View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CorpusUL

Le filtrage des bornes pour les contraintes Cumulative et Multi-Inter-Distance

Mémoire

Pierre Ouellet

Maîtrise en informatique
Maître ès sciences (M.Sc.)

Québec, Canada

© Pierre Ouellet, 2014

Résumé

Ce mémoire traite de la résolution de problèmes d'ordonnancement à l'aide de la programmation par contraintes. Il s'intéresse principalement aux contraintes globales et particulièrement à la contrainte cumulative. Il passe en revue les règles permettant de la filtrer et les principaux algorithmes qui les appliquent. Il explique le Edge-Finder de Vilím et son arbre cumulatif. Il propose un algorithme plus performant et plus général pour appliquer les règles découlant du raisonnement énergétique.

Le mémoire traite du cas particulier où toutes les tâches sont de durée identique. Pour modéliser efficacement ce type de problèmes, on y conçoit la contrainte multi-inter-distance. L'algorithme d'ordonnancement de López-Ortiz et Quimper est adapté pour réaliser un algorithme qui applique la cohérence de bornes. La contrainte multi-inter-distance s'avère efficace à résoudre le problème de séquençage des atterrissages d'avions du banc d'essai d'Artiouchine et Baptiste.

Abstract

This thesis discusses how to solve scheduling problems using constraint programming. We study global constraints and particularly the Cumulative constraint. We survey its main filtering rules and their state-of-the-art filtering algorithms. We explain the Vilím's Edge-Finder and its cumulative tree. We introduce a more efficient and more general algorithm that enforces the filtering rules from the energetic reasoning.

We study the special case where all tasks have identical processing times. To efficiently model such problems, we introduce the Multi-Inter-Distance constraint. The scheduling algorithm by López-Ortiz and Quimper is adapted to produce a filtering algorithm enforcing bounds consistency. The constraint Multi-Inter-Distance is proved efficient to solve the runway scheduling problem on the benchmark by Artiouchine and Baptiste.

Table des matières

Résumé iii

Abstract v

Table des matières vii

Liste des tableaux ix

Liste des figures xi

Remerciements xiii

Introduction 1

1 La programmation par contraintes 3
- 1.1 Problèmes de satisfaction de contraintes 3
- 1.2 Problèmes d'ordonnancement 7

2 Filtrage de la contrainte Cumulative 11
- 2.1 Background 13
- 2.2 Le Edge-Finder de Vilím 21
- 2.3 Contributions 34

3 Ordonnancement de tâches identiques 51
- 3.1 Background 51
- 3.2 L'algorithme de López-Ortiz et Quimper 55
- 3.3 Filtrage de la contrainte Multi-Inter-Distance 66

Conclusion 83

A Tableau des notations 85

Bibliographie 87

Liste des tableaux

2.1 Résultats comparés 50

3.1 Résultats pour piste unique 80
3.2 Résultats pour pistes multiples 81

A.1 Notations de base utilisées dans le document. 85

Liste des figures

1.1 Arbre de recherche . . . 4
1.2 Schématisation d'une tâche . . . 7
1.3 Schématisation d'un ordonnancement . . . 8
1.4 Représentation géométrique de l'enveloppe . . . 9

2.1 Extended-Edge-Finding . . . 15
2.2 Time-Tabling . . . 16
2.3 Time-Table-Extended-Edge-Finding . . . 17
2.4 Propriété gauche-droite . . . 18
2.5 Not-First/Not-Last . . . 20
2.6 Opposé mathématique . . . 21
2.7 Arbre binaire essentiellement complet . . . 22
2.8 Arbre cumulatif . . . 23
2.9 Détection par Edge-Finding . . . 24
2.10 Détection par Extended-Edge-Finding . . . 26
2.11 Mécanique de l'arbre cumulatif . . . 27
2.12 Condition d'énergie minimale . . . 29
2.13 Scission de l'arbre cumulatif . . . 31
2.14 Alternative à la scission . . . 33
2.15 Ajustement par Time-Tabling . . . 37
2.16 Intervalles contigus . . . 38
2.17 Les quatre détections . . . 40
2.18 Schématisation des valeurs d'enveloppe et d'énergie . . . 43
2.19 Tâches diminuées et tâches virtuelles . . . 49

3.1 Solution obtenue par EDD . . . 51
3.2 Graphe d'ordonnancement . . . 57
3.3 Graphe d'ordonnancement et sa matrice des distances . . . 58
3.4 Cycle négatif . . . 67
3.5 Initialisation du vecteur de distances . . . 75
3.6 Structure de données principale . . . 76
3.7 Boucle d'attente . . . 77
3.8 Exemple des données . . . 78
3.9 Nombre de problèmes résolus pour piste unique . . . 80
3.10 Nombre de problèmes résolus pour pistes multiples . . . 81

Remerciements

Remerciements au département d'informatique et de génie logiciel pour la qualité des cours dispensés. Remerciements particuliers au professeur Jean-Marie Beaulieu pour qui le format restreint de cette page ne permet pas d'expliquer pleinement la justification de l'épithète *captivant*. Remerciements au coloré directeur de programme, le professeur Chaib-draa, pour ses bons conseils. Remerciements à mes correcteurs : le professeur Gaudreault et le professeur Laviolette. Et remerciements tout particuliers à mon directeur de recherche, le professeur Claude-Guy Quimper, pour son talent et sa patience.

Introduction

La planification des opérations représente un défi de taille pour l'industrie. Dans les manufactures, plusieurs tâches doivent être planifiées sur les chaînes de montage. Tout en considérant un ordre partiel au sein des tâches, on doit par exemple installer la carosserie d'un voiture avant de poser son pare-brise, on cherche à optimiser l'utilisation des robots sans les surcharger. De façon générale, un problème d'ordonnancement consiste à déterminer à quel moment on doit exécuter une tâche et quelle ressource on doit y assigner. De plus en plus, l'industrie fait appel à des outils informatiques d'aide à la décision pour résoudre ses problèmes d'ordonnancement.

La rapidité dans la planification joue un rôle crucial pour la compétitivité de l'entreprise. Les fluctations dans la demande des produits manufacturés nécessitent une planification périodique. Si un bris d'équipement ou une indisponibilité subite de la main-d'oeuvre survient, l'entreprise doit être en mesure de replanifier ses opérations dans le but de relancer la chaîne de montage. Il peut prendre plusieurs jours à un ordinateur pour calculer la solution optimale. Par conséquent, on interrompt généralement le calcul avant terme et procède avec une solution sous-optimale, c'est-à-dire la meilleure solution trouvée jusqu'alors. Dans cette optique, une bonne performance de l'outil informatique nécessite de trouver une bonne solution dans un court laps de temps. Pour augmenter l'efficacité d'un outil de calcul, il faut améliorer la qualité de la solution ou réduire le temps de calcul pour l'obtenir. Dans ce mémoire, nous explorons deux approches qui mènent à la réduction du temps de calcul.

Dans un premier temps nous verrons comment la programmation par contraintes peut résoudre des problèmes combinatoires tels les problèmes d'ordonnancement. Nous verrons en particulier comment la contrainte cumulative peut modéliser un large éventail de problèmes d'ordonnancement. Nous proposons de nouveaux algorithmes qui filtrent l'espace de recherche de ces problèmes. Dans un second temps, nous étudions un cas particulier des problèmes d'ordonnancement où toutes les tâches et toutes les machines sur lesquelles elles s'exécutent sont similaires. Pour le modéliser et le résoudre, nous introduisons une nouvelle contrainte et un algorithme filtrant l'espace de recherche du problème. Les algorithmes que nous introduisons dans ce mémoire ont été en partie publiés dans des actes de conférences internationales avec comité de lecture.

1. Pierre Ouellet and Claude-Guy Quimper. The Multi-Interdistance Constraint. In Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 11), pages 629–634, 2011.
2. Pierre Ouellet and Claude-Guy Quimper. Time-Table-Extended-Edge-Finding for the Cumulative Constraint. In Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013), pages 562–577, 2013.

Chapitre 1

La programmation par contraintes

1.1 Problèmes de satisfaction de contraintes

Un *problème combinatoire* est un problème mathématique dont la résolution s'effectue dans un espace de recherche discret. Un *problème de satisfaction de contraintes* est un problème combinatoire défini par un tuple $\langle \mathcal{X}, \mathcal{C}, \text{dom} \rangle$ où $\mathcal{X}$ est un ensemble de variables, $\mathcal{C}$ est un ensemble de contraintes et dom est une fonction qui prend une variable et qui retourne le domaine associé à cette variable. Le *domaine* d'une variable X que l'on dénote $\text{dom}(X)$ est l'ensemble des valeurs permises pour cette variable. Un problème combinatoire considère toutes les combinaisons possibles parmi les valeurs des domaines de chacune des variables du problème.

Une *contrainte* définie une relation entre les valeurs d'un sous-ensemble des variables d'un problème. Ce sous-ensemble de variables régies par une contrainte s'appelle la *portée* et la cardinalité de la portée se nomme *l'arité*. Une solution au problème est une affectation d'une valeur $v_i \in \text{dom}(X_i)$ à une variable $X_i \in \mathcal{X}$ de sorte que toutes les contraintes soient satisfaites.

Exemple 1. *Considérons le problème de satisfaction de contraintes suivant :*

$$A < B \tag{1.1}$$

$$A \times B = 12 \tag{1.2}$$

$$dom(A) = \{1, 2, 3, 4, 5, 6\} \tag{1.3}$$

$$dom(B) = \{1, 2, 3, 4, 5, 6\} \tag{1.4}$$

Dans ce problème, l'ensembles des variables est $\{A, B\}$. Les contraintes sont les relations (1.1) et (1.2). Il existe deux solutions au problème : $\{A = 2, B = 6\}$ et $\{A = 3, B = 4\}$.

Un *problème d'optimisation* est un problème combinatoire où l'on cherche non-seulement à trouver une solution réalisable mais aussi une solution qui minimise ou maximise une fonction objectif. Dans l'exemple 1, si nous avions pour objectif de maximiser $2A + B$ l'unique solution au problème serait $\{A = 3, B = 4\}$.

Les problèmes combinatoires sont souvent résolus au moyen d'une fouille avec retours arrière. Une fouille explore un arbre de recherche. La racine de l'arbre correspond à une solution partielle vide, c'est-à-dire une solution où aucune valeur n'a été assignée aux variables. Un noeud hérite de la solution partielle de son parent auquelle il ajoute une affectation additionnelle d'une valeur à une variable. Une feuille au dernier niveau est donc une affectation complète des variables. La fouille débute à la racine de l'arbre qui devient le premier noeud courant. D'un noeud courant elle procède à la visite d'un enfant en affectant l'une des variables non-affectées. Si cette nouvelle affection ne viole aucune contrainte alors le noeud enfant devient le noeud courant sinon on poursuit l'exploration en affectant une nouvelle valeur à cette variable jusqu'à épuisement de son domaine. Après épuisement des valeurs du domaine de la variable, le noeud parent redevient le noeud courant. La fouille se termine lorsqu'une solution est trouvée ou lorsque l'arbre a été complétement exploré.

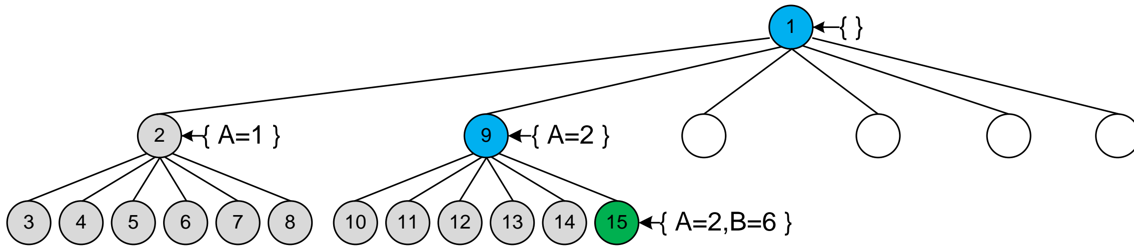


Figure 1.1: L'arbre de recherche de l'exemple 1 et l'ordre de visite de ses noeuds.

La taille de l'arbre de recherche croît exponentiellement en fonction du nombre de variables. La plupart des problèmes combinatoires étant NP-Difficile, il est normal de résoudre ces problèmes en temps exponentiel. Néanmoins, il existe des techniques pour réduire la taille d'un arbre de recherche. L'une d'entre elles est le filtrage qui repose sur des notions de support et de cohérence.

Un *support* est un témoin qui permet de croire qu'une contrainte peut être satisfaite. Une contrainte qui peut être satisfaite a nécessairement un support mais la converse n'est pas nécessairement vraie. Nous précisons la notion de support à l'aide des deux définitions suivantes :

Définition 1 (Support de domaine)**.** *Un support de domaine pour une contrainte C est une affectation des variables de la portée de la contrainte de sorte que chaque valeur affectée à une variable appartient au domaine de la variable. De plus, cette affectation doit satisfaire la contrainte.*

Définition 2 (Support de bornes)**.** *Un support de bornes pour une contrainte C est une affectation des variables de la portée de la contrainte de sorte que chaque valeur affectée à une variable X appartient à l'intervalle* $[\min(dom(X)), \max(dom(X))]$. *De plus, cette affectation doit satisfaire la contrainte.*

Exemple 2. *Considérons le problème suivant où* ALL-DIFFERENT *contraint les variables de sa portée à des affectations de valeurs distinctes :*

$$\text{All-Different}(X_1, X_2, X_3) \tag{1.5}$$

$$dom(X_1) = \{1, 3\} \tag{1.6}$$

$$dom(X_2) = \{1, 3\} \tag{1.7}$$

$$dom(X_3) = \{2, 3\} \tag{1.8}$$

Dans ce problème, les affectations $X_1 = 1, X_2 = 3$ *et* $X_3 = 2$ *sont un support de domaine alors que les affectations* $X_1 = 1, X_2 = 2$ *et* $X_3 = 3$ *sont un support de bornes. Notez qu'un support de bornes n'est pas nécesairement une solution au problème.*

Pour toutes les variables et leur domaine associé, une valeur pour laquelle il n'existe pas de support est une valeur qui ne fait partie d'aucune solution. De telles valeurs sont dites incohérentes. Le *filtrage* consiste à retirer les valeurs incohérentes du domaine des variables afin de réduire la taille de l'espace de recherche. Un algorithme de filtrage propre à une contrainte retire des domaines des valeurs pour lesquelles il n'existe pas de support. Si l'algorithme retire toutes les valeurs incohérentes, on dit des domaines qu'ils sont cohérents avec la contrainte. Tout comme il existe différents types de supports il existe différents types de cohérence. Nous précisons la notion de cohérence au moyen des deux définitions suivantes :

Définition 3 (Cohérence de domaine)**.** *Une contrainte* C *est dite cohérente de domaine si pour toutes les variables de la portée de* C*, pour toutes les valeurs du domaine de ses variables, il existe un support de domaine.*

Définition 4 (Cohérence de bornes)**.** *Une contrainte* C *est dite cohérente de bornes si pour toutes les variables de la portée de* C*, la valeur minimum et la valeur maximum du domaine de ses variables ont un support de bornes.*

Dans l'exemple 2, un algorithme de filtrage devrait retirer la valeur 3 du domaine de X_3 pour établir la cohérence de domaine. Par contre, même si l'affectation $X_3 = 3$ ne fait partie d'aucune des solutions au problème, la contrainte est cohérente de bornes. Un algorithme établissant la cohérence de bornes ne filtre jamais plus qu'un algorithme établissant la cohérence de domaine. La complexité des algorithmes établissant la cohérence de bornes est généralement moindre que celle des algorithmes établissant la cohérence de domaine. Par exemple, il existe des containtes [3, 30] pour lesquelles établir la cohérence de domaine est NP-Difficile alors que l'on peut établir la cohérence de bornes en temps polynomial.

Un algorithme de filtrage, aussi appelé *propagateur*, est idempotent si une deuxième exécution sur les domaines filtrés lors d'une exécution précédente ne mène jamais à un filtrage supplémentaire. Un algorithme qui maintient la cohérence de domaine ou la cohérence de bornes

est nécessairement idempotent. Il existe des algorithmes de filtrage non-idempotent. Pour ces algorithmes, il est nécessaire de relancer l'exécution jusqu'à l'obtention d'un point fixe. À chaque exécution, de nouvelles valeurs incohérentes peuvent être décelées.

Une contrainte globale [34] est une contrainte dont l'arité est un paramètre. Par exemple la contrainte ALL-DIFFERENT($X_1, \ldots, X_n$) [35, 26] contraint n variables à prendre des valeurs distinctes, où n est un paramètre de la contrainte. Les contraintes globales peuvent être exprimées par des contraintes d'arité fixe. Par exemple, la contrainte ALL-DIFFERENT peut être exprimée par les contraintes d'inégalités $X_i \neq X_j$ pour tout $1 \leq i < j \leq n$. Les algorithmes de filtrage des contraintes globales filtrent généralement plus de valeurs que les contraintes d'arité fixe. Dans l'exemple 2, établir la cohérence de domaines sur les contraintes d'inégalités n'aurait pas retiré la valeur 3 du domaine de la variable X_3.

La contrainte INTER-DISTANCE($[X_1, \ldots, X_n], p$) [3, 33] est satisfaite lorsque chaque variable est séparée d'au moins p unités de toutes les autres, où p est un paramètre. Plus formellement, la contrainte est satisfaite lorsque $|X_i - X_j| \geq p$ pour tout $i < j$. Lorsque le paramètre $p = 1$, la contrainte INTER-DISTANCE en est réduite à une contrainte ALL-DIFFERENT. Cette contrainte est utilisée pour modéliser des problèmes d'affectation de fréquences radio. Les variables représentent les fréquences d'émission d'antennes dans une même région géographique. Afin d'éviter les interférences, la différence entre deux fréquences ne doit pas être plus petite que p hertz.

La contrainte MULTI-INTER-DISTANCE($[X_1, \ldots, X_n], m, p$) [30] est satisfaite lorsqu'au plus m variables sont affectées dans un même intervalle de longueur p, où p et m sont des paramètres. La contrainte est satisfaite lorsque $|\{i \mid X_i \in [t, t+p)\}| \leq m$ pour tout entier t. Lorsque $m = 1$, la contrainte MULTI-INTER-DISTANCE en est réduite à une contrainte INTER-DISTANCE. Cette contrainte est utile pour modéliser des problèmes dans lesquels on a recours à des machines identiques. Par exemple, un garage dispose de $m = 2$ passerelles élévatrices et requiert $p = 30$ minutes pour effectuer un changement de pneus. Les variables sont les heures de rendez-vous des clients. La contrainte s'assure qu'en tout temps t au plus 2 passerelles sont utilisées.

La contrainte GLOBALE DE CARDINALITÉ($[X_1, \ldots, X_n], D, l, u$) [36] est satisfaite lorsque chacune des valeurs $v \in D$ est assignée au moins l_v fois et au plus u_v fois à une variable de X. Formellement, $\forall v_i \in D : l_i \leq |\{j \mid x_j = v_i\}| \leq u_i$. Lorsque $l_i = 0 \; \forall l$ et $u_i = 1 \; \forall u$, la contrainte GLOBALE DE CARDINALITÉ devient une contrainte ALL-DIFFERENT. Cette contrainte peut servir de noyau pour modéliser des horaires d'employés. Les variables X représentent les quarts de travail à combler alors que les valeurs $v \in D$ personnifient des employés et leur disponibilité spécifique(l_v, u_v).

Les contraintes globales sont particulièrement utilisées dans les problèmes d'ordonnacement que nous présentons à la section suivante. Nous y introduisons aussi les contraintes globales CUMULATIVE et DISJONCTIVE spécialisées dans ce genre de problème.

1.2 Problèmes d'ordonnancement

Nous appelons *tâche* un travail élémentaire nécessitant l'usage d'une ressource. Un *problème d'ordonnancement* consiste à planifier la réalisation d'un ensemble de tâches en considérant les contraintes temporelles propre à chacune d'entre-elles et la disponibilité des ressources. Pour une tâche A nous définissons les paramètres qui suivent. Le *temps de sortie* est_A (earliest starting time), est le moment le plus tôt où la tâche peut commencer. L'*échéance* lct_A (latest completion time), est le moment où la tâche doit être obligatoirement terminée. Le *temps de traitement* p_A (processing time) est la quantité de temps nécessaire pour réaliser la tâche. La *hauteur* h_A est la quantité de ressources nécessaire au traitement de la tâche. La variable associée à la tâche A est son *temps de départ* S_A (starting time), qui marque le moment où elle débute. Nous nous intéressons aux problèmes d'ordonnancement dans lesquels les tâches ne peuvent pas être interrompues, c'est-à-dire que la tâche A consomme nécessairement h_A unités de ressource sur tout l'intervalle $[S_A\ ,\ S_A + p_A)$.

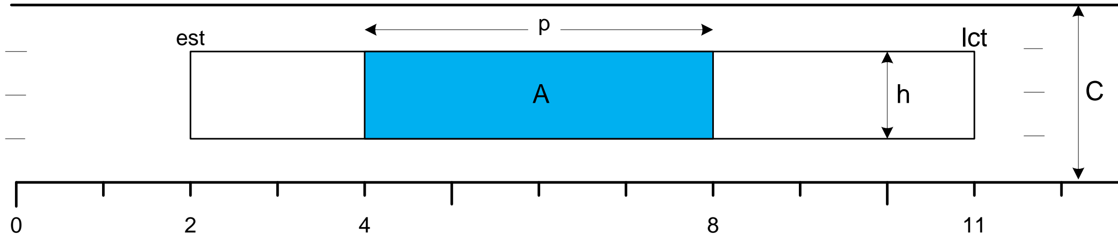


Figure 1.2: Schématisation de la tâche A dont la variable temps de départ S_A a été fixée à $S_A = 4$. Les paramètres de A sont son temps de sortie $est_A = 2$, son échéance $lct_A = 11$, son temps de traitement $p_A = 4$ et sa hauteur $h_A = 2$.

Pour une tâche i, nous notons par ect_i (earliest completion time), le temps minimum de terminaison.

$$ect_i = est_i + p_i \tag{1.9}$$

De même, nous notons lst_i (latest starting time), le temps maximum de départ de la tâche i.

$$lst_i = lct_i - p_i \tag{1.10}$$

Une solution au problème assigne un temps de départ S_i à toutes les tâches i tel que :

$$est_i \leq\ S_i\ \leq\ lst_i \tag{1.11}$$

La quantité de ressources C limite la hauteur totale des tâches pouvant être traitées simultanément. Pour un ensembles de tâches $\mathcal{I}$ nous avons :

$$\forall t, \forall i \in \mathcal{I} \sum_{\{i \mid S_i \leq t < S_i + p_i\}} h_i \leq\ C \tag{1.12}$$

Par exemple, sur la Figure 1.2, la quantité de ressources est $C = 4$.

Pour des raisons de commodité, la littérature emploie l'abbréviation CuSP (Cumulative Scheduling Problem) pour désigner un problème d'ordonnancement dans lequel plusieurs tâches peuvent s'exécuter conjointement.

La contrainte globale CUMULATIVE $([S_1, \ldots, S_n], [h_1, \ldots, h_n], [p_1, \ldots, p_n], C)$ [1] contraint les variables de temps de départ S_i pour toutes les tâches $i \in \mathcal{I}$. En tout temps t, la hauteur totale des tâches en cours d'exécution ne doit pas excéder la quantité de ressources disponible. La contrainte CUMULATIVE est satisfaite lorsque l'équation 1.12 est satisfaite pour $\text{dom}(S_i) = [est_i, lst_i]$. Si toutes les tâches $i \in \mathcal{I}$ sont de durée égale et de hauteur unitaire $(p_i = k,\ h_i = 1)$, la contrainte CUMULATIVE devient une contrainte MULTI-INTER-DISTANCE. Si par surcroît la quantité de ressource est de une unité ($C = 1$), elle devient une contrainte INTER-DISTANCE. Dans le cas ultime où toutes les tâches sont de durée et de hauteur unitaires de même que la quantité de ressources ($C = 1,\ h_i = 1,\ p_i = 1\ \forall i$), elle en est réduite à une contrainte ALL-DIFFERENT.

La contrainte DISJONCTIVE modélise les problèmes d'ordonnancement dans lesquels la ressource doit exécuter les tâches une à une. Elle est un cas particulier de la contrainte CUMULATIVE lorsque $h_i + h_j > C \quad \forall i \neq j$. La contrainte DISJONCTIVE est satisfaite si, en tout temps, pas plus d'une tâche n'est en cours d'exécution.

$$\bigwedge_{i \neq j} (S_i + p_i \leq S_j \vee S_j + p_j \leq S_i) \tag{1.13}$$

Dans un problème d'ordonnancement à plusieurs solutions, nous considérons optimales les solutions qui minimisent le temps total de traitement (makespan), c'est-à-dire le temps écoulé entre le début du traitement de la première tâche et la fin du traitement de la dernière. La figure 1.3 présente une solution où le temps total de traitement est minimal.

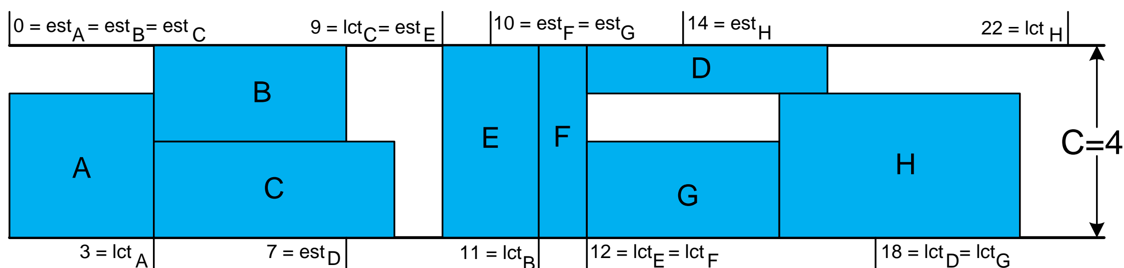


FIGURE 1.3: Un ensemble de tâches sur une ressource de capacité $C = 4$. Les hauteurs et durées sont à l'échelle et varient de 1 à 4 pour les hauteurs et de 1 à 5 pour les durées. Les valeurs de temps de sortie et d'échéance sont indiquées sur l'axe du haut et sur celui du bas.

Pour un CuSP, trouver une telle solution, ou simplement trouver un horaire réalisable, est NP-difficile. L'on peut alors procéder par une fouille avec retours arrière qui affecte des valeurs de temps de départ aux tâches et vérifie la faisabilité de l'horaire. Un algorithme de filtrage qui

ajuste les temps de sortie et les échéances des tâches contribue à réduire l'espace de recherche. Pour les contraintes globales régissant des tâches, il est d'usage de ne filtrer que les bornes des domaines. La méthode utilisée consiste à réduire le plus possible l'intervalle $[est_i, lct_i)$ de chacune des tâches (le domaine de la variable correspondante S_i est $[est_i, lst_i]$) en s'assurant de ne retirer que des valeurs qui n'ont pas de support de solution.

Pour y parvenir, il existe plusieurs méthodes ayant recours à un raisonnement énergétique qui s'appuie sur des notions de précédence, d'énergie, de slack et d'enveloppe. Nous les définissons dans les paragraphes qui suivent.

L'*énergie* est le produit entre le temps de traitement d'une tâche i et sa hauteur.

$$e_i = h_i \, p_i \tag{1.14}$$

Par exemple, pour un travail qui nécessite 3 ouvriers pendant 2 heures, nous disons qu'il requiert 6 *hommes* $\times$ *heure*. Pour une tâche régie par une contrainte Cumulative, nous disons que son énergie est de 6 unités.

L'*enveloppe* est une mesure d'un certain potentiel d'énergie consommé jusqu'à l'achèvement d'une tâche. Son évaluation prend en compte toutes les unités *ressource* $\times$ *temps* du passé sans égard à leur utilisation effective, les unités non-utilisées ne pouvant être récupérées.

$$Env_i = C \, est_i \; + \; e_i \tag{1.15}$$

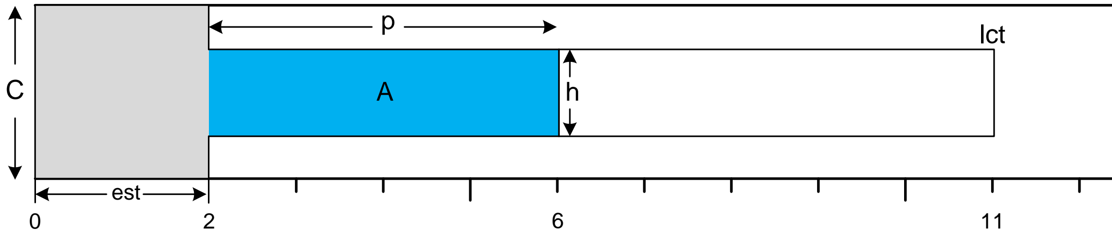


Figure 1.4: Représentation géométrique de l'enveloppe de la tâche A.

La figure 1.4 schématise l'enveloppe de la tâche A de la figure 1.2. L'évaluation de l'enveloppe considère la capacité totale de la ressource antérieurement au temps de sortie de la tâche A (le rectangle gris) et l'énergie de la tâche A (le rectangle bleu).

