$. Il reste donc le calcul de l'énergie du à Time-Table. En détaillant les sommes, on obtient :

$$\Big(\sum_{i \in \mathcal{A} \setminus \Omega, t_2 \in [\overline{e_i}, \underline{s_i}[} h_i \Big) + \qquad \text{add}(t_1, t_2, a) - \text{add}(t_1, t_2 - 1, a) \leq \text{capa}$$

Par construction, $\text{add}(t_1, t_2, a) - \text{add}(t_1, t_2 - 1, a)$ représente la quantité d'énergie libre ajoutée dans l'intervalle. Elle est égale à 0 ou h_a (*cf.* table 3.1, page 43). L'équation devient :

$$\Big(\sum_{i \in \mathcal{A} \setminus \Omega, t_2 \in [\overline{e_i}, \underline{s_i}[} h_i \Big) + \qquad h_a \leq \text{capa}$$

Il s'agit de la propriété au point fixe du raisonnement Time-Table (propriété 3.2, page 31) . Le raisonnement Time-Table ayant déjà été appliqué, cette condition est respectée. La démonstration dans le cas où t_1 n'est pas une date d'intérêt pour le raisonnement Time-Table-Edge-Finding est équivalente. □

Par conséquent il est suffisant d'effectuer les tests de détection de filtrage pour les intervalles d'activités libres. Lorsqu'un propagateur Time-Table est préalablement effectué, le raisonnement Time-Table-Edge-Finding domine effectivement le raisonnement Edge-Finding.

4.5 Conclusions et Perspectives

Dans ce chapitre, nous avons proposé une étude théorique des différents raisonnements pour la contrainte Cumulative. Nous avons montré que les principaux raisonnements pouvaient être vus comme des raisonnements basés sur l'énergie sur un intervalle de temps, tel que présenté par la propriété 4.1.

En ce basant sur cette notion de consommation sur un intervalle et en comparant directement les fonctions correspondantes, nous avons pu redonner des démonstrations simples sur les relations de dominances entre les différentes règles de détection d'incohérence. Nous avons aussi pu montrer qu'appliqué avec un propagateur Time-Table le raisonnement Time-Table-Edge-Finding domine le raisonnement Edge-Finding, comme annoncé dans [Vil11].

Dans le cadre du raisonnement énergétique, en étudiant plus finement les propriétés de la fonction de consommation sur un ensemble d'activités, nous avons obtenu les deux résultats théoriques suivants :

- nous avons montré qu'il était possible de réduire le nombre d'intervalle d'intérêt tout en conservant la complétude du filtrage (théorème 4.4, page 66), ce qui réponds à une question laissé ouverte par Baptiste *et al.* [BLN01, page 89].
- nous avons réduit significativement le nombre d'intervalles d'intérêt à considerer dans l'algorithme de détection d'incohérence et de filtrage (sections 4.2.4 et 4.4.1). D'un point de vue pratique, cette réduction du nombre d'intervalles à considerer a amené, sur le temps d'exécution du propagateur, à des gains d'un facteur compris entre deux et quatre.

Ces résultats ouvrent des perspectives intéressantes concernant le propagateur énergétique. Notre travail, principalement théorique, a naturellement aboutit à un nouvel algorithme du raisonnement énergétique. Une perspective est d'aboutir à un algorithme ayant un temps d'exécution encore plus faible que les algorithmes actuels du raisonnement énergétique, tout en gardant un pouvoir de filtrage plus élevé que l'algorithme Time-Table-Edge-Finding.

Ces travaux on fait l'objet d'une publication à la conférence internationale *CP'14* [DP14], puis d'une communication à la conférence nationale *JFPC'14* [Der14].

5

L'ordonnancement robuste

Sommaire

5.1 Introduction

Dans de nombreuses applications réelles [BN02, BS03, BS04, WBB09, ZVS+13], la notion d'optimalité du coût est souvent secondaire face à la capacité de mise en oeuvre de la solution dans un contexte où différents aléas peuvent survenir en production. De ce point de vue, la solution d'un problème d'ordonnancement est souvent peu robuste : si l'une des activités venait à être retardée, l'ensemble du planning pourrait être remis en question.

La littérature propose deux grandes approches pour traiter la robustesse dans les problèmes combinatoires. La première approche, probabiliste, repose sur la génération d'échantillons de scénarios résolus à l'aide de méthode usuelle [DJB01]. La seconde approche, exacte, propose d'intégrer la robustesse au sein du modèle grâce au concept de *super-solution* [Heb04]. Cependant, cette deuxième approche considère que toutes les variables ont le même statut (*cf.* section 3.2.1, page 49), alors qu'elles sont de nature hétérogène dans les problèmes d'ordonnancement cumulatif (dates, durées, consommations, etc.). De plus, cette deuxième approche permet de réparer une

solution en avançant une activité avant qu'un aléa ne survienne. Si cela reste pertinent dans le cas où l'aléa est connu suffisamment longtemps à l'avance, ce type d'approche ne convient pas lorsque les aléas surviennent tardivement. Nous cherchons donc à spécialiser la notion de super-solution, pour l'ordonnancement cumulatif.

Dans cette partie, nous définissons un problème générique dans lequel chaque activité peut être retardée d'un certain temps sans que l'on soit obligé de recalculer la planification. Nous introduisons FlexC, une contrainte cumulative dédiée à ce problème. Notre paradigme répond aux trois aspects suivants :

1. Un cadre de travail déclaratif : Nous nous plaçons dans le cadre où les solutions ne peuvent pas être recalculées à tout moment. Le délai maximal autorisé est une donnée et diffère d'une activité à une autre.

2. Une définition spécialisée : pour modéliser le problème, nous utilisons une définition de la robustesse basée sur la sémantique du problème lui-même. Toutes les variables n'ont pas nécessairement le même statut.

3. Une approche modulaire : notre approche doit être valide quelles que soient les contraintes métier que l'utilisateur peut être amené à ajouter.

Afin de valider la pertinence de ce travail, nous l'avons appliqué à un problème réel d'ordonnancement de grue dans un port [ZVS+13]. Ce problème est soumis à une grande variété de perturbations, allant du simple retard d'un employé à un ralentissement dû à de mauvaises conditions météorologiques. Notre approche a démontré sa capacité à traiter des instances de taille réelle.

5.2 FlexC : Une contrainte Cumulative robuste

5.2.1 Description du nouveau paradigme

Dans cette section, nous proposons une nouvelle définition d'un problème cumulatif dans lequel chaque activité a (définit par son début s_a, sa durée p_a, sa fin e_a et sa hauteur h_a) peut être retardée jusqu'à k_a unités de temps sans affecter l'ordonnancement des autres activités. Formellement, nous souhaitons, pour un paramètre r, autoriser à tout point de temps que r activités puissent être rallongées ou retardées. Nous utilisons la notation suivante pour la hauteur maximale d'ordre i d'un ensemble d'activités.

Notation 5.1 (i-ème plus haute activité)**.** *Étant donné $\mathcal{A}^{\downarrow}$ la collection d'activités d'un ensemble $\mathcal{A}$ trié par hauteur décroissante, $\max^{i}_{a\in\mathcal{A}}(h_a)$ est la hauteur de la i-ème activité dans $\mathcal{A}^{\downarrow}$.*

Définition 5.1 (RCuSP^r)**.** *Soit un ensemble de n activités $\mathcal{A} = \{a_1, \ldots, a_n\}$, un ensemble de n entiers positifs associés un à un à chaque activité $\mathcal{K} = \{k_{a_1}, \ldots, k_{a_n}\}$, et deux entiers positifs r, capa. Une solution au problème RCuSP^r satisfait les conditions (46) et (47) :*

$$s_a + p_a = e_a \qquad \forall a \in \mathcal{A} \tag{46}$$

$$\Big(\sum_{\substack{j\in\mathcal{A},\\ t\in[s_j,e_j[}} h_j\Big) + \sum_{i=1}^{i=r} \max^{i}_{\substack{j\in\mathcal{A},\\ t\in[e_j,e_j+k_j[}} (h_j) \leq \text{capa} \qquad \forall t \in \mathbb{N} \tag{47}$$

La définition du problème RCuSP^r reprend la contrainte (1) (page 28) d'intégrité d'une activité de la contrainte cumulative, et modifie la contrainte (2) (page 28) de ressource en ajoutant à chaque point de temps la consommation de r nouvelles activités. Les activités ordonnancées avant le point de temps t pouvant être repoussées jusqu'à t sont comptabilisées ; Le cas le plus critique étant que les r activités les plus demandeuses de ressources soient décalées ; leur hauteur est ajoutée à la consommation.

Cette définition considère la robustesse comme étant la capacité à retarder une activité a d'au plus k_a point de temps, ou alors d'allonger a de k_a point de temps.

Exemple 5.1. *La figure 5.1 montre une solution à un problème cumulatif,* Cumulative$(\mathcal{A}, \text{capa})$.

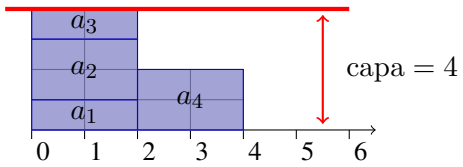


FIGURE 5.1 – Solution à un problème cumulatif robuste.

L'ordonnancement reste valide si l'on allonge une activité d'une durée quelconque : quelque soit $\mathcal{K}$ cet ordonnancement reste valide pour le RCuSP^1. De plus, il est possible d'allonger les activités a_1, a_3 et a_4 simultanément. Seuls les allongements simultanés des activités a_2 et a_1, ou, a_2 et a_3 mènent à des échecs. Bien que l'ordonnancement ne soit pas valide pour le RCuSP^2, il est possible de retarder plusieurs activité à la fois.

Une solution au problème RCuSP^r est une solution du problème CuSP à laquelle il est possible de repousser r activités sans impacter les autres activités, cela peut être vue comme une $(r; 0)$-super-solution spécialisée pour le CuSP. Plusieurs propriétés des super-solutions s'appliquent donc aussi au RCuSP^r. Plus la valeur de r est élevée, plus la solution sera robuste. Une solution pour RCuSP^r est aussi une solution pour RCuSP^{r-1}.

Si r est élevé, alors on obtient le même modèle que celui de l'approche naïve (approche dans laquelle on allonge la durée de toutes les activités). Cela arrive lorsque $r = n$, mais on peut améliorer cette borne en calculant le nombre d'activités pouvant être planifiées à un même point de temps, par exemple les r activités les moins hautes telles que leur somme dépasse la capacité.

Réciproquement, dans un cadre d'optimisation, le RCuSP^1 (qui est le moins contraint) est le modèle qui propose les solutions de meilleur coût.

Considérer qu'à chaque point de temps une seule activité peut être retardée ne signifie pas qu'une seule activité peut être retardée dans la solution proposée. Au contraire de nombreuses activités peuvent être repoussées (*cf.* exemple 5.1). C'est le cas pour des activités ne se recoupant pas ou de petite hauteur. Une solution du RCuSP^1 permettra en pratique de repousser, dans des fenêtres de temps disjointes, bien plus d'une activité.

Dans notre cadre appliqué, nous considérons que ce choix est le meilleur, d'une part parce qu'il est celui proposant une solution de meilleur coût, mais d'autre part parce que la robustesse qu'il apporte est suffisante. Nous travaillons donc sur les propriétés et algorithme du RCuSP^1 que nous exprimons à l'aide de la contrainte FlexC$(\mathcal{A}, \text{capa}, \mathcal{K})$, tout comme le CuSP est exprimé à l'aide de la contrainte Cumulative$(\mathcal{A}, \text{capa})$.

Définition 5.2 (Contrainte FlexC)**.** *Soient un ensemble $\mathcal{A}$ de n activités $\{a_1, \ldots, a_n\}$, une limite de consommation de ressource* capa *et un facteur de robustesse pour chaque activité $\mathcal{K} \in \mathcal{N}^n = \{k_{a_1}, \ldots, k_{a_n}\}$. La contrainte* FlexC$(\mathcal{A}, \text{capa}, \mathcal{K})$ *est satisfaite si et seulement si la condition d'intégrité des activités (48) et la condition de capacité (49) sont vérifiées :*

$$s_a + p_a = e_a \qquad \forall a \in \mathcal{A} \tag{48}$$

$$\Big(\sum_{\substack{j \in \mathcal{A}, \\ t \in [s_j, e_j[}} h_j\Big) + \max_{\substack{j \in \mathcal{A}, \\ t \in [e_j, e_j + k_j[}} (h_j) \leq \text{capa} \qquad \forall t \in \mathbb{N} \tag{49}$$

Remarquons que le problème cumulatif est une relaxation du problème cumulatif robuste. Cette propriété s'exprime de la façon suivante :

Propriété 5.1. FlexC$(\mathcal{A}, \text{capa}, \mathcal{K}) \Rightarrow$ Cumulative$(\mathcal{A}, \text{capa})$.

Démonstration. Les propriétés d'intégrité (48) et (1) sont identiques. Puisque h_i est positive, leur somme l'est aussi donc (49) $\Rightarrow$ (2). □

La propriété 5.2 montre qu'il y a une équivalence entre la contrainte FlexC et l'ensemble des contraintes Cumulative dans lesquelles on a allongé de k_a unités de temps une activité a.

Notation 5.2. *Notons $\mathcal{A}'_a$ l'ensemble des activités de $\mathcal{A}$ auquel seule l'activité $a = \langle s_a,\ p_a,\ e_a,\ h_a \rangle$ a été allongée. Soit $a' = \langle s_a,\ p_a + k_a,\ e_a + k_a,\ h_a \rangle$, $\mathcal{A}'_a = \mathcal{A} \setminus \{a\} \cup \{a'\}$*

Propriété 5.2.

$$\text{FlexC}(\mathcal{A}, \text{capa}, \mathcal{K}) \Leftrightarrow \bigwedge_{a \in \mathcal{A}} \text{Cumulative}(\mathcal{A}'_a, \text{capa})$$

Démonstration. Montrons que l'ensemble des contraintes est identique pour les deux problèmes.

Concernant les conditions (48) d'intégrité, pour chaque activité a' de Cumulative$(\mathcal{A}'_a, \text{capa})$ une contrainte supplémentaire est créée : $s'_a + p'_a = e'_a$ qui correspond à $s_a + (p_a + k_a) = (e_a + k_a)$. Cette dernière contrainte est redondant avec $s_a + p_a = e_a$. Les conditions d'intégrité sont donc équivalentes dans les deux cas.

L'ensemble des contraintes (49) de consommation peut être décomposé pour chaque activité prise en compte dans la fonction max :

$$\forall t \in \mathbb{N}, \forall j \in \mathcal{A}, t \in [e_j, e_j + k_j[\qquad \Big(\sum_{\substack{i \in \mathcal{A}, \\ t \in [s_i, e_i[}} h_i\Big) + h_j \leq \text{capa}$$

Ces contraintes sont identiques à celles induites par la contrainte Cumulative$(\mathcal{A}'_j, \text{capa})$.

□

Nous venons de montrer qu'il est possible de modéliser le problème avec n contraintes Cumulative. Il est possible de modéliser par la même décomposition lorsque r est plus grand que 1. Il suffit de générer pour chaque combinaison de r activités une contrainte Cumulative. Dans ce manuscrit nous nous focalisons sur le cas $r = 1$. Nous ne donnons donc pas la preuve pour tout r, qui suit exactement le modèle du cas $r = 1$.

Une telle décomposition en n propagateurs risque de ne pas être efficace en pratique, à cause du phénomène de "ping-pong" dû aux évènements survenant sur des variables partagées par ces propagateurs. Cette limitation sera illustrée dans la section 5.2.7 (page 87).

5.2.2 Détection d'incohérence

Nous avons vu, dans les chapitres précédents, que le raisonnement Time-Table permet de détecter efficacement les incohérences pour des problèmes à plusieurs milliers d'activités. Nous avons donc cherché à adapter ce raisonnement à la contrainte FlexC.

Nous venons de voir que le problème pouvait se modéliser à l'aide d'une décomposition en n contraintes Cumulative. Nous cherchons donc à obtenir un niveau de consistance équivalent à la consistance obtenue par l'application du raisonnement Time-Table sur chacune des n contraintes Cumulative sans créer explicitement ces n contraintes Cumulative. Pour cela nous introduisons la notion de partie $\mathcal{K}$-obligatoire, le complément entre la partie obligatoire de l'activité rallongée et la partie obligatoire de l'activité d'origine :

Définition 5.3 (partie $\mathcal{K}$-obligatoire)**.** *Soit $a \in \mathcal{A}$ une activité et $k_a \in \mathcal{K}$. La partie $\mathcal{K}$-obligatoire de a, noté KCP_a, est l'intervalle $[\max(\overline{s_a}, \underline{e_a}),\ \underline{e_a}+k_a[$.*

Exemple 5.2 (partie $\mathcal{K}$-obligatoire)**.** *Il existe trois cas possibles pour le placement au plus tôt de l'activité a, par rapport au placement au plus tard. Ils sont illustrés sur les figures 5.2, 5.3 et 5.4 par une activité a de durée 3, et de durée d'allongement 3. La fin au plus tard est 10 et le début au plus tôt change en fonction des cas.*

cas 1 : *si $\underline{e_a} + k_a \leq \overline{s_a}$, alors il n'existe ni partie obligatoire ni partie $\mathcal{K}$-obligatoire.*

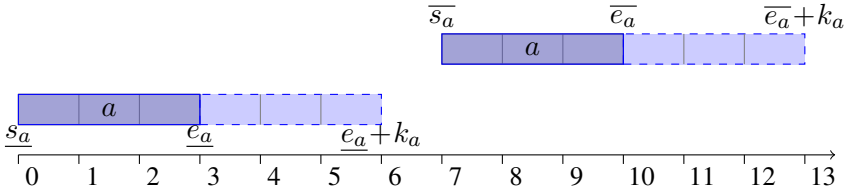


FIGURE 5.2 – Exemple d'activité sans partie $\mathcal{K}$-obligatoire.

cas 2 : *si $\underline{e_a} \leq \overline{s_a} < \underline{e_a} + k_a$, alors seule une partie $\mathcal{K}$-obligatoire existe, à savoir $[\overline{s_a},\ \underline{e_a}+k_a[$.*

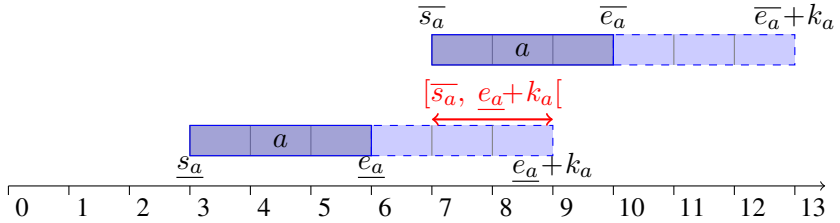


FIGURE 5.3 – Exemple de partie $\mathcal{K}$-obligatoire (en rouge) pour une activité n'ayant pas de partie obligatoire.

cas 3 : *si $\overline{s_a} < \underline{e_a}$, alors une partie obligatoire ainsi qu'une partie $\mathcal{K}$-obligatoire existent. Elles correspondent respectivement à $[\overline{s_a},\ \underline{e_a}[$ et $[\underline{e_a},\ \underline{e_a}+k_a[$.*

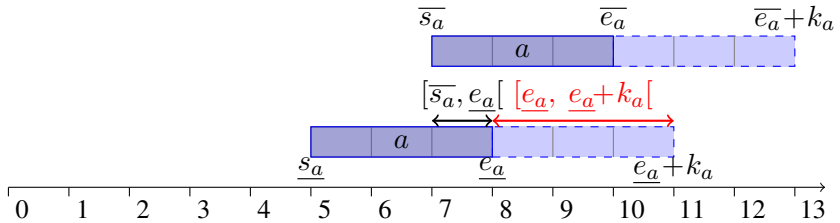


FIGURE 5.4 – Exemple de partie $\mathcal{K}$-obligatoire (en rouge) pour une activité ayant une partie obligatoire.

La règle de cohérence du raisonnement Time-Tabling (propriété 3.1, page 30) adaptée pour la contrainte FlexC est alors :

Propriété 5.3 (Règle de cohérence du raisonnement Time-Tabling pour FlexC)**.**

$$\text{FlexC}(\mathcal{A}, \text{capa}, \mathcal{K}) \implies \forall t, \Big(\sum_{i \in \mathcal{A}, t \in [\underline{s_i}, \underline{e_i}[} h_i\Big) + \Big(\max_{j \in \mathcal{A}, t \in KCP_j} h_j\Big) \leq \text{capa} \tag{50}$$

La démonstration que la règle 5.3 apporte la même déduction que l'ensemble des règles de cohérence Time-Table sur la décomposition est similaire à celle de la propriété 5.2 : à chaque activité j prise en compte par la fonction max correspond la contrainte $\text{Cumulative}(\mathcal{A}'_j, \text{capa})$. L'ensemble des contraintes pour FlexC et les Cumulative sont donc identiques. Ainsi les raisonnements sont identiques.

Le profil des parties $\mathcal{K}$-obligatoires représente, à chaque point temps, la hauteur cumulée de la consommation Time-Table pour FlexC (*cf.* (50)). Si, à un point de temps, ce profil dépasse la capacité, un échec est détecté (par la propriété 5.3).

5.2.3 Filtrage

Nous proposons maintenant une règle de filtrage Time-Table pour la contrainte FlexC. Nous l'appelons filtrage Time-Table, car elle apporte le même filtrage que celui apporté par le raisonnement Time-Table sur chacune des contraintes Cumulative de la décomposition (*cf.* propriété après filtrage 3.2 (page 31) pour une contrainte Cumulative).

Propriété 5.4 (Propriété après le filtrage Time-Table pour FlexC)**.** *Soit* $\text{FlexC}(\mathcal{A}, \text{capa}, \mathcal{K})$*, après application d'un propagateur Time-Table, les conditions* (51) *et* (52) *doivent être respectées pour tout* $a \in \mathcal{A}$ *:*

$$\forall t \in [\underline{s_a}, \underline{e_a}[, \qquad h_a + \sum_{\substack{i \in \mathcal{A}\setminus\{a\}, \\ t \in [\underline{s_i}, \underline{e_i}[}} h_i + \max_{\substack{j \in \mathcal{A}\setminus\{a\}, \\ t \in KCP_j}} h_j \qquad \leq C \tag{51}$$

$$\forall t \in [\underline{e_a}, \underline{e_a} + k_a[, \qquad h_a + \sum_{\substack{i \in \mathcal{A}\setminus\{a\}, \\ t \in [\underline{s_i}, \underline{e_i}[}} h_i \qquad \leq C \tag{52}$$

Intuitivement, la règle (52) correspond aux points de temps où c'est l'activité a qui est rallongée, aucune autre activité ne peut alors être allongée. C'est donc la règle (2) de filtrage Time-Table usuelle qui est appliquée.

La règle (51) correspond aux point de temps où il est possible qu'une autre activité que a soit rallongée. On doit alors prendre en compte toutes les activités pouvant être allongées à ce point de temps (celles dont la partie $\mathcal{K}$-obligatoire intersecte t). On retient alors la plus haute d'entre elles, car c'est le cas le plus contraignant.

L'ensemble des contraintes ainsi défini correspond à l'ensemble des contraintes défini par les Cumulative de la décomposition. La démonstration est identique à celle des propriétés 5.2 et 5.3 : à chaque j de la fonction max, correspond la contrainte d'une Cumulative pour l'activité a.

Exemple 5.3 (Les deux règles de filtrage de FlexC)**.** *Soit une activité a de hauteur 1, dont la date de début au plus tôt est 1, la durée 2 et l'allongement maximum 3. Le placement au plus tard est arbitrairement grand. Soit un profil des parties obligatoires matérialisé par la ligne noire, et un profil des parties $\mathcal{K}$-obligatoires matérialisé en noir pointillé.*

La règle 51 doit être respectée sur l'intervalle $[\underline{s_a}, \underline{e_a}[$.

La règle 52 doit être respectée sur l'intervalle $[\underline{e_a}, \underline{e_a}+k_a[$.

La figure 5.5 illustre le placement valide d'une activité au vu d'un profil de parties obligatoires.

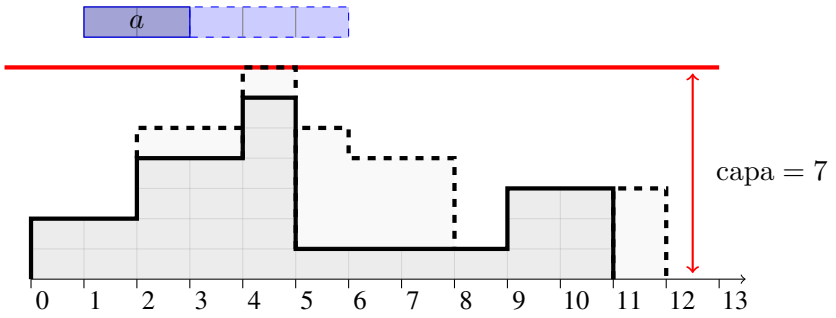


FIGURE 5.5 – Placement au plus tôt d'une activité au vu des profils des parties obligatoires et parties $\mathcal{K}$-obligatoires.

Sur l'intervalle de temps $[\underline{e_a}, \underline{e_a}+k_a[$, seule une partie $\mathcal{K}$-obligatoire doit être prise en compte, car seule une activité peut être allongée à un même point de temps. C'est pourquoi sur l'intervalle $[4, 5[$ la règle (52) ne prend pas en compte les autres parties $\mathcal{K}$-obligatoires. Le placement de a est donc valide.

5.2.4 Algorithme de filtrage

Dans la section 5.2.3, nous venons de proposer une adaptation du raisonnement Time-Table à la contrainte FlexC. Nous proposons maintenant un algorithme pour effectuer le filtrage correspondant. L'algorithme de balayage SWEEP_MIN est capable de résoudre des problèmes de grande taille, c'est-à-dire des problèmes pouvant aller jusqu'à plusieurs dizaines, voire centaines de milliers d'activités [LCB14].

Dans l'état de l'art (section 3.1.3, page 29), nous avons donné les détails de l'algorithme de balayage *dynamic sweep* de Letort *et al.* que nous adaptons pour la contrainte FlexC dans cette section. Pour rappel, le principe est de faire évoluer ce que l'on appelle une droite de balayage, de la gauche vers la droite de l'horizon de temps, en suivant les évènements stockés dans une structure de données nommée h_{event}. L'algorithme maintient, à chaque point de temps, la hauteur restant disponible pour les activités dans la variable gap, ainsi que l'état de chaque activité :

- Si une activité ne respecte pas la contrainte de capacité, elle est dite en conflit et est stockée dans une structure de données nommée $h_{conflict}$.
- Si elle n'est pas en conflit, alors elle est stockée dans h_{check}.

L'algorithme SWEEP_MIN gère les évènements et organise le balayage. Il délègue le filtrage sur un intervalle à l'algorithme FILTER_MIN et la synchronisation des évènements à l'algorithme SYNCHRONIZE.

Enfin, rappelons la signification des évènements de l'algorithme SWEEP_MIN que nous avons décrit en section 3.1.3 :

- SCP : désigne le début de partie obligatoire, si elle existe ;
- CCP : désigne le début de partie obligatoire, si elle n'existe pas ;
- ECPD : désigne la fin de partie obligatoire, si elle existe ;
- PR : désigne le début au plus tôt de l'activité, si elle n'est pas fixée.

Notons que le filtrage n'est plus symétrique, car les activités peuvent seulement être retardées et non pas avancées dans le temps. Dans les sections suivantes nous détaillons les modifications que nous avons apportées. Tout d'abord nous présentons des adaptations génériques, permettant de simplifier la gestion des évènements dynamiques de l'algorithme que nous allons adapter. Puis, dans un deuxième temps, nous détaillons les modifications que nous avons apportées pour FlexC, pour filtrer le début au plus tôt des activités. Enfin, nous présenterons les changements que nous avons apportés pour le filtrage de la fin au plus tard des activités.

Adaptation générique

Dans un premier temps, nous avons supprimé l'évènement CCP et créé un évènement SCP pour chaque activité, qu'elle ait ou non une partie obligatoire. On peut en effet remarquer que dans l'algorithme SYNCHRONIZE (ligne 14 et 16) l'évènement dynamique CCP, recrée un évènement SCP au même moment.

Par ailleurs, nous ne créons pas l'évènement ECPD durant l'initialisation, mais uniquement lors de la gestion de l'évènement SCP. En effet, dans l'algorithme originel, l'évènement ECPD, créé au début de l'algorithme, doit être synchronisé car il est possible qu'un filtrage ait pu repousser la fin de partie obligatoire de l'activité. Or, il est possible de statuer sur la date définitive de fin de partie obligatoire lorsque la droite de balayage coïncide avec le début de partie obligatoire. Lors de la gestion de l'évènement CCP dans l'algorithme originel SYNCHRONIZE (ligne 17), la variable $evup_t$ est mise à $vrai$, ce qui montre qu'à ce moment l'évènement ECPD est définitivement positionné. Il est donc possible d'attendre l'évènement SCP et de le créer seulement lorsque la date définitive est connue.

Adaptation pour le filtrage du début au plus tôt

Dans ce paragraphe, nous montrons les adaptations de l'algorithme principal, SWEEP_MIN.

Pour simplifier la lecture, les adaptations sont surlignées en gris dans l'algorithme. Dans la section précédente, nous avons modifié la génération des évènements, nous détaillons donc la ligne 3 dans l'algorithme ROBUST_GENERATE_EVENT_MIN (page 80). Nous avons également externalisé le traitement des activités (lignes 25 et 26) dans la méthode ROBUST_PROCESS_EVENT_MIN (page 81).