Les valeurs et paramètres déterminant une tâche peuvent s'étendre à l'ensemble de toutes les tâches ou à tous sous-ensembles de celui-ci. $\forall \, \Omega \; \subseteq \; \mathcal{I}$ nous avons pour le temps de sortie :

$$est_\Omega = \min_{i \, \in \, \Omega} \; est_i \tag{1.16}$$

pour l'échéance :

$$lct_\Omega = \max_{i \, \in \, \Omega} \; lct_i \tag{1.17}$$

pour le temps minimum de terminaison :

$$ect_{\Omega} = \max_{i \in \Omega} \ ect_i \tag{1.18}$$

pour le temps maximum de départ :

$$lst_{\Omega} = \min_{i \in \Omega} \ lst_i \tag{1.19}$$

pour l'énergie :

$$e_{\Omega} = \sum_{i \in \Omega} e_i \tag{1.20}$$

et pour l'enveloppe :

$$Env_{\Omega} = \max_{\Theta \subseteq \Omega} \ (\ C \ est_{\Theta} + e_{\Theta} \ ) \tag{1.21}$$

Pour un sous-ensemble de tâches $\Omega \subseteq \mathcal{I}$ nous appellons *premier temps de terminaison* et notons $minEct_{\Omega}$, le plus petit temps minimum de terminaison.

$$minEct_{\Omega} = \min_{i \in \Omega} \ ect_i \tag{1.22}$$

Symétriquement, nous appellons *dernier temps de départ* et notons $maxLst_{\Omega}$, le plus grand temps maximum de départ parmi les tâches de Ω.

$$maxLst_{\Omega} = \max_{i \in \Omega} \ lst_i \tag{1.23}$$

Pour un intervalle donné, $[est_{\Omega}, lct_{\Omega})$, le *slack* ($Sl_{\Omega}$) est la quantité d'énergie disponible à l'intérieur de l'intervalle une fois le traitement de toutes les tâches $i \in \Omega$ effectué.

$$Sl_{\Omega} = C(lct_{\Omega} - est_{\Omega}) - e_{\Omega} \tag{1.24}$$

Un CuSP est dit *énergétiquement cohérent* (E-Feasible) s'il ne contient aucun intervalle de slack négatif.

Une *précédence* est une relation d'ordonnancement entre une tâche et un ensemble de tâches. On écrit $\Omega \prec A$ si la tâche A ne peut être complétée avant que toutes les tâches de Ω ne soient complétées. C'est entre autres la découverte de précédences qui permet de déduire l'ajustement des bornes des tâches. Sur la figure 1.3, on peut voir plusieurs de ces relations. Les tâches B et C sont précédées par la tâche A. On note $A \prec B$ et $A \prec C$ ou $A \prec \{B, C\}$. Pour la tâche D on note $\{A, B, C, E, F\} \prec D$.

Pour une tâche i, nous appelons *précédence naturelle* la relation d'ordonnancement triviale qui relie la tâche i à l'ensemble des tâches dont l'échéance n'est pas plus grand que son temps minimum de terminaison.

$$\forall i, \forall j \neq i, \{j \mid lct_j \leq ect_i\} \prec i \tag{1.25}$$

Chapitre 2

Filtrage de la contrainte Cumulative

La contrainte globale CUMULATIVE $([S_1, \ldots, S_n], [h_1, \ldots, h_n], [p_1, \ldots, p_n], C)$ encode le cas général des problèmes d'ordonnancement. Elle contraint un ensemble de tâches de hauteurs quelconques et de durées quelconques à s'exécuter sans consommer au-delà d'une quantité de ressources C uniforme [1]. Sa formulation souple dans laquelle le vecteur S représente les varaiables temps de départ et les vecteurs constants h et p respectivement les hauteurs et les durées des tâches, permet de modéliser un large éventail de problèmes. Par exemple, l'usinage de biens à produire qui sont des tâches et les machines qui les produisent des ressources, la gestion de personnel dans laquelle les fonctions à combler sont des tâches et les travailleurs qui les accomplissent des ressources. On peut l'utiliser pour résoudre des problèmes d'optimisation informatique, les instructions étant des tâches et les micro-processeurs des ressources.

Son utilité en modélisation ne se limite pas à l'ordonnancement de tâches à proprement parler. La disponibilité régulière de ses ressources dans le temps, assimilable à une forme rectangulaire, permet de traiter efficacement des problèmes géométriques comme le découpage ou l'emballage dans lesquels l'on veut minimiser les pertes de la ressource disponible. Il n'est pas rare qu'elle fasse partie de la résolution de problèmes originaux s'apparentant au bien connus *BinPacking* ou *Knapsack* qui consistent à remplir des contenants de façon optimale. Elle s'avère aussi utile en gestion de l'énergie où l'optimisation peut consister à consommer le plus régulièrement possible de la ressource.

Il va de soi que la formulation des problèmes pratiques est plus élaborée que les problèmes théoriques sur lesquels la modélisation s'appuie. C'est-à-dire qu'un problème fait généralement face à des contraintes extérieures. Par exemple, une tâche ou un ensemble de tâches peut devoir s'exécuter avant une autre. La disponibilité des ressources ou les critères d'optimalité peuvent varier dans le temps. On peut avoir à se plier aux nombreux caprices d'une convention collective de travail ou tout simplement ajuster les travaux en fonction des prévisions météorologiques

1. Pour modéliser une quantité de ressource non-uniforme, on ajoute au problème des tâches supplémentaires avec pour caractéristique $est + p = lct$. Ces ajouts font en sorte de diminuer la quantité de ressource disponible au traitement des vraies tâches sur l'intervalle $[est, lct)$.

ou de la durée du jour qui varie avec les saisons. La programmation par contraintes se veut un univers de prédilection pour ce genre de problèmes. En effet, il lui est facile de modéliser les particularités extérieures par l'ajout de contraintes supplémentaires sans avoir à modifier le noyau du modèle initial et les algorithmes de filtrage fournis par le solveur de contraintes. Dans cette optique, la très souple contrainte CUMULATIVE sert de noyau à la résolution d'une multitude de problèmes en optimisation combinatoire. Par conséquent, elle est l'objet d'une recherche intense dans la communauté de programmation par contraintes.

Aggoun et Baldiceanu [1] introduisent la contrainte en 1993. Leur article traite de problèmes géométriques (le placement de rectangles quelconques, le problème du carré parfait...) issus du domaine de la recherche opérationnelle. Il y est montré que la programmation par contraintes avec la contrainte CUMULATIVE comme noyau donne de très bons résultats. Leur article met aussi en relief la grande versatilité de cette toute nouvelle contrainte. Baptiste, Le Pape et Nuijten [5, 6] procèdent à une étude en profondeur de la contrainte. La CUMULATIVE s'inscrivant dans leurs travaux en ordonnancement comme la suite logique de l'étude de la contrainte DISJONCTIVE, les règles de filtrage qu'ils établissent pour la CUMULATIVE sont généralement l'extension de celles utilisées pour la DISJONCTIVE. Baptiste et al. montrent qu'obtenir la cohérence de bornes pour la CUMULATIVE est NP-difficile. Il en serait de même si l'on disposait de plusieurs ressources unitaires (C=m>1). Dans cette optique, les auteurs présentent la CUMULATIVE comme la conjugaison de deux problèmes NP-difficile. Pour Baptiste et Le Pape [7], la propagation de la contrainte CUMULATIVE n'est réellement efficace que dans les problèmes hautement disjonctifs ou hautement cumulatifs [2]. Depuis, les expérimentations leur donnent raison. Toutefois, la programmation par contraintes est un travail d'équipe. Chacune des contraintes d'un modèle contribue à réduire l'espace de recherche [3] et aucun filtrage ne doit être négligé.

Il existe plusieurs règles qui permettent de filtrer les bornes des tâches de la contrainte CUMULATIVE. Les propagateurs issus de ces règles sont tous non-idempotents et nécessitent de relancer leur exécution jusqu'à l'obtention d'un point fixe. La prochaine section présente un survol des règles de filtrage et des propagateurs actuels qui les exploitent.

2. Hautement disjonctif réfère à des CuSP dans lesquels la hauteur des tâches permet difficilement d'en exécuter deux à la fois. Hautement cumulatif qualifie les CuSP où l'on retrouve beaucoup de tâches dans des intervalles de temps relativement courts.

3. Considérant que les problèmes que la programmation par contraintes permet d'optimiser sont généralement NP-difficile, que celui qui détient une meilleure approche lève la main !

2.1 Background

La règle du *Overload Checking* vérifie que l'énergie requise pour le traitement d'un sous-ensemble de tâches Ω n'est pas supérieure à l'énergie totale que la ressource peut dispenser.

$$\forall\, \Omega \subseteq \mathcal{I},\; e_\Omega \leq C(lct_\Omega - est_\Omega) \tag{2.1}$$

Il s'agit en fait d'un test qui vérifie la cohérence énergétique (*E-Feasibility*, équation 1.24) d'un CuSP [4]. Nuijten [29] teste en temps quadratique tous les intervalles $[est_i, lct_j)$ pour $i, j \in \mathcal{I}$. Wolf et Schrader [49] mettent à profit un arbre binaire et obtiennent le même résultat en $O(n \log n)$. Leur algorithme ne teste pas tous les intervalles mais seulement les *intervalles critiques* c'est-à-dire ceux susceptibles de mener à la découverte de l'infaisabilité [5]. Vilìm [47] produit un propagateur qui s'appuie sur la règle du Overload Checking. Son algorithme est, en quelque sorte, une transformation de [49] pour obtenir un propagateur. Pour respecter la complexité de l'algorithme dont il s'inspire, Vilìm utilise une approximation de l'évaluation de l'énergie et le filtrage résultant est faible. Il est probable que réduire progressivement le domaine de la variable associée à la tâche responsable du est_Ω et de celle responsable du lct_Ω jusqu'à ce que l'Overload soit détecté pourrait aussi mener à un bon filtrage. [6] L'approche n'a toutefois jamais été tentée. La véritable utilité de l'Overload Checking en propagation se fait sentir dans les deux règles qui suivent.

La régle du *Edge-Finding* [7] consiste à déterminer si la quantité de ressources est suffisante pour débuter une tâche i au temps est_i conjointement à un sous-ensemble Ω de tâches. Sinon, la tâche est précédée par toutes les tâches du sous-ensemble Ω.

$$\forall \Omega, \forall i \notin \Omega \wedge est_i \geq est_\Omega,\; C(lct_\Omega - est_\Omega) < e_\Omega + e_i \Rightarrow \Omega \prec i \tag{2.2}$$

Vilìm [46] trouve toutes les précédences en $O(n \log n)$. L'ajustement du temps de sortie nécessite d'évaluer l'énergie disponible pour la tâche i dans l'intervalle $[est_\Omega, lct_\Omega)$. Nuijten [29] propose la technique suivante. Pour un sous-ensemble optimal $\Theta \subseteq \Omega$, elle répartit l'énergie e_Θ en deux blocs. Le premier bloc, de hauteur $C - h_i$, consomme uniformément de la ressource sur tout l'intervalle $[est_\Theta, lct_\Theta)$. Le second bloc, de hauteur h_i, débute au début de l'intervalle

4. Les règles de filtrage Edge-Finding, Extended-Edge-Finding et NotFirst/NotLast dont il est question plus loin, ne sont valides que pour des CuSP énergétiquement cohérents. Par conséquent, il est généralement nécessaire de tester la cohérence énergétique avant d'exécuter les propagateurs qui les appliquent.

5. Leur structure de données en arbre binaire, appelé arbre cumulatif, permet de déterminer l'intervalle critique en $O(\log n)$ pour chacune des bornes testées. Puisqu'il y a un maximum de $2n$ bornes, la complexité générale de l'algorithme est de $O(n \log n)$. Nous présentons l'arbre cumulatif à la Section 2.2 et en détaillons toute la mécanique.

6. Une approche similaire est utilisée dans [30] pour obtenier le cohérence de bornes pour la contrainte Multi-Inter-Distance.

7. Edge-Finding fait référence à la découverte d'arcs (edges) dans un graphe d'ordonnancement où les sommets représentent des tâches et les arcs des relations de précédence qui forcent un certain ordre de traitement des tâches.

et consomme la portion restante de l'énergie e_Θ. La tâche i peut débuter lorsque le deuxième bloc se termine.

$$\Omega \prec i \Rightarrow est_i' = \max_{\Theta \subseteq \Omega} est_\Theta + \left\lceil \frac{e_\Theta - (C - h_i)(lct_\Theta - est_\Theta)}{h_i} \right\rceil \tag{2.3}$$

Vilìm [46] parvient à évaluer tous les ajustements en $O(kn \log n)$ où $k = |\{h_i \mid i \in \mathcal{I}\}|$ est le nombre de hauteurs de tâche distinctes. Son algorithme est présenté en détails à la Section 2.2. Kameugne et al. [21] procèdent en une seule étape. Leur algorithme détecte en temps quadratique toutes les tâches dont la règle du *Edge-Finding* permet d'ajuster le temps de sortie et les ajuste immédiatement. Bien qu'il n'applique pas nécessairement le meilleur filtrage [8], l'algorithme parvient au même point fixe au terme de ses exécutions. Son astuce réside dans l'utilisation, pour chaque échéance lct, de deux intervalles optimaux susceptibles de mener à un ajustement : l'intervalle de slack minimum et l'intervalle de densité maximum. L'algorithme ajuste la tâche, le cas échéant, avec l'intervalle qui produit l'ajustement maximal. L'intervalle de densité maximale est celui qui satisfait l'équation 2.3 avec $lct_\Theta = lct$. L'intervalle de slack minimum Θ est celui dont la borne inférieure est_Θ cerne l'intervelle de slack minimum avec $lct_\Theta = lct$. L'ajustement qu'il effectue équivaut à appliquer tout le slack au traitement de la tâche i à ajuster.

$$est_i' = \max(est_i\ ,\ lct_\Theta - \left\lfloor \frac{Sl_\Theta}{h_i} \right\rfloor) \tag{2.4}$$

Si le temps de sortie de la tâche i est inférieur au temps de sortie du sous-ensemble Ω et que la quantité de ressources est insuffisante pour traiter conjointement Ω et la tâche i lorsque celle-ci débute à son temps de sortie, la règle porte le nom de *Extended-Edge-Finding*. Elle s'énonce ainsi : $\forall \Omega, \forall i \notin \Omega$, si $est_i < est_\Omega < ect_i$ alors

$$e_\Omega + h_i(\min(ect_i, lct_\Omega) - est_\Omega) > C(lct_\Omega - est_\Omega) \Rightarrow \Omega \prec i \tag{2.5}$$

Mercier et Van Hentenryck [28] exploitent cette règle et produisent un algorithme qui filtre en $O(kn^2)$ où $k = |\{h_i \mid i \in \mathcal{I}\}|$ est le nombre de hauteurs de tâche distinctes. L'algorithme itère sur tous les intervalles $[est_i, lct_j)$ et détermine, pour toutes les k hauteurs de tâche du problème, la longueur du *segment de longueur maximale* [9] qui peut s'exécuter conjointement aux tâches confinées à l'intervalle. Pour toute tâche qui requiert un temps d'exécution dans l'intervalle supérieur au segment de longueur maximale, l'algorithme en ajuste le temps de sortie si elle chevauche l'intervalle $[est_i, lct_j)$. L'ajustement appliqué fait en sorte que la tâche requiert un temps d'exécution dans l'intervalle égal à la longueur du segment de longueur maximale. La figure 2.1 illustre les deux règles.

8. Voir [21] section Abstract.
9. La technique s'apparente à la notion d'enveloppe partielle présentée à la section 2.2.3.

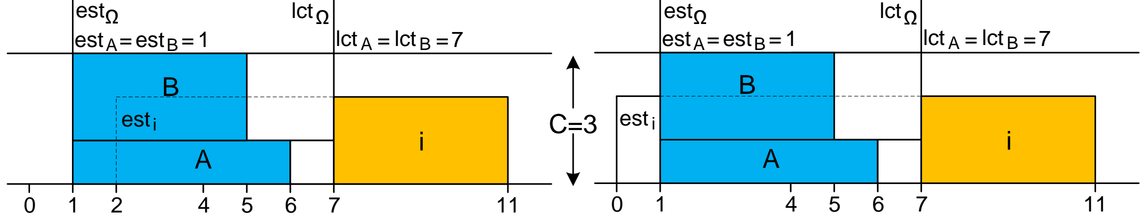


FIGURE 2.1: Illustration de deux situations de précédence en vertu de la règle *Edge-Finding* à gauche et de la règle *Extended-Edge-Finding* à droite.

La figure 2.1 montre l'intervalle $[est_\Omega, lct_\Omega) \subset [1, 7)$ disposant de $3 \times 6 = 18$ unités d'énergie. Les tâches A et B qui y sont confinées consomment un total de 13 unités d'énergie ($e_A = 5$, $e_B = 8$). À gauche, la tâche i ($e_i = 8$) ne peut s'exécuter entièrement dans l'intervalle $[est_\Omega, lct_\Omega)$ provoquant la détection de la précédence $\Omega \prec i$ par la règle du Edge-Finding. Dans le portion de droite, le temps de sortie de la tâche i est fixé à 0 ($est_i = 0$). Lorsque la tâche i débute à son temps de sortie, elle consomme 6 unités d'énergie dans l'intervalle $[est_\Omega, lct_\Omega)$. L'énergie disponible dans l'intervalle étant insuffisant, la règle du Extended-Edge-Finding détecte la précédence $\Omega \prec i$. Dans les deux cas de précédence, le temps de sortie de la tâche i est ajusté à 5 ($est_i' = 5$).

La règle du *Time-Tabling* consiste à déterminer la quantité de ressources qui doit nécessairement être allouée au traitement d'une ou de plusieurs tâches spécifiques. Pour une tâche i qui satisfait la relation $lst_i < ect_i$, l'intervalle $[lst_i, ect_i)$ détermine la *partie fixe* de la tâche i, notée pf_i. C'est-à-dire que la tâche i consomme nécessairement h_i quantité de ressources sur l'intervalle $[lst_i, ect_i)$. Soit $f(\Omega, t)$, la hauteur totale de toutes les parties fixes des tâches du sous-ensemble Ω au temps t et soit $f(\Omega, [a, b))$, le *bloc cumulé* dans l'intervalle $[a, b)$ des parties fixes des tâches du sous-ensemble Ω.

$$f(\Omega, t) = \sum_{i \in \Omega | t \in [lst_i, ect_i)} h_i \qquad\qquad f(\Omega, [a, b)) = \sum_{t \in [a, b)} f(\Omega, t) \tag{2.6}$$

Si la tâche i ne peut pas terminer avant le temps t et que sa hauteur additionnée à la hauteur cumulative des parties fixes de toutes les autres tâches au temps t excède la quantité de ressources, alors la tâche i doit débuter après t.

$$ect_i > t \geq est_i \wedge C < h_i + f(\Omega \setminus \{i\}, t) \Rightarrow est_i' > t \tag{2.7}$$

Baptiste et al. [4] parlent d'un test de faisabilité et d'un algorithme de filtrage qui s'exécutent respectivement en $O(n \log n)$ et en $O(n^2)$ [10]. Beldiceanu et Carlsson [9] introduisent une méthode appelée *Sweep Algorithm* qui tient à jour la hauteur du bloc cumulé. Leur algorithme itère sur tous les temps critiques de est_I à lct_I et procède à la gestion d'*évènements* dont l'un

10. Nous n'avons pas réussi à nous procurer le rapport technique. Cependant nous pensons que le test de faisabilité est similaire à l'algorithme 5 présenté plus loin.

d'eux est l'ajustement d'une tâche. Sa complexité est de $O(n^2)$ pour un ressource unique bien que le nombre de temps critiques soit borné par $O(n)$ [11]. Les auteurs s'adjoignent Letort [24] et améliorent leur méthode qu'ils appellent *dynamic sweep*. L'algorithme obtenu peut itérer jusqu'à l'obtention d'un point fixe ou fournir rapidement une très bonne approximation. L'algorithme amélioré permet entre autres de traiter de très gros ensembles de tâches. Bien que sa complexité soit de $O(n^2 \log n)$, il supplante dans les tests la version précédente pour tous les problèmes mis en test. Beldiceanu et al. [8] propose une façon originale d'exploiter la règle du Time-Tabling en la mettant en relation avec le problème du placement de rectangles quelconques [12]. Sa méthode se distingue en ce qu'elle considère les unités d'énergie de la ressource nécessairement inutilisables en raison de la configuration des tâches. La figure 2.2 illustre la règle du Time-Tabling.

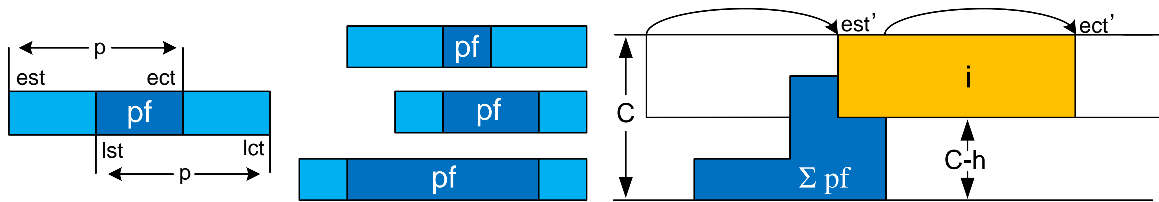


FIGURE 2.2: Illustration de la règle du Time-Tabling. À gauche, les caractéristiques d'une tâche qui possède une partie fixe. Au centre, toutes les tâches avec partie fixe du problème. On somme les parties fixes pour former le bloc cumulé de la portion de droite de la figure. On constate que la ressource est insuffisante pour traiter le bloc cumulé (les parties fixes) et la tâche i en entier lorsqu'elle débute à son temps de sortie. La règle du Time-Tabling ajuste le temps de sortie de la tâche i au moment le plus tôt où la ressource suffit à traiter la tâche i conjointement avec le bloc cumulé.

Vilím [48] introduit l'idée d'augmenter le filtrage du Edge-Finding en tenant compte de l'énergie des parties fixes. Il appelle cette nouvelle règle *Time-Table-Edge-Finding*. Soit e^f_Ω l'énergie des tâches du sous-ensemble Ω additionnée à l'énergie du bloc cumulé dans l'intervalle $[est_\Omega, lct_\Omega)$ des parties fixes des tâches non-confinées à l'intervalle.

$$e^f_\Omega = e_\Omega + f(\mathcal{I} \setminus \Omega, [est_\Omega, lct_\Omega)) \tag{2.8}$$

En substituant 2.8 dans la règle de détection du Edge-Finding (équation 2.2) et en soustrayant, le cas échéant, l'énergie fixe de la tâche i dans l'intervalle $[est_\Omega, lct_\Omega)$, on obtient la règle de détection du Time-Table-Edge-Finding :

$$\forall \Omega, \forall i \notin \Omega,\ C(lct_\Omega - est_\Omega) < e^f_\Omega + e_i - f(\{i\}, [est_\Omega, lct_\Omega)) \Rightarrow \Omega \prec i \tag{2.9}$$

11. Mis à part les tris, l'algorithme effectue $O(n)$ itérations. Sa complexité déclarée de $O(n \log n + np)$ où $p \leq n$ est attribuable au fait que l'algorithme peut théoriquement ajuster la même tâche jusqu'à n fois. Dans la pratique, cet algorithme s'avère aussi performant qu'une version améliorée (voir la note 35) de l'algorithme 6 présenté à la section 2.3 qui s'exécute en $O(n \log \log n)$.

12. Le problème consiste à positionner des rectangles de dimension quelconque de manière à utiliser une surface rectangulaire totale de dimension minimale.

De façon similaire, en substituant 2.8 dans la règle de détection du Extended-Edge-Finding (équation 2.5), on obtient la règle de détection du Time-Table-Extended-Edge-Finding qui s'énonce ainsi : $\forall \Omega, \forall i \notin \Omega$, si $est_i < est_\Omega < ect_i$ alors

$$e^f_\Omega + h_i(\min(ect_i, lct_\Omega) - est_\Omega) - f(\{i\}, [est_\Omega, lct_\Omega)) > C(lct_\Omega - est_\Omega) \Rightarrow \Omega \prec i \qquad (2.10)$$

L'algorithme de Vilím [48] itère sur tous les $O(n^2)$ intervalles. Il considère quatre situations d'interaction distinctes entre les tâches i et les intervalles testés [13]. La complexité déclarée de son algorithme est de $O(n^2)$. Toutefois, pour le pire ensemble de tâches $\mathcal{I}$, il nécessite $n-2$ itérations [14] pour filtrer toutes les tâches qui peuvent être ajustées en vertu de la règle du Time-Table-Edge-Finding. Ce qui lui confère une complexité de $O(n^3)$. Il serait néanmoins facile de traiter toutes les tâches dans une seule itération en ajoutant 3 files prioritaires de complexité $O(\log n)$. Nous affirmons donc que la complexité réelle de son algorithme est de $O(n^2 \log n)$. Shutt et al. [38] obtiennent des résultats impressionnants sur le banc d'essais SMRCPSP [22] en combinant l'algorithme de Vilím et l'utilisation de *NOGOODS* [15]. la Figure 2.3 illustre l'efficacité de la règle combinée.

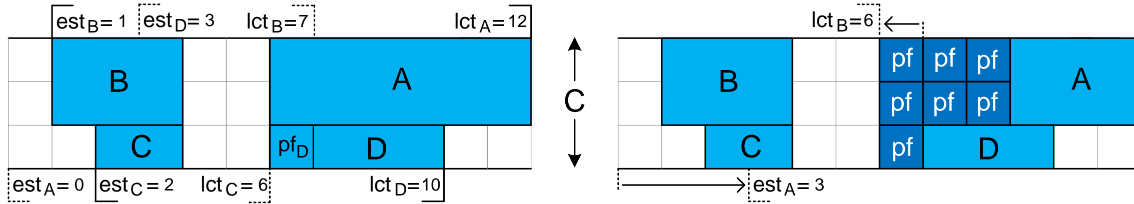


FIGURE 2.3: On peut voir à gauche un CuSP à quatre tâches. Les temps de sortie et de terminaison sont indiqués sur l'axe du haut et celui du bas. Les temps de traitement et les hauteurs des tâches sont à l'échelle. En ne tenant pas compte de la partie fixe de la tâche D, ni la règle du Extended-Edge-Finding pas plus que celle du Time-Tabling ne parviennent à déduire un ajustement. En considérant l'unité d'énergie unique de la partie fixe de la tâche D, la règle du Time-Table-Extended-Edge-Finding ajuste la borne inférieure de la tâche A au temps $t = 3$. L'ajustement a pour effet de créer une nouvelle partie fixe de 6 unités d'énergie qui entraîne à son tour l'ajustement de la borne supérieure de la tâche B au temps $t = 6$ par la règle du Time-Tabling. La partie de droite montre l'état du CuSP au point fixe.

13. Pour chacune des deux règles, il distingue les cas où les tâches i doivent se terminer à l'extérieur ou à l'intérieur de l'intervalle testé lorsqu'elles débutent à leur temps de sortie.

14. Voir [48] section 2.5.

15. Cette technique consiste à déterminer deux ou plusieurs couples variable-assignation qui sont incohérents entre eux plutôt que de compléter les calculs qui mèneraient au filtrage des incohérences du domaine de la variable fautive. Ultimement, le solveur parvient à filtrer un ou plusieurs domaines en combinant et précisant les relations d'incohérence découvertes.

Baptiste, Le Pape et Nuijten [5][6] énoncent les conditions nécessaires au *Energetic Reasoning* qui domine les règles du Time-Tabling du Edge-Finding et du Extended-Edge-Finding [16]. Soit une tâche i et un intervalle $[t_1, t_2)$, la propriété gauche-droite détermine la quantité minimum d'énergie nécessaire à la tâche i dans l'intervalle $[t_1, t_2)$.

$$gd_i[t_1, t_2) = h_i \min(t_2 - t_1, p_i^+(t_1), p_i^-(t_2)) \quad \text{où} \tag{2.11}$$

$$p_i^+(t_1) = \max(0, p_i - \max(0, t_1 - est_i))$$
$$p_i^-(t_2) = \max(0, p_i - \max(0, lct_i - t_2))$$

La propriété gauche-droite s'explique géométriquement. On considère le temps d'exécution qui s'effectue dans l'intervalle $[t_1, t_2)$ testé pour les deux positionnements limites de la tâche. Soit lorsque la tâche débute le plus tôt possible (p^+) ou lorsqu'elle débute le plus tard possible (p^-). Le plus petit des deux temps d'exécution est multiplié par la hauteur de la tâche et donne l'énergie minimum. La figure 2.4 illustre la propriété.

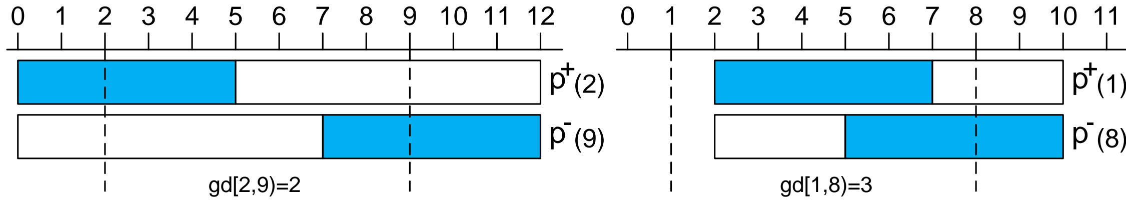


Figure 2.4: Deux illustrations de la propriété gauche-droite. La première partie montre une tâche de hauteur $h = 1$, de temps de traitement $p = 5$, de temps de sortie $est = 0$ et de temps maximum de départ $lst = 7$. Lorsque la tâche débute au temps $t = 0$ elle consomme 3 unités d'énergie dans l'intervalle $[2, 9)$ alors qu'elle en consomme 2 lorsqu'elle débute au temps $t = 7$. Sa propriété gauche-droite est le minimum des deux valeurs, $gd[2, 9) = 2$. La deuxième partie montre une tâche partiellement incluse dans l'intervalle testé $[1, 8)$. Sa propriété gauche-droite nous est donnée par son positionnement tardif $p^- = 3$. La hauteur de la tâche étant $h = 1$, sa propriété gauche-droite dans l'intervalle $[1, 8)$ est de 3 unités d'énergie, $gd[1, 8) = 3$.

Considérons les 3 ensembles suivants composés de marqueurs de temps.

$$Q_1 = \{est_i\} \cup \{lst_i\} \cup \{ect_i\} \text{ pour } 1 \leq i \leq n$$
$$Q_2 = \{lct_i\} \cup \{ect_i\} \cup \{lst_i\} \text{ pour } 1 \leq i \leq n$$
$$Q(t) = \{est_i + lct_i - t\} \text{ pour } 1 \leq i \leq n$$

16. La dominance pour une règle de filtrage signifie qu'elle détecte et ajuste toujours également ou mieux que la règle qu'elle domine. Dans le cas qui nous occupe, un algorithme de filtrage qui propage la Cumulative selon la règle du Energetic Reasoning domine tous les propagateurs de la Cumulative qui font appel aux règles Time-Tabling, Edge-Finding, Extended-Edge-Finding ou à leurs combinaisons, notamment le Time-Table-Extended-Edge-Finding.

Soit deux marqueurs de temps $a < b$, les deux alternatives :

$$a \in Q_1, b \in Q_2$$
$$a \in Q_1 \cup Q_2, b \in Q(a)$$

constituent toutes les combinaisons d'intervalles de la forme $[a, b)$ significatifs pour le Energetic Reasoning. C'est-à-dire que la condition nécessaire pour qu'un CuSP dispose d'un horaire réalisable au sens du Energetic Reasoning, est qu'aucun de ces intervalles ait un slack négatif ($Sl_{[a,b)} \geq 0 \quad \forall\ a, b$) [17]. Les auteurs [6] proposent un algorithme qui s'exécute en $O(n^2)$ pour évaluer l'énergie minimum requise dans tous les intervalles significatifs au sens du Energetic Reasoning. Sans produire d'algorithme de filtrage, ils proposent de comparer chacune des tâches avec les intervalles précédemment calculés. Si la tâche ne peut s'exécuter entièrement dans l'intervalle testé, son temps de sortie est ajusté en fonction de l'énergie disponible. Un tel algorithme appliquerait la règle du Energetic Reasoning en $O(n^3)$.

La règle du *Not-Last* est en quelques sortes le complément de la règle Edge-Finding. Elle consiste à déterminer si une tâche i peut terminer la dernière parmi un sous-ensemble de tâches. Sinon, on en déduit qu'au moins une tâche du sous-ensemble doit terminer avant que la tâche i ne se termine. $\forall \Omega, \forall i \notin \Omega$, si $lct_i > maxLst_\Omega$ alors :

$$e_\Omega + h_i(lct_\Omega - \max(lst_i, est_\Omega)) > C(lct_\Omega - est_\Omega) \Rightarrow S_i + p_i \leq maxLst_\Omega \tag{2.12}$$

Schutt, Wolf et Schrader [37] transposent et améliorent un algorithme écrit plus tôt par Nuijten [29] qui filtre en vertu de la règle *Not-First*. Ils obtiennent un algorithme de filtrage pour la règle Not-Last qui s'exécute en $O(n^3 \log n)$. Leur principale contribution est l'extension de la règle aux tâches i qui débutent avant l'intervalle borné par le sous-ensemble Ω [18]. Schutt et Wolf [39] utilisent une technique s'apparentant à une décomposition qu'ils nomment *pseudo-tasks*. La technique consiste à constituer des tâches virtuelles associées aux tâches du problème initial et permettant de déterminer sur quelles seules tâches le règle NOT-LAST peut déduire un ajustement. Leur algorithme s'exécute en $O(n^2 \log n)$ quoiqu'il n'applique pas toujours le meilleur ajustement [19] en vertu de la règle. Dans le pire cas, il peut nécessiter jusqu'à n

17. Cette condition est plus restrictive que la cohérence énergétique (voir 1.24) testée par la règle du OverloadChecking. Baptiste et al. [5][6] définissent trois formes de relaxation pour le CuSP. Elles sont, de la plus forte à la plus faible, la *leftshift/rightshift*, la *partially elastic* et la *fully elastic*. La règle du Energetic Reasoning découle de la relaxation la plus forte alors que la règle du OverloadChecking appartient à la plus faible. Par relaxation, on entend un compromis dans la formulation du problème. Les relaxations partially elastic et fully elastic sont deux manières de ne pas restreindre les tâches à s'exécuter sans interruption. Carlier et Pinson [13] présentent d'autres formes de relaxation du CuSP.

18. Avant leur article, il était d'usage d'appliquer la règle sur les tâches i tel que $est_i > est_\Omega$. Cette condition faisait d'ailleurs partie de l'énoncé de la règle. Les modifications apportées à l'algorithme original permettent de traiter le cas où $est_i \leq est_\Omega$ sans en compromettre la complexité. Cette extension n'apporte toutefois pas de filtrage supplémentaire à la contrainte CUMULATIVE. Les tâches i tel que $est_i \leq est_\Omega$ qui sont en situation de détection par la règle Not-Last sont nécessairement en situation de détection par la règle Extended-Edge-Finding dont le plus petit ajustement fait en sorte de placer la tache i en situation de détection par la règle Not-Last avec $est_i > est_\Omega$. Quoi qu'il en soit, tous les algorithmes publiés depuis adoptent la version étendue de la règle.

19. Voir [39] p.456

itérations pour effectuer les ajustements optimaux. Quant aux tâches virtuelles, leur nombre est borné supérieurement par $2n - 1$.

La règle du Not-First est l'exacte symétrie de la règle du Not-Last. Elle consiste à déterminer si une tâche i peut s'exécuter la première parmi un sous-ensemble de tâches. Sinon, on en déduit qu'au moins une tâche du sous-ensemble doit débuter avant que la tâche i ne débute. $\forall \Omega, \forall i \notin \Omega$, si $est_i < minEct_\Omega$ alors :

$$e_\Omega + h_i(\min(ect_i, lct_\Omega) - est_\Omega) > C(lct_\Omega - est_\Omega) \Rightarrow S_i \geq minEct_\Omega \tag{2.13}$$

Les règles Not-First et Not-Last auxquelles on réfère généralement par l'appellation commune *Not-First/Not-Last*, jouent un rôle essentiel dans la propagation de la contrainte CUMULATIVE. Elles détectent les ordonnancements maladroits qui impliquent un gaspillage d'énergie de la ressource. Pour cette raison, elles ne sont pas dominées par le Energetic Reasoning. La figure 2.5 illustre les particularités de la règle.

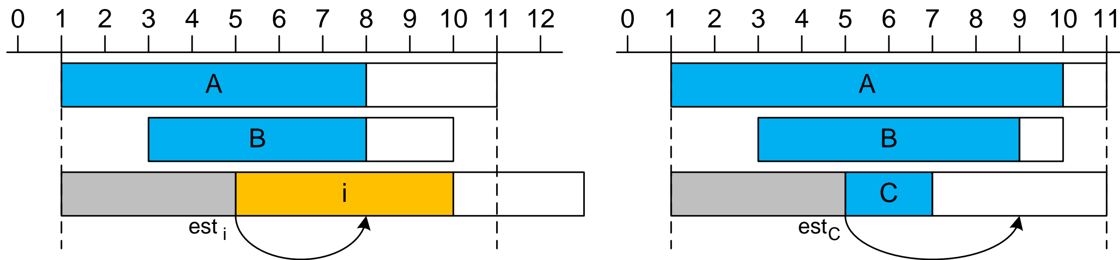


FIGURE 2.5: Trois tâches de hauteur $h = 1$ sur une ressource de capacité $C = 2$. L'intervalle critique $[1, 11)$ dispose donc de 20 unités d'énergie. À gauche, la tâche i est extérieure à l'intervalle critique qui est composé des tâches A et B. Ni A pas plus que B ne peuvent terminer avant le temps $t = 8$. En faisant débuter i avant $t = 8$, toutes les unités d'énergie entre le début de l'intervalle et le départ de i (en gris sur la figure) sont perdues car inutilisables par A ou par B. L'ajustement du temps de sortie de i fait en sorte de récupérer les unités d'énergie perdues. La portion de droite illustre la même situation pour 3 tâches confinées à l'intervalle critique. Bien que l'énergie totale des tâches soit de 3 unités inférieure aux capacités de la ressource, l'ordonnacement est irréalisable en raison du mauvais positionnement de la tâche C qui occasionne une perte de 4 unités d'énergie. Puisqu'aucune des tâches A ou B ne peut terminer avant le temps $t = 9$, le temps de sortie de la tâche C est ajustée à $est'_C = 9$. La partie de droite de la figure met en évidence une autre particularité du Not-First. La règle doit aussi détecter au sein des tâches confinées à l'intervalle testé.

Les règles Not-First/Not-Last illustre bien la symétrie inhérente au problème du filtrage des bornes des tâches. Il est d'ailleurs d'usage de ne filtrer que les temps de sortie des tâches. Pour les échéances, on procède à leur filtrage en exécutant l'algorithme une deuxième fois sur l'opposé mathématique de l'instance originale. Les échéances (lct) de la première instance deviennent les temps de sorties (est) de la seconde. La figure 2.6 illustre le procédé.

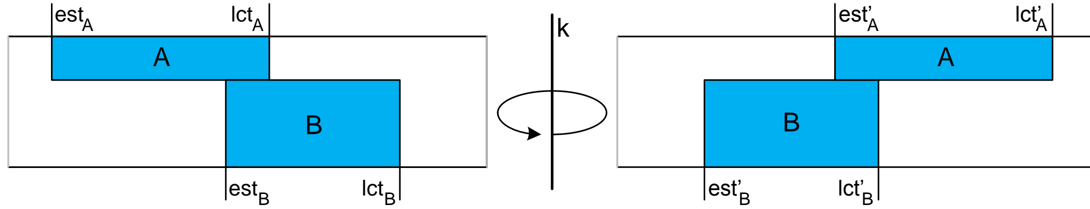


FIGURE 2.6: Une instance et son opposé mathématique.

Pour composer l'opposé mathématique, on choisit une constante $k \geq \max\ lct_i$. Ensuite pour les bornes de la deuxième instance on procède de la façon suivante :

$$est'_i = k - lct_i \quad et \quad lct'_i = k - est_i$$

Au terme de la seconde exéctution de l'algorithme, on recompose l'instance originale en se servant de la même constante k.

$$lct_i = k - est'_i$$

2.2 Le Edge-Finder de Vilím

Dans cette section nous présentons l'algorithme de filtrage pour la contrainte CUMULATIVE de Petr Vilím [46]. Il applique la règle Edge-Finding et un cas particulier de la règle Extended-Edge-Finding. Sa caractéristique principale est l'utilisation astucieuse d'un arbre binaire qu'il nomme arbre cumulatif Θ,Λ. Cette structure de données, introduite [20] par l'auteur [45] pour filtrer la contrainte DISJONCTIVE, fait partie intégrante de la majorité des algorithmes utilisés aujourd'hui pour filtrer la contrainte CUMULATIVE. Le Edge-Finder de Vilím constitue le modèle à partir duquel l'Extended-Edge-Finder proposé à la section 2.3 est construit.

2.2.1 Préalables

Il nous est utile d'estimer une borne maximale pour le temps minimum à partir duquel l'achèvement de toutes les tâches d'un sous-ensemble est possible. Cette quantité diffère dans sa nature du temps minimum de terminaison (ect) des tâches individuelles en ce qu'elle constitue une estimation [21]. Pour la distinguer nous utilisons Ect avec un E majuscule dans sa notation. La *borne* $Ect(\Omega)$ est la valeur optimale parmi toutes les combinaisons de tâches possibles au sein d'un sous-ensemble $\Omega \subseteq \mathcal{I}$. Nous désignons par Θ le sous-ensemble qui inclut les tâches de Ω qui génèrent la combinaison optimale. Pour l'évaluation de la borne Ect, on considère

20. À notre connaissance, un arbre binaire balancé est antérieurement utilisé dans [17] pour cumuler des valeurs en $O(n \log n)$ sans être dûment identifié comme tel.

21. Dans [3], les auteurs ont recours au *earliest completion time* d'un ensemble de tâches pour forcer la cohérence de bornes pour la contrainte INTER-DISTANCE . Toutefois, il s'agit là d'une *estimation exacte* dont l'évaluation est possible compte tenu que les régions tabous (zones prohibées pour le départ de tâches voir section 3.1) sont connues et que les tâches sont de durée fixe et de hauteur unitaire.

qu'aucune unité de production n'est perdue, c'est-à-dire que les tâches du sous-ensemble Θ s'emboîtent parfaitement tout en utilisant la ressource de façon optimale [22].

$$Ect(\Omega) = \max_{\Theta \subseteq \Omega} \left(est_\Theta + \left\lceil \frac{e_\Theta}{C} \right\rceil \right) = \left\lceil \frac{\max_{\Theta \subseteq \Omega}(C\, est_\Theta + e_\Theta)}{C} \right\rceil \tag{2.14}$$

En substituant (1.21) dans (2.14) nous obtenons [23] :

$$Ect(\Omega) = \left\lceil \frac{Env_\Omega}{C} \right\rceil \tag{2.15}$$

Pour cumuler des valeurs d'énergie et d'enveloppe et pour trouver le sous-ensemble Θ qui optimise Env_Ω, l'algorithme utilise une structure de données appelée par l'auteur arbre Θ, Λ. Pour éviter la confusion, nous utilisons l'appellation arbre cumulatif. L'*arbre cumulatif* est un arbre binaire entier. C'est-à-dire que chacun de ses noeuds internes possède exactement deux fils et que la distance de ses feuilles à la racine varie d'au plus une unité. Ainsi, pour un arbre binaire entier à n feuilles, la distance entre une feuille et la racine est d'au plus $\lceil \log n \rceil$.

Si aucune de ses feuilles n'a de noeud interne sur sa droite, l'arbre binaire entier porte le nom d'arbre binaire essentiellement complet. Quoique pas une exigence, nous préconisons l'utilisation d'un arbre binaire essentiellement complet [24] pour faciliter l'implémentation. On

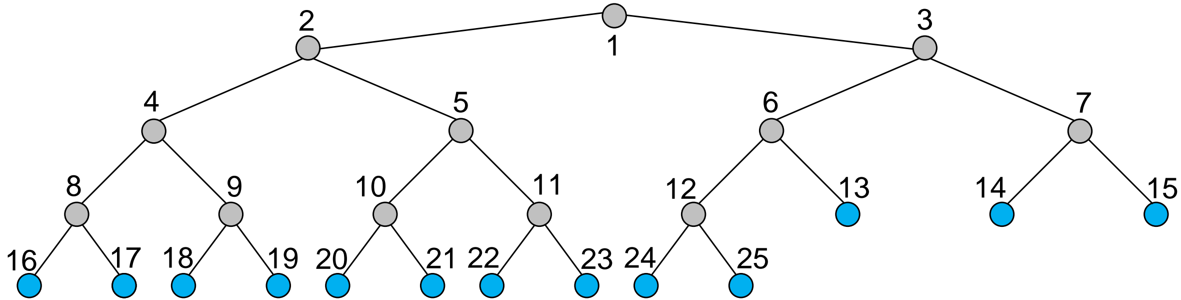


FIGURE 2.7: Un arbre binaire essentiellement complet à 13 feuilles

construit l'arbre cumulatif de la façon suivante. Chaque feuille correspond à une tâche $i \in \mathcal{I}$ et on les dispose en commençant par la gauche en ordre croissant des temps de sortie est_i. Par exemple, sur la figure 2.7, on positionne la première tâche dans la feuille 16 et la tâche de la

22. Baptiste et al.[5, 6] nomment cette relaxation *fully elastic* (voir la note 17). Elle consiste à remplacer chaque tâche i par $p_i h_i$ tâches de durée unitaire et de hauteur unitaire avec comme valeur de temps de sortie $est = est_i$ et d'échéance $lct = lct_i$.

23. La démonstration complète de l'évaluation de la borne Ect est présentée par l'auteur dans [47].

24. De façon similaire à un monceau, cette structure peut être encodée avec un vecteur ayant autant de composantes qu'il y a de noeuds dans l'arbre. Le rang du noeud (voir figure 2.7) est utilisé comme index pour accéder à l'information contenue dans le noeud. Il est ainsi aisé de se déplacer dans l'arbre grâce aux relations suivantes :

- $rang_{père} = rang_{fils} \;/\; 2$
- $rang_{filsGauche} = 2 \times rang_{père}$
- $rang_{filsDroit} = 2 \times rang_{père} + 1$

feuille 13 succède à celle de la feuille 25. Une feuille $\{i\}$ est identifiée par sa tâche associée et leur association perdure jusqu'au terme de l'exécution de l'algorithme. Les feuilles $\{i\}$ contiennent des valeurs spécifiques à leur tâche i associée. Lorsqu'une tâche change de statut pendant l'exécution de l'algorithme, les valeurs contenues dans sa feuille associée sont réinitialisées en conséquence. Pour les noeuds internes, ces valeurs sont fonction de celles contenues dans leurs noeuds fils. Ainsi, pour tout changement de valeurs opéré sur une feuille, il est possible de mettre à jour l'arbre cumulatif jusqu'à la racine en $O(\log n)$.

De l'équation de l'enveloppe (1.21) $Env_\Omega = \max_{\Theta \subseteq \Omega}(C\ est_\Theta + e_\Theta)$, on constate que l'une des composantes (e_Θ) de la valeur de l'enveloppe est l'énergie du sous-ensemble Θ. Or celle-ci est la sommation de l'énergie de toutes les tâches confinées à $[est_\Theta, lct_\Theta)$. L'autre composante ($Cest_\Theta$) prend pour acquis toute l'énergie que la ressource peut dispenser jusqu'au temps est_Θ. Ainsi, trouver le sous-ensemble $\Theta \subseteq \Omega$ qui maximise la valeur de Env_Ω équivaut à cerner l'intervalle $[est_\Theta, lct_\Theta)$ qui maximise $Cest_\Theta + e_\Theta$ avec $lct_\Theta = lct_\Omega$. Pour trouver la borne est_Θ, on se sert de l'arbre cumulatif de la façon suivante. Chaque noeud de l'arbre contient une valeur d'énergie et une valeur d'enveloppe. Pour une feuille $\{i\}$, ces valeurs sont celles de sa tâche associée i, $e_{\{i\}} = p_i h_i$ et $Env_{\{i\}} = Cest_i + e_i$. Pour les noeuds internes v, les valeurs sont calculées à partir de celles contenues dans leur fils de gauche (g) et de droite (d) :

$$e_v = e_g + e_d \qquad\qquad Env_v = \max(Env_g + e_d\,,\, Env_d)$$

On itére sur les tâches $j \in \mathcal{I}$ en ordre décroisant des échéances lct_j. On procède en les retirant une à une de l'arbre en affectant à zéro les quantités contenues dans leur feuille associée $\{j\}$. Ainsi au début de chaque itération, la racine de l'arbre contient la valeur optimale Env_Ω avec $lct_\Omega = lct_\Theta = lct_j$. La figure 2.8 illustre le processus.

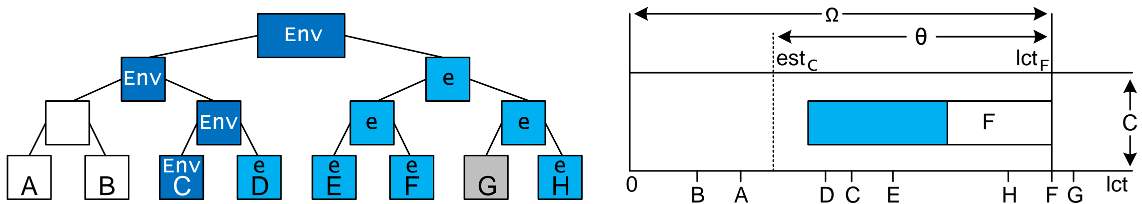


FIGURE 2.8: À gauche, un arbre cumulatif et ses feuilles $\{i\}$ disposées en ordre croissant des temps de sortie est_i de leur tâche asociée. A droite, une représentation schématique de la ressource cumulative avec l'axe du temps marqué des échéances lct. La feuille $\{G\}$ a été retirée de l'arbre à l'itération précédente. On itère maintenant sur la tâche F. Ce qui implique que tous les intervalles testés sont bornés supérieurement par lct_F. Dans cette instance hypothétique, l'enveloppe maximale est induite par la feuille associée à la tâche C. La partie de droite montre l'intervalle optimal $[est_\Theta, lct_\Theta) = [est_C, lct_F)$. Il contient les tâches $\{C, D, E, F, H\}$. La partie de gauche montre les valeurs qui sont cumulées jusqu'à la racine de l'arbre et qui résulte en la valeur maximale de Env_Ω.

2.2.2 La détection

À l'étape de détection, l'algorithme de Vilím détecte toutes les précédences issues de la règle Edge-Finding. Les tâches sont ensuite ajustées, le cas échéant, à l'étape d'évaluation (voir section 2.2.3). Le vecteur $precedence[i]$, $1 \leq i \leq n$, tient à jour la plus grande précédence détectée pour la tâche i, $\Omega \prec i \rightarrow precedence[i] = lct_\Omega$. Le vecteur fait ensuite la jonction entre l'étape de détection et l'étape d'évaluation.

Soit un sous-ensemble de tâches Ω et son intervalle associé $[est_\Omega, lct_\Omega)$. La règle du Edge-Finding nécessite de tester pour toute tâche i dont le temps de sortie est_i se situe dans l'intervalle $[est_\Omega, lct_\Omega)$ et dont l'échéance lct_i est plus grande que celle de l'intervalle, $lct_i > lct_\Omega$, si elle peut s'exécuter entièrement dans l'intervalle. Dans le cas contraire, on en déduit que toutes les tâches du sous-ensemble Ω doivent se terminer avant que i ne se termine. On appelle cette relation une précédence et on note $\Omega \prec i$ (voir section 1.2). De l'equation (2.2) : $\forall \Omega, \forall i \notin \Omega \wedge est_i \geq est_\Omega,\ C(lct_\Omega - est_\Omega) < e_\Omega + e_i \Rightarrow \Omega \prec i$.

L'équation (2.15) montre comment estimer la borne maximale du le temps minimum à partir duquel l'achèvement de toutes les tâches d'un sous-ensemble est possible (Ect_Ω). L'estimation optimale est donnée par l'intervalle $[est_\Theta, lct_\Theta) \subseteq [est_\Omega, lct_\Omega)$ qui maximise la valeur de Env_Ω. En incluant la tâche i au sous-ensemble Ω nous avons :

$$Ect(\Omega \cup \{i\}) = \left\lceil \frac{Env_\Omega + e_i}{C} \right\rceil \tag{2.16}$$

En combinant les équations 2.2 et 2.16 nous obtenons :

$$\forall \Omega, \forall i \notin \Omega \wedge est_i \geq est_\Omega\ ,\ Env_\Omega + e_i > Clct_\Omega \Rightarrow \Omega \prec i \tag{2.17}$$

La figure 2.9 schématise une ressource cumulative de capacité $C = 3$. On constate qu'il manque au minimum une unité d'énergie à la ressouce dans l'intervalle $[est_C, lct_E)$ pour y exécuter toutes les tâches du sous-ensemble optimal Θ conjointement à la tâche i.

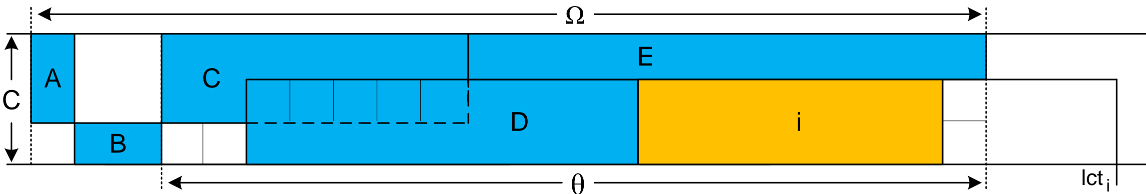


FIGURE 2.9: Illustratation d'une précédence en vertu de la règle du Edge-Finding. La figure montre toutes les tâches de l'ensemble Ω (en bleu) et le sous-ensemble Θ qui maximise la valeur Env_Ω en relation avec la borne supérieure $lct_\Omega = lct_E$. Le sous-ensemble optimal est formé des tâches C,D et E. La tâche i, dont le temps de sortie se situe dans l'intervalle $[est_\Theta, lct_\Theta)$, ne peut s'exécuter entièrement dans l'intervalle. On en déduit que toutes les tâches dont l'échéance n'est pas supérieure à $lct_\Theta = lct_E$ précédent la tâche i.

L'algorithme utilise l'arbre cumulatif pour cerner l'intervalle $[est_\Theta, lct_\Theta)$ et trouver la tâche qui maximise la fonction $Ect(\Omega \cup \{i\})$. L'algorithme itère (voir explications Section 2.2.1) sur les tâches $j \in \mathcal{I}$ en ordre décroissant des échéances lct_j. Nous désignons par Ω_j le sous-ensemble formé des tâches $i \in \mathcal{I}$ dont l'échéance n'est pas supérieure à lct_j.

$$\Omega_j = \{i \in \mathcal{I} \mid lct_i \leq lct_j\} \tag{2.18}$$

Au départ, toutes les tâches font partie de Ω_j. Au terme de l'itération sur la tâche j, l'algorithme la déplace de Ω_j vers Λ. Lorsqu'une précédence est découverte pour l'une ou l'autre des tâches de $i \in \Lambda$, elle est retirée de l'ensemble Λ. Au cours de l'exécution de l'algorithme une tâche $i \in \mathcal{I}$ est soit élément de Ω_j, élément de Λ ou d'aucun des deux ensembles.

$$\forall i \in \mathcal{I},\ i \in \Omega_j \text{ ou } i \in \Lambda \text{ ou } i \notin\ \Omega_j \cup \Lambda \qquad \text{et } \Omega_j \cap \Lambda = \emptyset$$

Nous appelons *enveloppe Lambda* et notons Env^Λ la valeur maximale de $Ect(\Omega_j \cup \{i\})$ à l'itération portant sur la tâche j. Nous avons :

$$Env^\Lambda = \max_{\Theta \subseteq \Omega_j, i \in \Lambda, est_\Theta \leq est_i} Ect(\Theta \cup \{i\}) \tag{2.19}$$

Nous désignons par *tâche la plus contraignante*, la tâche $i \in \Lambda$ responsable de l'enveloppe Lambda. Par exemple, sur la figure 2.9, la tâche i est la tâche la plus contraignante. L'intervalle $[est_\Theta, lct_\Theta)$ cerne les tâches qui générent la valeur d'enveloppe maximum parmi l'ensemble Ω_j en relation avec i, c'est-à-dire que $est_\Theta \leq est_i$ [25]. Pour obtenir la valeur de l'enveloppe Lambda à la racine de son arbre cumulatif, l'algorithme cumule quatre quantités. Pour une feuille (f), elles prennent les valeurs suivantes [26] :

$$e_f = \begin{cases} e_i & si \quad i \in \Omega_j \\ 0 & si \quad i \in \Lambda \end{cases} \qquad Env_f = \begin{cases} C\ est_i\ +\ e_i & si \quad i \in \Omega_j \\ -\infty & si \quad i \in \Lambda \end{cases}$$

$$e_f^\Lambda = \begin{cases} -\infty & si \quad i \in \Omega_j \\ e_i & si \quad i \in \Lambda \end{cases} \qquad Env_f^\Lambda = \begin{cases} -\infty & si \quad i \in \Omega_j \\ C\ est_i\ +\ e_i & si \quad i \in \Lambda \end{cases}$$

Pour les noeuds internes (v) incluant la racine, ces quatre quantités sont obtenues à partir des valeurs contenues dans les fils de gauche (g) et de droite (d) comme ceci :

$$\begin{aligned} e_v &= e_g + e_d \\ Env_v &= \max(Env_g + e_d,\ Env_d) \\ e_v^\Lambda &= \max(e_g^\Lambda + e_d,\ e_g + e_d^\Lambda) \\ Env_v^\Lambda &= \max(Env_g^\Lambda + e_d,\ Env_g + e_d^\Lambda,\ Env_d^\Lambda) \end{aligned}$$

25. Cette condition (voir équation 2.17) n'a pas à être testée. Les feuilles de l'arbre cumulatif hébergent les tâches disposées en ordre croissant des temps de sortie. En combinant une valeur d'enveloppe et une valeur d'énergie qui lui succède dans l'arbre, la tâche qui induit l'enveloppe est toujours à gauche de la tâche la plus contraignante $i \in \Lambda$. Par conséquent, son temps de sortie n'est pas supérieur à est_i.

26. L'algorithme affecte aux quantités la valeur $-\infty$, élément absorbant de l'addition, lorsqu'une composante essentielle (tâche) à la quantité est manquante.