La condition de mise en conflit d'une activité est renforcée par rapport à l'algorithme originel, en prenant en compte la partie $\mathcal{K}$-obligatoire (*cf.* condition (51)). Ce changement est reporté à la ligne 15. Enfin, pour respecter les contraintes (51) et (52), un placement au plus tôt doit être assuré sur un intervalle correspondant à la durée rallongée $p_a + k_a$, au lieu de p_a pour l'algorithme originel. Ce changement est reporté à la ligne 16.

Algorithm 8 ROBUST_SWEEP_MIN

```
1:  fonction ROBUST_SWEEP_MIN() : bool
2:      [INITIALISATION]
3:      ROBUST_GENERATE_EVENT_MIN()
4:      h_check, h_conflict ← ∅ ; newTasksToPrune ← ∅ ; ℒ ← ∅
5:      pour a = 0 to n − 1 faire
6:          evup_a ← (s_a = s̄_a) ; mins_a ← s_a
7:      fin pour
8:      δ ← get(h_events) ; δ_next ← δ ; gap ← capa
9:      [BOUCLE PRINCIPALE]
10:     tant que ¬vide(h_events) faire
11:         [BALAYAGE]
12:         si δ ≠ δ_next alors
13:             tant que ¬vide(newTasksToPrune) faire
14:                 extraire une activité a de newTasksToPrune
15:                 si h_a > gap − max(ℒ) alors ajouter ⟨h_a, a⟩ dans h_conflict
16:                 sinon si p_a + k_a > δ_next − δ alors {ajouter ⟨h_a, a⟩ dans h_check; mins_a ← δ;}
17:                 sinon evup_a ← Vrai
18:             fin tant que
19:             si ¬ROBUST_FILTER_MINδ, δ_next alors retourne Faux fin si
20:             δ ← δ_next
21:         fin si
22:         [GESTION DE L'EVENEMENT COURANT]
23:         δ ← ROBUST_SYNCHRONIZE_MIN(δ)
24:         extraire ⟨type, a, δ, dec⟩ de h_events
25:         ROBUST_PROCESS_EVENT_MIN(⟨type, a, δ, dec⟩)
26:         _
27:         [GETTING NEXT EVENT]
28:         si vide(h_event) alors retourne ROBUST_FILTER_MIN(δ, +∞)
29:         sinon δ_next ← ROBUST_SYNCHRONIZE_MIN(δ)
30:     fin tant que
31:     retourne Vrai
32: fin fonction
```

Le symbole « _ » est utilisé lorsque une instruction est supprimé, afin de visualiser la suppression, tout en gardant la numérotation des lignes sur le reste de l'algorithme. C'est en particulier le cas pour les lignes 25 et 26, qui sont remplacées par un unique appel de fonction.

La phase de création d'évènements (ligne 3 de l'algorithme ROBUST_SWEEP_MIN) est effectuée par l'algorithme ROBUST_GENERATE_EVENT_MIN :

Algorithm 9 Génération des évènements pour l'algorithme ROBUST_SWEEP_MIN

1: **procédure** ROBUST_GENERATE_EVENT_MIN()
2: **pour** chaque activité $a \in \{\mathcal{A}\}$ **faire**
3: $h_{event} \leftarrow h_{event} \cup \{\langle SCP, a, \overline{s_a}, -h_a \rangle\}$
4: **si** $\underline{s_a} \neq \overline{s_a}$ **alors** $h_{event} \leftarrow h_{event} \cup \{\langle PR, a, \underline{s_a}, 0 \rangle\}$ **fin si**
5: **fin pour**
6: **fin procédure**

À la vue des modifications que nous avons apportées à la gestion des évènements, la fonction de synchronisation dans le cas du filtrage du début au plus tôt consiste simplement à une récupération du point temps suivant :

Algorithm 10 ROBUST_SYNCHRONIZE_MIN

1: **fonction** ROBUST_SYNCHRONIZE_MIN(δ) : entier
2: **si** $vide(h\_events)$ **alors retourne** $-\infty$ **fin si**
3: $\langle type, a, date, dec \rangle \leftarrow$ consulter premier évènement de h_events ;
4: **retourne** $date$
5: **fin fonction**

Par ailleurs, nous proposons des modifications pour les algorithmes de gestion d'évènements et de filtrage.

Comme exposé en section 5.2.3, l'adaptation au cas FlexC de la règle après filtrage (4) (page 31), pour le raisonnement Time-Table de la contrainte Cumulative, donne lieu à deux nouvelles règles : (51) et (52) (page 76).

Afin de prendre en compte la règle (51) dans l'algorithme, il faut connaître la hauteur maximale des activités pour lesquelles l'intervalle courant ($[\delta, \delta_{next}[$) est dans leur partie $\mathcal{K}$-obligatoire. Pour rappel, la partie $\mathcal{K}$-obligatoire d'une activité a est l'intervalle $[\max(\overline{s_a}, \underline{e_a}), \underline{e_a} + k_a[$. Pour cela, nous introduisons une nouvelle structure de données, $\mathcal{L}$, dans laquelle est stocké l'ensemble des activités dont la partie $\mathcal{K}$-obligatoire intersecte l'intervalle courant. La méthode $\max(\mathcal{L})$ permet de récupérer la hauteur maximale des activités de $\mathcal{L}$.

Une activité a est ajoutée à $\mathcal{L}$ à son début de partie $\mathcal{K}$-obligatoire. Si l'activité a une partie obligatoire, l'ajout est réalisé en fin de partie obligatoire, pendant la gestion de l'évènement ECPD. Si l'activité n'a pas de partie obligatoire, l'ajout est réalisé lors de la gestion de l'évènement SCP.

L'activité doit être supprimée de $\mathcal{L}$ à sa fin de partie $\mathcal{K}$-obligatoire. Nous introduisons pour cela un nouvel évènement, EKCP. Cet évènement est ajouté dynamiquement à la liste des évènements au début de la partie $\mathcal{K}$-obligatoire (en même temps que a est ajoutée à $\mathcal{L}$).

Dans l'algorithme ROBUST_PROCESS_EVENT_MIN, la ligne 11 traduit l'ajout dans le cas où l'activité n'a pas de partie obligatoire, la ligne 17 l'ajout dans le cas où l'activité a une partie obligatoire et la ligne 20 la suppression en fin de partie obligatoire.

L'algorithme de gestion des évènements ROBUST_PROCESS_EVENT_MIN applique les modifications que nous venons de voir. La gestion de la partie obligatoire y est effectuée comme vu dans les adaptations génériques (page 78), ainsi que la gestion de la partie $\mathcal{K}$-obligatoire.

L'évènement SCP analyse si une partie obligatoire existe (ligne 6), ou si seule une partie $\mathcal{K}$-obligatoire existe (ligne 10), ou si aucune des parties obligatoires n'existe (ligne 13).

Algorithm 11 Traitement des évènements pour l'algorithme ROBUST_SWEEP_MIN

1: **procédure** ROBUST_PROCESS_EVENT_MIN($\langle date, a, type, dec \rangle$)
2: **si** l'évènement est de type PR **alors**
3: $newTasksToPrune \leftarrow newTasksToPrune \cup \{a\}$
4: **fin si**
5: **si** l'évènement est de type SCP **alors**
6: **si** $\delta < mins_a + p_a$ **alors**
7: $gap \mathrel{+}= dec$
8: $h_{event} \leftarrow h_{event} \cup \{\langle ECPD, a, mins_a + p_a, h_a \rangle\}$
9: $h_{event} \leftarrow h_{event} \cup \{\langle EKCP, a, mins_a + p_a + k_a, h_a \rangle\}$
10: **sinon si** $\delta < mins_a + p_a + k_a$ **alors**
11: $\mathcal{L} \leftarrow \mathcal{L} \cup \{a\}$
12: $h_{event} \leftarrow h_{event} \cup \{\langle EKCP, a, mins_a + p_a + k_a, 0 \rangle\}$
13: **fin si**
14: **fin si**
15: **si** l'évènement est de type ECPD **alors**
16: $gap \mathrel{+}= dec$
17: $\mathcal{L} \leftarrow \mathcal{L} \cup \{a\}$
18: **fin si**
19: **si** l'évènement est de type EKCP **alors**
20: $\mathcal{L} \leftarrow \mathcal{L} \setminus \{a\}$
21: **fin si**
22: **fin procédure**

Dans l'algorithme de filtrage FILTER_MIN, les activités sont séparées en fonction de leur état : l'activité respecte la règle de capacité ($h \leq gap$) ou elle ne la respecte pas ($h > gap$). Dans le cas robuste, deux règles de capacité (51) et (52), doivent être respectées. Nous proposons donc d'introduire un troisième état, h_{robust}.

Dans un premier temps, nous contraignons davantage que nécessaire le problème sur l'intervalle $[mins_a + p_a,\ mins_a + p_a + k_a[$, en y appliquant la contrainte (51) au lieu d'y considérer la contrainte (52). Ainsi, le fonctionnement est très similaire à l'algorithme FILTER_MIN, ne nécessitant que deux états, puisqu'une unique contrainte est considérée. La contrainte (52) sera prise en compte dans un second temps.

En appliquant la contrainte (51) à tout l'intervalle, les modifications de l'algorithme sont alors les lignes 5 et 23 pour prendre en compte la contrainte de hauteur renforcée et les lignes 7 et 29 pour prendre en compte la durée allongée.

Cependant, si ces seules lignes sont considérées, la hauteur est contrainte plus que nécessaire sur l'intervalle $[mins_a + p_a,\ mins_a + p_a + k_a[$. En effet, selon la contrainte (52), portant sur cet intervalle, une activité dont la hauteur est inférieure à gap n'est pas en conflit. L'état h_{robust} contient les activités se trouvant dans cette situation.

Avant de changer l'état de l'activité de h_{check} à $h_{conflict}$, nous vérifions si le point de temps courant est dans l'intervalle $[mins_a + p_a,\ mins_a + p_a + k_a[$ (ligne 10), dans ce cas l'état devient h_{robust} (ligne 11).

Le traitement de h_{robust} est similaire à celui de h_{check} dans l'algorithme originel, à l'exception de la longueur de l'intervalle pour laquelle le placement est valide ($p_a + k_a$ au lieu de p_a) (lignes 14 à 21).

Algorithm 12 ROBUST_FILTER_MIN

1: **fonction** ROBUST_FILTER_MIN(δ, δ_{next}) : boolean
2: `[VERIFICATION DE LA COHERENCE]`
3: **si** $gap - \max(\mathcal{L}) < 0$ **alors retourne** Faux
4: `[MISE A JOUR DES ACTIVITES DE` h_{check}`]`
5: **tant que** $\neg vide(h_{check}) \wedge \Big(vide(h\_events) \vee get(h_{check}) > gap - \max(\mathcal{L})\Big)$ **faire**
6: extraire $\langle h_a, a\rangle$ de h_{check}
7: **si** $\delta \geq \overline{s_a} \vee \delta - mins_a \geq p_a + k_a \vee vide(h\_events)$ **alors**
8: $adjust\_min\_var(s_a, mins_a); adjust\_min\_var(e_a, mins_a + p_a);$
9: **si** $\neg evup_a$ **alors** { _ ; $evup_a \leftarrow Vrai$;}
10: **sinon si** $mins_a + p_a \leq \delta$ **alors**
11: ajout de $\langle h_a, t\rangle$ dans h_{robust}
12: **sinon**
13: ajout de $\langle h_a, t\rangle$ dans $h_{conflict}$
14: `[MISE A JOUR DES ACTIVITES DE` h_{robust}`]`
15: **tant que** $\neg vide(h_{robust}) \wedge (vide(h\_events) \vee get(h_{robust}) > gap)$ **faire**
16: extraire $\langle h_a, a\rangle$ de h_{robust}
17: **si** $\delta \geq \overline{s_a} \vee \delta - mins_a \geq p_a + k_a \vee vide(h\_events)$ **alors**
18: $adjust\_min\_var(s_a, mins_a); adjust\_min\_var(e_a, mins_a + p_a);$
19: **si** $\neg evup_a$ **alors** { _ ; $evup_a \leftarrow Vrai$;}
20: **sinon**
21: ajout de $\langle h_a, t\rangle$ dans $h_{conflict}$
22: `[MISE A JOUR DES ACTIVITES DE` $h_{conflict}$`]`
23: **tant que** $\neg vide(h_{conflict}) \wedge get(h_{conflict}) \leq gap - \max(\mathcal{L})$ **faire**
24: extraire $\langle h_a, a\rangle$ de $h_{conflict}$
25: **si** $\delta \geq \overline{s_a}$ **alors**
26: $adjust\_min\_var(s_a, \overline{s_a}); adjust\_min\_var(e_a, \overline{e_a});$
27: **si** $\neg evup_a$ **alors** { _ ; $evup_a \leftarrow Vrai$;}
28: **sinon**
29: **si** $\delta_{next} - \delta \geq p_a + k_a$ **alors**
30: $adjust\_min\_var(s_a, \delta); adjust\_min\_var(e_a, \delta + p_a);$
31: **si** $\neg evup_a$ **alors** { _ ; $evup_a \leftarrow Vrai$;}
32: **sinon**
33: add $\langle h_a, a\rangle$ into h_{check} ; $mins_a \leftarrow \delta$;
34: **retourne** Vrai

Adaptations de SWEEP_MIN pour FlexC dans le cas du filtrage de la fin au plus tard

Contrairement à la contrainte Cumulative, le filtrage de FlexC n'est pas symétrique. En effet la partie $\mathcal{K}$-obligatoire se situe après l'activité et le sens de balayage n'est donc pas anodin. Dans cette section, nous présentons la solution que nous avons produite pour adapter l'algorithme de filtrage au cas du filtrage de la fin au plus tard.

Pour comprendre le fonctionnement du filtrage de la fin au plus tard, il est important de remarquer que l'on se situe dans le "miroir" du problème. La partie robuste se situe donc maintenant avant le placement de l'activité. En d'autres termes, c'est le problème robuste dans lequel l'activité peut être commencée plus tôt que prévu.