Algorithme 1 : DétectionDesPrécédences

```
1  precedence[i] ← ect_i  ∀ i
   Ω_j ← I , Λ ← ∅
   for j ∈ I en ordre non-croissant des échéances lct_j do
2      while Env^Λ > C lct_j do
3          i ← tâche ∈ Λ responsable de Env^Λ
           if lct_j > precedence[i] then precedence[i] ← lct_j
           Λ ← Λ \ {i}
       Ω_{j−1} ← Ω_j \ {j}
       Λ ← Λ ∪ {j}
```

L'algorithme 1 initialise son arbre cumulatif en plaçant toutes les tâches dans l'ensemble Ω_j. Il itère sur les tâches j en ordre décroissant des échéances lct_j. Une itération se termine en deplaçant la tâche j vers l'ensemble Λ. À chacune des itérations, il détecte toutes les précédences en relation avec la borne supérieure $lct_\Theta = lct_j$. La détection s'effectue (ligne 2) en appliquant l'équation 2.19 : $Env^{\Lambda}_{racine} = \max_{\Theta,i} Ect(\Theta, \{i\})$ supérieur à $Clct_j$ indique qu'un intervalle optimal $[est_\Theta, lct_\Theta)$ restreint une tâche $i \in \Lambda$. L'algorithme procède en trouvant la tâche la plus contraignante (ligne 3), en mémorisant sa précédence et en la retirant de l'ensemble Λ. Après la mise à jour de l'arbre, on teste si l'on peut détecter (ligne 2) une autre précédence. Dans la négative, l'algorithme termine l'itération sur la tâche j. Pour trouver la feuille hébergeant la tâche la plus contraignante, on part de la racine de l'arbre en suivant d'abord la trace de l'enveloppe Env^{Λ} et par la suite celle de l'énergie e^{Λ}.

Trouver et mémoriser une précédence s'effectue en $O(\log n)$. L'algorithme 1 détecte un maximum de n précédences. Sa complexité est dans la famille $O(n \log n)$.

En guise d'initialisation du vecteur *precedence* (ligne 1), l'algorithme 1 affecte la valeur de la précédence naturelle ect_i (équation 1.25) pour chaque tâche i. Ce faisant, il traite un cas particulier de la règle du Extended-Edge-Finding [27] sans alourdir son exécution. La figure 2.10 illustre une situation du cas particulier qui mène à un ajustement.

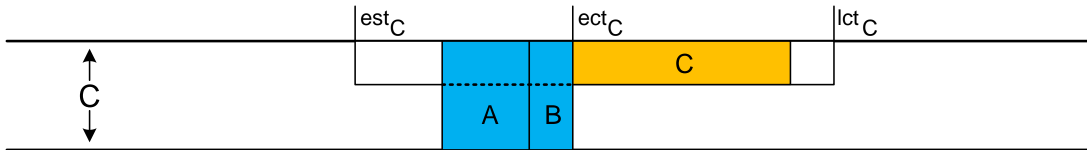


FIGURE 2.10: Illustration d'une précédence en vertu du Extended-Edge-Finding. La tâche C ne peut s'exécuter entièrement dans l'intervalle $[est_B, lct_B)$. Il est donc correct d'ajuster son temps de sortie. Bien qu'elle débute avant l'intervalle critique, l'initialisation judicieuse du vecteur *precedence* permet de traiter ce cas particulier de la règle Extended-Edge-Finding.

27. L'ajustement est le même pour la règle du Extended-Edge-Finding que pour la règle du Edge-Finding (équation 2.3). Il n'est fonction que de la valeur de la précédence et de la hauteur de la tâche à ajuster.

La figure 2.11 illustre la quatrième itération de la phase de détection de l'instance présentée sur la figure 1.3 à la fin du premier chapitre. Les états successifs des noeuds intérieurs qui sont mis à jour plus d'une fois lors de l'itération y sont juxtaposés en changeant les couleurs du rouge au jaune et finalement au bleu qui marque l'état des noeuds au sortir de la boucle *while*. Les trois premières itérations n'ont découvert aucune relation de précédence. Les tâches

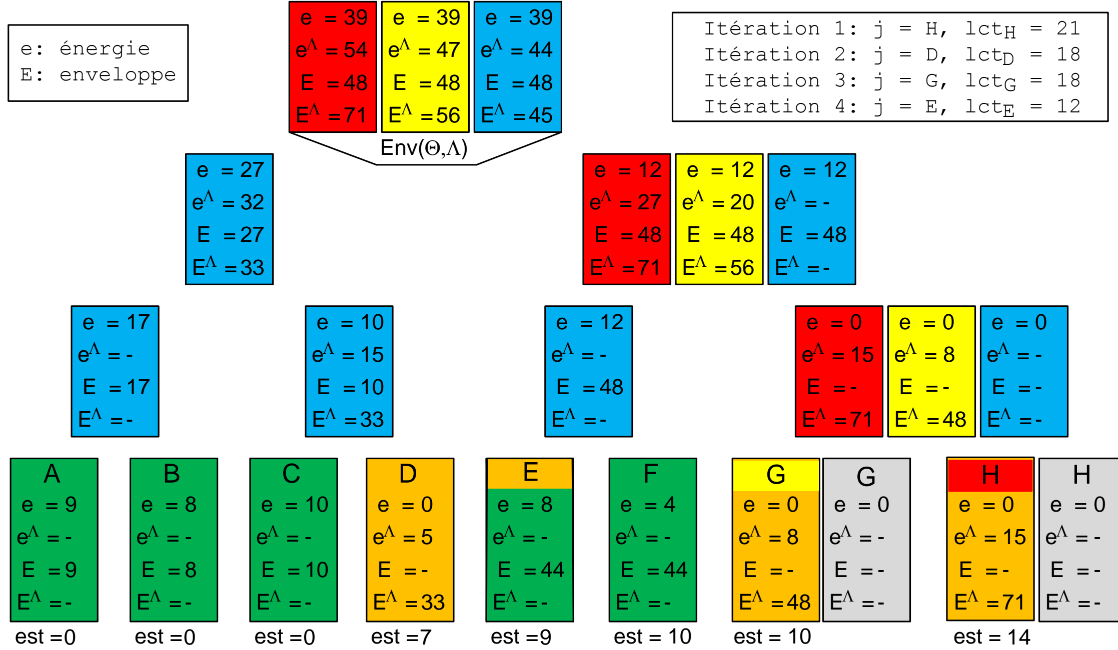


FIGURE 2.11: Illustration de la mécanique de l'arbre cumulatif.

H, D et G ont été déplacées de l'ensemble Ω_j (feuilles vertes de la figure 2.11) à l'ensemble Λ (feuilles orangés). On itère maintenant sur la tâche E. À la racine de l'arbre on découvre une relation de précédence ($Env^\Lambda > C\ lct_E$). L'étape suivante consiste à découvrir la tâche la plus contraignante. On part de la racine en suivant la trace de i (chemin en rouge) ce qui nous conduit à la feuille associée à la tâche H. La tâche H est retirée de l'ensemble Λ et les valeurs de sa feuille associée sont réinitialisées en conséquence. On remet alors à jour l'arbre cumulatif et vérifions de nouveau si une relation de précédence existe. Cette séquence d'opérations (découverte de la tâche la plus contraignante, réinitialisation de sa feuille associée et mise à jour de l'arbre) est répétée jusqu'à ce qu'aucune relation de précédence ne soit détectée. Dans l'exemple illustrée par la figure 2.11, une seconde relation de précédence est détectée et la trace (chemin jaune) nous conduit à la feuille associée à la tâche G qui est à son tour réinitialisée. La mise à jour de l'arbre qui s'ensuit produit un état stable ($Env^\Lambda \leq C\ lct_E$) et on termine l'itération en faisant passer la tâche E de l'ensemble Ω_j à l'ensemble Λ. Ce qui entraîne une réaffectation des valeurs de sa feuille associée. Cette dernière mise à jour n'est pas illustrée sur la figure 2.11.

2.2.3 L'évaluation

Le but de cette étape est d'évaluer, pour chacune des valeurs d'échéance des tâches $j \in \mathcal{I}$, la quantité de ressources disponible une fois le traitement complété de toutes les tâches dont l'échéance n'est pas supérieur à lct_j. Les tâches $i \in \mathcal{I}$ pour lesquelles une précédence existe à cette échéance ($lct_j \prec i$) sont comparées à l'évaluation et leur temps de sortie est ajusté si la capacité de la ressource à traiter une tâche supplémentaire au temps $t = est_i$ s'avère insuffisante. Or cette capacité à traiter une tâche supplémentaire dépend de la hauteur h_i de la tâche i à traiter. L'algorithme effectue donc une évaluation pour chacune des k différentes hauteurs de tâche présentes dans le problème. Nous utilisons la notation $update[j, h]$ pour désigner le résultat de l'évaluation où h est la hauteur d'une tâche subséquente à traiter et j, l'une des tâches du problème $j \in \mathcal{I}$ caractérisée par son échéance lct_j. C'est-à-dire qu'une tâche de hauteur h, précédée par le sous-ensemble Ω_j, ne pourra débuter avant la valeur de temps $update[j, h]$. Nous utilisons la notation $geo^\sharp[\Omega_j, h]$ pour exprimer les résultats temporaires jusqu'au définitif $geo[\Omega_j, h]$ utilisé par l'algorithme pour évaluer l'$update[\Omega_j, h]$ [28]. Une fois tous les $update[j, h]$ calculés, on procède à l'ajustement des bornes de sortie (est_i) ainsi [29] :

$$\Omega_j \prec i \;\Rightarrow\; est'_i = \max(est_i, update[j, h_i]) \;\text{ où }\; \Omega_j = \{k \in \mathcal{I} \mid lct_k \leq lct_j\} \tag{2.20}$$

On évalue l'$update[j, h]$ au temps $t = lct_j$ de façon géométrique en découpant toute l'énergie de l'ensemble Ω_j en petits blocs unitaires. On remplit d'abord l'espace $C - h$ de est_{Ω_j} à lct_{Ω_j} et on utilise les blocs restants pour compléter la capacité totale de la ressource en commençant à est_{Ω_j}.

$$geo^2[\Omega_j, h] = est_{\Omega_j} + \left\lceil \frac{e_{\Omega_j} - (C - h)(lct_{\Omega_j} - est_{\Omega_j})}{h} \right\rceil$$

Cette évaluation n'est toutefois pas optimale car elle considère tout l'ensemble Ω_j. À la section 2.2.1, on montre comment cerner l'intervalle $[est_\Theta, lct_\Theta)$ avec $lct_\Theta = lct_{\Omega_j}$ qui maximise la borne $Ect(\Omega_j)$. En appliquant un raisonnement similaire, à savoir trouver l'intervalle $[est_\Theta, lct_\Theta)$ qui maximise $geo^2[\Omega_j, h]$, nous obtenons :

$$geo^1[\Omega_j, h] = \max_{\Theta \subseteq \Omega_j} est_\Theta + \left\lceil \frac{e_\Theta - (C - h)(lct_\Theta - est_\Theta)}{h} \right\rceil \tag{2.21}$$

À l'étape de détection, on cherche à savoir si l'énergie que la ressource peut dispenser dans l'intervalle optimal $[est_\Theta, lct_\Theta)$ est suffisant pour traiter toutes les tâches qui y sont confinées

28. Nous suivons, pas à pas, la démonstration de l'article [46].

29. À l'étape de détection, le vecteur *precedence* mémorise les précédences pour chaque tâche i. La valeur qu'il enregistre est celle du temps $t = lct_j$ où j est la tâche dont l'échéance lct_j est la limite supérieure de l'intervalle traité. La raison en est la suivante. L'algorithme 1 traite aussi un cas particulier de la règle du Extended-Edge-Finding et chaque position du vecteur *precedence* est initialisée au temps minimum de terminaison de la tâche i correspondante ($t = ect_i$) qui constitue la précédence naturelle de la tâche i. Or, il n'existe pas nécessairement une tâche $j \in \mathcal{I}$ tel que $lct_j = ect_i$. En conséquence, pour ajuster les tâches, on doit utiliser l'$update[j, h_i]$ du plus grand j tel que $lct_j \leq precedence[i]$.

conjointement avec la tâche $i \in \Lambda$ dans sa totalité. Il est sans conséquence de considérer la valeur totale de l'énergie de la tâche i pour les fins du calcul de maximisation. Pour l'évaluation, on cherche à trouver la restriction maximale que le sous-ensemble de tâches optimal $\Theta \subseteq \Omega_j$ peut exercer sur une tâche i de hauteur h_i. Puisque la tâche i peut se terminer après lct_Θ, on ne doit pas considérer toute son énergie, mais bien une seule tranche de hauteur $h = h_i$ que l'on cherche à exécuter le plus tôt possible conjointement aux tâches confinées à l'intervalle optimal. La figure 2.12 illustre la problématique.

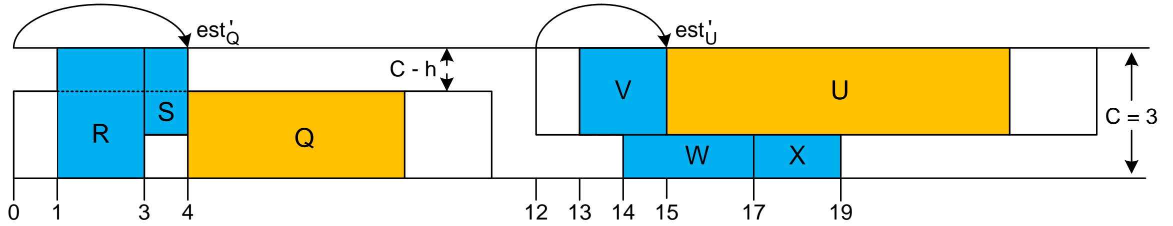


FIGURE 2.12: Deux situations de précédence. À gauche, la tâche Q doit obligatoirement se terminer après que les tâches R et S soient complétées. On applique l'Équation 2.21 pour $\Theta = \{R, S\}$ et on trouve $geo^1[\Omega_S, 2] = 4$. Nous obtenons le même résultat pour $\Theta = \{S\}$. Les deux sous-ensembles sont optimaux pour l'évaluation $geo^1[\Omega_S, 2]$. À droite, la tâche U est précédée par l'ensemble des tâches dont l'échéance n'est pas supérieur à $lct_X = 19$. En choisissant $\Theta = \{X\}$ on trouve $geo^1[\Omega_X, 2] = 17$ et en choisissant $\Theta = \{W, X\}$ notre évaluation de $geo^1[\Omega_X, 2]$ est de 14. Ces deux valeurs sont inexactes car il n'y a pas assez d'énergie dans aucun des deux sous-ensembles pour restreindre une tâche subséquente de hauteur $h = 2$: sur la figure on constate qu'il est possible d'insérer W et X entre U et la limite de la ressource. La bonne évaluation nous est donnée à partir du sous-ensemble $\Theta = \{V, W, X\}$ pour lequel $geo^1[\Omega_X, 2] = 15$.

Il est impératif d'ajouter la condition d'*énergie minimale* au sous-ensemble Θ dont l'énergie devra être supérieure à celle requise pour remplir l'espace de hauteur $C - h$ sur toute la longueur de l'intervalle $[est_\Theta, lct_\Theta)$. L'équation qui suit est correcte.

$$geo^0[\Omega_j, h] = \max_{\substack{\Theta \subseteq \Omega_j \\ e_\Theta > (C-h)(lct_\Theta - est_\Theta)}} est_\Theta + \left\lceil \frac{e_\Theta - (C-h)(lct_\Theta - est_\Theta)}{h} \right\rceil$$

À l'étape d'évaluation, l'ensemble Ω_j est construit progressivement en y ajoutant une à une les tâches en ordre croissant des échéances. Il est donc inutile de considérer toutes les combinaisons possibles de $\Theta \subseteq \Omega_j$ mais bien seulement celles incluant la tâche j tout juste ajoutée à l'ensemble Ω_{j-1}. S'il s'avère que la valeur maximale est obtenue à partir d'un sous-ensemble excluant cette tâche, une évaluation antérieure l'a déjà considérée. Nous définissons $update[j, h]$ récursivement de la façon suivante :

$$update[j, h] = \max(update[j-1, h], geo[\Omega_j, h]) \text{ et } update[0, h] = 0 \tag{2.22}$$

Il n'est pas dans l'esprit de l'algorithme de vérifier la condition d'énergie minimale sur toutes les combinaisons de $\Theta \subseteq \Omega_j$. Or, considérant que la condition d'énergie minimale dépend de la longueur de l'intervalle $[est_\Theta, lct_\Theta)$ et que $lct_\Theta = lct_j$ pour tous les sous-ensembles évalués à l'itération j. Considérant aussi que l'arbre cumulatif est constitué de feuilles associées aux tâches de Ω_j triées en ordre croissant des temps de sortie (est_j). L'algorithme contourne l'obstacle combinatoire via l'arbre cumulatif en trouvant le plus petit ensemble de feuilles consécutives se terminant à la feuille associée à la tâche j et respectant la condition d'énergie minimale. Nous notons $estMax[j, h]$ le temps de sortie de la première tâche du sous-ensemble Θ ainsi formé.

$$estMax[j, h] = \max\{\ est_\Theta \mid \Theta \subseteq \Omega_j\ ,\ e_\Theta > (C-h)(lct_j - est_\Theta)\ \} \tag{2.23}$$

Par exemple, dans la partie de droite de la figure 2.12, le plus petit sous-ensemble est $\{V, W, X\}$ et $estMax[X, 2] = est_V = 13$. Ce qui mène à la version finale de l'équation utilisée par l'algorithme pour l'évaluation des updates :

$$geo[\Omega_j, h] = \max_{\substack{\Theta \subseteq \Omega_j \\ est_\Theta \leq estMax[j,h]}} est_\Theta + \left\lceil \frac{e_\Theta - (C-h)(lct_\Theta - est_\Theta)}{h} \right\rceil \tag{2.24}$$

En considérant que le sous-ensemble optimal Θ inclut nécessairement la tâche j, on obtient après manipulations et simplifications de la partie de droite de l'Équation (2.24) :

$$est_\Theta + \left\lceil \frac{e_\Theta - (C-h)(lct_\Theta - est_\Theta)}{h} \right\rceil = \left\lceil \frac{Cest_\Theta + e_\Theta - (C-h)lct_j}{h} \right\rceil$$

Pour trouver le sous-ensemble qui génère la valeur maximale tout en s'abstenant de considérer ceux dont le temps de sortie est trop grand ($est_\Theta > estMax[j, h]$), on sépare l'ensemble Ω_j en deux sous-ensembles α et β tel que :

$$\alpha = \{k \mid k \in \Omega_j\ \wedge\ est_k \leq estMax[j, h]\} \tag{2.25}$$
$$\beta = \{k \mid k \in \Omega_j\ \wedge\ est_k > estMax[j, h]\} \tag{2.26}$$

Ainsi, la condition d'énergie minimale est respectée si et seulement si le sous-ensemble optimal Θ inclut au moins une tâche de α. En appliquant la même opération de séparation à Θ et en lui substituant Θ_α et Θ_β dans l'équation (2.24) sous sa forme simplifiée on obtient :

$$geo[\Omega_j, h] = \max_{\substack{\Theta_\alpha \subseteq \alpha \\ \Theta_\beta \subseteq \beta}} \left\lceil \frac{Cest_{\Theta_\alpha} + e_{\Theta_\alpha} + e_{\Theta_\beta} - (C-h)lct_j}{h} \right\rceil = \left\lceil \frac{Env_\alpha + e_\beta - (C-h)lct_j}{h} \right\rceil$$

L'auteur préconise de procéder à une scission de l'arbre cumulatif. Les deux sous-arbres obtenus sont nommées α et β. Les sous-arbres sont constitués des feuilles associées aux tâches de Ω_j

selon leur appartenance à α ou à β (2.26). Une fois les sous-arbres mis à jour, la valeur Env_α se trouve à la racine du sous-arbre α et la valeur e_β à la racine du sous-arbre β[30]. La figure 2.13 schématise une scission.

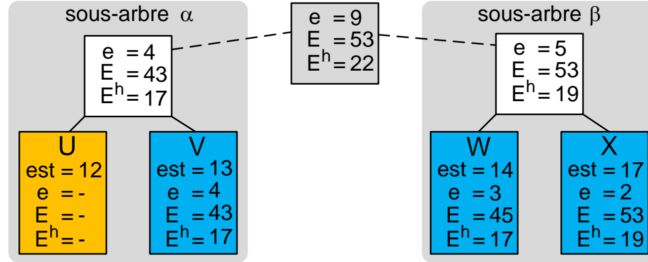


FIGURE 2.13: Instance de droite de la figure 2.12. Le $estMax[X, 2]$ est donné par la feuille associée à la tâche V. On scinde l'arbre cumulatif immédiatement après la feuille V.

Algorithme 2 : CalculDesUpdates

---

for $h \in \{h_i, i \in \mathcal{I}\}$ **do**
  $\Omega_0 \leftarrow \emptyset$
  $update[0, h] \leftarrow 0$
  for $j \in \mathcal{I}$ *en ordre non-decroissant des échéances* lct_j **do**
    $\Omega_j \leftarrow \Omega_{j-1} \cup \{j\}$
1 $\alpha, \beta = Scission(\Omega_j, estMax[j, h])$
2 $evaluation \leftarrow \left\lceil \frac{Env_\alpha + e_\beta - (C-h)\, lct_j}{h} \right\rceil$
    $update[j, h] \leftarrow \max(update[j-1, h], evaluation)$
    $\Omega_j \leftarrow \alpha \cup \beta$

L'algorithme 2 doit connaître (ligne 1) la feuille associée à la valeur $estMax[j, h]$. Pour la trouver, il véhicule dans l'arbre cumulatif une nouvelle quantité appelée *enveloppe partielle* que l'on note (Env^h). L'enveloppe partielle est une mesure de l'enveloppe adaptée à la hauteur h des tâches sur lesquelles l'évaluation courante $geo[\Omega_j, h]$ s'applique. Tout comme l'enveloppe, la notion d'enveloppe partielle est définie pour une tâche i et peut s'étendre à un sous-ensemble de tâches Ω :

$$Env_i^h = (C - h)est_i + e_i \qquad Env_\Omega^h = \max_{\Theta \subseteq \Omega}(C - h)est_\Theta + e_\Theta \tag{2.27}$$

L'arbre cumulatif utilisé par l'algorithme 2 porte les valeurs d'énergie, d'enveloppe et d'enveloppe partielle. Ces valeurs sont mises à jour de façon similaire aux valeurs de l'arbre cumulatif de l'étape de détection. Pour l'enveloppe partielle, la valeur d'un noeud interne (v) est calculée à partir des fils de gauche (g) et de droite (d) comme suit :

$$Env_v^h = \max(Env_g^h + e_d,\ Env_d^h)$$

30. Cette méthode est laborieuse. Elle nécessite de considérer plusieurs cas de positionnement de la feuille de scission. Nous proposons une façon simple d'obtenir les valeurs Env_α et e_β à la section 2.2.4.

L'exécution débute avec un arbre vide et itère en y ajoutant les tâches, une à une, en ordre croissant des échéances. [31] À la ligne 1, on cherche la feuille la plus à droite dont la tâche associé ($estMax[j, h]$) assure la satisfaction de la condition d'énergie minimale. Pour y parvenir, l'algorithme utilise une version transformée de l'équation 2.23 :

$$e_\Theta > (C-h)(lct_j - est_\Theta) \;\Leftrightarrow\; (C-h)\,est_\Theta + e_\Theta > (C-h)\,lct_j \;\Leftrightarrow\; Env^h_\Omega > (C-h)\,lct_j$$

Algorithme 3 : Calcul du $estMax[j, h]$

$n \leftarrow racine$
$E \leftarrow 0$
while n *n'est pas une feuille* **do**
  if $Env^h_{filsdroit(n)} + E > (C-h)\,lct_j$ **then**
    $n \leftarrow filsdroit(n)$
  else
    $E \leftarrow E + e_{filsdroit(n)}$
    $n \leftarrow filsgauche(n)$
return n

La découverte de la feuille de scission (algorithme 3) s'effectue en $O(\log n)$ tout comme chacune des mises à jour de l'arbre cumulatif. Elles s'exécutent à l'intérieur de deux boucles imbriquées. La première itère sur toutes les k différentes hauteurs de tâche du problème ($k = |\{h_i \mid i \in \mathcal{I}\}|$) et la seconde sur toutes les n échéances de tâche. La complexité générale de cette étape de l'algorithme est donc de $O(kn \log n)$ et elle domine la complexité de l'étape de détection.

Disposant de tous les *update*$[j, h]$, il est possible de mettre à jour les temps de sortie (est_i) en temps linéaire à partir de la relation (2.20).

2.2.4 Amélioration proposée

Dans cette section, nous suggérons une méthode simple pour trouver les quantités Env_α et e_β requise par l'algorithme 2 (ligne 2) sans avoir recours à la scission de l'arbre cumulatif qui doit, de toute façon, être réunifié par la suite. Au moment de la scission, l'arbre cumulatif porte toutes les tâches de Ω_j. Nous avons :

$$e_{\Omega_j} = e_\alpha + e_\beta \;\Leftrightarrow\; e_\beta = e_{\Omega_j} - e_\alpha \quad \text{où } \; \Omega_j = \alpha \cup \beta$$

Ce qui nous dispense d'avoir à mettre à jour le sous-arbre β puisque la quantité e_β est l'énergie totale de Ω_j, dont la valeur est directement accessible à la racine de l'arbre cumulatif, de laquelle on retranche l'énergie du sous-arbre α.

31. En fait, les tâches sont ajoutées à l'arbre dans l'ordre inverse à celui par lequel on les retire de l'arbre à l'étape de détection, celle-ci débutant avec un arbre contenant toutes les tâches $i \in \mathcal{I}$.

Pour évaluer l'énergie et l'enveloppe du sous-arbre α, on procède comme suit en partant de la feuille de scission. On initialise avec les valeurs contenues dans la feuille de scission et on remonte l'arbre jusqu'à la racine. À chaque nouveau palier, si l'on provient d'un fils droit, on met à jour les valeurs en cumulant avec les valeurs contenues dans le fils gauche. Pour la mise à jour, on procède de la même façon que pour l'arbre cumulatif de l'étape de détection. Pour l'enveloppe on choisit le maximum entre la valeur d'enveloppe courante et l'enveloppe du fils de gauche auquelle on ajoute l'énergie courante et pour l'énergie on additionne celle du fils gauche avec l'énergie courante. Si l'on provient d'un fils gauche on n'effectue aucune mise à jour. L'algorithme 4 s'exécute en $O(\log n)$.

Algorithme 4 : Calcul de la valeur $Env_\alpha + e_\beta$

$n \leftarrow estMax[j, h]$
$E \leftarrow Env_n$
$e \leftarrow e_n$
while *n n'est pas la racine de l'arbre cumulatif* **do**
  if *n est un fils droit* **then**
    $g \leftarrow filsGauche(parent_n)$
    $E \leftarrow \max(\, Env_g + e \,,\, E \,)$
    $e \leftarrow e + e_g$
  $n \leftarrow parent_n$
return $e_n - e + E$

Cette amélioration n'a pas la prétention d'apporter un gain en temps significatif. Toutefois elle simplifie l'algorithme et son implémentation tout en s'inscrivant dans les opérations naturelles d'un arbre binaire. La figure 2.14 schématise la méthode.

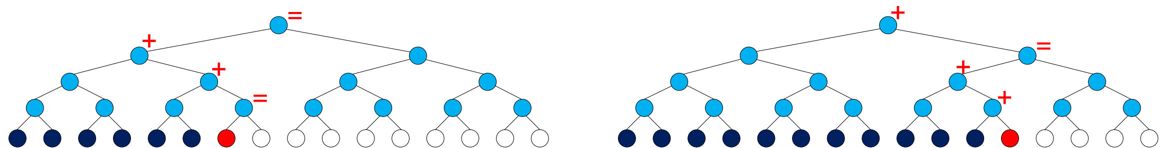


FIGURE 2.14: Deux exemples de l'évaluation des valeurs Env_α et e_α à partir de la feuille de scission (les feuilles en rouge). Les noeuds marqués d'un plus rouge sont ceux sur lesquels on effectue un cumul des valeurs courantes avec celles du fils de gauche. Les noeuds marqués d'un signe d'égalité sont ceux sur lesquels aucune opération n'est effectuée et on transporte les valeurs courantes jusqu'au noeud parent.

2.3 Contributions

Dans cette section, nous proposons des algorithmes de filtrage pour la contrainte CUMULATIVE qui exploitent une règle ou une combinaison de règles de filtrage. Leur complexité est meilleure que les algorithmes actuels exploitant les mêmes règles.

Pour la règle du Time-Tabling, nous montrons comment filtrer en $O(n \log n)$ en utilisant un arbre balancé pour colliger des intervalles qui ne se chevauchent pas. Notre algorithme obtient une complexité équivalente [32] à celle de l'algorithme de Beldiceanu et Carlsson [9] $O(n \log n + np)$.

Pour la règle du Extended-Edge-Finding, notre propagateur s'exécute en $O(kn \log n)$. Ce qui constitue une amélioration sur l'algorithme de Mercier et Van Hentenryck [28] dont la complexité est de $O(kn^2)$. Le propagateur proposé met en scène un modèle aux fondements géomètriques pour la contrainte CUMULATIVE. Il considère quatre types d'interaction entre un intervalle critique et une tâche extérieure qui mène à quatre règles de détection. Parce qu'il compare les tâches à filtrer avec leur intervalle de slack minimum, il s'inspire de la méthode de Kameugne, Fotso, Scott et Ngo-Kateu [21] tout en s'inscrivant directement dans le raisonnement énergétique de Baptiste, Le Pape et Nuijten [5, 6]. Le modèle permet d'ailleurs, par une décomposition appropriée des tâches, d'utiliser le même algorithme pour appliquer la règle combinée du Time-Table-Extended-Edge-Finding. Notre algorithme applique cette règle en $O(kn \log n)$ surpassant la complexité de $O(n^2 \log n)$ de l'algorithme de Vilím.