Exemple 5.4 (partie $\mathcal{K}$-obligatoire, point de vue miroir)**.** *Il existe trois cas possibles pour le placement au plus tard de a, par rapport au placement au plus tôt :*

cas 1 *si $\underline{e_a} \leq \overline{s_a} - k_a$, alors il n'existe ni partie obligatoire ni partie $\mathcal{K}$-obligatoire.*

cas 2 *si $\overline{s_a} - k_a < \underline{e_a} \leq \overline{s_a}$, alors il n'y a pas de partie obligatoire mais une partie $\mathcal{K}$-obligatoire existe ($[\overline{s_a} - k_a[,\ \underline{e_a}[$).*

cas 3 *si $\overline{s_a} < \underline{s_a}$, alors une partie obligatoire et une partie $\mathcal{K}$-obligatoire existe ($[\overline{s_a} - k_a,\ \overline{s_a}[$).*

Les trois cas sont illustrés par la figure 5.6, en reprenant les notations des figures 5.2, 5.3 et 5.4 (page 75) :

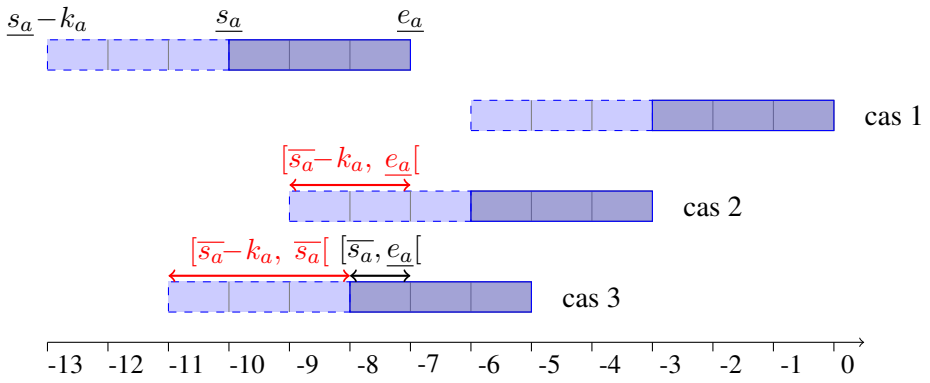


FIGURE 5.6 – Exemples de partie $\mathcal{K}$-obligatoire dans le cas miroir.

Pour rappel, lorsque l'on raisonne sur le miroir, le filtrage s'effectue sur les valeurs minimales.

Si elle existe, le début de la partie $\mathcal{K}$-obligatoire est fixé à $\overline{s_a} - k_a$. Ainsi, nous créons initialement l'évènement SKCP. La méthode ROBUST_GENERATE_EVENT_MAX génère deux évènements.

Algorithm 13 Génération des évènements pour l'algorithme ROBUST_SWEEP_MAX

```
1: procédure ROBUST_GENERATE_EVENT_MAX()
2:     pour chaque activité a ∈ {A} faire
3:         h_event ← h_event ∪ {⟨SKCP, a, s̄_a − k_a[, −h_a⟩}
4:         si e̲_a ≠ ē_a alors h_event ← h_event ∪ {⟨PR, a, ē_a, 0⟩} fin si
5:     fin pour
6: fin procédure
```

Gestion de l'évènement SKCP. Lorsque la droite de balayage est sur un évènement SKCP, il faut vérifier qu'il existe une partie $\mathcal{K}$-obligatoire.

- Dans le cas où un placement valide de l'activité est trouvé avant le début de partie $\mathcal{K}$-obligatoire ($mins_a + p_a \leq \delta$), alors il n'existe pas de partie $\mathcal{K}$-obligatoire (et par conséquent, pas de partie obligatoire).
- Dans le cas où l'activité doit être placée après δ, alors la partie $\mathcal{K}$-obligatoire existe : l'activité a est ajoutée à $\mathcal{L}$. Pour connaître sa date de fin, il faut savoir si une partie obligatoire est créée : $mins_a + p_a > \overline{s_a}$.
  - Si une partie obligatoire est détectée, alors un évènement SCP est créé pour notifier le début de partie obligatoire. L'évènement montre aussi la fin de partie $\mathcal{K}$-obligatoire.
  - Si la partie obligatoire n'est pas encore détectée, un évènement EKCPD de fin de partie $\mathcal{K}$-obligatoire est créé. Cet évènement étant dynamique, il est possible que la fin soit repoussée (*cf.* exemple 5.5).

Exemple 5.5 (Filtrage pendant la partie robuste)**.** *Soit un problème cumulatif avec 3 activités, qui, vues en miroir, ont pour valeurs :* $a_0 = \langle s_{a_0} = 5, p_{a_0} = 1, e_{a_0} = 6, h_{a_0} = 1\rangle\ k_{a_0} = 2$,
$a_1 = \langle s_{a_1} = 7, p_{a_1} = 2, e_{a_1} = 9, h_{a_1} = 1\rangle\ k_{a_1} = 2$ *et*
$a_2 = \langle s_{a_2} \in [5, 9], p_{a_2} = 2, e_{a_2} \in [6, 9], h_{a_2} = 1\rangle\ k_{a_2} = 5$.
La figure 5.7 montre les placements extrêmes de a_2 *ainsi que le profil robuste induit :*

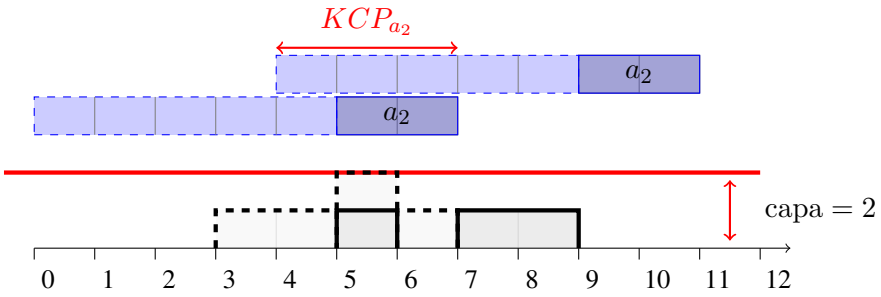


FIGURE 5.7 – Filtrage durant la partie robuste.

Le positionnement de l'activité a_2 *sur l'intervalle* $[5, 7[$ *n'est pas valide. Un filtrage est détecté sur l'intervalle* $[5, 6[$*, car la règle (51) n'est pas respectée :* $1 + 1 + 1 \not\leq 2$*. La valeur* 5 *doit être supprimée du domaine de* s_{a_2}*. On remarque que la partie* $\mathcal{K}$*-obligatoire de* a_2 *est toujours présente sur l'intervalle* $[5, 6[$ *: dans ce cas miroir, la règle la plus forte (c.-à-d. la règle (51)) est appliquée en premier et il est possible qu'en repoussant l'activité, la deuxième règle n'induise pas de filtrage ; c'est le cas ici, car le positionnement de l'activité* a_2 *sur l'intervalle* $[6, 8[$ *est valide.*

Gestion de la fin de partie $\mathcal{K}$-obligatoire. Durant la partie $\mathcal{K}$-obligatoire, il est possible qu'un filtrage ait lieu, et que l'évènement EKCPD doive être synchronisé. Dans la méthode de synchronisation il faut donc vérifier si un filtrage a eu lieu ($mins_a + p_a > \delta$). Si c'est le cas, il faut recréer un évènement de fin de partie $\mathcal{K}$-obligatoire, similairement à ce qui est fait pour l'évènement SKCP (on regarde si une partie obligatoire est créée et si c'est le cas, on ajoute un évènement SCP ou EKCPD).

Les méthodes de synchronisation et de gestion des évènements sont alors :

Algorithm 14 ROBUST_SYNCHRONIZE_MAX

```
1: fonction ROBUST_SYNCHRONIZE_MAX(δ) : entier
2:     si vide(h_events) alors retourne −∞
3:     repeat
4:         sync ← Vrai ; ⟨date, a, type, dec⟩ ← consulter premier évènement de h_events ;
5:         [GESTION DE L'EVENEMENT DYNAMIQUE (EKCPD)]
6:         si type = EKCPD ∧ mins_a + p_a > δ alors
7:             si mins_a + p_a > s̄_a alors ajout de ⟨s̄_a, a, SCP, h_a⟩
8:             sinon mise à jour de l'évènement à la date mins_a + p_a
9:             sync ← Faux ;
10:    until sync
11:    retourne date
```

Algorithm 15 Traitement des évènements pour l'algorithme ROBUST_SWEEP_MAX

```
1: procédure ROBUST_PROCESS_EVENT_MAX(⟨date, a, type, dec⟩)
2:     si l'évènement est de type PR alors
3:         newTasksToPrune ← newTasksToPrune ∪ {a}
4:     fin si
5:     si l'évènement est de type SKCP alors
6:         si mins_a + p_a > δ alors
7:             L ← L ∪ {a}
8:             si mins_a + p_a > s̄_a alors
9:                 h_event ← h_event ∪ {⟨SCP, a, s̄_a, h_a⟩}
10:            sinon
11:                h_event ← h_event ∪ {⟨EKCPD, a, mins_a + p_a, h_a⟩}
12:            fin si
13:        fin si
14:    fin si
15:    si l'évènement est de type EKCPD alors
16:        L ← L \ {a}
17:    fin si
18:    si l'évènement est de type SCP alors
19:        L ← L \ {a}
20:        gap += dec
21:        h_event ← h_event ∪ {⟨ECPD, a, mins_a + p_a, h_a⟩}
22:    fin si
23: fin procédure
```

Algorithm 16 ROBUST_FILTER_MAX

1: **fonction** ROBUST_FILTER_MAX(δ, δ_{next}) : boolean
2: [VERIFICATION DE LA COHERENCE]
3: **si** $gap < 0$ **alors retourne** Faux
4: [MISE A JOUR DES ACTIVITES DE h_{check}]
5: **tant que** $\neg vide(h_{check}) \wedge \Big(vide(h\_events) \vee hauteurOk\,(get(h_{check}))\Big)$ **faire**
6: extraire $\langle h_a, a\rangle$ de h_{check}
7: **si** $\delta \geq \overline{s_a} \vee \delta - mins_a \geq p_a + k_a \vee vide(h\_events)$ **alors**
8: $adjust\_min\_var(s_a, mins_a); adjust\_min\_var(e_a, mins_a + p_a);$
9: **si** $\neg evup_a$ **alors** { _ ; $evup_a \leftarrow Vrai$;}
10: **sinon si** $get(h_{check}) \leq gap$ **alors**
11: ajout de $\langle h_a, a\rangle$ dans h_{robust}
12: **sinon**
13: ajout de $\langle h_a, a\rangle$ dans $h_{conflict}$
14: [MISE A JOUR DES ACTIVITES DE h_{robust}]
15: **tant que** $\neg vide(h_{robust}) \wedge (vide(h\_events) \vee get(h_{robust}) > gap)$ **faire**
16: extraire $\langle h_a, a\rangle$ de h_{robust}
17: ajout de $\langle h_a, t\rangle$ dans $h_{conflict}$
18: **tant que** $\neg vide(h_{robust}) \wedge \Big(vide(h\_events) \vee hauteurOk\,(get(h_{robust}))\Big)$ **faire**
19: extraire $\langle h_a, a\rangle$ de h_{robust}
20: add $\langle h_a, a\rangle$ into h_{check} ; $mins_a \leftarrow \delta + k_a$;
21: [MISE A JOUR DES ACTIVITES DE $h_{conflict}$]
22: **tant que** $\neg vide(h_{conflict}) \wedge get(h_{conflict}) \leq gap$ **faire**
23: extraire $\langle h_a, a\rangle$ de $h_{conflict}$
24: **si** $\delta \geq \overline{s_a}$ **alors**
25: $adjust\_min\_var(s_a, \overline{s_a}); adjust\_min\_var(e_a, \overline{e_a});$
26: **si** $\neg evup_a$ **alors** { _ ; $evup_a \leftarrow Vrai$;}
27: **sinon**
28: **si** $\delta_{next} - \delta \geq p_a + k_a$ **alors**
29: $adjust\_min\_var(s_a, \delta); adjust\_min\_var(e_a, \delta + p_a);$
30: **si** $\neg evup_a$ **alors** { _ ; $evup_a \leftarrow Vrai$;}
31: **sinon**
32: add $\langle h_a, a\rangle$ into h_{check} ; $mins_a \leftarrow \delta$;
33: **retourne** Vrai

Le filtrage dans le cas de la fin au plus tard est une généralisation du cas précédent, dans lequel nous avons trois états possibles en fonction de la hauteur de l'activité. La différence majeure tient dans le fait qu'un filtrage peut être effectué durant la partie robuste (*cf.* exemple 5.5, page 84). Il est alors possible que le point de temps courant, δ, intersect la partie robuste de l'activité que l'on cherche à placer. Dans le cas ou le plus grand élément de $\mathcal{L}$ est l'activité courante, il faut alors prendre en compte le deuxième plus grand élément. Nous externalisons alors la comparaison de la hauteur dans la fonction booléenne $hauteurOk(get(h_{check}))$

$$hauteurOk(a) = \begin{cases} h_a \leq gap - \max 2(\mathcal{L}) & \text{si } a \text{ est l'activité la plus haute} \\ h_a \leq gap - \max(\mathcal{L}) & \text{si } a \text{ n'est pas l'activité la plus haute} \end{cases}$$

avec $\max 2()$ la fonction renvoyant la hauteur de la deuxième plus grande activité de $\mathcal{L}$.

5.2.5 Analyse de l'algorithme

L'adaptation que nous proposons induit un certain nombre d'évènements dynamiques. À chaque traitement de l'évènement de fin de partie obligatoire, si l'activité a été repoussée, l'évènement est repoussé. Cela ce produit un nombre de fois de l'ordre de k_a/p_a (*cf.* annexe A.2). Notons d le plus grand ratio entre la durée de l'activité et sa partie robuste ($d = \max_{a \in \mathcal{A}}(k_a/p_a)$). Dans le pire cas, l'algorithme doit traiter $\mathcal{O}(d\ n)$ évènements. La complexité de l'algorithme est alors $\mathcal{O}(d\ n^2\ log\ n)$. Dans la plupart des cas, on peut admettre que la partie robuste k_a est plus petite que la durée de l'activité. Dans ce cas au plus 1 nouvel évènement est créé, l'algorithme est alors en $\mathcal{O}(n^2\ log\ n)$.

Nous avons adapté plusieurs versions différentes de l'algorithme de balayage Time-Table, *Dynamic Sweep* ([LCB14, section 3]) et *Synchronized Sweep* ([LCB14, section 4]). Dans le cadre d'une adaptation robuste les deux implémentations ne diffèrent que très légèrement, aussi bien en matière d'adaptation algorithmique, qu'en temps d'exécution. Puisque ces différentes versions n'ajoutent rien à la capacité de modélisation, que nous souhaitons privilégier, nous ne nous sommes pas attardés dessus. Cette piste d'amélioration purement algorithmique reste donc ouverte.

5.2.6 Contraintes annexes

Pour modéliser un problème robuste, l'ensemble des contraintes du réseau doit prendre en compte la robustesse et pas uniquement la contrainte Cumulative. L'adaptation de certaines contraintes usuelles, telle que les précédences, se fait aisément en ajoutant la partie robuste à la durée de l'activité. Une contrainte de précédence entre l'activité a et l'activité b s'écrit alors : $e_a + k_a \leq s_b$.

5.2.7 Tests expérimentaux

Notre implémentation robuste de l'algorithme SWEEP_MIN pour Cumulative est plus complexe que l'originel, notamment parce qu'il n'y a plus de symétrie entre le filtrage du début au plus tôt et de la fin au plus tard. Nous avons choisi l'algorithme SWEEP_MIN car il passe à l'échelle. Dans un premier temps, nous avons cherché à nous assurer que notre adaptation conserve cette propriété. Nous avons donc effectué des expérimentations similaires à celles de Letort *et al.* [LCB14] et avons généré des instances aléatoires de grande taille avec p_a entre 5 et 10, h_a entre 1 et 5 et une capacité égale à 30. Les valeurs pour les k_a sont strictement positives avec une moyenne de 4. La figure 5.8 montre que notre algorithme passe à l'échelle sur des problèmes de satisfiabilité ayant 12800 activités. La décomposition atteint la limite de temps de 3600 secondes (c.-à-d. 1h) avec 1600 activités, et mène à un dépassement mémoire avec 6400 contraintes Cumulative.

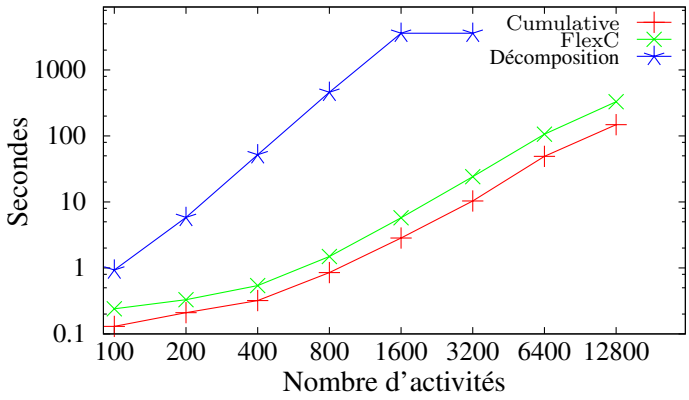


FIGURE 5.8 – Performance de l'algorithme de balayage robuste pour FlexC.

5.3 Application à un problème de déchargement de grues

Dans cette section, nous présentons le problème de déchargement de grues tel qu'il a été présenté par Zampelli [DPZ14a]. Nous comparons ensuite le modèle utilisant la contrainte Cumulative à une approche *ad-hoc* rallongeant chacune des activités.

5.3.1 Description du problème

Le *CAP* (*Crane Assignment Problem* en anglais) est une spécialisation du *berth and crane problem* [ZVS+13], dans lequel nous portons un regard approfondi sur l'ordonnancement du déchargement d'un unique porte-conteneurs dans un port. Un porte-conteneurs est constitué de baies. Les baies sont des sections transverses contenant les conteneurs. Chaque baie est constituée d'une partie haute et d'une partie basse. Un nombre prédéterminé de grues est affecté à chaque bateau. Le but est de minimiser la durée du déchargement. Une taxe proportionnelle au temps de retard de déchargement est appliqué au terminal. La vitesse de déchargement d'un conteneur dépend de conditions extérieures : les conditions de vent et de mer par exemple, ainsi que du conducteur et de la position des conteneurs dans le bateau.

```
range A=1..40//Range of acts                                    1
int nbc=4; //nbr of cranes                                      2
int pos[A]=...; //act position                                  3
int tt[A,A]=...;//acts transition time                          4
bool preced[A,A]=...;//act precedence                           5
Solver m();                                                     6
IntVar s[A](...);//start                                        7
IntVar p[A](m,rand[5,800]));//duration                          8
IntVar e[A](...);//end                                          9
IntVar h[A](m,1);//resource                                     10
IntVar c[A](m,[0,nbc-1]);//crane                                11
IntVar k[i∈A](m,rand([0,.25])*p[i]);                            12
```

FIGURE 5.9 – Donnée d'entrée du *Crane Assignement Problem*.

La figure 5.9 présente le pseudo-code pour les données et les variables de décision du problème de déchargement de grues. La variable A (ligne 1) correspond à la plage des activités. Le cargo a 20 baies avec une activité sur et sous le pont. Le nombre usuel de grues pour un cargo de 20 baies est de 4 (ligne 2). Les lignes 3 à 5 déclarent la position dans les baies de chaque activité ainsi que le temps de transition et les relations de précédence. Les temps de transition sont calculés proportionnellement à la distance entre leurs positions dans la baie. Pour chacune des baies, il existe une relation de précédence entre les activités en-dessous et au-dessus du pont. Les lignes 7 à 10 déclarent les activités (début, fin, durée et hauteur). Pour les différents benchmarks, la durée est choisie entre 5 et 800 minutes. Les lignes 11 et 12 déclarent les variables de grues et de robustesse pour chaque activité. La durée de la partie robuste est prise aléatoirement entre 0 et 25% de la durée de l'activité.

```
//1.cumu cstr, nbc resources                                          1
m.post(Cumulative(s,p,e,h,nbc))                                       2
//2.precedence constraints                                            3
for (i,j) s.t. preced[i,j]==1:                                        4
   m.post(e[i]<s[j]);                                                 5
//3.crane alloc, transition times                                     6
for (i,j) i!=j∧pos[i]<pos[j]:                                         7
   m.post( ((s[i]<e[j]+tt[i,j])∧(s[j]<e[i]+tt[i,j]))                  8
              => c[i]<c[j] );                                         9
//4.no intersec for nearby acts                                       10
for (i,j) i<j∧|pos[i]−pos[j]|<=2:                                     11
   m.post(s[i]>e[j] ∨ e[i]<s[j]);                                     12
minimize obj = min({e[i]} i∈ A)                                       13
```

FIGURE 5.10 – Modèle non robuste du *CAP*.

La figure 5.10 présente les contraintes du modèle sans robustesse. En suivant [ZVS+13], nous modélisons cette application comme un problème cumulatif. Les lignes 2 à 5 posent les contraintes de précédence et cumulatives. Les lignes 8 et 9 posent les contraintes du problème de déchargement de grues : si la distance dans le temps entre deux activités ne permet pas qu'une même grue s'en occupe (condition de la ligne 8) alors l'activité d'indice la plus petite doit être affectée à la grue d'indice la plus petite, (les activités comme les grues sont triées de la gauche vers la droite). La ligne 12 pose une contrainte de distance temporelle entre activités ayant une distance physique faible. Ceci permet d'éviter une collision lors du déchargement simultané de deux conteneurs trop proches.

```
//1.flex cumu cstr, nbc resources                                     1
m.post( FlexC(s,p,e,h, k ,nbc))                                       2
//2.precedence constraints                                            3
for (i,j) s.t. preced[i,j]==1:                                        4
   m.post(e[i] +k[i] <s[j]);                                          5
//3.crane alloc, transition times                                     6
for (i,j) i!=j∧pos[i]<pos[j]:                                         7
  m.post(((s[i]<e[j]+tt[i,j] +k[j] )∧(s[j]<e[i]+tt[i,j] +k[i] ))      8
              => c[i]<c[j] );                                         9
//4.no intersec for nearby acts                                       10
for (i,j) i<j∧|pos[i]−pos[j]|<=2:                                     11
   m.post(s[i]>e[j] +k[i] ∨ e[i] +k[i] <s[j]);                        12
minimize obj = min({e[i]+ k[i] } i∈ A)                                13
```

FIGURE 5.11 – Modèle robuste du *CAP*.

La figure 5.11 présente le modèle robuste. FlexC est utilisé à la place de Cumulative (ligne 2). les contraintes de précédences et de distance considère également qu'une activité doit pouvoir être repoussée ou allongée (lignes 5, 8 et 12).

5.3.2 Résultats

Nous avons traité des problèmes à 20 zones par cargo et 4 grues. Les activités ont des durées allant de 5 à 800 minutes, avec un facteur de robustesse simulé aléatoirement pour chaque activité correspondant à un pourcentage entre 0 et 25% de sa durée. Ces données sont réalistes pour des bateaux de taille moyenne.

La technique d'exploration utilisée est une recherche à voisinage large guidée par la propagation [PSF04], avec une remise en cause aléatoire de 30% des activités et une stratégie de choix de la variable de début ayant le plus petit domaine, affectée avec la date la plus petite d'abord. Nous avons fixé pour les dix instances une limite de temps de cinq minutes.

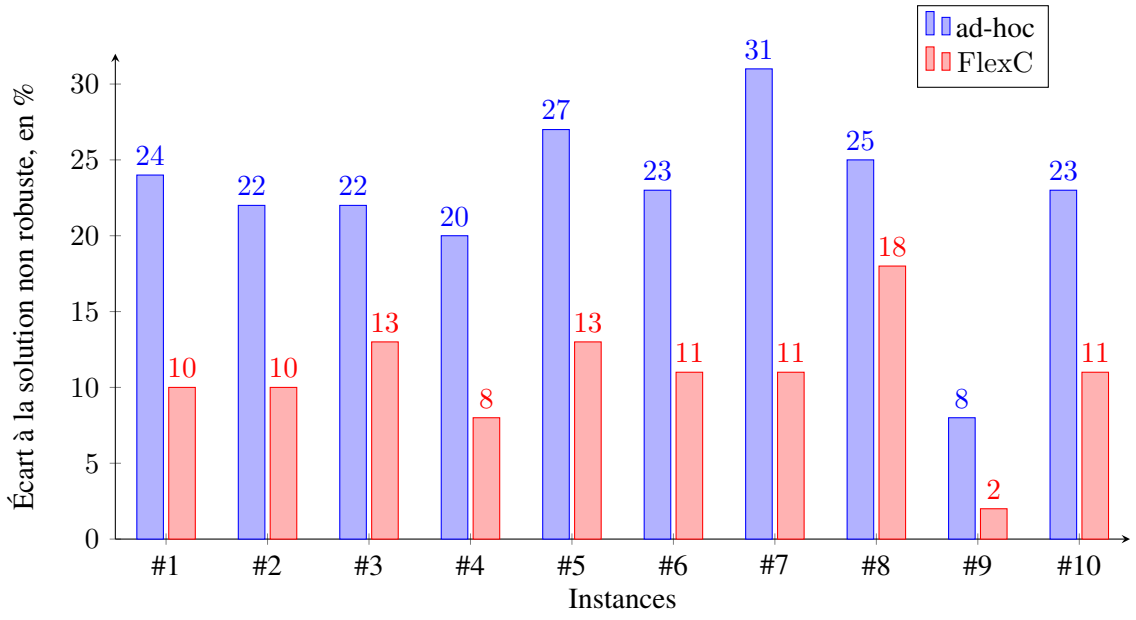


FIGURE 5.12 – Comparaison de l'écart moyen à une solution non robuste entre notre approche et une méthode ad-hoc.

La figure 5.12 montre l'écart moyen sur chacune des instances de notre approche et une méthode ad-hoc (où toutes les activités sont allongées) avec la solution non robuste. On remarque qu'en moyenne la solution ad-hoc dégrade la solution de 22%, alors que notre approche dégrade la solution de seulement 10%. Cela montre qu'il est possible, grâce à notre approche, d'avoir des solutions robustes tout en gardant une solution de bonne qualité.

Nous donnons en annexe A.3.1 l'ensemble des résultats correspondant à la version publiée à la conférence *CP'14* de ces travaux [DPZ14a].

5.4 Conclusions et Perspectives

Dans certaines applications la robustesse d'une solution prend le pas sur l'optimalité de la fonction de coût. En effet les conséquences d'un aléa sur une solution non robuste peuvent engendrer une dégradation importante de la solution. Cette dégradation peut aussi bien correspondre à une augmentation du coût que de la nécessité de réviser complètement l'ordonnancement initialement proposé.

Dans ce chapitre nous avons proposé une approche nouvelle permettant de traiter des problématiques sur l'incertain en ordonnancement. Les nouveautés du paradigme présentés viennent du fait que l'on traite en même temps tous les points suivants :

- Le paradigme présenté est déclaratif car le niveau de robustesse est spécifié dans la formulation même du problème.
- Le paradigme introduit est dédié au problème cumulatif.
- Le paradigme introduit est modulaire.
- Le paradigme proposé est complètement stable car la solution proposée n'a pas a être recalculé lorsque un aléa survient.

Dans ce contexte de robustesse nous avons introduit dans ce chapitre la contrainte globale FlexC. Cette nouvelle contrainte généralise la contrainte Cumulative. Dans la section 3.1.3 (page 31), nous avons également proposé un algorithme de filtrage, basé sur un algorithme Time-Table passant à l'échelle [LCB14]. En effet la décomposition de la contrainte FlexC en contraintes cumulatives de base ne permet pas de passer à l'échelle : une délai d'une heure ne permet pas de trouver une première solution à un problème à 1600 activités (*cf.* figure 5.8, page 87). En comparaison avec cette décomposition la nouvelle contrainte globale FlexC permet de gagner un ordre de grandeur sur la taille de problème résolu. Comparé à l'algorithme SWEEP_MIN dont nous sommes partis pour implémenter FlexC, la perte en temps d'execution est minime : la perspective de gain par une implémentation plus fine est donc limitée.

Au-delà des résultats théoriques apportés dans ce chapitre, les avantages de cette approche sont prometteurs. Nous avons par exemple pu modéliser simplement un problème réel d'ordonnancement robuste de grues dans un port tout en conservant les performances d'une approche non robuste. Il ressort de nos expérimentations sur cette application que notre approche offre un gain supérieur à 10% par rapport à une approche ou toute les activités sont systématiquement allongées.

Dans ce chapitre, nous avons considéré un contexte dans lequel la robustesse prend en considération le retardement d'une activité. Cependant, nous pouvons noter que, grâce à l'approche miroir, il serait également possible de modéliser une situation dans laquelle l'incertitude serait sur l'avancement d'une activité. Il serait intéressant de regarder comment il serait possible de modéliser un problème dans lequel une incertitude existe à la fois sur l'avancement et le retardement d'une activité.

Par ailleurs, nous avons noté que pour obtenir un ordonnancement robuste il est essentiel que la robustesse soit également prise en compte dans les contraintes annexes, telles que les précédences. En suivant le schéma proposé par Letort *et al.* [LCB14], nous pourrions les incorporer au sein même de notre contrainte d'ordonnancement robuste.

La perspective d'adaptation de notre modèle robuste à d'autre contraintes d'ordonnancement, tel que les problèmes de job-shop et open-shop permettrait, outre une approche générique de la robustesse en ordonnancement de plus facilement comparer notre outil à des méthodes de résolution existantes.

Ces travaux ont fait l'objet d'une publication à la conférence *CP'14* [DPZ14a]. Un rapport technique vient compléter cette publication en donnant les détails d'implémentation présentés dans la section 5.2.4 [DPZ14b].

6

Conclusion

La programmation par contraintes se base sur des techniques d'inférence souvent hétérogènes. Cette hétérogénéité est une force car elle permet de résoudre efficacement certaines classes de problèmes. C'est aussi une faiblesse car il peut s'avérer complexe de déterminer quels algorithmes choisir pour résoudre au mieux un problème.