2.3.1 Un algorithme simple pour le Time-Tabling

Le Time-Tabling consiste à déterminer la quantité de ressources qui doit nécessairement être allouée au traitement d'une ou de plusieurs tâches spécifiques. Pour chaque tâche i qui satisfait la relation $lst_i < ect_i$, l'intervalle $[lst_i, ect_i)$ détermine sa partie fixe (pf_i) qui requiert nécessairement h_i quantité de ressources sur l'intervalle $[lst_i, ect_i)$. Soit $f(\Omega, t)$, la hauteur totale de toutes les parties fixes des tâches du sous-ensemble Ω au temps t et soit $f(\Omega, [a, b))$, le bloc cumulé dans l'intervalle $[a, b)$ des parties fixes des tâches de Ω. L'équation 2.6 indique comment former le bloc cumulé :

$$f(\Omega, t) = \sum_{i \in \Omega | t \in [lst_i, ect_i)} h_i \qquad\qquad f(\Omega, [a, b)) = \sum_{t \in [a, b)} f(\Omega, t)$$

Soit R, une liste strictement croissante $r_1, \ldots, r_x$, composée du temps maximum de départ lst_i et du temps minimum de terminaison ect_i de toutes les tâches i qui satisfont la relation $lst_i < ect_i$. Puisque le bloc cumulé est la sommation de toutes les parties fixes des tâches i et qu'une partie fixe pf_i est de la forme $[lst_i, ect_i)$, tous les temps t qui marquent une variation de hauteur du bloc cumulé sont nécessairement éléments de la liste R.

32. Voir la note 11 à la section 2.1.

Nous représentons une section du bloc cumulé par un tuple $bc = \langle [a, b), h \rangle$, où $[a, b)$ est un intervalle avec a et b éléments de la liste R et $h \geq 0$ est la hauteur du bloc cumulé sur cet intervalle. Soit $BC = \{bc_y, 0 \leq y < x\}$, l'ensemble de cardinalité minimale des tuples bc qui découpent le bloc cumulé en intervalles contigus sur $[r_1, r_x)$. L'algorithme 5 assemble l'ensemble BC.

Algorithme 5 : FormationDuBlocCumuléMinimum ($\mathcal{I}, C$)

```
L ← {lst_i | i ∈ I, lst_i < ect_i}
E ← {ect_i | i ∈ I, lst_i < ect_i}
R ← trier(L ∪ E)      // Liste strictement croissante
Rh_x ← 0 ∀x ∈ 1..|R|
BC ← ∅
for i ∈ I do
    if lst_i < ect_i then
        Rh_index(lst_i,R) ← Rh_index(lst_i,R) + h_i
        Rh_index(ect_i,R) ← Rh_index(ect_i,R) − h_i
if |R| > 0 then hauteur ← Rh_1
k ← 1 , position ← 1
while k < |R| do
    k ← k + 1
    if hauteur > C then return L'horaire est infaisable
    if Rh_k ≠ 0 then
        BC ← BC ∪ {⟨[R_position, R_k), hauteur⟩}
        position ← k
        hauteur ← hauteur + Rh_k
return BC
```

L'algorithme 5 accepte un ensemble de tâches $\mathcal{I}$ et la capacité de la ressource C en entrée et retourne l'ensemble BC formé des tuples bc qui découpent le bloc cumulé en intervalles contigus. La liste R contient un maximum de $2n$ éléments. On peut la trier en $O(n \log n)$. L'algorithme itère sur n tâches. Au sein d'une itération, les fonctions $index(lst_i)$ et $index(ect_i)$ s'exécute en $O(\log n)$. Construire un élément de BC s'effectue en temps constant et BC contient un maximum de $2n - 1$ éléments. La complexité générale de l'algorithme 5 est $O(n \log n)$.

L'équation 2.7 énonce la règle de filtrage par Time-Tabling :

$$ect_i > t \geq est_i \wedge C < h_i + f(\Omega \setminus \{i\}, t) \Rightarrow est'_i > t$$

La règle nécessite de comparer chaque tâche i avec le bloc cumulé généré par toutes les autres tâches de l'ensemble $f(\Omega \setminus \{i\}, t)$. Ce qui implique d'évaluer jusqu'à n blocs cumulés. Nous contournons cette obligation en considérant deux cas de tâches à filtrer selon que la tâche i satisfait ou non la relation $lst_i < ect_i$.

Pour une tâche i qui satisfait $lst_i \geq ect_i$ (cas 1), sa partie fixe pf_i est nulle. Conséquemment $f(\Omega \setminus \{i\}, t) = f(\Omega, t)$ pour tout temps t. Pour une tâche i du premier cas la règle s'énonce comme suit. S'il existe un intervalle $[a, b)$ entre le temps de sortie de la tâche i et son temps minimum de terminaison dont la hauteur cumulée des tâches fixes ne permet pas d'exécuter la tâche i conjointement, alors le temps de sortie de la tâche i doit être repoussé au premier temps t_r, $t < b \leq t_r$ où la ressource est suffisante pour débuter [33] la tâche i conjointement avec le bloc cumulé. Pour une tâche i qui satisfait $lst_i < ect_i$ (cas 2), la contribution de la tâche i au bloc cumulé débute au temps $t = lst_i$. Ainsi nous avons $f(\Omega \setminus \{i\}, t) = f(\Omega, t)\ \forall t < lst_i$. Pour le deuxième cas, nous limitons l'application de la règle aux temps t qui satisfont $t < lst_i$ autant à la détection qu'à l'ajustement [34]. Les équations qui suivent adaptent la règle du Time-Tabling à leur cas respectif :

$$\begin{aligned}
&(\text{cas } 1)\ lst_i \geq ect_i \wedge t \in [est_i, ect_i) \wedge f(\Omega, t) > C - h_i \\
&\qquad\qquad\qquad \Rightarrow est'_i = \min\{t_r \mid f(\Omega, t_r) \leq C - h_i,\ t_r > t\} \\
&(\text{cas } 2)\ lst_i < ect_i \wedge t \in [est_i, lst_i) \wedge f(\Omega, t) > C - h_i \\
&\qquad\qquad\qquad \Rightarrow est'_i = \min\{t_r \mid f(\Omega, t_r) \leq C - h_i,\ t_r > t\} \cup \{lst_i\}
\end{aligned}$$

Pour le filtrage des tâches $i \in \mathcal{I}$, nous trions les tuples $bc \in BC$ issus de l'algorithme 5 en ordre non-croissant des hauteurs. On itère sur les tâches i en ordre non-décroissant des hauteurs. Au fil des itérations, on tient à jour un arbre binaire balancé (type AVL) qui contient des intervalles qui ne se chevauchent pas [35]. Il est formé par tous les tuples bc dont la hauteur est suffisante pour restreindre les tâches i qu'il reste à tester ($h_{bc} + h_i > C$).

Nous appelons *aboutement* l'opération consistant à joindre deux intervalles contigus.

$$[a_1, b_1) \cup [a_2, b_2) = [a_1, b_2) \quad , b_1 = a_2$$

De même, nous appelons *colmatage* l'opération consistant à joindre deux intervalles disjoints par l'ajout d'un troisième intervalle contigu aux deux premiers.

$$\{[a_1, b_1), [a_3, b_3)\} \cup [a_2, b_2) = [a_1, b_3) \quad , b_1 = a_2, b_2 = a_3, b_1 < a_3$$

33. Si la tâche i ne peut pas s'exécuter entièrement en débutant au temps t_r, l'itération suivante du propagateur le détecte. Il est impossible à un propagateur non-idempotent de déduire à coup sûr le temps de sortie de la tâche i au point fixe. En effet, un ajustement sur une tâche j, $j \neq i$ peut entraîner une nouvelle partie fixe qui fera en sorte d'entraîner un nouvel ajustement sur la tâche i. Pour cette raison, il est d'usage en propagation de la contrainte CUMULATIVE de se contenter d'ajuster le plus précisément toutes les tâches qui peuvent être ajustées en vertu de la règle de filtrage. On construit les propagateurs les plus rapides et on relance leur exécution jusqu'à l'obtention d'un point fixe.

34. Cet affaiblissement de la règle est sans conséquence. Si l'on peut déduire que la tâche i doit débuter après son temps maximum de départ lst_i en vertu de la règle du Time-Tabling, une itération ultérieure de l'algorithme le détecte et le solveur conclut alors à l'infaisabilité de l'horaire.

35. Puisqu'il ne contient que des intervalles qui ne se chevauchent pas, l'arbre utilisé par l'algorithme est assimilable à un arbre à clefs uniques. Par conséquent, la structure de données appelée *van Emde Boa tree* [14] pp.531–560 est utilisable. Elle supporte en $O(\log \log n)$ les opérations *insertion*, *recherche* et *successeur* auxquelles l'algorithme 6 a recours. Avis aux puristes.

Considérant que les tuples bc déterminent des intervalles $[a, b)$, tous contigus ou disjoints entre eux. Considérant que l'ensemble BC contient les tuples bc qui couvrent toute la distance de r_1 à r_x, l'ajout d'un intervalle dans l'arbre mène soit à une insertion, un colmatage ou à un aboutement. Dans les cas de colmatages ou d'aboutements, l'intervalle résultant n'est pas nécessairement de hauteur uniforme quoique nécessairement de hauteur suffisante sur toute sa longueur $[a, b)$ pour restreindre les tâches sur lesquelles l'algorithme n'a pas encore itérées.

Soit t_{limite}, le temps limite d'application de la règle pour une tâche i selon qu'elle répond du cas 1 ou du cas 2 : si $lst_i \geq ect_i$ alors $t_{limite} = ect_i$ et si $lst_i < ect_i$ alors $t_{limite} = lst_i$. À l'aide de l'arbre AVL, on recherche l'intervalle contraignant $[a, b)$ tel que $a < t_{limite}$ et a est maximum. Puisque l'arbre contient des intervalles disjoints, le temps $t_r = b$ constitue le premier temps $t < b \leq t_r$ où les ressources peuvent débuter la tâche i conjointement avec le *bloc cumulé*. Pour l'ajustement nous avons :

$$lst_i \geq ect_i \text{ (cas 1) : } est'_i = \max(est_i, b) \tag{2.28}$$

$$lst_i < ect_i \text{ (cas 2) : } est'_i = \max(est_i, \min(lst_i, b)) \tag{2.29}$$

La figure 2.15 illustre un filtrage issu du premier cas.

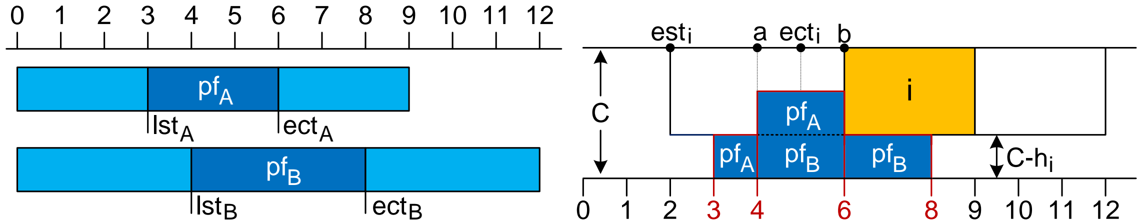


FIGURE 2.15: Illustration d'une instance à trois tâches. À gauche, les tâches A et B et leur partie fixe respective. À droite, l'étape du filtrage de la tâche i dont la partie fixe est nulle. Le temps limite est donc fixé à $t_{limite} = ect_i = 5$. À cette étape, l'arbre ne contient qu'un seul intervalle, $[a, b)$=$[4, 6)$. Il est le seul intervalle du *bloc cumulé* de hauteur suffisante pour restreindre les tâches dont la hauteur $h = h_i$. La recherche retourne l'intervalle $[4, 6)$ puisque 4<5 ($a < t_{limite}$). Le temps de sortie de la tâche i est ajusté à $t = b = 6$. Si l'itération suivante portait sur une tâche j de hauteur $h_j = C$, les intervalles $[3, 4)$ et $[6, 8)$ seraient aboutés à l'intervalle $[4, 6)$. L'arbre ne contiendrait toujours qu'un seul intervalle, $[3, 8)$, de hauteur non-uniforme mais suffisamment haut sur toute sa longueur pour restreindre une tâche i de hauteur $h_i = C$.

L'algorithme 6 applique la règle du Time-Tabling aux tâches de l'ensemble $\mathcal{I}$. Il requiert le bloc cumulé minimum BC assemblé par l'algorithme 5. La liste G est composée d'un maximum de $2n - 1$ éléments. On peut la trier en $O(n \log n)$. L'opération d'insertion (ligne 1) s'effectue en $O(\log n)$, balancement de l'arbre inclus. L'algorithme effectue un maximum de $2n - 1$ insertions. La recherche dans l'arbre (ligne 2) s'effectue en $O(\log n)$. Elle est incluse dans une boucle qui itère sur n tâches. La complexité générale de l'algorithme 6 est de $O(n \log n)$.

Algorithme 6 : FiltrageParTimeTabling ($BC, \mathcal{I}, C$)

$A \leftarrow [-\infty, -\infty + 1)$: Arbre AVL initialisé avec une sentinelle
$G \leftarrow$ Liste des tuples $bc \in BC$ en ordre décroissant des hauteurs h
$g \leftarrow 1$: index du prochain élément de G
for $i \in \mathcal{I}$ *en ordre croissant des hauteurs* **do**
    while $g < |G| \wedge h_{G_g} + h_i > C$ **do**
1       $A \leftarrow A \cup [a, b)_{G_g}$
        $g \leftarrow g + 1$
    if $lst_i < ect_i$ **then**
        $t_{limite} \leftarrow lst_i$
    else
        $t_{limite} \leftarrow ect_i$
2   $ajustement \leftarrow \max_{[a,b) \in A, a < t_{limite}} b$
    if $lst_i < ect_i$ **then**
        $est'_i = \max(est_i, \min(lst_i, ajustement))$
    else
        $est'_i = \max(est_i, ajustement)$

Dans l'algorithme 6, l'ajout (ligne 1) d'un intervalle dans l'arbre est l'opération la plus coûteuse. Notre implémentation vise à réduire le nombre d'instructions pour cette opération. L'algorithme a recours à la structure de données *Union-Find* avec union par rang et compression des chemins [36] pour pister les intervalles déjà insérés dans l'arbre. Chaque tuple bc (voir figure 2.16) est initialisé membre unique de son ensemble. Lorsqu'un intervalle est inséré dans l'arbre, son ensemble mémorise l'adresse mémoire du noeud qu'il occupe. S'il y a aboutement, on unit les deux ensembles associés aux tuples des intervalles que l'on joint. Avant d'insérer un intervalle, on vérifie le statut (inséré ou non) du tuple de gauche et de celui de droite. Si l'un ou l'autre est déjà dans l'arbre, on procède à un aboutement et si les deux voisins sont dans l'arbre, alors on opère un colmatage. Dans les deux cas, la structure *Union-Find* nous dispense de parcourir l'arbre puisque nous connaissons l'adresse mémoire des noeuds en cause. Ainsi, nous aboutons en temps constant et pour le colmatage, nous ne payons que pour balancer l'arbre à partir du noeud supprimé [37]. Quant au coût de la structure de données elle-même, sa complexité est de $O(\alpha(n))$ où $\alpha(n)$ est l'inverse de la fonction d'Ackermann.

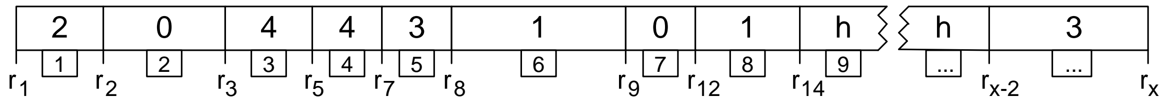


Figure 2.16: Représentation schématique d'une quelconque instance d'ensemble BC dont les tuples bc déterminent des intervalles qui couvre toute la surface s'étalant de r_1 à r_x. Sur la figure on voit les intervalles contigus, leur hauteur et leur indice.

36. Voir les notes 11,22 et 23 du chapitre 3.
37. On supprime le noeud le moins élevé, tel que suggéré par l'analyse de Brass [11] (p.50–58).

2.3.2 Un nouveau modèle pour la contrainte Cumulative

Nous proposons un algorithme [31] original pour la Cumulative. Le modèle repose sur des fondements géométrique comme l'algorithme de Beldiceanu, Carlsson et Poder [8] qui assimile la Cumulative à un rectangle. Sa mécanique est la même que celle de l'algorithme de Vilím [46] dont il pourrait être vu comme l'extension. Pour la détection et l'ajustement, nous procédons en une seule étape adaptant l'idée de l'algorithme de Kameugne, Fotso, Scott et Ngo-Kateu [21] qui trouve les intervalles de slack minimum.

L'algorithme détecte toute situation où une tâche i qui, lorsque débutant à son temps de sortie est_i, entraîne une demande excédentaire en énergie pour un intervalle critique. L'ajustement résultant sur le temps de sortie de la tâche i est celui qui fait en sorte d'éliminer exactement l'excédant. Par conséquent, l'algorithme applique simultanément les règles du Edge-Finding et du Extended-Edge-Finding.

Soit Re_i^Θ, l'*énergie requise* dans l'intervalle $[est_\Theta, lct_\Theta)$ par la tâche i, $lct_i > lct_\Theta$, lorsqu'elle débute à son temps de sortie $S_i = est_i$. Nous appelons *déficit énergétique* et notons De_i^Θ, la quantité de ressources manquante dans l'intervalle $[est_\Theta, lct_\Theta)$ pour y exécuter conjointement la tâche i lorsqu'elle débute à son temps de sortie est_i.

$$Re_i^\Theta = h_i \times \max(0, \min(lct_\Theta, ect_i) - \max(est_\Theta, est_i)) \ , \ lct_i > lct_\Theta \tag{2.30}$$

$$De_i^\Theta = e_\Theta + Re_i^\Theta - C(lct_\Theta - est_\Theta) \tag{2.31}$$

De la section 2.1, nous reformulons la détection par les règles du Edge-Finding (2.2) et du Extended-Edge-Finding (2.5). Soit un intervalle $[est_\Theta, lct_\Theta)$ de slack non-négatif et soit une tâche i avec $lct_i > lct_\Theta$. Si le déficit énergétique De_i^Θ est supérieur à zéro, alors la tâche i est en situation de détection pour l'une ou l'autre des règles selon que son temps de sortie est_i est inférieur ou non à l'intervalle. $\forall\ [est_\Theta, lct_\Theta),\ Sl_\Theta \geq 0$:

$$est_i \geq est_\Theta \wedge lct_i > lct_\Theta \wedge De_i^\Theta > 0 \Rightarrow i \text{ est détectée par } \textit{Edge-Finding}$$
$$est_i < est_\Theta \wedge lct_i > lct_\Theta \wedge De_i^\Theta > 0 \Rightarrow i \text{ est détectée par } \textit{Extended-Edge-Finding}$$

Soit la tâche $j \in \mathcal{I}$ caractérisée par son échéance lct_j. À la section 2.2.2, on introduit le sous-ensemble Ω_j composé des tâches $i \in \mathcal{I}$ dont l'échéance n'est pas supérieure à lct_j. Pour simplifier la notation nous notons ce sous-ensemble Ω [38].

$$\Omega = \{i \in \mathcal{I} \mid lct_i \leq lct_j\} \tag{2.32}$$

38. L'utilité du sous-ensemble Ω_j est algorithmique. Au départ de l'exécution, il contient toutes les tâches du problème et on les lui retire une à une à chaque itération. La démonstration quant à elle, se situe à l'intérieur de la même itération, celle qui porte sur la tâche j.

On sépare les tâches $i \in \mathcal{I} \setminus \Omega$ en deux sous-ensembles, Λ et Ψ, selon leur capacité à terminer avant toutes les tâches du sous-ensemble Ω.

$$\Lambda = \{i \in \mathcal{I} \setminus \Omega \mid ect_i < lct_\Omega\} \qquad \Psi = \{i \in \mathcal{I} \setminus \Omega \mid ect_i \geq lct_\Omega\} \tag{2.33}$$

Pour la détection, nous considérons quatre types d'interaction entre un intervalle $[est_\Theta, lct_\Theta)$ et une tâche i, $lct_i > lct_\Theta$. Ils proviennent des combinaisons des deux alternatives suivantes : $est_i \geq est_\Theta$ ou $est_i < est_\Theta$ et $i \in \Lambda$ ou $i \in \Psi$. Nous référons aux quatre détections par *Edge-Finding-Lambda* (EF^Λ) si $est_i \geq est_\Theta$ et $i \in \Lambda$, par *Edge-Finding-Psi* (EF^Ψ) si $est_i \geq est_\Theta$ et $i \in \Psi$, par *Extended-Edge-Finding-Lambda* (XF^Λ) si $est_i < est_\Theta$ et $i \in \Lambda$ et par *Extended-Edge-Finding-Psi* (XF^Ψ) si $est_i < est_\Theta$ et $i \in \Psi$. La figure 2.17 schématise les quatre cas de détection.

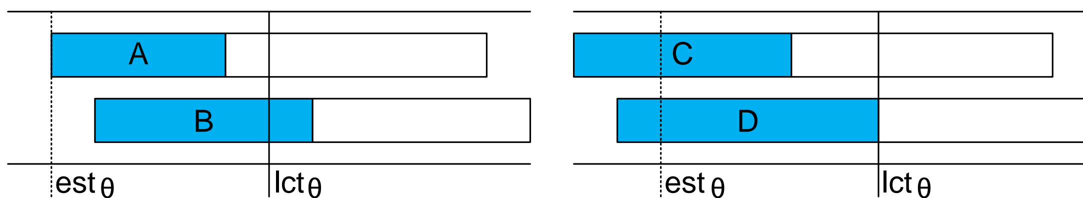


FIGURE 2.17: Quatre tâches qui débutent à leur temps de sortie et leur intervalle critique $[est_\Theta, lct_\Theta)$. À gauche, les tâches A et B débutent dans l'intervalle, leur détection relève du Edge-Finding alors qu'à droite, les tâches C et D débutent avant l'intervalle et leur détection relève du Extended-Edge-Finding. Les tâche A et C appartiennent à Λ, c'est-à-dire qu'elles peuvent terminer avant que l'intervalle termine, alors que B et D appartiennent à Ψ puisqu'il leur est impossible de terminer avant que l'intervalle ne termine. En relation avec l'intervalle $[est_\Theta, lct_\Theta)$, la tâche A relève de la détection EF^Λ, la tâche B du EF^Ψ, la tâche C du XF^Λ et la tâche D du XF^Ψ.

Soit $Rp_i^\Theta = \min(lct_\Theta, ect_i) - \max(est_\Theta, est_i)$, le *temps de traitement requis* dans l'intervalle $[est_\Theta, lct_\Theta)$ par la tâche i, $lct_i > lct_\Theta$, lorsqu'elle débute à son temps de sortie $S_i = est_i$. Nous spécifions Rp_i^Θ pour les quatre détections (voir figure 2.17) :

$$\text{cas } EF^\Lambda : Rp_i^\Theta = ect_i - est_i \tag{2.34}$$

$$\text{cas } EF^\Psi : Rp_i^\Theta = lct_\Theta - est_i \tag{2.35}$$

$$\text{cas } XF^\Lambda : Rp_i^\Theta = ect_i - est_\Theta \tag{2.36}$$

$$\text{cas } XF^\Psi : Rp_i^\Theta = lct_\Theta - est_\Theta \tag{2.37}$$

et pour une tâche i, $lct_i > lct_\Theta$, on a $Re_i^\Theta = h_i Rp_i^\Theta$ pour les quatre cas de détection.

Soit le sous-ensemble Ω de l'équation 2.32 caractérisé par son échéance lct_Ω. Pour chacune des quatre détections, on cherche à trouver la combinaison d'un intervalle $[est_\Theta, lct_\Theta)$, $lct_\Theta = lct_\Omega$, et d'une tâche i, $lct_i > lct_\Omega$, qui maximise le déficit énergétique. De l'équation 2.31 nous avons : $De_i^\Omega = e_\Omega + Re_i^\Omega - C(lct_\Omega - est_\Omega)$. Pour les détections EF^Λ, EF^Ψ, XF^Λ et XF^Ψ on obtient les fonctions qui suivent :

$$\begin{aligned}
\text{cas } EF^\Lambda : \max_i De_i^\Omega &= \max_{\substack{\Theta\subseteq\Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i\in\Lambda \\ est_\Theta \leq est_i}} e_\Theta + (h_i ect_i - h_i est_i) - C lct_\Theta + C est_\Theta \\
\text{cas } EF^\Psi : \max_i De_i^\Omega &= \max_{\substack{\Theta\subseteq\Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i\in\Psi \\ est_\Theta \leq est_i}} e_\Theta + (h_i lct_\Theta - h_i est_i) - C lct_\Theta + C est_\Theta \\
\text{cas } XF^\Lambda : \max_i De_i^\Omega &= \max_{\substack{\Theta\subseteq\Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i\in\Lambda \\ est_i < est_\Theta < ect_i}} e_\Theta + (h_i ect_i - h_i est_\Theta) - C lct_\Theta + C est_\Theta \\
\text{cas } XF^\Psi : \max_i De_i^\Omega &= \max_{\substack{\Theta\subseteq\Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i\in\Psi \\ est_i < est_\Theta}} e_\Theta + (h_i lct_\Theta - h_i est_\Theta) - C lct_\Theta + C est_\Theta
\end{aligned}$$

On veut utiliser l'arbre cumulatif de la section 2.2.2 pour évaluer ces fonctions. Pour ce faire, on doit éliminer les termes $h_i lct_\Theta$ et lct_Θ du côté droit des équations[39]. Soit $Hor = \max_i lct_i$, la plus grande des échéances parmi les tâches de l'ensemble $\mathcal{I}$. Nous appliquons les transformations qui suivent :

$$\begin{aligned}
\max_i De_i^\Omega + C lct_\Omega &= \max_{\substack{\Theta\subseteq\Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i\in\Lambda \\ est_\Theta \leq est_i}} e_\Theta + (h_i ect_i - h_i est_i) + C est_\Theta \\
\max_i De_i^\Omega + C lct_\Omega + h_i(Hor - lct_\Omega) &= \max_{\substack{\Theta\subseteq\Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i\in\Psi \\ est_\Theta \leq est_i}} e_\Theta + (h_i Hor - h_i est_i) + C est_\Theta \\
\max_i De_i^\Omega + C lct_\Omega &= \max_{\substack{\Theta\subseteq\Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i\in\Lambda \\ est_i < est_\Theta < ect_i}} e_\Theta + (h_i ect_i - h_i est_\Theta) + C est_\Theta \\
\max_i De_i^\Omega + C lct_\Omega + h_i(Hor - lct_\Omega) &= \max_{\substack{\Theta\subseteq\Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i\in\Psi \\ est_i < est_\Theta}} e_\Theta + (h_i Hor - h_i est_\Theta) + C est_\Theta
\end{aligned}$$

Les termes en h sont fonction de la hauteur de tâche sur laquelle l'algorithme itère. À l'itération portant sur la hauteur h, on ne filtre que les tâches i de hauteur $h_i = h$. Ainsi, pour les fins de la maximisation, il y a équivalence entre la valeur h et h_i.

39. Comme expliqué à la section 2.2.1 et illustré à la section 2.2.2, l'arbre cumulatif cerne en $O(\log n)$ un intervalle optimal parmi $O(n)$ possibilités. Le mécanisme d'optimisation est le suivant. Sitôt les valeurs contenues dans une feuille modifiées, on met à jour l'arbre de la feuille jusqu'à la racine. Au cours de son exécution, l'algorithme itère sur chacune des tâches en changeant leur statut (déplacement d'un ensemble vers un autre) ce qui entraîne une réinitialisation des valeurs contenues dans leur feuille associée. Or, la tâche sur laquelle on itère est justement celle qui marque la borne supérieure lct_Θ. Ainsi, la valeur lct_Θ change constamment. Par conséquent, toute feuille qui contiendrait des valeurs fonction de lct_Θ serait mise à jour à chacune des itérations, entraînant une cascade de mises à jour jusqu'à la racine.