Dans cette thèse, nous avons étudié les problèmes d'ordonnancement cumulatifs. Nous avons proposé une caractérisation unifiée des raisonnements de la contrainte $\mathrm{Cumulative}$: Time-Table, Edge-Finding, Time-Table-Edge-Finding et le Raisonnement Energétique. Grâce à ce nouveau point de vue, nous avons formellement comparé les différentes techniques. Outre une simplification des comparaisons entre les méthodes de détection d'incohérence (section 4.3), notre approche a permis de montrer que le raisonnement Time-Table-Edge-Finding domine le raisonnement Edge-Finding à condition d'associer ces algorithmes à une technique de type Time-Table, ce qui est usuellement le cas dans les solveurs de contraintes (section 4.2.3). Nous avons démontré qu'il est possible de réduire le travail nécessaire pour effectuer l'ensemble des déductions du raisonnement énergétique (sections 4.2.4 et 4.4.1), répondant à une question qui était restée ouverte dans la littérature. Nous avons proposé un nouveau propagateur pour le raisonnement énergétique.

Ces travaux on fait l'objet d'une publication à la conférence avec sélection et actes *CP'14* [DP14].

Une perspective de ces contributions est de chercher à améliorer la complexité théorique du Raisonnement Energétique, qui souffre actuellement d'une complexité en $\mathcal{O}(n^3)$. À défaut, il serait intéressant de concevoir un propagateur de coût opérationnel raisonnable dont le filtrage domine Time-Table-Edge-Finding.

Nous avons traité un cas d'étude concernant l'intégration de notions de qualité des solutions indépendantes d'une fonction d'optimisation classique : la notion de robustesse en ordonnancement cumulatif (section 5.2.1). Nous avons démontré que les propriétés théoriques de l'un des algorithmes de référence pour le problème cumulatif classique peuvent être vérifiées dans le cas de problèmes cumulatifs robustes (section 5.2.2). Nous avons adapté cet algorithme au problème d'affectation de grues (section 5.2.4). Nous avons montré expérimentalement que cette démarche est utile dans un contexte réel (section 5.3).

Ces travaux on fait l'objet d'une publication à la conférence *CP'14* [DPZ14a].

Les perspectives qui se dégagent de cette approche sont les suivantes. D'un point de vue pratique, il serait intéressant d'adapter l'algorithme « sweep+précédences » de Letort *et al.* [LCB14]. Il est en outre envisageable d'étudier l'adaptation d'algorithmes effectuant un filtrage plus fort, e.g., Time-Table-Edge-Finding [Vil11]. D'un point de vue plus théorique, il serait intéressant d'étudier la généralisation de notre approche à n'importe quel nombre d'activités r, possiblement ordonnancées dans une même fenêtre temporelle et simultanément sujettes à des aléas.

Dans le but de traiter les problèmes d'ordonnancement en programmation par contraintes, une tendance actuelle semble être de chercher à optimiser les algorithmes les plus rapides, ayant un filtrage faible. Nos travaux semblent montrer qu'un problème lié à la contrainte $\mathrm{Cumulative}$ est au contraire son manque de filtrage. Il me semble donc intéressant de pousser plus loin les raisonnements, afin de concevoir des algorithmes de filtrage plus puissants, susceptibles de « casser » la complexité des problèmes.

Annexe

A.1 Travaux en Parallèle de la thèse.

En marge de mes travaux de thèse j'ai été amené à collaborer sur un projet d'ordonnancement industriel. Ces travaux ont fait l'objet d'une publication à *CP'15* [DFPP15], en voici le résumé :

Ce papier est motivé par une application dont l'objectif est de générer automatiquement des résumés de vidéo [BBG14]. Dans une vidéo, par exemple un match de tennis, des extraits caractérisent la présence d'applaudissements, de paroles, de jeu, de point gagnant etc. Dans cette application, la programmation par contraintes est utilisée pour sélectionner les intervalles de temps qui feront le résumé en fonction des caractéristiques, générées préalablement, de la vidéo. Les règles métier sont exprimées par l'algèbre d'Allen [All83]. Les meilleurs résultats sont obtenus par des solution *ad-hoc*, une par règle : une combinaison des 13 relations d'Allen. Développer une telle solution nécessite une expertise en programmation par contraintes, ce qui est un point critique pour les spécialistes en vidéo. Dans notre papier, nous proposons une contrainte générique *ExistAllen*, dédiée à une classe de problèmes temporels, qui couvre l'ensemble des règles. *ExistAllen* prend en argument un ensemble de tâches (les segments video qui seront sélectionnés), un ensemble de règles d'Allen et un ensemble d'intervalles (les données du problème : les caractéristiques de la video). *ExistAllen* est satisfaite si, et seulement si, les tâches sont ordonnées et si pour chaque tâche au moins une relation avec un intervalle est satisfaite. Nous avons aussi proposé un algorithme de filtrage borne-consistant dont la complexité temporelle est $\mathcal{O}(n+m)$, avec n le nombre de tâches et m le nombre d'intervalles. Ce propagateur prend en charge n'importe quelle des 2^{13} combinaisons de relations d'Allen, sans paramétrage. Par conséquent notre approche permet une modélisation et une mise en application ne nécessitant pas de connaissance poussée en programmation par contraintes. Les expérimentations, réalisées sur des instances réelles, confirment l'intérêt de notre approche.

A.2 Nombre de décalages de EKCPD

Dans cette annexe, nous montrons que le nombre de fois où l'évènement EKCPD de fin de partie $\mathcal{K}$-obligatoire d'une activité a, de durée p_a et de longueur d'allongement maximale k_a, est repoussé de l'ordre de k_a/p_a fois (*cf.* section 5.2.5).

Intuitivement, lorsqu'un filtrage est détecté, l'événement de fin de partie obligatoire est repoussé de la durée de l'activité. Le filtrage suivant ne sera détecté qu'à ce moment là, la distance entre ces deux détections est donc la durée de l'activité. Par conséquent, lorsque l'évènement de fin de partie obligatoire est repoussé deux fois, la durée sur laquelle il a été repoussé est supérieure ou égale à la durée de l'activité.

Démonstration. Notons x_i le point de temps où le début au plus tôt est repoussé la i-ème fois. Le début au plus tôt de l'activité à l'étape 0 servira de référence, il vaut alors $x_0 = 0$. La fin au plus tôt est alors au point de temps p_a. Supposons que la fin au plus tard et la partie $\mathcal{K}$-obligatoire soient arbitrairement grands.

Notons x_1 le début au plus tôt, repoussant la fin au plus tôt, détecté au point de temps p_a lors du traitement de la fin de partie $\mathcal{K}$-obligatoire. Il se situe nécessairement dans l'intervalle $[0, p_a[$. La figure A.1 montre cette situation.

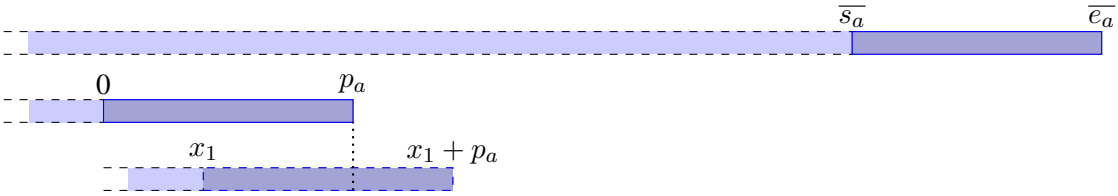


FIGURE A.1 – Positionnement de l'activité après le premier filtrage.

Lors de l'évènement de fin de la partie $\mathcal{K}$-obligatoire, au point de temps p_a, le dernier filtrage a pour conséquence de repousser l'évènement en $x_1 + p_a$.

Le deuxième filtrage, repoussant le début au plus tôt de a se situera alors dans l'intervalle $[p_a,\ x_1 + p_a[$. Notons le nouveau début au plus tôt x_2. La figure A.2 montre cette situation.

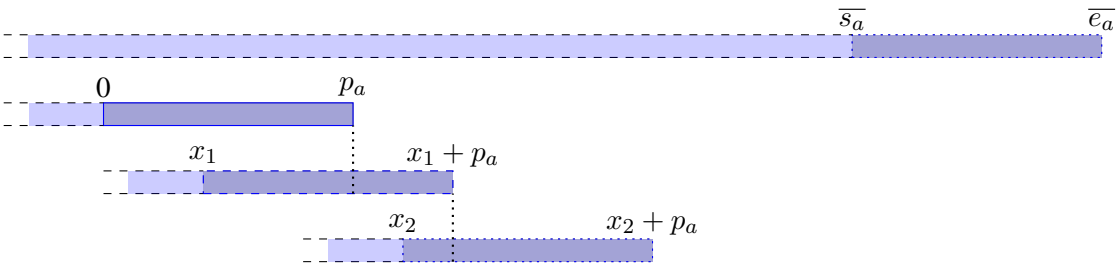


FIGURE A.2 – Positionnement de l'activité après le deuxième filtrage.

Lors de l'évènement de fin de partie $\mathcal{K}$-obligatoire, au point de temps x_1+p_a, le dernier filtrage a pour conséquence le décalage de l'évènement en $x_2 + p_a$.

Le troisième filtrage, repoussant le début au plus tôt se situe dans l'intervalle $[x_1+p_a,\ x_2+p_a[$. Donc $x_3 \geq x_1 + p_a$. La figure A.3 montre cette situation.

Nous venons de montrer que, quels que soient les filtrages effectués, la distance entre x_1 et x_3 est supérieure ou égale à la durée de l'activité, p_a. La distance entre deux filtrages x_i et x_{i+2n} est donc supérieure ou égale à $n \times p_a$. Soit e le nombre maximum de fois où l'évènement est repoussé. La durée entre x_1 et x_e est donc supérieure ou égale à $\lfloor \frac{e}{2} \rfloor \times p_a$. Par conséquent $k_a \geq \lfloor \frac{e}{2} \rfloor \times p_a$ d'où $e \leq 2 \times k_a/p_a + 1$. Le nombre maximum de fois où l'évènement est repoussé est donc de l'ordre de k_a/p_a. □

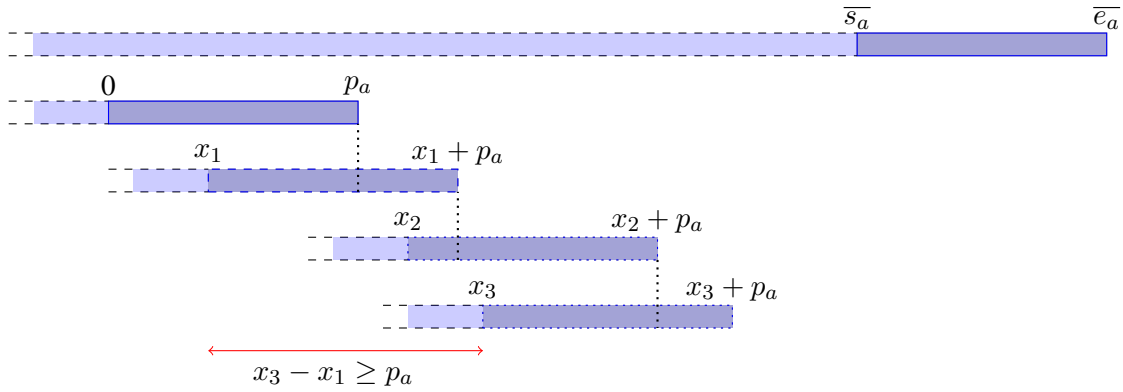


FIGURE A.3 – Positionnement de l'activité après le troisième filtrage.

A.3 Tableaux Benchs

A.3.1 Approche robuste

Nous donnons ici la version complete du bench fournit à CP [DPZ14a], dont un graphique est extrait en section 5.3.2 (page 90).

Nous avons traité des problèmes à 20 zones par cargo et 4 grues. Les activités ont des durées variant de 5 à 800 minutes, avec un facteur de robustesse simulé aléatoirement pour chaque activité correspondant à un pourcentage entre 0 et 25% de sa durée. Ces données sont réalistes pour des bateaux de taille moyenne.

La technique d'exploration utilisée est une recherche à voisinage large guidée par la propagation [PSF04], avec une remise en cause aléatoire de 30% des activités et une stratégie de choix de la variable de début ayant le plus petit domaine, affectée avec la date la plus petite d'abord. Nous avons fixé pour les dix instances une limite de temps de cinq minutes.

Nous avons calculé une borne inférieure sans prendre en compte les contraintes annexes, et avons comparé trois approches à cette borne : dans le cas d'un CuSP classique (colonne CAP), d'un modèle robuste utilisant FlexC (colonne FlexC) et d'un modèle robuste où le slack est ajouté en dur à la durée de chaque activité (colonne Ad-Hoc). Dans chacune de ces trois colonnes est indiqué la distance moyenne à la borne inférieure ainsi que, entre parenthèse, la deviation moyenne.

La colonne α donne la position relative de la valeur d'objectif de l'approche robuste (FlexC) comparée au makespan du CuSP et de l'approche Ad-Hoc. La colonne γ est le ratio entre le modèle Ad-Hoc et celui utilisant FlexC. Une valeur $\gamma = 2$ indique que le modèle Ad-Hoc double la distance à la borne inférieure théorique, par rapport au modèle utilisant FlexC.

#	CAP	FlexC	Ad-Hoc	α	γ
1	10.2 (0.08)	20.7 (2.20)	36.2 (3.28)	0.40	2.47
2	8.6 (0)	19.4 (2.13)	32.5 (1.62)	0.45	2.21
3	5.7 (0)	19.2 (3.08)	28.5 (0.60)	0.59	1.68
4	9.3 (0)	17.7 (0.47)	30.9 (3.66)	0.38	2.57
5	6.3 (0)	20.2 (2.10)	35.3 (0)	0.47	2.08
6	6.4 (0)	17.8 (1.37)	30.7 (0.15)	0.46	2.13
7	4 (0)	15.5 (1.08)	35.8 (2.40)	0.36	2.76
8	1.8 (0)	19.8 (2.35)	27.1 (1.42)	0.71	1.40
9	13.2 (0)	15.5 (2.61)	22 (0.31)	0.26	3.82
10	7.2 (0)	19.4 (1.63)	31.6 (0.07)	0.5	2.0

TABLE A.1 – Distances en % de trois approches comparées à une borne inférieure théorique.

A.3.2 Comparaison des propagateurs

Instances *Pack* en satisfaction

Nous avons comparé les nombres de noeuds, d'échecs, de backtrack et le temps moyen, sur 10 exécutions avec comme heuristique *activity based*. Nous avons cherché à retrouver la meilleure solution connue, dans un temps limite de 1 minute.