Après simplifactions et manipulations, on reconnait dans les expressions à droite du signe d'égalité la forme des fonctions $Env_\Omega = \max_\theta(Cest_\Theta + e_\Theta)$ de la section 2.2.2 et celle de $Env_\Omega^h = \max_\theta((C - h)est_\Theta + e_\Theta)$ de la section 2.2.3. Pour les quatre cas de détections, nous définissons une fonction d'enveloppe qui peut être maximisée par l'entremise de l'arbre cumulatif. Elle permet de détecter l'interaction entre un intervalle optimal $[est_\Theta, lct_\Theta)$, $lct_\Theta = lct_\Omega$, et sa tâche i la plus contraignante, $lct_i > lct_\Theta$, $i \in \Lambda$ ou $i \in \Psi$, qui génère le déficit énergétique De_i^Θ maximal. On obtient :

$$Env^\Lambda = \max_{\substack{\Theta \subseteq \Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i \in \Lambda \\ est_\Theta \leq est_i}} Cest_\Theta + e_\Theta + h_i p_i \tag{2.38}$$

$$Env^\Psi = \max_{\substack{\Theta \subseteq \Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i \in \Psi \\ est_\Theta \leq est_i}} Cest_\Theta + e_\Theta + h_i(Hor - est_i) \tag{2.39}$$

$$XEnv^\Lambda = \max_{\substack{\Theta \subseteq \Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i \in \Lambda \\ est_i < est_\Theta < ect_i}} (C - h)est_\Theta + e_\Theta + h_i ect_i \tag{2.40}$$

$$XEnv^\Psi = \max_{\substack{\Theta \subseteq \Omega \\ lct_\Theta = lct_\Omega}} \max_{\substack{i \in \Psi \\ est_i < est_\Theta}} (C - h)est_\Theta + e_\Theta + h_i Hor \tag{2.41}$$

Pour maximiser les quatre fonctions par l'entremise de l'arbre cumulatif, nous nous servons de onze quantités. Pour une feuille (f), les quantités sont fonction des valeurs d'énergie et d'enveloppe de sa tâche associée i :

$$e_f = \begin{cases} e_i & \text{si } i \in \Omega \\ 0 & sinon \end{cases} \qquad Env_f = \begin{cases} Cest_i + e_i & \text{si } i \in \Omega \\ -\infty & \text{sinon} \end{cases} \tag{2.42}$$

$$e_f^\Lambda = \begin{cases} e_i & \text{si } i \in \Lambda \\ -\infty & \text{sinon} \end{cases} \qquad e_f^\Psi = \begin{cases} h_i(Hor - est_i) & \text{si } i \in \Psi \\ -\infty & \text{sinon} \end{cases} \tag{2.43}$$

$$xe_f^\Lambda = \begin{cases} h_i ect_i & \text{si } i \in \Lambda \\ -\infty & \text{sinon} \end{cases} \qquad xe_f^\Psi = \begin{cases} h_i Hor & \text{si } i \in \Psi \\ -\infty & \text{sinon} \end{cases} \tag{2.44}$$

$$Env_f^\Lambda = \begin{cases} Cest_i + e_i & \text{si } i \in \Lambda \\ -\infty & \text{sinon} \end{cases} \qquad Env_f^\Psi = \begin{cases} Cest_i + e^\Psi & \text{si } i \in \Psi \\ -\infty & \text{sinon} \end{cases} \tag{2.45}$$

$$Env_f^h = \begin{cases} (C - h)est_i + e_i & \text{si } i \in \Omega \\ -\infty & \text{sinon} \end{cases} \tag{2.46}$$

$$XEnv_f^\Lambda = -\infty \qquad XEnv_f^\Psi = -\infty \tag{2.47}$$

Les fonctions $XEnv_f^\Lambda$ et $XEnv_f^\Psi$ trouve une valeur maximale de l'interaction entre un intervalle et une tâche qui le précède. Les feuilles étant la représentation d'une unique tâche, ces fonctions n'y sont pas définies et leur valeur est initialisée à $-\infty$. Il en va autrement pour les fonctions Env_v^Λ et Env_v^Ψ car une tâche $i \in \Lambda$ peut interagir avec un intervalle critique dont la borne inférieure est_Θ coïncide avec son temps de sortie est_i.

Pour un noeud interne (v), les quantités sont évaluées à partir des quantités contenues dans son fils de gauche (g) et de celles contenues dans son fils de droite (d) :

$$e_v = e_g + e_d \tag{2.48}$$

$$Env_v = \max(Env_g + e_d, Env_d) \tag{2.49}$$

$$Env_v^h = \max(Env_g^h + e_d, Env_d^h) \tag{2.50}$$

$$e_v^\Lambda = \max(e_g^\Lambda + e_d, e_g + e_d^\Lambda) \tag{2.51}$$

$$xe_v^\Lambda = \max(xe_g^\Lambda, xe_d^\Lambda) \tag{2.52}$$

$$e_v^\Psi = \max(e_g^\Psi + e_d, e_g + e_d^\Psi) \tag{2.53}$$

$$xe_v^\Psi = \max(xe_g^\Psi, xe_d^\Psi) \tag{2.54}$$

$$Env_v^\Lambda = \max(Env_g^\Lambda + e_d, Env_g + e_d^\Lambda, Env_d^\Lambda) \tag{2.55}$$

$$Env_v^\Psi = \max(Env_g^\Psi + e_d, Env_g + e_d^\Psi, Env_d^\Psi) \tag{2.56}$$

$$XEnv_v^\Lambda = \max(XEnv_g^\Lambda + e_d, xe_g^\Lambda + Env_d^h, XEnv_d^\Lambda) \tag{2.57}$$

$$XEnv_v^\Psi = \max(XEnv_g^\Psi + e_d, xe_g^\Psi + Env_d^h, XEnv_d^\Psi) \tag{2.58}$$

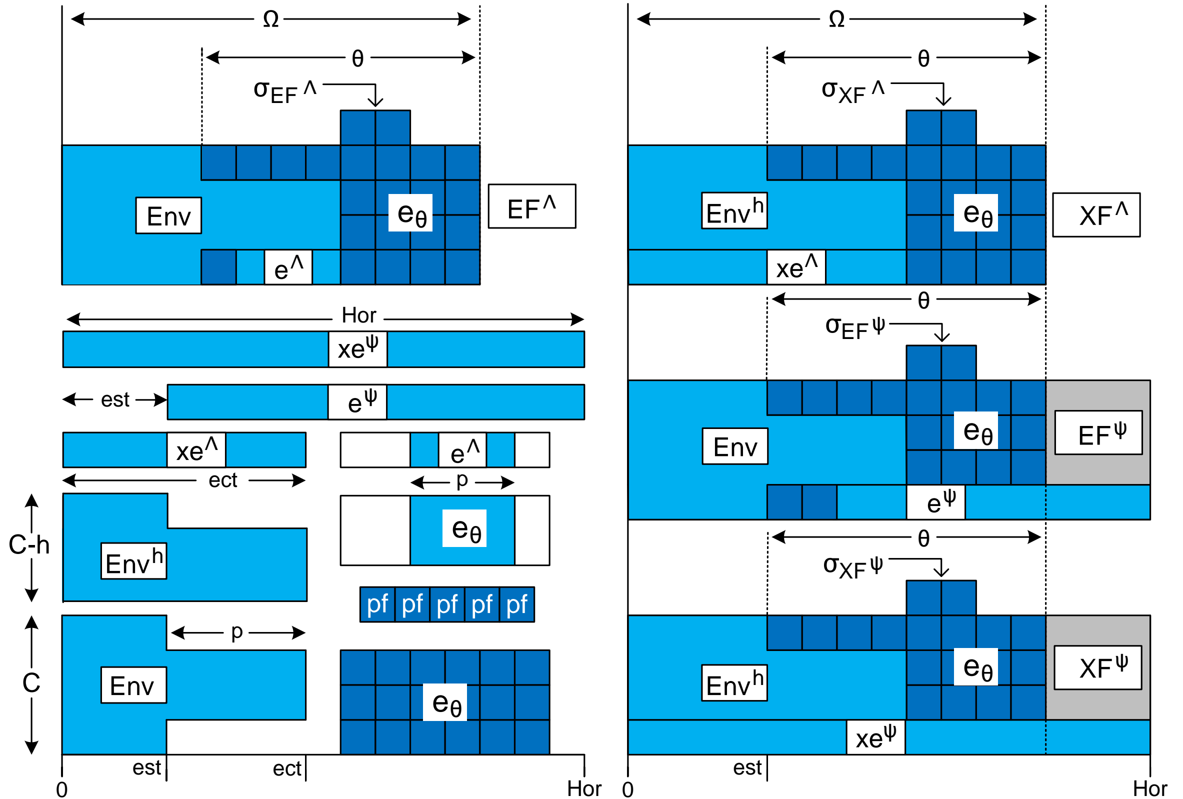


FIGURE 2.18: Schématisation des valeurs d'enveloppe et d'énergie utilisées par l'arbre cumulatif. Les carrés foncés représentent le découpage en unité de l'énergie des tâches confinées à $[est_\Theta, lct_\Theta)$ mis à part l'énergie de la tâche responsable de l'enveloppe (Env ou Env^h).

À la racine de l'arbre cumulatif, les valeurs Env^{Λ}, Env^{Ψ}, $XEnv^{\Lambda}$ et $XEnv^{\Psi}$ contiennent la valeur maximale de l'enveloppe des quatre détections à chacune des itérations (tout le bleu sur la figure 2.18). Les enveloppes nous permettent de détecter tout déficit énergétique. Nous les notons respectivement $\sigma_{EF^{\Lambda}}$, $\sigma_{EF^{\Psi}}$, $\sigma_{XF^{\Lambda}}$, $\sigma_{XF^{\Psi}}$. Par exemple, l'instance schématisée sur la figure 2.18 montre un déficit énergétique de 2 unités pour chacune des quatre détections. Les déficits énergétiques maximaux sont décelés en comparant les valeurs d'enveloppe contenues à la racine de l'arbre avec la capacité totale de la ressource sur l'intervalle $[0, lct_{\Theta})$. En considérant les modifications apportées aux équations 2.39 et 2.41 :

$$\sigma_{EF^{\Lambda}} = Env^{\Lambda} - Clct_{\Omega} \tag{2.59}$$

$$\sigma_{EF^{\Psi}} = Env^{\Psi} - Clct_{\Omega} \tag{2.60}$$

$$\sigma_{XF^{\Lambda}} = XEnv^{\Lambda} - Clct_{\Omega} - h(Hor - lct_{\Omega}) \tag{2.61}$$

$$\sigma_{XF^{\Psi}} = XEnv^{\Psi} - Clct_{\Omega} - h(Hor - lct_{\Omega}) \tag{2.62}$$

Un déficit énergétique positif (voir équation 2.31) résulte d'un manque d'énergie dans l'intervalle $[est_{\Theta}, lct_{\Theta})$ pour traiter conjointement toutes les tâches confinées à l'intervalle et la tâche i lorsque celle-ci débute à son temps de sortie est_i. Nous appelons *segment excédentaire* et notons sx_i^{Θ}, le temps minimum d'exécution de la tâche i dans l'intervalle $[est_{\Theta}, lct_{\Theta})$ qui doit être repoussé hors de l'intervalle pour éliminer le déficit énergétique. Nous avons :

$$sx_i^{\Theta} = \left\lceil \frac{De_i^{\Theta}}{h_i} \right\rceil \text{ si } De_i^{\Theta} > 0 \qquad\qquad sx_i^{\Theta} = 0 \text{ sinon} \tag{2.63}$$

Nous appelons *segment conjoint* et notons sc_i^{Θ}, la portion de la durée de traitement p_i de la tâche i, $lct_i > lct_{\Theta}$ qui peut être exécutée conjointement à l'intervalle $[est_{\Theta}, lct_{\Theta})$. Pour éliminer le déficit énergétique dans l'intervalle $[est_{\Theta}, lct_{\Theta})$, on ajuste le temps de sortie de la tâche i de telle sorte que son temps de traitement dans l'intervalle $[est_{\Theta}, lct_{\Theta})$ soit égal à la longueur du segment conjoint sc_i^{Θ}.

$$sc_i^{\Theta} = Rp_i^{\Theta} - sx_i^{\Theta} \qquad\qquad est' = lct_{\Theta} - sc_i^{\Theta} \tag{2.64}$$

Nous précisons l'ajustement appliqué au temps de sortie est_i selon la détection. Des équations du traitement requis Rp_i^{Θ} (2.34) à (2.37), de celle du segment excédentaire sx_i^{Θ} (2.63) et des équations du segment conjoint sc_i^{Θ} et de l'ajustement est'_i (2.64) :

$$\sigma_{EF^{\Lambda}} > 0 \Rightarrow est'_i = lct_{\Theta} - p_i + \left\lceil \frac{\sigma_{EF^{\Lambda}}}{h_i} \right\rceil \tag{2.65}$$

$$\sigma_{EF^{\Psi}} > 0 \Rightarrow est'_i = est_i + \left\lceil \frac{\sigma_{EF^{\Psi}}}{h_i} \right\rceil \tag{2.66}$$

$$\sigma_{XF^{\Lambda}} > 0 \Rightarrow est'_i = lct_{\Theta} - ect_i + est_{\Theta} + \left\lceil \frac{\sigma_{XF^{\Lambda}}}{h_i} \right\rceil \tag{2.67}$$

$$\sigma_{XF^{\Psi}} > 0 \Rightarrow est'_i = est_{\Theta} + \left\lceil \frac{\sigma_{XF^{\Psi}}}{h_i} \right\rceil \tag{2.68}$$

Algorithme 7 : ExtendedEdgeFinder($\mathcal{I}$)

$Hor \leftarrow \max_{i \in \mathcal{I}} lct_i$

for $h \in \{h_i \mid i \in \mathcal{I} \wedge ect_i < lct_i\}$ **do**

1 $\Omega \leftarrow \mathcal{I}$

$\Lambda \leftarrow \emptyset,\ \Psi \leftarrow \emptyset$

for $j \in \mathcal{I}$ *En ordre décroissant des échéances* lct_j **do**

2 **if** $Env > Clct_j$ **then return** Horaire infaisable

3 $\Delta \leftarrow \{i \in \Lambda \mid ect_i \geq lct_j\}$

4 $\Lambda \leftarrow \Lambda \setminus \Delta$

5 $\Psi \leftarrow \Psi \cup \Delta \setminus \{i \in \Psi \mid est_i \geq lct_j\}$

repeat

6 $\sigma_{EF^\Lambda} \leftarrow Env^\Lambda - Clct_j$

7 $\sigma_{XF^\Lambda} \leftarrow XEnv^\Lambda - Clct_j$

8 $\sigma_{EF^\Theta} \leftarrow Env^\Theta - Clct_j - h(Hor - lct_j)$

9 $\sigma_{XF^\Theta} \leftarrow XEnv^\Theta - Clct_j - h(Hor - lct_j)$

$m \leftarrow \max\{\sigma_{XF^\Lambda}, \sigma_{XF^\Theta}, \sigma_{EF^\Lambda}, \sigma_{EF^\Psi}\}$

if $\sigma_{XF^\Lambda} = m > 0$ **then**

Soit $i \in \Lambda$ la tâche responsable de la valeur xe^Λ dans $XEnv^\Lambda$

Soit $k \in \Omega$ la tâche responsable de la borne est_k dans Env^h

$est'_i \leftarrow lct_j - (ect_i - est_k) + \left\lceil \frac{\sigma_{XF^\Lambda}}{h_i} \right\rceil$

$\Lambda \leftarrow \Lambda \setminus \{i\}$

else if $\sigma_{XF^\Theta} = m > 0$ **then**

Soit $i \in \Psi$ la tâche responsable de la valeur xe^Ψ dans $XEnv^\Psi$

Soit $k \in \Omega$ la tâche responsable de la borne est_k dans Env^h

$est'_i \leftarrow est_k + \left\lceil \frac{\sigma_{XF^\Theta}}{h_i} \right\rceil$

$\Psi \leftarrow \Psi \setminus \{i\}$

else if $\sigma_{EF^\Lambda} = m > 0$ **then**

Soit $i \in \Lambda$ la tâche responsable de la valeur e^Λ dans Env^Λ

$est'_i \leftarrow lct_j - p_i + \left\lceil \frac{\sigma_{EF^\Lambda}}{h_i} \right\rceil$

$\Lambda \leftarrow \Lambda \setminus \{i\}$

else if $\sigma_{EF^\Psi} = m > 0$ **then**

Soit $i \in \Psi$ la tâche responsable de la valeur e^Ψ dans Env^Ψ

$est'_i \leftarrow est_i + \left\lceil \frac{\sigma_{EF^\Theta}}{h_i} \right\rceil$

$\Psi \leftarrow \Psi \setminus \{i\}$

until $m \leq 0$

10 **if** $h_j = h$ **then** $\Lambda \leftarrow \Lambda \cup \{j\}$

11 $\Omega \leftarrow \Omega \setminus \{j\}$

L'algorithme 7 accepte un ensemble de tâches $\mathcal{I}$ en entrée et procède au filtrage de leur temps de sortie en vertu des quatre règles de détection. Il itère sur les k différentes hauteurs parmi les tâches de $\mathcal{I}$, $k = |\{h_i \mid i \in \mathcal{I}\}|$. Les opérations qui suivent se passent à l'intérieur d'une itération sur la hauteur h. L'algorithme forme l'ensemble Ω avec toutes les tâches de $\mathcal{I}$ et construit l'arbre cumulatif (ligne 1). Les tâches sont associées, une à une, aux feuilles de l'arbre

cumulatif en partant de la gauche en ordre croissant des temps de sortie est_i (voir section 2.2.1). On départage les égalités par les échéances (lct), la plus petite en premier [40]. L'algorithme itère sur les tâches de l'ensemble Ω en ordre décroissant des échéances. Nous appelons j la tâche sur laquelle l'algorithme itère. À la fin de l'itération, la tâche j est déplacée (ligne 10) vers l'ensemble Λ si elle est de hauteur égale à la hauteur en cours de traitement ($h_j = h$). Sinon, la tâche j est retirée de Ω (ligne 11). Les tâches déplacées vers Λ à cette étape (ligne 10) sont celles parmi lesquelles on cherche l'interaction maximale avec l'intervalle optimal $[est_\Theta, lct_\Theta)$ aux cours des itérations subséquentes. Elles sont alors référées par la lettre i le cas échéant. L'algorithme effectue un Overload Check à la ligne 2 (voir les explications détaillées de la section 2.2.1) qui vérifie la consistance énergétique du CuSP (voir section 2.1). La mécanique des lignes 3 à 5 est la suivante. L'ensemble Λ contient toutes les tâches j devenues i (ligne 10) lors d'une itération antérieure, dont le temps minumum de terminaison ect_i est inférieur à l'échéance en cours d'itération lct_j. Les tâches $i \in \Lambda$ qui satisfont $ect_i \geq lct_j$ sont déplacées vers Ψ. On retire aussi les tâches $i \in \Psi$ ne pouvant plus interagir avec un intervalle optimal de Ω, $est_i \geq lct_j = lct_\Omega$. La boucle *repeat* constitue la phase de filtrage de l'algorithme. Les quatre valeurs d'enveloppe contenues à la racine de l'arbre sont comparées à la capacité totale de la ressource (lignes 6 à 9) pour détecter tout déficit énergétique positif. Lorsqu'un tel déficit est détecté, on trouve la feuille $i \in \Lambda \cup \Psi$ qui en est responsable. Pour ce faire, on part de la racine et on parcourt l'arbre en suivant la trace de l'énergie xe^Λ, xe^Ψ, e^Λ ou e^Ψ (dans l'ordre des blocs if) jusqu'à la feuille qui la porte. Sa tâche associée est alors filtrée en fonction de la valeur du déficit énergétique. Après filtrage, la tâche i est retirée de son ensemble et on procède à une mise à jour de l'arbre dans le but de détecter un autre déficit énergétique positif. Pour les détections provenant du Extended-Edge-Finding, la valeur de la borne inférieure est_Θ de l'intervalle optimal est requise pour le filtrage. On procède de façon similaire en pistant à partir de la racine de l'arbre, la feuille $\{k\}$ dont le temps de sortie de sa tâche associée k cerne l'intervalle optimal ($est_k = est_\Theta$) de l'enveloppe (Env^h).

À la ligne 3 l'algorithme doit comparer les temps minimum de terminaison des tâches de Λ avec le nouvel échéance lct_j. On implémente un monceau (priority queue) et l'opération s'effectue en $O(\log n)$ à chacune des entrées ou sorties. L'algorithme procède de la même façon à la ligne 5 pour les tâches de Ψ.

40. Le bris d'égalité est arbitraire. Il n'influence pas l'exactitude de l'algorithme. Toutefois, il nous assure que la tâche la plus contraignante i soit détectée par le bon cas selon la position relative de son temps de sortie est_i et le début de l'intervalle optimal est_Θ. Lorsque i débute avec l'intervalle, $est_i = est_\Theta$, les deux détections mènent au même résultat. Or nous avons statué qu'en pareille circonstance, le déficit énergétique est détecté par les cas découlant du Edge-Finding, se conformant ainsi aux travaux antérieurs dans le domaine (voir la règle de la section 2.1 tel qu'énoncée par Nuijten [29] et Baptiste et al. [6]). Cet algorithme est le premier à appliquer simultanément les règles du Edge-Finding et du Extended-Edge-Finding et il était d'usage d'appliquer les deux règles aux cas où $est_i = est_\Theta$.

L'algorithme 7 itère sur les k différentes hauteurs parmi les tâches de $\mathcal{I}$. À l'intérieur de l'itération sur les hauteurs, chaque tâche i suit l'un des chemins suivants :

$$i \rightarrow \Omega \rightarrow \text{rebut}$$
$$i \rightarrow \Omega \rightarrow \Lambda$$
$$i \rightarrow \Omega \rightarrow \Lambda \rightarrow \text{filtrage} \rightarrow \text{rebut}$$
$$i \rightarrow \Omega \rightarrow \Psi$$
$$i \rightarrow \Omega \rightarrow \Psi \rightarrow \text{rebut}$$
$$i \rightarrow \Omega \rightarrow \Psi \rightarrow \text{filtrage} \rightarrow \text{rebut}$$
$$i \rightarrow \Omega \rightarrow \Lambda \rightarrow \Psi$$
$$i \rightarrow \Omega \rightarrow \Lambda \rightarrow \Psi \rightarrow \text{rebut}$$
$$i \rightarrow \Omega \rightarrow \Lambda \rightarrow \Psi \rightarrow \text{filtrage} \rightarrow \text{rebut}$$

Chacun des déplacements indiqués ci-haut s'effectue en $O(\log n)$. Il y a n tâches dans le problème. La complexité générale de l'algorithme est donc de $O(kn \log n)$ où $k = |\{h_i \mid i \in \mathcal{I}\}|$.

2.3.3 Time-Table-Extended-Edge-Finding

Le Time-Table-Edge-Finding consiste à appliquer la règle du Edge-Finding en tenant compte de l'énergie des parties fixes des tâches de $\mathcal{I}$. Il en est de même de l'application du Time-Table-Extended-Edge-Finding.

De l'équation 2.8 : soit e^f_Ω l'énergie des tâches du sous-ensemble Ω additionnée à l'énergie du bloc cumulé dans l'intervalle $[est_\Omega, lct_\Omega)$.

$$e^f_\Omega = e_\Omega + f(\mathcal{I} \setminus \Omega, [est_\Omega, lct_\Omega)) = e_\Omega - f(\Omega, [est_\Omega, lct_\Omega)) + f(\mathcal{I}, [est_\Omega, lct_\Omega)) \tag{2.69}$$

Nous formons 3 ensembles de tâches T_i, T_d et T_v. T_i est composé des tâches de $\mathcal{I}$ qui n'ont pas de parties fixes ($ect_i \leq lst_i$). T_d est composé des tâches de $\mathcal{I}$ qui satisfont $lst_i < ect_i$ et auxquelles on a retiré la partie fixe. Les tâches i qui satisfont $lst_i < ect_i$ sont remplacées par une *tâche diminuée* i^d de durée $p_{i^d} = p_i - ect_i + lst_i$. Les tâches diminuées conservent le temps de sortie $est_{i^d} = est_i$, l'échéance $lct_{i^d} = lct_i$ et la hauteur $h_{i^d} = h_i$ de leur tâche originelle. On somme toutes les parties fixes pour obtenir le bloc cumulé, $f(\mathcal{I}, [est_I, lct_I))$. Soit R, une liste strictement croissante $r_1, \ldots, r_x$, composée du temps de sortie est_i, du temps minimum de terminaison ect_i, du temps maximum de départ lst_i et de l'échéance lct_i de toutes les tâches de $\mathcal{I}$. Pour chaque élément $r_x \in R$ qui correspond à une hauteur non-nulle du bloc cumulé $f(\mathcal{I}, r_x) > 0$, on crée la tâche virtuelle v_x avec les propriétés suivantes : $est_{v_x} = r_x$, $lct_{v_x} = r_{x+1}$, $p_{v_x} = r_{x+1} - r_x$ et $h_{v_x} = f(\mathcal{I}, r_x)$. On regroupe toutes les tâches virtuelles dans l'ensemble T_v. La figure 2.19 illustre la formation des ensembles T_d et T_v.

Algorithme 8 : DécompositionDesTâches ($\mathcal{I}$,C)

```
R ← trier{est_i, ect_i, lst_i, lct_i} ∀ i ∈ I    // Liste strictement croissante
c_k ← 0 ∀k ∈ 1..|R|
T_i ← ∅
T_d ← ∅
T_v ← ∅
for i ∈ I do
    if ect_i > lst_i then
        a ← index(lst_i, R)
        b ← index(ect_i, R)
        c_a ← c_a + h_i
        c_b ← c_b − h_i
        T_d ← T_d ∪ {Tache(est = est_i, lct = lct_i, h = h_i, p = p_i − ect_i + lst_i)}
    else
        T_i ← T_i ∪ {i}
hauteur ← 0
for k = 1..|R| − 1 do
    hauteur ← hauteur + c_k
    if hauteur > C then Horaire infaisable
    if hauteur > 0 then
        T_v ← T_v ∪ {Tache(est = r[k], lct = r[k + 1], h = hauteur, p = r[k + 1] − r[k])}
return (T_i ∪ T_d ∪ T_v)
```

L'algorithme 8 accepte un ensemble de tâches $\mathcal{I}$ et la capacité de la ressource C en entrée et retourne les ensembles T_i, T_d et T_v. La liste R contient un maximum de $4n$ éléments. On peut la trier en $O(n \log n)$. L'algorithme itère sur n tâches. Au sein d'une itération, les fonctions $index(lst_i)$ et $index(est_i)$ s'exécute en $O(\log n)$. Construire chacune des tâches de T_i, T_d ou T_v s'effectue en temps constant. T_i et T_d contiennent un maximum de n tâches. T_v contient un maximum de $4n - 1$ tâches. La complexité générale de l'algorithme 8 est de $O(n \log n)$.

Soit $Te_\Omega^H = \sum_{h \in H | est_\Omega \leq est_h \wedge lct_h \leq est_\Omega} e_h$, l'énergie totale des tâches de l'ensemble H confinées à l'intervalle $[est_\Omega, lct_\Omega)$. De l'équation 2.69 on obtient :

$$e_\Omega - f(\Omega, [est_\Omega, lct_\Omega)) = Te_\Omega^{T_i} + Te_\Omega^{T_d} \qquad e_\Omega^f = Te_\Omega^{T_i} + Te_\Omega^{T_d} + Te_\Omega^{T_v}$$

Nous proposons d'appliquer les règles du Time-Table-Edge-Finding et du Time-Table-Extended-Edge-Finding en exécutant l'algorithme 7 sur l'union des trois ensemble T_i, T_d et T_v issus de la décomposition des tâches de $\mathcal{I}$. À la section 2.3.1, nous avons vu que les bornes des intervalles critiques pour le Time-Tabling sont constituées du temps minimum de terminaison ect_i et du temps maximum de départ lst_i des tâches de $\mathcal{I}$. Les mêmes bornes significatives de la composante Time-Tabling des règles conjuguées proviennent, grâce à la décomposition, des tâches virtuelles de l'ensemble T_v. La décomposition de $\mathcal{I}$ génère un maximum de $5n$ tâches. La complexité de $O(kn \log n)$ de l'algorithme 7 n'est pas altérée par la décomposition proposée.

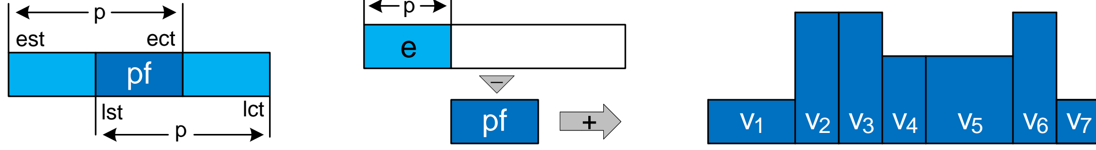


FIGURE 2.19: À gauche, une tâche avec une partie fixe. Au centre, on retire à une tâche sa partie fixe en diminuant son temps de traitement de la longueur de la partie fixe. Toutes les tâches ainsi diminuées forment l'ensemble T_d. À droite, un hypothétique bloc cumulé résultat de la sommation de toutes les parties fixes. Chaque point du bloc cumulé de hauteur non-nulle qui correspond à un élément de la liste R délimite une tâche virtuelle v_x. Toutes les tâches virtuelles ainsi créées forment l'ensemble T_v.

Pour appliquer les règles du Time-Table-Edge-Finding et du Time-Table-Extended-Edge-Finding sur l'ensemble de tâches $\mathcal{I}$, on décompose d'abord les tâches avec un appel à l'algorithme 8 qui retourne trois ensembles de tâches. On exécute ensuite l'algorithme 7 sur l'union des trois ensembles $T_i \cup T_d \cup T_v$. Les tâches de T_v ne peuvent être filtrées. Pour les tâches de T_i, on procède à leur filtrage normalement. Si l'algorithme de filtrage conclut à l'ajustment du temps de sortie d'une tâche $i^d \in T_d$, on procède au filtrage de sa tâche originelle $i \in \mathcal{I}$ comme suit. On détermine la longueur de la partie fixe de la tâche i qui s'exécute à l'intérieur de l'intervalle optimal $[est_\Theta, lct_\Theta)$. Cette longueur est ensuite ajoutée à celle du segment conjoint de sa tâche diminuée $sc_{i^d}^{\Theta}$ pour déterminer l'ajustement à appliquer. Considérant qu'un déficit énergétique positif implique que la tâche la plus contraignante requiert de la ressource à l'intérieur de l'intervalle optimal lorsqu'elle débute à son temps de sortie. Considérant aussi que le temps minimum de terminaison d'une tâche diminuée ect_{i^d} coïncide avec le début de la partie fixe de sa tâche originelle (voir figure 2.19). La partie fixe pf_i de la tâche originelle i d'une tâche diminuée $i^d \in T_d$ pour laquelle l'algorithme 7 déduit un ajustement du temps de sortie, débute nécessairement après le début de l'intervalle optimal est_Θ.

$$De_{i^d}^{\Theta} > 0 \Rightarrow ect_{i^d} > est_\Theta \quad \text{et} \quad lst_i = ect_{i^d} \quad \text{alors} \quad De_{i^d}^{\Theta} > 0 \Rightarrow lst_i > est_\Theta$$

Soit $Rpf_{i^d}^{\Theta}$ la longueur de la portion de la partie fixe de la tâche originelle i qui s'exécute à l'intérieur de l'intervalle optimal $[est_\Theta, lct_\Theta)$, nous avons :

$$Rpf_{i^d}^{\Theta} = \max(0, \min(ect_i, lct_\Theta) - lst_i) \qquad est'_i = lct_\Theta - sc_{i^d}^{\Theta} - Rpf_{i^d}^{\Theta} \tag{2.70}$$

2.3.4 Expérimentations

Nous testons nos différentes versions de l'algorithme sur le banc d'essai PSLIB (Projection Scheduling Problem Library) [22]. Plus précisément, nous résolvons des instances du Single-Mode resource-constrained project scheduling problem (SMRCPSP). Ces instances sont formées d'un ensemble de tâches qui peuvent terminer avant un horizon donné. Des ressources cumulatives sont allouées, chacune ayant une capacité spécifique C. Chaque tâche de l'ensemble est caractérisée par une durée de traitement et un profil de consommation (hauteur h)

de chacune des ressources allouées. Les tâches sont accompagnées d'une liste de 0 à 3 tâches, leurs successeurs, qui ne doivent pas débuter avant qu'elles soient terminées.

Notre modèle principal est basé sur deux contraintes. Nous utilisons une contrainte de précédence pour s'assurer que l'ordre des successeurs est respecté. Nous utilisons aussi une contrainte CUMULATIVE pour chacune des ressources cumulatives allouées. Les contraintes CUMULATIVE procèdent à des ajustements sur les temps de sortie des tâches pour s'assurer que leur capacité C n'est jamais excédée. Nous fixons le makespan à la meilleure valeur reportée sur le banc d'essai pour chacune des instances. Nous avons recours à des variables binaires pour forcer une précédence entre chaque paire de tâches. Nous branchons d'abord sur les paires de tâches avec un profil de consommation de ressources le plus similaire et le plus grand temps de traitement.

Nous utilisons le solveur Choco version 2.1.5 sur un ordinateur avec un processeur AMD Athlon II P340 Dual-Core séquencé à 2.20 GHz. Nous exécutons simultanément deux instances, une par noyau. Nous comparons le propagateur de Choco qui applique la règle du Edge-Finding et du Extended Edge-Finding avec l'algorithme de Mercier et Van Hentenryck [28] et la règle du Time-Tabling avec l'algorithme de Beldiceanu et Carlsson [9] avec notre propagateur. Nous notons EEF+TT, la combinaison de l'algorithme 6 et de l'algorithme 7 et TTEEF lorsque l'algorithme 7 est utilisé avec une décomposition préalable des tâches, l'algorithme 8. Le tableau 2.1 présente les résultats.

Banc d'essai			Choco			EEF+TT			TTEEF		
n	nb	tl	rsl	r-a	temps	rsl	r-a	temps	rsl	r-a	temps
30	480	10	364	8757	223	377	8757	50	377	8379	54
60	480	20	332	3074	1527	340	3074	269	341	2861	291
90	480	50	321	5024	5522	327	5024	857	329	4635	913

TABLE 2.1: Les résultats expérimentaux. La section Banc d'essai indique le nombre de tâches n, le nombre d'instances (nb) et le temps limite de calcul (tl). Pour chaque algorithme de filtrage, nous montrons le nombre d'instances résolues (rsl). Nous montrons aussi la nombre total de retours-arrière (r-a) et le temps total (temps) requis pour résoudre les instances communément résolues par les trois algorithmes.

Le nombre de retours-arrière pour Choco et EEF+TT est le même car les propagateurs appliquent exactement le même filtrage. La colonne temps montre que l'écart de performance se creuse avec la taille des instances. Ce qui est cohérent avec la complexité des propagateurs en cause. La section TTEEF présente des résultats décevants en ce qui a trait au nombre d'instances résolues. Toutefois, le Banc d'essai PSLIB est hautement disjonctif (voir la note 2 et la section 2), ce qui ne met pas en valeur la règle conjuguée du Time-Table-Extended-Edge-Finding. Il est probable qu'un banc d'essai hautement cumulatif permettrait de mieux mettre en valeur l'algorithme qui applique la règle conjuguée.

Chapitre 3

Ordonnancement de tâches identiques

Dans ce chapitre, nous nous intéressons à l'ordonnancement et au filtrage des bornes de tâches de durée fixe et de hauteur unitaire ($p_i = k, h_i = 1 \; \forall i \in \mathcal{I}$). La capacité du système correspond au nombre de ressources identiques aptes à réaliser une tâche. Nous désignons par *machines* ces ressources et notons leur nombre par m. Nous présentons des problèmes dans lesquels les tâches sont réalisées entièrement par la même machine sans interruption.

3.1 Background

Nous posons d'abord le problème d'ordonnancer un ensemble de tâches de durée unitaire ($p = 1$) de manière à en minimiser le temps total de traitement. Si tous les temps de sortie sont des valeurs entières, le problème peut être résolu en $O(n \log n)$ [20] peu importe le nombre de machines en utilisant la méthode *EDD* (Earliest Due Date). La règle est la suivante : dès qu'une machine se libère, on démarre la tâche ayant l'échéance la plus serrée parmi celles éligibles en raison de leur temps de sortie. Le résultat se généralise aux cas où les temps de sortie sont multiples de la durée (p), pour toutes durées et pour tout nombre de machines.

$$\forall p, \forall m, \forall i \in \mathcal{I}, \exists k, est_i = kp \;\Rightarrow\; EDD \text{ parvient à minimiser le } makespan$$

Si les temps de sortie ne sont pas tous multiples de la durée, la méthode peut faillir même dans les problèmes à machine unique. La figure 3.1 illustre une telle instance.

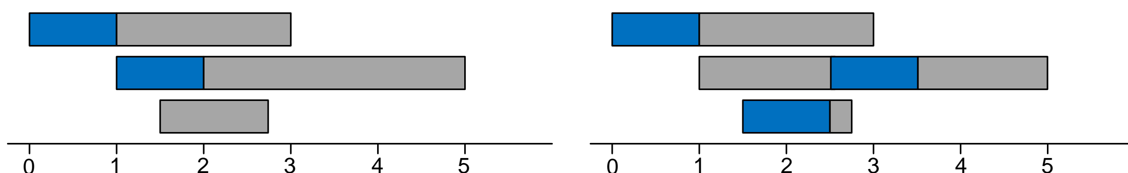


FIGURE 3.1: La solution obtenue par EDD à gauche et la solution optimale à droite.

Simons [41] présente un algorithme en $O(n^2 \log n)$ pour résoudre le problème à une seule machine. Garey et al. [17] proposent de résoudre le même problème en le séparant en deux étapes. Ils évaluent d'abord des *régions tabous* (forbidden regions) dans lesquelles aucune tâche ne doit débuter. Fort de cette information, ils appliquent la règle *EDD* transformée ainsi : dès que la machine se libère, on débute la tâche éligible à l'échéance la plus serrée si on se situe à l'extérieur d'une région tabou. Dans le cas contraire, on patiente jusqu'au sortir de la région tabou. Pour déterminer les régions tabous, on procède comme suit. Pour chaque intervalle $[est_i, lct_j)$, on calcule le nombre de tâches devant nécessairement s'y exécuter. On ordonne chacune de ces tâches le plus tardivement possible à l'intérieur de l'intervalle $[est_i, lct_j)$ sans tenir compte de leur temps de sortie ni de leur échéance quoique se gardant de les faire débuter dans une région tabou précédemment déterminée. On prend ensuite en considération le temps de départ de la première tâche. La différence entre ce temps de départ et la borne inférieure de l'intervalle (est_i) constitue le temps excédentaire, c'est-à-dire celui qui peut être utilisé par une tâche précédant l'intervalle sans conséquence sur la faisabilité du problème. Si le temps excédentaire est plus petit que la durée, alors on ajoute une région tabou de longueur complémentaire au temps excédentaire. La nouvelle région tabou fera en sorte d'empêcher que le traitement d'une tâche précédente empiète au-delà du temps de départ le plus tardif des tâches confinées à l'intervalle $[est_i, lct_j)$ considéré.

Par exemple sur la figure 3.1, la tâche du bas est confinée à l'intervalle $[1.5, 2.75)$. Son temps de départ le plus tardif est de $2.75 - 1 = 1.75$ et le temps excédentaire de l'intervalle est $1.75 - 1.5 = 0.25 < 1$. Le temps excédentaire plus petit de 0.75 que la durée permet de déterminer la région tabou $[0.75, 1.50)$. La portion de droite de la figure 3.1 montre qu'en maintenant en attente la machine jusqu'au temps 1.50, l'instance est réalisable et son *makespan* minimisé. Grâce à l'utilisation d'une structure de données d'arbre binaire pour cumuler et comparer des *temps critiques*, les auteurs parviennent à déterminer toutes les régions tabous en $O(n \log n)$. Quant à leur nombre, il est borné supérieurement par le nombre de tâches n. L'ensemble des régions tabous est noté F.

Artiouchine et Baptiste [3] exploitent les régions tabous pour filtrer les bornes des tâches de la contrainte INTER-DISTANCE. Ils introduisent le concept de temps minimum de terminaison $ect(F, t, q)$ d'un ensemble de q tâches ne débutant pas avant le temps t considérant un ensemble de régions tabous F. Symétriquement, le concept de temps maximum de démarrage [1] $lst(F, t, q)$ d'un ensemble de q tâches ne se terminant pas après t considérant un ensemble de régions tabous F est introduit. Ces concepts permettent de déterminer des tranches d'ajustement internes qui régularisent le positionnement des tâches à l'intérieur d'un intervalle $[est_i, lct_j]$ de sorte que toutes puissent s'y exécuter. Ils déterminent aussi des tranches d'ajustement externes qui restreignent les temps de départ des tâches extérieures à l'intervalle $[est_i, lct_j)$ de

1. Le temps maximum de démarrage d'un ensemble de tâches est le temps de départ le plus tardif de la tâche que l'on traite en premier parmi tous les ordonnancements possibles de l'ensemble.

manière à ce que leur traitement n'empiète pas sur le temps nécessaire aux tâches confinées à l'intervalle $[est_i, lct_j)$. Les auteurs prouvent que ces deux ensembles de tranches suffisent à forcer la cohérence de bornes de la contrainte INTER-DISTANCE et ils produisent un algorithme qui s'exécute en $O(n^3)$ [2]. Grâce à la découverte de relations de dominance au sein des tranches d'ajustement et par une pirouette algorithmique qui en permet l'exploitation, Quimper et al. [33] font passer la complexité de la méthode à $O(n^2)$; le nombre minimal de tranches d'ajustements à considérer étant borné supérieurement par n^2.

Le problème consistant à minimiser le makespan d'un ensemble de tâches de durée unitaire et de temps de sortie arbitraire mais fixe s'effectuant sur $m > 1$ machines est résolu pour la première fois en temps polynomial par Simons [42]. Elle transforme la notion de régions tabous en *régions limitées* qui sont des intervalles pendant lesquels un nombre limité de tâches peuvent débuter [3]. En divisant le problème en plusieurs sous-problèmes et à l'aide d'une structure de données pour exégètes elle résout en $O(n^3 \log\log n)$. La complexité de l'algorithme est plus tard ramenée à $O(mn^2)$ [40] grâce à une meilleure séquence des opérations incluant des calculs préliminaires. La découverte de relations de dominance permet aussi d'abréger ou d'éliminer l'exécution de sous-problèmes de l'algorithme initial. Vakhania [43] résout le même problème en appliquant l'heuristique du *plus grand bout* (longest tail heuristic [4]) sur différentes instances modifiées du problème original. Les ordonnancements obtenus, appelés complémentaires, sont placés dans un arbre de recherche et l'arbre ainsi constitué contient l'une des solutions optimales. La complexité de l'algorithme est fonction de la plus grande échéance $(q_{max} = max_i(lct_i))$ parmi toutes les tâches du problème et s'exprime ainsi $O(q_{max} mn \log n + O(mn^2))$. Bien qu'il ne fasse pas appel à la notion de régions tabous, des *régions vides* [5] apparaissent dans les ordonnancements spécifiques à chacune des machines.

La complexité des algorithmes précédents témoigne de la difficulté à résoudre ce problème de façon conventionnelle. L'approche la plus performante à ce jour consiste à séparer le problème en deux phases. Si l'on connaît tous les temps auxquels le départ d'une tâche quelconque s'effectue dans la solution optimale, le problème se limite alors à un couplage dans un graphe bipartite [6]. Les temps de départ dans la solution sont les sommets du premier ensemble et les tâches ceux du deuxième. Il y a une arête entre la tâche i et le temps de départ t, si et seulement

2. Cette méthode comporte des similitudes avec la règle du Time-Tabling utilisée pour le filtrage de la CUMULATIVE notamment les tranches d'ajustements qui sont de la forme $[lst(), ect())$.

3. La notion de régions tabous ne peut s'appliquer dans sa forme originelle car une région n'est tabou que pour quelques machines dans certaines séquences d'ordonnancement.

4. L'appellation *tail* fait référence au temps séparant la terminaison d'une tâche et son échéance. Le plus long bout spécifie une tâche parmi celles qui n'ont pas encore débuté.

5. Les régions vides sont le résultat d'un manque de tâches à traiter ou d'un arrêt forcé. La nature des arrêts forcés peut être assimilée à la notion de régions limitées de l'algorithme de Simons.

6. Un graphe bipartite est un graphe dont les noeuds peuvent être partitionnés en deux ensembles disjoints X et Y de sorte que toutes les arêtes relient uniquement un noeud de X à un noeud de Y.

si, $est_i \leq t \leq lst_i$ [7]. Ce problème est adressé sous une forme plus générale [8] par Glover [18] et résolu en $O(n \log n)$. La méthode pour résoudre tient dans la convexité des arêtes reliant les éléments des deux ensembles [9]. Or le problème en cause peut se représenter sous la forme d'un graphe bipartite convexe. Lipski et Preparata [25] trouvent un couplage maximum (*maximum matching*) en $O(n\alpha(n))$ [10] si les éléments sont préalablement ordonnés. La complexité linéaire est plus tard atteinte par Gabow et Tarjan [16] grâce à l'utilisation d'un cas particulier de la structure de données *Union-Find* [11]. Dans le problème qui nous occupe, puisque nous sommes en présence d'un couplage complet, on peut résoudre simplement en appliquant l'adaptation de la méthode *EDD* de Garey et al.[17] à plusieurs machines. On trie d'abord les temps de départ en ordre croissant. Pour assigner les tâches on procède de la sorte. Dès qu'un temps de départ se présente, on assigne à l'une des machines disponibles la tâche à l'échéance la plus serrée parmi celles dont le temps de sortie le permet. Il ne reste qu'à trouver les temps de départ dans la solution optimale.

Brucker et Kravchenko [12] construisent le problème en utilisant la programmation linéaire et produisent un algorithme dont la complexité avoisinent $O(n^{10})$. Leur modèle assigne une tâche à chaque temps de départ. Il permet aussi de traiter les cas ou un poids (w), rendant compte de l'importance relative des tâches, est assigné à chaque tâche. Leur modèle ne permet toutefois pas d'optimiser sur le poids [12]. En se libérant des poids assignés aux tâches et en ne cherchant que les temps de départ, tel qu'expliqué au paragraphe précédent, Dürr et Hurand [15] parviennent à simplifier grandement la formulation du problème. Leur système d'équations final assure une solution entière et optimale si une solution existe. De plus, leur matrice des contraintes admet un dual qui est la matrice des incidences d'un graphe orienté. Sans produire d'algorithme, ils promettent une résolution en $O(n^4)$ en s'appuyant sur un résultat démontré de la théorie des graphes. López-Ortiz et Quimper [27] poursuivent sur cette lancée et produisent un algorithme sub-quadratique qui tire profit de techniques algorithmiques, de structures de données, de résultats empruntés à la théorie des graphes et de propriétés propres au graphe hautement structuré présenté dans [15]. Leur algorithme est, somme toute, un tour guidé de l'optimisation et constitue la pierre angulaire de nos travaux et expérimentations. Pour ces raisons, l'algorithme, les résultats sur lesquels il s'appuie et la formulation initiale du problème issue de la programmation linéaire sont décrits en détails dans la section suivante.

7. Cette approche est, en quelque sorte, un raffinement de la méthode préconisée par Garey et al.[17] qui résolvent le problème pour une machine unique en trouvant les régions tabous précédant les temps de départ de la solution optimale. Ici on cherche directement les temps de départ de la solution.

8. L'article présente un algorithme qui trouve un nombre maximum de couples, *maximum matching*, parmi les sommets de deux ensembles de cardinalité quelconque

9. Un graphe bipartite convexe requiert que les sommets de l'ensemble Y soient ordonnés de sorte que si les arêtes (x, y_1) et (x, y_3) appartiennent au graphe pour $x \in X$ et $y_1, y_3 \in Y$, alors l'arête (x, y_2) appartient nécessairement au graphe pour $y_1 < y_2 < y_3$.

10. $\alpha(n)$ est l'inverse de la fonction récursive d'Ackermann qui croît extrêmement rapidement.

11. En français, cette structure est parfois appelée *structure de données des ensemble disjoints.*

12. Le problème visant à maximiser le poids total des tâches traitées pour des tâches de durée fixe sur plusieurs machines identiques n'a toujours pas été résolu bien qu'il ne soit pas reconnu NP-Difficile.

3.2 L'algorithme de López-Ortiz et Quimper

Le problème à résoudre est celui consistant à minimiser le makespan d'un ensemble de tâches $\mathcal{I}$ de durée fixe (p) s'effectuant sur $m \geq 1$ machines identiques. L'algorithme [27] retourne les temps de départ dans la solution optimale, si une telle solution existe. Dans le cas contraire, il détecte l'infaisabilité.

Nous appelons *temps minimum* et notons r_{min} le plus petit temps de sortie parmi les tâches de l'ensemble $\mathcal{I}$.

$$r_{min} = \min_{i \in \mathcal{I}} \ est_i \tag{3.1}$$

De même, nous appelons *dernière échéance* et notons d_{max} la plus grande des échéances parmi les tâches de l'ensemble $\mathcal{I}$.

$$d_{max} = \max_{i \in \mathcal{I}} \ lct_i \tag{3.2}$$

L'intervalle $[r_{min}, d_{max} - p]$ délimite alors l'espace discret à l'intérieur duquel une solution est cherchée. Nous notons par x_t le nombre de tâches qui débutent au temps $t \in [r_{min}, d_{max} - p]$ dans la solution.

$$x_t = \sum_{\{i | S_i = t\}} 1 \tag{3.3}$$

Pour une tâche spécifique i, nous notons par μ_i le *temps limite* à partir duquel il est impossible de la compléter si elle n'a pas déjà débuté.

$$\mu_i = lct_i - p + 1 \tag{3.4}$$

Conséquemment, nous appelons *fenêtre de faisabilité* et notons F_i, la séquence de temps pendant laquelle il est possible de procéder au départ d'une tâche i et de la compléter dans les délais prescrits. Pour des raisons de commodité, nous la représentons par un intervalle ouvert à droite.

$$F_i = [est_i, \mu_i) \tag{3.5}$$

Considérant la solution à un problème donné, nous appellons *dernière limite* et notons μ_{max}, le moment le plus tôt à partir duquel plus aucune tâche ne peut débuter.

$$\mu_{max} = \max_{i \in \mathcal{I}} \ \mu_i \tag{3.6}$$

Dans un espace de recherche discret comme celui en cause ici, $\mu_{max} = d_{max} - p + 1$.

3.2.1 Le modèle

Puisque les durées des tâches sont fixes, minimiser le makespan équivaut à minimiser la somme des temps de départ[15]. On veut minimiser la fonction objectif suivante :

$$\sum_{t=r_{min}}^{\mu_{max}-1} t x_t \tag{3.7}$$

Le problème est sujet à trois ensembles de contraintes temporelles. Le premier assure de chercher une solution dans l'espace naturel $\mathbb{N}$. A tout temps t, un nombre non-négatif de tâches doivent débuter.

$$x_t \geq 0 \quad \forall\, r_{min} \leq t \leq d_{max} \tag{3.8}$$

Le second restreint la disponibilité des machines. Pour chacune des fenêtres de temps de longueur p, au plus m machines peuvent être au travail.

$$\sum_{t=s}^{s+p-1} x_t \leq m \quad \forall\, r_{min} \leq s \leq \mu_{max} - 1 \tag{3.9}$$

Le troisième est en relation avec la faisabilité du problème. Le positionnement des temps de départ optimaux doit permettre un horaire réalisable qui tient compte des restrictions s'appliquant sur chaque tâche. Il est nécessaire que chacune d'entre elles puisse débuter dans sa fenêtre de faisabilité. Ainsi, pour chaque combinaison distincte de deux bornes de temps, l'une marquant le début d'une fenêtre de faisabilité et l'autre marquant la fin, l'on délimite un intervalle à l'intérieur duquel doit débuter un nombre de tâches au minimum égal à la cardinalité de l'ensemble des tâches dont la fenêtre de faisabilité est entièrement incluse. Cette condition est suffisante et nécessaire pour assurer un couplage complet entre les tâches et les temps de départs. [19]

$$\sum_{t=est_i}^{\mu_j-1} x_t \geq \; |\{k \mid F_k \subseteq [est_i, \mu_j)\}| \quad \forall\, \mu_j > est_i \;\text{ pour } i, j \in \{1, \dots, n\} \tag{3.10}$$

Nous notons Y_t, le nombre total de tâches qui débutent antérieurement à t.

$$Y_t = \sum_{s=r_{min}}^{t-1} x_s \quad \text{et} \quad Y_{r_{min}} = 0 \tag{3.11}$$

En procédant à une substitution par Y dans chacune des équations du modèles nous obtenons les trois ensembles de contraintes qui suivent.

$$Y_t - Y_{t+1} \leq 0 \quad \forall\, r_{min} \leq t \leq \mu_{max} - 1 \tag{3.12}$$

$$Y_{t+p} - Y_t \leq m \quad \forall\, r_{min} \leq t \leq \mu_{max} - p \tag{3.13}$$

$$Y_{est_i} - Y_{\mu_j} \leq -|\{k \mid F_k \subseteq [est_i, \mu_j)\}| \quad \forall\, \mu_j > est_i \tag{3.14}$$

Nous sommes alors en présence d'un système dit de contraintes de différence. C'est-à-dire que chaque inégalité linéaire ne possède que deux variables dont les coefficients sont 1 et -1. Qui plus est, le membre de droite de chaque contrainte est une valeur entière ce qui assure une solution entière et optimale si une solution existe [13]. Formulée de la sorte, la matrice des contraintes admet un dual qui est la matrice des incidences d'un graphe orienté. Chaque variable Y_t correspond à un noeud et pour chaque contrainte de la forme $a - b \leq w$ correspond une arête allant de b à a et de poids w. Nous baptisons le graphe résultant *graphe d'ordonnancement*. La figure 3.2 illustre un tel graphe pour un problème dont les tâches sont de durée

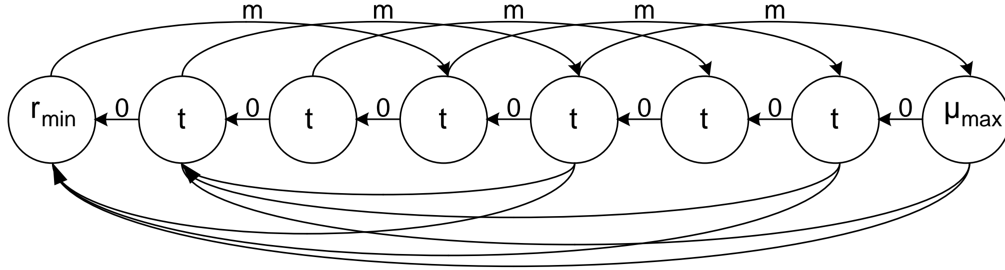


FIGURE 3.2: Exemple d'un graphe d'ordonnancement pour un problème avec $p = 3$.

$p = 3$. Les arêtes de poids 0 qui relient chaque noeud au noeud qui le précède sont issues de l'ensemble des contraintes (3.12). Nous les nommons *arêtes nulles*. L'ensemble d'arêtes de poids m qui relient chaque noeud au noeud qu'il devance de p positions, appelées *arêtes ascendantes*, est la schématisation des contraintes de (3.13). Quant aux arêtes sous les noeuds que nous nommons *arêtes descendantes*, résultats des contraintes de (3.14), leur poids est égal à l'opposé du nombre de tâches dont la fenêtre de faisabilité est incluse dans l'intervalle borné par leurs sommets incidents. Par exemple, l'arête reliant μ_{max} à r_{min} est de poids $-n$. Il y a une arête descendante pour chaque tuple distinct (μ_j, est_i) pour lequel $\mu_j > est_i$ et pour lequel l'intervalle $[est_i, \mu_j)$ inclut la fenêtre de faisabilité d'au moins une tâche de $\mathcal{I}$.

Nous appellons *plus courte distance* et notons $\delta(j, i)$, le poids total des arêtes constituant la chaîne au poids minimum en partance du noeud j et reliant le noeud i dans un graphe. Quant à la chaîne elle-même, nous la nommons *chemin le plus court*. Nous appellons *sommet horizon* et notons v_0, un noeud qui peut joindre tous les autres noeuds d'un graphe par une arête de coût $w = 0$. Cormen et al.[14] démontrent le théorème proposé par Bellman qui suit [14] :

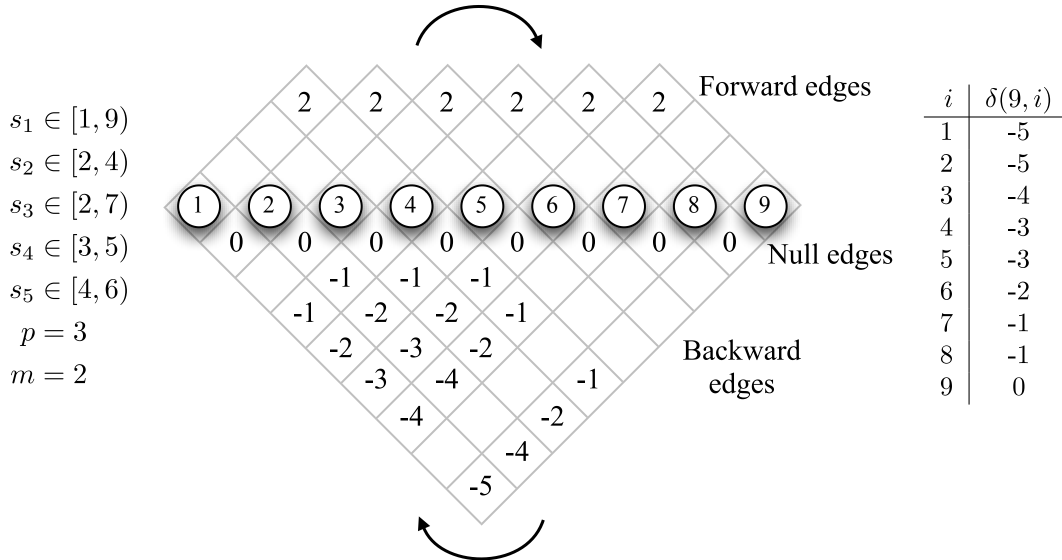
Théorème 1. *Pour un système de contraintes de différences $Ax \leq b$, soit $G = (V, E)$ son graphe d'ordonnancement correspondant et soit v_0 un sommet horizon de G. Si G ne contient pas de cycle négatif alors $x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$ est une solution réalisable. Si G contient un cycle négatif alors il n'y a pas de solution.*

13. Pour une matrice des contraintes A unimodulaire et un vecteur des valeurs b entier tel que $Ay \leq b$, la solution est entière pour chaque vecteur de b [32] p.316.

14. Le lien entre un système de contraintes de différence et la plus courte distance dans un graphe de même que la preuve du théorème se trouve dans [14] p.664-668.

La structure régulière des arêtes nulles confère à μ_{max} le statut de sommet horizon au sens de la définition ci-haut. Le problème se résoud en calculant la plus courte distance entre μ_{max} et tous les autres noeuds du graphe d'ordonnancement [15]. Une version modifiée de l'algorithme de Bellman-Ford [10] avec μ_{max} comme noeud source est utilisée. Puisque μ_{max} a accès à tous les noeuds du graphe, l'algorithme détecte le cycle négatif si un tel cycle existe [16]. Les plus courtes distances évaluées par l'algorithme sont les valeurs des variables Y_t dans la solution optimale. Or, les variables Y_t indiquent, selon l'équation (3.11), le nombre total de tâches déjà démarrées antérieurement au temps t. Ainsi le nombre de tâches qui débutent à chaque temps t est déduit de la façon suivante :

$$x_t = \delta(\mu_{max}, t+1) - \delta(\mu_{max}, t) \quad \forall\ r_{min} \leq t < \mu_{max} \tag{3.15}$$



i	$\delta(9,i)$
1	-5
2	-5
3	-4
4	-3
5	-3
6	-2
7	-1
8	-1
9	0

FIGURE 3.3: Un graphe d'ordonnancement à 9 noeuds. Le poids des arêtes reliant deux noeuds est inscrit à l'intersection des diagonales passant par ces noeuds. Les arêtes nulles et les arêtes descendantes apparaissent sous la ligne des noeuds alors que les arêtes ascendantes apparaissent au-dessus. Une intersection vide indique l'absence d'une arête d'orientation correspondante entre deux noeuds. Par exemple, le poids de l'arête (9,2) est -4 alors que celui de l'arête (5,8) est 2 et il n'y a pas d'arête (8,5). Le chemin le plus court entre les noeuds 9 et 6 passe par les noeuds 9,1,4,7,2,5,3,6.