#	Noeud	Fail	BackTrack	ms
1	110248	110227	220418	6985
2	3988	3964	7918	241
3	13351	13330	26641	804
4	897204	897191	1794361	60000
5	20	0	0	3
6	1062081	1062062	2124109	59384
7	27	16	24	2
8	24	6	6	2
9	25	3	3	2
10	173	156	308	11
11	887800	887784	1775544	60000
12	923550	923536	1847041	60000
13	23	13	21	2
14	944931	944916	1889819	60000
15	1159511	1159500	2318980	60000
16	918380	918367	1836678	60000
17	994932	994896	1989732	60000
18	694789	694770	1389498	60000
19	884673	884654	1769266	60000
20	606249	606230	1212422	60000
21	21	4	4	3
22	672886	672875	1345721	60000
23	541699	541678	1083318	48576
24	702926	702911	1405779	60000
25	664697	664677	1329330	60000
26	333289	333269	666525	24462
27	1175	1151	2285	105
28	704404	704390	1408723	60000
29	372072	372052	744062	24912
30	221	209	413	16
31	1443900	1443881	2887738	60000
32	1319627	1319601	2639176	60000
33	453358	453346	906670	22134
34	1573011	1572998	3145965	60000
35	294661	294633	589240	10153
36	1543714	1543693	3087353	60000
37	1258846	1258824	2517576	60000
38	1243022	1243000	2485978	60000
39	1609270	1609239	3218449	60000
40	3376	3339	6656	151
41	967364	967341	1934602	60000
42	946508	946484	1892903	60000
43	34	0	0	2
44	2494	2470	4927	256
45	791239	791219	1582425	36295
46	2379	2365	4696	206
47	42199	42178	84334	2420
48	1434933	1434916	2869792	60000
49	984206	984188	1968329	60000
50	822376	822361	1644682	60000
51	942278	942256	1884455	60000
52	1310842	1310832	2621638	60000
53	1541	1519	3014	114
54	15	0	0	2
55	1000106	1000090	2000147	60000

TABLE A.2 – Statistique de satisfaction sur les instances pack, avec Time-Table

#	Noeud	Fail	BackTrack	ms
1	97232	97211	194386	14994
2	1113	1089	2168	242
3	8768	8747	17475	1536
4	306828	306815	613596	60000
5	25	1	1	5
6	447142	447133	894237	60000
7	21	10	13	3
8	17	0	0	3
9	25	3	3	5
10	143	126	248	26
11	259639	259626	519230	60000
12	256673	256662	513292	60000
13	14	4	5	3
14	288571	288558	577098	60000
15	438930	438920	877801	60000
16	196688	196674	393284	60000
17	147760	147742	295465	60000
18	174996	174982	349936	60000
19	190235	190215	380401	60000
20	159632	159614	319184	60000
21	21	4	4	6
22	158233	158223	316414	60000
23	143753	143733	287436	40565
24	3134	3117	6223	1222
25	162586	162566	325100	60000
26	56167	56147	112283	16037
27	264	241	465	96
28	148563	148546	297061	60000
29	256743	256724	513406	51889
30	191	179	353	24
31	30016	29995	59945	3357
32	163389	163365	326716	24045
33	310855	310843	621664	29241
34	518205	518192	1036347	60000
35	66759	66731	133436	4910
36	419183	419162	838306	60000
37	412233	412214	824377	60000
38	383940	383924	767800	60000
39	698257	698230	1396423	60000
40	2408	2371	4720	281
41	256848	256828	513611	60000
42	302371	302350	604643	60000
43	34	0	0	4
44	2333	2308	4603	547
45	560232	560207	1120395	60000
46	2339	2325	4616	442
47	40763	40742	81462	5464
48	491204	491189	982335	60000
49	217126	217106	434173	60000
50	359925	359907	719779	60000
51	303393	303373	606698	60000
52	541028	541017	1082006	60000
53	1536	1514	3004	252
54	15	0	0	4
55	391679	391665	783302	60000

TABLE A.3 – Statistique de satisfaction sur les instances pack, avec Edge-Finding

#	Noeud	Fail	BackTrack	ms
1	58152	58135	116265	15889
2	253	230	449	94
3	1899	1879	3743	362
4	203965	203952	407885	60000
5	25	1	1	10
6	269170	269163	538312	60000
7	21	10	13	7
8	17	0	0	4
9	25	3	3	6
10	141	124	244	33
11	164710	164696	329368	60000
12	137748	137735	275446	60000
13	14	4	5	3
14	146166	146152	292283	60001
15	252135	252124	504209	60000
16	101295	101280	202531	60000
17	94680	94660	189293	60000
18	172548	172532	345038	60000
19	138750	138730	277439	60000
20	60902	60888	121745	60000
21	20	4	4	10
22	82432	82421	164803	60000
23	92085	92063	184097	52673
24	2932	2916	5815	2761
25	84758	84743	169459	60000
26	28860	28840	57669	15206
27	91	68	122	41
28	91435	91422	182820	60000
29	173132	173123	346214	60000
30	24	12	20	8
31	103	87	149	20
32	110850	110826	221638	28793
33	206916	206904	413786	32926
34	305855	305842	611642	60000
35	8504	8476	16926	1884
36	3785	3750	7477	719
37	248251	248232	496447	60000
38	180410	180392	360759	60000
39	402085	402063	804094	60000
40	319	283	544	93
41	112041	112025	224000	60000
42	139909	139887	279737	60000
43	34	0	0	5
44	2309	2284	4555	897
45	479325	479305	958595	60000
46	2312	2298	4562	681
47	34653	34632	69242	5792
48	288990	288973	577896	60000
49	142923	142906	285791	60000
50	212663	212647	425258	60000
51	140024	140003	279967	60000
52	222230	222216	444418	51555
53	1536	1514	3004	418
54	15	0	0	6
55	227029	227015	454004	60000

TABLE A.4 – Statistique de satisfaction sur les instances pack, avec Time-Table-Edge-Finding

#	Noeud	Fail	BackTrack	ms
1	58454	58437	116869	29852
2	319	295	580	216
3	1865	1845	3675	776
4	136744	136729	273437	60000
5	25	1	1	8
6	188526	188516	377014	60000
7	18	7	7	7
8	17	0	0	6
9	25	3	3	10
10	133	116	228	48
11	97642	97627	195236	60000
12	81353	81342	162655	60000
13	14	4	5	4
14	98616	98602	197182	60000
15	184202	184193	368367	60000
16	72826	72811	145591	60000
17	51776	51759	103494	60002
18	85200	85185	170349	60000
19	64684	64667	129314	60002
20	33512	33495	66960	60000
21	20	4	4	15
22	42208	42201	84368	60000
23	70134	70124	140229	60000
24	2043	2027	4037	3327
25	42369	42352	84672	60000
26	25078	25058	50105	23466
27	68	46	80	50
28	64945	64931	129837	60000
29	122555	122545	245061	60000
30	23	11	18	11
31	16	2	2	5
32	99304	99280	198546	44389
33	109943	109931	219840	34485
34	176823	176811	353588	60000
35	5057	5029	10032	2285
36	8631	8608	17197	3030
37	107532	107512	214980	60000
38	83488	83471	166907	60001
39	1132	1103	2192	250
40	296	260	498	190
41	71462	71445	142834	60000
42	90947	90929	181826	60000
43	34	0	0	11
44	2296	2271	4529	1497
45	209459	209440	418862	60000
46	2151	2137	4240	1193
47	24284	24263	48504	9660
48	125293	125277	250510	60000
49	77212	77192	154366	60000
50	110929	110913	221787	60000
51	91099	91080	182130	60000
52	52556	52542	105070	25892
53	1536	1514	3004	679
54	15	0	0	9
55	110568	110552	221071	60000

TABLE A.5 – Statistique de satisfaction sur les instances pack, avec le Raisonnement Energétique

Instances *Pack* en optimisation

Nous avons comparé les nombres de noeuds, d'échecs, de backtrack et le temps pour prouver l'optimalité avec comme heuristique *activity based*, dans un temps limite de 10 minutes.

#	KnownLB	KnownUB	obj	nbNodes	nbFails	nbBT	tempsMs
1	23	23	23	14425770	14412215	28821398	600000
2	32	32	32	11178551	11157055	22310524	600000
3	29	29	29	12386818	12369093	24735193	600000
4	40	44	44	12959939	12945293	25886271	600000
5	42	42	42	13515143	13499740	26994855	600000
6	47	47	47	15503021	15490335	30976159	600000
7	41	41	41	19414160	19407923	38813564	600000
8	44	44	44	15432275	15421353	30838984	600000
9	50	72	72	16923041	16912570	33821283	600000
10	38	38	38	17307926	17300392	34598049	600000
11	44	44	44	12945667	12923237	25840288	600000
12	45	45	45	12376552	12359653	24714581	600000
13	36	36	36	16956132	16946913	33890670	600000
14	45	45	45	12346180	12334291	24665065	600000
15	43	43	43	16116273	16107428	32212070	600000
16	63	63	63	10084310	10067058	20129262	600000
17	62	62	63	9812538	9792241	19579481	600000
18	60	60	61	12384253	12359029	24712462	600000
19	59	59	59	11058110	11037034	22068512	600000
20	62	62	63	9311449	9296924	18589995	600000
21	51	51	51	87487	77410	151965	7556
22	59	59	59	9002370	8996865	17991585	600000
23	51	51	52	11640239	11630452	23258132	600000
24	56	56	57	9938143	9932381	19863013	600000
25	49	70	71	10446098	10416100	20825206	600000
26	54	54	54	13011266	13000054	25996755	600000
27	55	55	56	11153775	11129360	22252267	600000
28	64	64	64	9377864	9360897	18717897	600000
29	43	43	44	14099627	14060858	28111616	600000
30	20	20	20	13663841	13643335	27282586	600000
31	70	70	70	22272280	22259904	44516883	600000
32	80	80	84	19215991	19205540	38408426	600000
33	78	78	78	26868363	26860492	53718747	600000
34	73	73	74	20392108	20382242	40761640	600000
35	72	77	77	25399007	25393926	50786081	600000
36	85	106	106	23437657	23416590	46828942	600000
37	98	138	138	18152750	18130042	36255681	600000
38	86	86	92	18384124	18371460	36740353	600000
39	90	111	111	19738752	19720385	39435800	600000
40	80	91	91	18529973	18504755	37005031	600000
41	27	27	27	12203631	12185977	24369190	600000
42	29	29	29	11686406	11667996	23332967	600000
43	105	105	105	11747120	11733760	23462670	600000
44	103	103	103	12202239	12193047	24382444	600000
45	86	87	87	20338829	20329664	40656815	600000
46	98	128	128	16126852	16118558	32234290	600000
47	103	107	107	19590867	19578616	39154004	600000
48	76	77	77	13460862	13449616	26896424	600000
49	29	29	30	11697265	11677862	23352613	600000
50	88	109	109	18000133	17989717	35976377	600000
51	29	29	29	11732153	11713744	23424455	600000
52	85	85	86	16548243	16539570	33076308	600000
53	93	113	113	15000960	14993020	29983917	600000
54	91	100	100	14773310	14753355	29501282	600000
55	91	97	97	15659386	15648011	31292807	600000

TABLE A.6 – Statistique en optimisation sur les instances pack, avec Time-Table

#	KnownLB	KnownUB	obj	nbNodes	nbFails	nbBT	tempsMs
1	23	23	23	6057614	6043548	12083959	600000
2	32	32	32	4268879	4248510	8493658	600000
3	29	29	29	4521898	4504032	9005060	600000
4	40	44	44	5405037	5390521	10776749	600000
5	42	42	42	3851908	3836451	7668369	600000
6	47	47	47	7308172	7295500	14586497	600000
7	41	41	41	9176377	9170139	18337996	600000
8	44	44	44	5510571	5499687	10995635	600000
9	50	72	72	7787945	7777279	15550788	600000
10	38	38	38	7336387	7328679	14654601	600000
11	44	44	44	4504306	4481869	8957631	600000
12	45	45	45	4572763	4555902	9107101	600000
13	36	36	36	5617564	5608160	11213139	600000
14	45	45	45	4153127	4140925	8278208	600000
15	43	43	43	6672313	6663392	13324017	600000
16	63	63	63	4531673	4514359	9023737	600000
17	62	62	63	4727632	4707320	9409624	600000
18	60	60	61	4871363	4845686	9685836	600000
19	59	59	59	3513007	3492320	6979332	600000
20	62	62	63	4038515	4024180	8044671	600000
21	51	51	51	67594	58100	113740	22930
22	59	59	59	1124785	1119766	2237700	600000
23	51	51	52	3341711	3330649	6658036	600000
24	56	56	57	1530907	1525308	3048753	600000
25	49	70	71	3991486	3962571	7918411	600000
26	54	54	54	3981034	3969275	7935203	600000
27	55	55	56	3977568	3953461	7900548	600000
28	64	64	64	3455719	3439650	6875412	600000
29	43	43	44	5407258	5369081	10728251	600000
30	20	20	20	4497869	4477418	8950752	600000
31	70	70	70	10352374	10343759	20685502	600000
32	80	80	84	9343475	9333033	18663499	600000
33	78	78	78	12779086	12771226	25540651	600000
34	73	73	74	7792001	7782286	15561825	600000
35	72	77	77	12264075	12259312	24516949	600000
36	85	106	106	10483945	10462226	20920453	600000
37	98	138	138	8185066	8162351	16320192	600000
38	86	86	92	9025283	9012600	18022555	600000
39	90	111	111	7473136	7454774	14904624	600000
40	80	91	91	7843638	7818396	15632664	600000
41	27	27	27	4732823	4715384	9427999	600000
42	29	29	29	3769981	3751886	7500851	600000
43	105	105	105	2893624	2880209	5755595	600000
44	103	103	103	4765392	4756201	9508733	600000
45	86	87	87	8699331	8690160	17378062	600000
46	98	128	128	7464910	7456636	14910351	600000
47	103	107	107	9459339	9446906	18890719	600000
48	76	77	77	6082592	6071411	12140264	600000
49	29	29	30	3300830	3280790	6558449	600000
50	88	109	109	9011377	9001005	17999362	600000
51	29	29	29	3769625	3751531	7500145	600000
52	85	85	86	6233766	6223701	12444591	600000
53	93	113	113	6914812	6906894	13811756	600000
54	91	100	100	5430650	5411152	10817010	600000
55	91	97	97	7635134	7623928	15244812	600000

TABLE A.7 – Statistique en optimisation sur les instances pack, avec Time-Table-Edge-Finding

#	KnownLB	KnownUB	obj	nbNodes	nbFails	nbBT	tempsMs
1	23	23	23	14894	1409	33	5907
2	32	32	32	61736	41524	79943	32483
3	29	29	29	24568	6836	10861	10501
4	40	44	44	1513205	1498955	2993839	600000
5	42	42	42	2328012	2312554	4620582	600000
6	47	47	47	2196298	2183657	4363068	600000
7	41	41	41	4233554	4227362	8452447	600000
8	44	44	44	2715818	2704896	5406058	600000
9	50	72	72	1775974	1765538	3527359	600000
10	38	38	38	34311	26639	50899	9943
11	44	44	44	43439	20926	36048	16966
12	45	45	45	160113	143211	281965	72446
13	36	36	36	20232	10905	18910	3981
14	45	45	45	52353	40634	77994	15826
15	43	43	43	10191	1379	399	1849
16	63	63	63	96675	79379	154034	48701
17	62	62	63	1050655	1029306	2053438	600000
18	60	60	61	963637	937548	1869498	600000
19	59	59	59	378518	358020	710886	214212
20	62	62	63	836690	822524	1641454	600000
21	51	51	51	21446	11208	19793	22744
22	59	59	59	636839	631804	1262020	600000
23	51	51	52	61130	50064	97330	33439
24	56	56	57	660899	655201	1308982	600000
25	49	70	71	770286	742038	1477561	600000
26	54	54	54	1439378	1427209	2850956	600000
27	55	55	56	1258341	1234127	2461972	600000
28	64	64	64	851716	835274	1666841	600000
29	43	43	44	1701289	1661833	3313476	600000
30	20	20	20	21104	2041	628	8003
31	70	70	70	10615	2297	3081	2903
32	80	80	84	3244721	3234370	6466165	600000
33	78	78	78	5160631	5152783	10303788	600000
34	73	73	74	3757685	3747938	7493446	600000
35	72	77	77	3837834	3832498	7663349	600000
36	85	106	106	2468345	2448744	4893978	600000
37	98	138	138	3065297	3042623	6081005	600000
38	86	86	92	1646693	1633961	3265427	600000
39	90	111	111	2556655	2538318	5072138	600000
40	80	91	91	1767359	1742124	3479948	600000
41	27	27	27	79503	62164	121717	36738
42	29	29	29	56240	38273	73837	29579
43	105	105	105	1546414	1533085	3061421	600000
44	103	103	103	1557088	1547898	3092404	600000
45	86	87	87	3678838	3669380	7336453	600000
46	98	128	128	2722609	2714287	5425929	600000
47	103	107	107	3284694	3271503	6539642	600000
48	76	77	77	2067036	2055566	4108521	600000
49	29	29	30	62667	44562	86554	40298
50	88	109	109	3036248	3026402	6050196	600000
51	29	29	29	56240	38273	73837	29130
52	85	85	86	2495171	2486216	4969937	600000
53	93	113	113	2585984	2578076	5154007	600000
54	91	100	100	1932457	1912761	3820377	600000
55	91	97	97	2872217	2861050	5719170	600000

TABLE A.8 – Statistique en optimisation sur les instances pack, avec le Raisonnement Energétique

Bibliographie

[AB93] Abder Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Math. Comp. Model*, 17(7) :57–73, April 1993. 28

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11) :832–843, November 1983. 95

[ALT11] Christian Artigues, Roel Leus, and Fabrice Talla Nobibon. Robust optimization for resource-constrained project scheduling with uncertain activity durations. In *IEEM*, pages 101–105, 2011. 50

[APM+99] Giorgio Ausiello, Marco Protasi, Alberto Marchetti-Spaccamela, Giorgio Gambosi, Pierluigi Crescenzi, and Vigo Kann. *Complexity and Approximation : Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999. 21

[Bak74] K.R. Baker. *Introduction to sequencing and scheduling*. Wiley, 1974. 27

[BBG14] Haykel Boukadida, Sid-Ahmed Berrani, and Patrick Gros. A novel modeling for video summarization using constraint satisfaction programming. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Ryan McMahan, Jason Jerald, Hui Zhang, Steven M. Drucker, Chandra Kambhamettu, Maha El Choubassi, Zhigang Deng, and Mark Carlson, editors, *Advances in Visual Computing - 10th International Symposium, ISVC 2014, Las Vegas, NV, USA, December 8-10, 2014, Proceedings, Part II*, volume 8888 of *Lecture Notes in Computer Science*, pages 208–219. Springer, 2014. 95

[BEG+11] Thierry Benoist, Bertrand Estellon, Frédéric Gardi, Romain Megel, and Karim Nouioua. Localsolver 1.x : a black-box local-search solver for 0-1 programming. *4OR*, 9(3) :299–316, 2011. 14

[Bes06] Christian Bessière. Constraint propagation. Technical Report 06020, Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier, 2006. 22

[BHLS04] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, pages 146–150. IOS Press, 2004. 24

[BLN01] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling : Applying Constraint Programming to Scheduling Problems*. International Series in Operations Research and Management Science. Kluwer, 2001. 11, 28, 45, 46, 47, 48, 58, 63, 66, 70

[BMS08] Jean-Charles Billaut, Aziz Moukrim, and Eric Sanlaville, editors. *Flexibility and Robustness in Scheduling*. Wiley, 2008. 49, 50

[BN02] Aharon Ben-Tal and Arkadi Nemirovski. Robust optimization : methodology and applications. *Mathematical Programming*, 92(3) :453–480, 2002. 49, 71

[BS03] Dimitris Bertsimas and Melvyn Sim. Robust discrete optimization and network flows. *Mathematical Programming*, 98(1-3) :49–71, 2003. 49, 71

[BS04] Dimitris Bertsimas and Melvyn Sim. The price of robustness. *Oper. Res.*, 52(1) :35–53, January 2004. 49, 71

[Cha13] Gilles Chabert. IBEX 2.0, 2013. [page web d'IBEX, en ligne au 01-Sept-2015]. 14

[CL94] Yves Caseau and Francois Laburthe. Improved clp scheduling with task intervals. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 369–383, 1994. 54

[Der14] Alban Derrien. Une nouvelle caractérisation des intervalles d'intérêt pour le raisonnement énergétique. In *JFPC*, 2014. 70

[DFPP15] Alban Derrien, Jean-Guillaume Fages, Thierry Petit, and Charles Prud'homme. A global constraint for a tractable class of temporal optimization problems. In Gilles Pesant, editor, *Proc. CP*, pages 105–120, 2015. 15, 95

[DJB01] Andrew J. Davenport, Christophe Jefflot, and J. Christopher Beck. Slack-based techniques for robust schedules. In *European Conference on Planning*, pages 7–18, 2001. 49, 50, 71

[DP13] Alban Derrien and Thierry Petit. The Energetic Reasoning Checker Revisited. In *CP Doctoral Program 2013*, pages 55–60, September 2013. 48, 59, 61

[DP14] Alban Derrien and Thierry Petit. A new characterization of relevant intervals for energetic reasoning. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 289–297, 2014. 15, 70, 93

[DPZ14a] Alban Derrien, Thierry Petit, and Stéphane Zampelli. A declarative paradigm for robust cumulative scheduling. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 298–306, 2014. 15, 49, 88, 90, 91, 93, 97

[DPZ14b] Alban Derrien, Thierry Petit, and Stéphane Zampelli. Dynamic sweep filtering algorithm for flexc. *CoRR*, abs/1408.5377, 2014. 91

[ELT89] Jacques Erschler, Pierre Lopez, and Catherine Thuriot. Scheduling under time and resource constraints. In *Proc. of Workshop on Manufacturing Scheduling, 11th IJCAI*, 1989. 14, 29, 44, 45

[FQ14] Hamed Fahimi and Claude-Guy Quimper. Linear-time filtering algorithms for the disjunctive constraint. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014. 40

[GB06] Laurent Granvilliers and Frédéric Benhamou. Algorithm 852 : Realpaver : an interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32(1) :138–156, 2006. 14

[GJ79] Michael Randolph Garey and David Stifler Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. 20, 29

[Glo65] Fred Glover. A multiphase-dual algorithm for the zero-one integer programming problem. *OR*, 13(6) :879–919, 1965. 55

[HE79] Robert Martin 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*, IJCAI'79, pages 356–364. Morgan Kaufmann Publishers Inc., 1979. 24, 61, 68

[Heb04] Emmanuel Hebrard. Robust solutions for constraint satisfaction and optimization. In *AAAI*, pages 952–953, 2004. 49, 71

[HHW04] Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Super solutions in constraint programming. In *CPAIOR*, pages 157–172, 2004. 49, 50

[Kar14] Ashok Kumar Karunanithi. A survey, discussion and comparison of sorting algorithms, 2014. 20

[KFF13] Roger Kameugne, Sévérine Betmbe Fetgo, and Laure Pauline Fotso. Energetic extended edge finding filtering algorithm for cumulative resource constraints. *American Journal of Operations Research*, 3(06) :589, 2013. 48

[KFSN14] Roger Kameugne, Laure Pauline Fotso, Joseph D. Scott, and Youcheu Ngo-Kateu. A quadratic edge-finding filtering algorithm for cumulative resource constraints. *Constraints*, 19(3) :243–269, 2014. 43, 68, 69

[KFSNK11] 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'11), 17th International Conference*, volume 6876 of *Lecture Notes in Computer Science*, pages 478–492. springer, 2011. 40

[Kre88] Mark W. Krentel. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36(3) :490 – 509, 1988. 21

[KS96] Rainer Kolisch and Arno Sprecher. Psplib – a project scheduling problem library. *EUROPEAN JOURNAL OF OPERATIONAL RESEARCH*, 96 :205–216, 1996. 41, 61, 68

[Lah79] Abdelkader Lahrichi. *Ordonnancements : la notion de partie obligatoire et son application aux problèmes cumulatifs*. PhD thesis, Paris 6, 1979. In French. 30

[Lah82] Abdelkader Lahrichi. Ordonnancements. La notion de "parties obligatoires" et son application aux problèmes cumulatifs. *RAIRO - Operations Research - Recherche Opérationnelle*, 16(3) :241–262, 1982. 14, 29

[LCB14] Arnaud Letort, Mats Carlsson, and Nicolas Beldiceanu. Synchronized sweep algorithms for scalable scheduling constraints. *Constraints*, pages 1–52, 2014. 11, 29, 31, 32, 33, 34, 35, 77, 87, 91, 94

[LEE92] Pierre Lopez, Jacques Erschler, and Patrick Esquirol. Ordonnancement de tâches sous contraintes : une approche énergétique. *Automatique, Productique, Informatique Industrielle*, 26 :453–481, 1992. 44

[Lop91] Pierre Lopez. *Energy-based approach for task scheduling under time and resource constraints*. Theses, Université Paul Sabatier - Toulouse III, September 1991. In french. 44

[Mon74] Ugo Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Inf. Sci.*, 7 :95–132, 1974. 22

[MV08] Luc Mercier and Pascal Van Hentenryck. Edge finding for cumulative scheduling. *INFORMS Journal on Computing*, 20(1) :143–153, 2008. 39

[MV12] Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *CPAIOR*, pages 228–243, 2012. 24

[Nui94] Wilhemus Petronella Maria Nuijten. *Time and resource constrained scheduling : A constraint satisfaction approach*. PhD thesis, Eindhoven University of Technology, 1994. 14, 29, 36, 37, 39

[Pap94] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994. 20, 21

[PFL14] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014. 22, 24, 61, 68

[Pin88] Eric Pinson. *Le problème de job-shop*. PhD thesis, Université Paris VI, 1988. in French. 36

[Pin12] MichaelL. Pinedo. Overview of stochastic scheduling problems. In *Scheduling*, pages 607–610. Springer US, 2012. 49

[PSF04] Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. In *Proc. CP*, pages 468–481, 2004. 90, 97

[Pug13] Jean-Francois Puget. No, The TSP Isn't NP Complete, 2013. [page web d'IBM, en ligne au 01-Sept-2015]. 21

[Ref04] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP*, pages 557–571, 2004. 24

[SFS13] Andreas Schutt, Thibaut Feydy, and Peter J. Stuckey. Explaining time-table-edge-finding propagation for the cumulative resource constraint. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 234–250, 2013. 41

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Proc. CP*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin Heidelberg, 1998. 14

[Vil09] Petr Vilím. Edge finding filtering algorithm for discrete cumulative resources in o(kn log n). In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, CP'09, pages 802–816. Springer-Verlag, 2009. 39

[Vil11] 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. 14, 29, 41, 42, 43, 56, 69, 70, 94

[VSD95] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). In Andreas Podelski, editor, *Constraint Programming : Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 293–316. Springer Berlin Heidelberg, 1995. 26

[WBB09] Christine Wei Wu, Kenneth N. Brown, and J. Christopher Beck. Scheduling with uncertain durations : Modeling beta-robust scheduling with constraints. *Computers & OR*, 36(8) :2348–2356, 2009. 49, 50, 71

[WS06] Armin Wolf and Gunnar Schrader. O(n log n) overload checking for the cumulative constraint and its application. In *Proceedings of the 16th International Conference on Applications of Declarative Programming and Knowledge Management*, INAP'05, pages 88–101, Berlin, Heidelberg, 2006. Springer-Verlag. 40, 56

[ZVS+13] Stéphane Zampelli, Yannis Vergados, Rowan Van Schaeren, Wout Dullaert, and Birger Raa. The berth allocation and quay crane assignment problem using a cp approach. In *Proc. CP*, pages 880–896, 2013. 49, 71, 72, 88, 89

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

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Alban DERRIEN

Ordonnancement cumulatif en programmation par contraintes

Caractérisation énergétique des raisonnements et solutions robustes

Cumulative scheduling in constraint programming

energetic characterization of reasoning and robust solutions

Résumé

La programmation par contraintes est une approche régulièrement utilisée pour traiter des problèmes d'ordonnancement variés. Les problèmes d'ordonnancement cumulatifs représentent une classe de problèmes dans laquelle des tâches non morcelable peuvent être effectuées en parallèle. Ces problèmes apparaissent dans de nombreux contextes réels, tels que par exemple l'allocation de machines virtuelles ou l'ordonnancement de processus dans le "cloud", la gestion de personnel ou encore d'un port. De nombreux mécanismes ont été adaptés et proposés en programmation par contraintes pour résoudre les problèmes d'ordonnancement. Les différentes adaptations ont abouti à des raisonnements qui semblent a priori significativement distincts.

Dans cette thèse nous avons effectué une analyse détaillée des différents raisonnements, proposant à la fois une notation unifiée purement théorique mais aussi des règles de dominance, permettant une amélioration significative du temps d'exécution d'algorithmes issus de l'état de l'art, pouvant aller jusqu'à un facteur sept. Nous proposons aussi un nouveau cadre de travail pour l'ordonnancement cumulatif robuste, permettant de trouver des solutions supportant qu'à tout moment une ou plusieurs tâches soit retardées, sans remise en cause de l'ordonnancement généré et en gardant une date de fin de projet satisfaisante. Dans ce cadre, nous proposons une adaptation d'un algorithme de l'état de l'art, Dynamic Sweep.

Mots clés

Ordonnancement, Programmation par Contraintes, Cumulatif, Optimisation Combinatoire, Robustesse.

Abstract

Constraint programming is an approach regularly used to treat a variety of scheduling problems. Cumulative scheduling problems represent a class of problems in which non-preemptive tasks can be performed in parallel. These problems appear in many contexts, such as for example the allocation of virtual machines, the ordering process in the "cloud", personnel management or a port. Many mechanisms have been adapted and offered in constraint programming to solve scheduling problems. The various adaptations have resulted in reasoning that appear a priori significantly different.

In this thesis we performed a detailed analysis of the various arguments, offering both a theoretical unified caracterization but also dominance rules, allowing a significant improvement in execution time of algorithms from the state of the art, up to a factor of seven. we also propose a new framework for robust cumulative scheduling, to find solutions that support at any time one or more tasks to be delayed while keeping a satisfactory end date of the project and without calling into question the generated scheduling. In this context, we propose an adaptation of an algorithm of the state of the art, Dynamic Sweep.

Key Words

Scheduling, Constraint Programming, Cumulative, Combinatorial Optimisation , Robustess.