15. Le problème appelé *Single Source Shortest Path* consistant à trouver la plus courte distance entre un sommet d'un graphe et tous les autres sommets est abondamment documenté.

16. La présence d'arêtes au poids négatif dans le graphe d'ordonnancement contraint à se prémunir contre l'existence d'un cycle négatif. L'article de Dürr et Hurand [15] traite de 3 problèmes formulés en programmation linéaire dans lesquels d'habiles transformations permettent de ramener la matrice des contraintes à une forme unimodulaire qui admet un dual aux allures de *graphe d'ordonnancement*. Pour l'un d'entre eux, le *Prefetch Caching*, leur acharnement à décomposer le problème résulte en un graphe qui ne contient que des arêtes au poids non-négatif. Il leur est alors aisé de résoudre en ayant entre autres recours à l'algorithme de Floyd-Warshall *All Source/Sink pair Min Cost Flow*. Dans les faits, ils font passer la vitesse de résolution d'un problème tout frais éclos de son oeuf polynomial de $O(n^{18})$ à $O(n^3 \log n)$.

La figure 3.3 est tirée de [27]. Elle illustre une exécution sur une instance à 5 tâches de durée $p = 3$ disposant de $m = 2$ machines identiques. Les fenêtres de faisabilité de chacune des tâches sont indiquées dans la colonne de gauche. Le résultat de l'exécution (les plus courtes distances) est détaillé dans le tableau à droite du graphe. De l'équation (3.15), nous déduisons que les temps de départ des tâches dans la solution optimale sont 2,3,5,6 et 8. Ces valeurs sont ensuite couplées aux tâches. Pour la solution du problème illustré par la figure 3.3, il n'y a qu'un seul couplage complet. Les tâches doivent être traitées dans l'ordre suivant : I_2, I_4, I_5, I_3 et I_1.

3.2.2 Optimisation et implémentation

Théoriquement, l'algorithme de Bellman-Ford s'exécute en $O(|V||E|)$, où $|V|$ est le nombre de noeuds et $|E|$ le nombre d'arêtes. Le graphe d'ordonnancement est constitué de $\mu_{max} - r_{min} + 1$ noeuds. Son nombre d'arêtes nulles est de 1 inférieur au nombre de noeuds, soit $\mu_{max} - r_{min}$. Quant aux arêtes ascendantes, leur nombre est donné précisement par $\mu_{max} - r_{min} - p + 1$. Le nombre d'arêtes descendantes dépend des propriétés spécifiques des tâches. Il est borné inférieurement par 1 (toutes les tâches ont une fenêtre de faisabilité identique) et supérieurement par n^2 (toutes les combinaisons $[est_i, \mu_j)$ pour $i, j \in \{1, \ldots, n\}$ incluent au moins une fenêtre de faisabilité). Ce qui confère à l'algorithme une complexité totale de $O((\mu_{max} - r_{min})^2 + n^2(\mu_{max} - r_{min}))$. Cette complexité est pseudo-polynomiale dû au terme $(\mu_{max} - r_{min})$.

L'algorithme présenté dans l'article tire profit de propriétés spécifiques au graphe d'ordonnancement qui découlent de la structure forte et régulière conférée par la présence des arêtes ascendantes ainsi que des arêtes nulles [17]. L'exploitation de ces propriétés combinée à l'utilisation de structures de données adaptées font en sorte de ramener la complexité de l'algorithme au voisinage de $O(n^2)$.

Yen a traité d'améliorations qui peuvent être apportées à l'algorithme de Bellman-Ford pour des configurations particulières de graphe. Dans [50], le résultat suivant est démontré : pour un graphe sans cycle négatif comportant des arêtes au poids négatif [18], on améliore le pire cas si l'on en ordonne les noeuds et que l'on sépare ensuite les arêtes en deux ensembles distincts selon leur direction : ascendante ou descendante. On itère en traitant tour à tour les deux ensembles, toutes les arêtes d'un même ensemble devant être relaxées avant de passer à l'ensemble suivant. Il est alors possible de réduire le nombre d'itérations au nombre de fois que le chemin le plus court alterne entre une arête ascendante et une arête descendante [19].

17. Pour les preuves et démonstrations complètes, le lecteur est invité à se référer à [27].
18. S'il n'y a que des arêtes positives, Yen suggère d'avoir recours à l'algorithme de Dijkstra.
19. Yen parle de tronçons croissants et de tronçons décroissants, un tronçon étant une chaîne d'arêtes de même sens. L'article constate que des tronçons optimaux se forment au sein des deux ensembles et qu'un chemin le plus court peut en être réduit à une alternance entre tronçons optimaux croissants et tronçons optimaux décroissants. D'une certaine façon, Yen tire profit de la propriété triviale selon laquelle chaque chaîne d'un chemin le plus court est elle-même un chemin le plus court.

Les lemmes qui suivent sont spécifiques aux graphes d'ordonnancement de la forme de celui présenté à la figure 3.2 et permettent de trouver le chemin le plus court au nombre minimum d'alternances tout en itérant sur un nombre minimum d'arêtes descendantes. En ce qui concerne les arêtes nulles et ascendantes, leur structure régulière permet de calculer leur effet sans itérer sur elles à proprement parler.

Lemme 2 ([27])**.** *Soit a<b<c<d, quatre noeuds d'un graphe d'ordonnancement sans cycle négatif. Si les arêtes (d,b) et (c,a) font partie d'un chemin le plus court alors il existe un chemin équivalent de même poids qui n'inclut pas ces arêtes.*

Soit a, b, c et d, 4 noeuds d'un graphe ordonné tel que $a < b < c < d$. Nous disons des arêtes (d, b) et (c, a) qu'elles se croisent. De même, nous qualifions la paire d'arêtes (d, c) et (b, a) d'arêtes disjointes et la paire d'arêtes (d, a) et (c, b) d'arêtes imbriquées [20].

Lemme 3 ([27])**.** *Pour tout chemin le plus court constitué d'arêtes descendantes qui se croisent, il existe un chemin le plus court dans lequel aucune paire d'arêtes descendantes ne se croisent.*

Les lemmes 2 et 3 interviennent dans le cas où il y a plus d'une arête descendante dans un chemin le plus court. Par extension, ils stipulent que pour toute combinaison de deux arêtes descendantes, les arêtes en cause ne peuvent être que disjointes ou imbriquées. Par exemple, si les arêtes (d,b) et (c,a) du lemme 2 font partie d'un chemin le plus court, l'on peut toujours leur substituer soit l'arête unique (d,a) ou une configuration d'arêtes imbriquées (d,a) + (f,e) tel que $a < e < f < d$. Cette propriété permet de négliger de tester certaines combinaisons. C'est-à-dire qu'il est inutile d'itérer sur toutes les arêtes.

Nous nommons *vecteur de distances* et notons $\Delta[t]$, la plus courte distance du noeud μ_{max} au noeud t.

$$\Delta[t] = \delta(\mu_{max}, t) \tag{3.16}$$

Lemme 4 ([27])**.** *Le vecteur des distances est monotone croissant.*

Lemme 5 ([27])**.** *Pour un graphe d'ordonnancement sans cycle négatif,* $\Delta[r_{min}] = -n$.

Le lemme 4 s'explique par la présence des arêtes nulles qui permettent de passer de $t + 1$ à t au coût $w = 0$ pour tout t. Le lemme 5 est un corrolaire du théorème 1 qui promet une solution au problème s'il n'y a pas de cycle négatif dans le graphe. Ainsi, exactement n tâches doivent débuter entre r_{min} et $\mu_{max} - 1$. Or la plus courte distance est la différence du nombre de tâches débutées antérieurement au temps t et aucune tâche ne peut débuter avant r_{min}.

20. En traçant un graphe dans lequel tous les noeuds se succèdent sur une droite comme dans la Figure 3.2, les épithètes *croisé*, *disjoint* et *imbriqué* décrivent physiquement l'assemblage formé de la paire d'arêtes qui relient les noeuds incidents en cause.

Lemme 6 ([27]). *Soit J_{est_i}, l'ensemble des tâches dont le temps de sortie est est_i. Les arêtes descendantes aboutissant aux noeuds correpondants à est_i et à est_{i+1} sont en relation comme suit :* $w(t, est_i) = w(t, est_{i+1}) - |\{k \in J_{est_i} \mid \mu_k \leq t\}|$.

Le lemme 6 découle de la définition même, voir l'équation (3.14), des contraintes donnant naissance aux arêtes descendantes. Le poids d'une arête est l'opposé du nombre de tâches devant débuter dans la fenêtre de temps définie par les sommets incidents à l'arête. Appellons *distance temporaire* et notons $d'[t]$, la plus courte distance $\delta(\mu_{max}, t)$ trouvée par l'algorithme tout juste avant une itération. Itérer sur les arêtes descendantes consiste à évaluer s'il vaut mieux passer par un noeud μ_t pour atteindre un noeud est_i, c'est-à-dire si $d'[t] + w(\mu_t, est_i) < d'[i]$ pour chaque noeud est_i. Le lemme suivant montre que pour parvenir à cette fin, il est inutile de considérer toutes les arêtes.

Lemme 7 ([27]). *Soit deux temps limites $\mu_a < \mu_b$, si $d'[\mu_a] + w(\mu_a, est_{i+1}) \geq d'[\mu_b] + w(\mu_b, est_{i+1})$ alors $d'[\mu_a] + w(\mu_a, est_i) \geq d'[\mu_b] + w(\mu_b, est_i)$.*

En pratique, les lemmes 6 et 7 expliquent que lorsqu'on itère sur les arêtes descendantes, si l'arête en provenance du noeud correpondant au temps limite μ_b nous donne une meilleure distance temporaire que celle en provenance du noeud de μ_a pour le noeud associé à est_{i+1}, elle donnera aussi une meilleure distance temporaire pour le noeud associé à est_i.

Algorithme 9 : ÉvaluationDesTempsDeDépart $(est[1, \ldots, n], lct[1, \ldots, n], m, p)$

```
E ← {est_i, ..., est_n}
U ← {lct_i − p + 1, ..., lct_n − p + 1}
β ← Tri croissant sur les éléments distincts de E ∪ U
l_i ← index(β, est_i)
u_i ← index(β, lct_i − p + 1)
d ← [μ_max, r_min, ..., r_min]  // vecteur de longueur n+1
d_0 ← [0, ..., 0]               // vecteur de longueur n+1
while d ≠ d_0 do
    d_0 ← d
    d ← Relaxation des arêtes ascendantes (d)    // Algorithme 11
    d ← Relaxation des arêtes descendantes (d)   // Algorithme 10
    if Un cycle négatif est découvert then
        return Horaire infaisable
return d[n], d[n − 1], ..., d[1] // le vecteur est défini sur [0,n]
```

L'algorithme 9 trouve les temps de départs qui minimise le makespan. Son vecteur d, de longueur $n + 1$, fait la jonction entre l'algorithme 11 et l'algorithme 10 dans lesquels il est constamment mis à jour. Il contient dans chacune de ses a positions, $a \in \{0, \ldots, n\}$, le plus grand des noeuds accessibles à une plus courte distance d'au plus $-a$ du noeud horizon μ_{max}. Ultimement, après l'obtention du point fixe, les temps de départ dans la solution optimale sont donnés par $d[n], d[n-1], ..., d[1]$.

Algorithme 10 : RelaxationDesArêtesDescendantes (d[ ])

```
d' ← []                    // vecteur de longueur |β| -distances temporaires
j ← −n
for b ∈ β en ordre croissant do
    while b > d[−j] do
        j ← j + 1
    d'[b] ← j
T ← UnionFind(|β|)         // structure de données UnionFind
k ← [0, ..., 0]            // vecteur de longueur |β| -compteurs de modifications
c ← []                     // vecteur de longueur |β| -distances inter-représentants
for j ← 1 to |β| − 1 do
    c[j] ← d'[j + 1] − d'[j]
    if c[j] = 0 then
        Unir(T, j, j + 1)  // ensemble de représentants équidistants
for i ∈ I en ordre décroissant des l_i do
    q ← TrouveMaximum(T, u_i)
    t ← TrouveMaximum(T, TrouveMinimum(T, u_i) − 1)
    if t < β_1 then
        return Un cycle négatif est détecté
    c[t] ← c[t] − 1
    if c[t] = 0 then
        Unir(T, q, t)
        k[q] ← k[q] + k[t]
    k[q] ← k[q] + 1
    e ← TrouveMaximum(T, l_i)
    a ← d'[e] − k[e]
    if a < d'[l_i] then
        d'[l_i] ← a
        d[−a] ← β[l_i]
return d[ ]
```

Pour itérer sur les arêtes descendantes, l'algorithme 10 construit une liste strictement croissante de représentants, notée β, formée de la valeur de tous les noeuds qui marquent soit un temps limite (μ_i) et/ou un temps de sortie (est_i). Par exemple, sur la figure 3.2, la liste serait constituée des cinq noeuds incidents aux arêtes descendantes, de r_{min} à μ_{max}. L'algorithme a recours à la structure de données *Union-Find* [14] pour regrouper les représentants à égale plus courte distance de μ_{max}. Puisque le vecteur des distances est monotone, l'opération *Union* s'effectue toujours sur des représentants consécutifs. Il n'y a donc pas de chevauchement entre les ensembles. Ainsi, un ensemble détermine l'intervalle des représentants équidistants et la valeur de son plus petit membre et celle de son plus grand membre, appelé *membre dominant*, sont maintenues à jour à la racine de l'ensemble. Le vecteur c mesure la différence de distance entre les ensembles et une opération d'*Union* résulte de l'assignation de la valeur 0 à l'une des positions du vecteur c. La routine tient aussi à jour un vecteur d' de longueur égale au

nombre de représentants. Le vecteur d' contient les distances temporaires, soit la plus courte distance évaluée jusqu'alors de μ_{max} à chaque représentant de la liste β. Il permet de vérifier si la distance du noeud β_{est_i} est diminuée après traitement de l'arête correspondant à F_i.

Exemple 3. *L'exemple qui suit est tiré de [27] et constitue la première itération du problème illustré par la figure 3.3. Nous cherchons à minimiser le makespan pour $n = 5$ tâches de durée $p = 3$ qui s'exécutent sur $m = 2$ machines identiques.*

$$
\begin{aligned}
&F_1 = [1, 9), F_2 = [2, 4), F_3 = [2, 7), F_4 = [3, 5), F_5 = [4, 6) \\
&\beta = \{1, 2, 3, 4, 5, 6, 7, 9\} \\
&d_0 = [9, 1, 1, 1, 1, 1] \\
&d_1 = [9, 7, 4, 4, 1, 1]
\end{aligned}
$$

La première ligne ci-haut énumère les fenêtres de faisabilité des tâches du problème. Les lignes qui suivent présentent la liste des représentants (β) qui en découle, le vecteur d après initialisation (d_0) et le même vecteur d résultant de la première itération (d_1) sur les arêtes ascendantes. L'algorithme relaxe ensuite les arêtes descendantes. À partir de la liste des représentants, le vecteur des distances temporaires (d') est construit.

$$
\begin{aligned}
&d' = [-5, -3, -3, -3, -1, -1, -1, 0] \\
&T = \{\boldsymbol{1}\} \leftarrow \underline{2} \leftarrowtail \{2, 3, \boldsymbol{4}\} \leftarrow \underline{2} \leftarrowtail \{5, 6, \boldsymbol{7}\} \leftarrow \underline{1} \leftarrowtail \{\boldsymbol{9}\} \\
&T' = \{\boldsymbol{1}\} \leftarrow \underline{2} \leftarrowtail \{2, 3, 4, 5, 6, \boldsymbol{7}\} \leftarrow \underline{1} \leftarrowtail \{\boldsymbol{9}\} \\
&d_2 = [9, 7, 4, 4, 2, 2]
\end{aligned}
$$

La deuxième ligne ci-haut montre la structure de données (T) après initialisation. Elle est constituée des quatres ensembles dont les représentants sont à égale distance de μ_{max} (-5,-3,-1,0). Le membre dominant, ou borne supérieure de chaque ensemble est marqué en gras. Les valeurs du compteur c[] qui indique l'écart de distance entre chaque représentant, sont inscrites et soulignées entre les flèches séparant les ensembles. L'itération sur les arêtes correspondantes de F_5 et F_4 font en sorte de décrémenter de 2 unités le compteur c[4] qui jalonne l'écart entre les noeuds β_4 et β_5 et dont la valeur devient 0. Il en résulte (T') une union des deux ensembles. La dernière ligne montre l'état du vecteur des distances (d_2) après le traitement de toutes les arêtes descendantes.

Dans son exécution, l'algorithme de Bellman-Ford nécessite de trouver l'arête descendante optimale incidente à tous noeuds correspondants à un temps de sortie est_i. Ainsi, nous devons trouver l'arête qui minimise la distance $d'[b_{\mu_j}] + w(b_{\mu_j}, b_{est_i})$ où b_{est_i} est le représentant du temps de sortie est_i. Lorsque toutes les tâches de temps de sortie supérieur ou égal à est_i ont été traitées, l'arête optimale nous est donnée directement sur la structure de données par le membre dominant de l'ensemble auquel appartient le représentant de est_i. De l'exemple 3,

la ligne T' nous montre l'état de la structure T après traitement de l'arête $F_4 = [3, 5)$. Le membre dominant de l'ensemble comprenant le représentant 3 étant 7, l'arête (7,3) est celle qui minimise la distance temporaire $d'[3]$ à cette itération. S'il s'avère qu'un représentant est le membre dominant de son ensemble après le traitement de toutes les tâches au temps de sortie non-inférieur, on en déduit que la distance temporaire $d'[est_i]$ ne peut être diminuée à cette itération [21]. L'arête optimale est alors l'arête virtuelle (b_{est_i}, b_{est_i}) de poids $w = 0$.

L'algorithme 11 procède artificielleement à la relaxation des arêtes ascendantes. Il itère sur les noeuds associés au membre dominant de chaque ensemble disjoint. Cette façon de procéder suffit à identifier les noeuds (l'ensemble de représentants) dont la plus courte distance pourrait être améliorée à l'étape de relaxation des arêtes descendantes.

Algorithme 11 : RelaxationDesArêtesAscendantes (d[ ])

for $j \leftarrow -n$ *to* $-m$ **do**
 $d[-j-m] \leftarrow \max\{d[-j]+p, d[-j-m]\}$
return $d[\,]$

3.2.3 Analyse

Pour les ensembles de représentants, l'algorithme 10 a recours à la structure de données *Union-Find* avec union par rang et compression des chemins [22] dont le pire cas est initialement borné par $O(\log n)$. En 1975, Tarjan [11] démontre que la véritable complexité [23] de cette structure est $O(n\alpha(n))$ pour n appels aux fonctions *Union* et *Find* et où α est l'inverse de la fonction d'Ackermann. Gabow et Tarjan [16] en abaisse [24] plus tard la complexité à $O(n)$ par l'ajout d'une structure complémentaire et en imposant des contraintes sur les paramètres qui peuvent être passés à la fonction *Union*. En ce qui concerne l'algorithme 10, ces contraintes sont satisfaites et la structure de données avec les améliorations de Tarjan est utilisable. La routine de relaxation des arêtes descendantes (algorithme 10) peut alors prétendre à une complexité linéaire puisqu'elle itère sur un maximum de $2n$ représentants. Pour la relaxation des arêtes ascendantes (algorithme 11), la routine requiert $(n - m + 1)$ itérations ce qui place

21. L'exactitude de cette affirmation est démontrée dans [27].
22. Si l'on réfère à chaque ensemble comme à un arbre et à la strucutre dans sa globalité comme à une forêt, l'union par rang signifie que lorsque l'on unit deux arbres, on choisit comme racine de l'arbre résultant la racine de l'arbre le plus haut avant l'union. La compression des chemins consiste à relier chaque noeud parcourru lors d'une requête directement à la racine de son arbre. C'est à dire que le lien parent des noeuds devient la racine de l'arbre. Ainsi, lors d'une requête subséquente passant par l'un ou l'autre desdits noeuds, on accède à la racine en temps constant.
23. Sa démonstration tient principalement sur le fait que pour un certain nombre de requêtes *FIND*, il est impossible de construire un arbre pour lequel chacune des requêtes prenne autant que $\log n$ opérations. Selon [14], l'inverse de la fonction d'Ackermann est toujours inférieure à 5 pour des quantités pouvant être manipulées par les ordinateurs actuels.
24. L'amélioration de [16] nécessite une grosse structure avec large constante (42) et elle n'est pas implémentée dans la version utilisée pour les expérimentations. La rapidité conférée par la structure *Union-Find* avant cette dernière amélioration dépasse probablement dans la pratique celle de Gabow et Tarjan [16] pour des instances de la taille de celles que nous mettons en test ($n \leq 165$).

aussi sa complexité dans la famille $O(n)$. L'amélioration de Yen [50] apportée à l'algorithme de Bellman-Ford promet une convergence en un nombre d'itérations égal au nombre de fois que le chemin le plus court alterne entre un tronçon ascendant et un tronçon descendant. López-Ortiz et Quimper [27] montrent que pour un graphe d'ordonnancement tel le graphe de la Figure 3.3, le nombre d'alternances dans le chemin le plus court est borné supérieurement par $\min(n, (\lceil \frac{n}{m} \rceil + 1)p)$. Outre l'initialisation qui comprend quelques tris, l'algorithme 9 lance tour à tour les routines algorithme 11 et algorithme 10 toutes deux de complexité linéaire. La complexité générale de l'algorithme d'évaluation des temps de départs est donc de $O(n(\min(n, (\lceil \frac{n}{m} \rceil + 1)p)))$ que nous approximons par $O(n^2)$.

Le résultat de Yen est valide pour un graphe sans cycle négatif. Pour détecter un cycle négatif dans un graphe général comportant des arêtes négatives, Yen [50] démontre qu'un maximum de $\frac{1}{2}n(n-1)(n-2)$ additions et comparaisons sont nécessaires. Il en va autrement pour le graphe d'ordonnancement traité par l'algorithme 9. À chacune des itérations, l'algorithme découvre au minimum l'une des alternances dans le chemin le plus court entre μ_{max} et r_{min}. L'influence de chaque alternance est partie intégrante de la structure de données aux itérations suivantes. S'il existe un cycle négatif dans le graphe d'ordonnancement, l'alternance qu'il constitue est parcourue plus d'une fois. Conséquemment, un cycle négatif est détecté en un maximum de $n+1$ itérations et résulte du noeud r_{min} que l'on tente de fixer à la distance $-n-1$ de μ_{max}. La complexité de l'algorithme demeure alors de l'ordre de $O(n^2)$ [25]. Cette dernière propriété est exploitée à la section suivante.

25. Dans les faits, détecter un cycle négatif prend une itération de plus qu'il n'en faut pour établir les temps de départ avec l'Algorithme 9. López-Ortiz et Quimper [27] évaluent la complexité de l'opération à $O(n^2 \min(1, \frac{p}{m}))$, ce qui équivaut à la complexité de l'opération visant à trouver les temps de départ. Les auteurs préconisent de laisser itérer l'algorithme $\min(n, (\lceil \frac{n}{m} \rceil + 1)p)))$ fois et que si un point fixe n'est pas encore atteint, de conclure à la présence d'un cycle négatif dans le graphe d'ordonnancement. Nous qualifions cette méthode de passive. Dans l'implémentation finale, nous ajoutons à la routine algorithme 10 la ligne de code : SI $t \leq \beta_1$ ALORS il y a un cycle négatif. Cette détection supplémentaire assure de stopper l'exécution à l'itération suivant exactement celle dans laquelle on aurait obtenu un point fixe s'il n'y avait pas de cycles négatifs dans le graphe d'ordonnacement.

3.3 Filtrage de la contrainte Multi-Inter-Distance

Dans cette section, nous montrons comment obtenir la cohérence de bornes pour la contrainte Multi-Inter-Distance en temps cubique[30]. La méthode présentée découle directement du Théorème 1 et adapte l'algorithme de López-Ortiz et Quimper[27] de la section précédente pour obtenir un propagateur efficace. Obtenir la cohérence de domaines pour la contrainte Inter-Distance est un problème NP-Difficile[3]. À plus forte raison il en est de même pour la contrainte Multi-Inter-Distance qui est une généralisation de la contrainte Inter-Distance pour laquelle $m \in \mathcal{N}^*$.

L'algorithme présenté dans cette section trouve des intervalles dans lesquels aucune valeur ne peut être assignée à une ou plusieurs variables X tout en satisfaisant la contrainte. Les intervalles cherchés sont ceux qui incluent la valeur est_i pour un certain i. Il s'agit donc des intervalles qui mèment au filtrage de la borne inférieure du domaine de la variable X_i. Les autres intervalles sont ignorés. Pour le filtrage des bornes supérieures, on procède en exécutant l'algorithme sur l'opposé mathématique du problème comme illustré par la figure 2.6.

3.3.1 Prélude

La contrainte Multi-Inter-Distance$([X_1, \ldots, X_n], m, p)$ est satisfaite lorsqu'au plus m variables sont affectées dans tout intervalle de longueur p, où p et m sont des paramètres. De façon formelle, la contrainte est satisfaite lorsque $|\{i \mid X_i \in [t, t+p)\}| \leq m$ pour tout entier t.

Une contrainte est cohérente de bornes si pour toutes les variables de sa portée, la valeur minimum et la valeur maximum de leur domaine ont un support d'intervalle. Un support d'intervalle pour une contrainte est une affectation des variables de sa portée de sorte que chaque valeur affectée à une variable X appartienne à l'intervalle $[\min\{\text{dom}(X)\}, \max\{\text{dom}(X)\}]$. De plus, cette affectation doit satisfaire la contrainte.

Pour toute tâche i d'un problème d'ordonnancement et sa variable associée du temps de départ X_i, nous appellons domaine de la variable X_i, la fenêtre de faisabilité de la tâche i. Le domaine de X_i comprend tous les nombres entiers inclus dans l'intervalle quoique nous nous intéressons plus particulièrement aux bornes de l'intervalle. Pour des raisons de commodité, nous représentons le domaine par un intervalle ouvert à droite. Nous désignons par temps limite et notons μ_i l'entier suivant immédiatement la borne supérieure du dom(X_i). Il représente le temps à partir duquel il n'est plus possible de terminer la tâche i si elle n'a pas encore débuté.

$$\text{dom}(X_i) = F_i = [est_i, \mu_i) \ , \ \forall i \ \in \ \mathcal{I}$$

Du Théorème 1 nous obtenons la propriété suivante. Pour un ensemble de tâches $\mathcal{I}$ et son graphe d'ordonnancement associé, il existe un horaire réalisable si et seulement si le graphe d'ordonnancement ne contient pas de cycles négatifs.

Pour la contrainte All-Different, Leconte [23] réfère à des valeurs successives qui ne peuvent être affectées à un sous-ensemble de variables par l'appellation *Intervalle de Hall*. Lorsque les domaines des variables sont continus, Garey et al. [17] introduisent l'appellation région tabou. Cette désignation est plus tard reprise par Artiouchine et Baptiste [3] pour la contrainte Inter-Distance. Nous utilisons cette même désignation pour la contrainte Multi-Inter-Distance. La détection des régions tabous est une application directe du Théorème 1. Pour une variable X_i, on procède en diminuant progressivement la borne supérieure de son domaine. À chaque étape, on construit le graphe d'ordonnancement associé et testons s'il existe un horaire réalisable, c'est-à-dire si le domaine transformé possède un support d'intervalle. Dans la négative, on conclut que la tâche i doit obligatoirement débuter dans la portion tronquée de sa fenêtre de faisabilité originelle. L'exemple 4 et la figure 3.4 illustrent le procédé.

Exemple 4. *Posons le problème visant à ordonner $n = 3$ tâches de durée $p = 5$ sur $m = 2$ machines identiques.*

$$X_1 \in [2, 5)$$
$$X_2 \in [3, 6)$$
$$X_3 \in [1, 9)$$

L'algorithme ÉvaluationDesTempsDeDépart (algorithme 9) nous permet de trouver les temps $X_1 = 2,\ X_2 = 3$ et $X_3 = 7$ qui minisent le makespan. Si l'on diminue progressivement la borne supérieure du domaine de la troisième tâche (I_3), le même algorithme nous retourne qu'il n'y a pas d'horaire réalisable pour $\mu_3 = 7$. On en déduit que pour obtenir un horaire réalisable, la tâche I_3 doit débuter au plus tôt au temps $t = 7$. Son domaine est ajusté à $X_3 \in [7, 9)$. Sur la figure 3.4 on peut voir à gauche le graphe d'ordonnancement associé au problème initial. Dans la portion de droite, le graphe transformé avec $\mu_3 = 7$ contient le cycle négatif formé des noeuds 7-1-6-2-7. La somme des poids des quatre arêtes qui forment le cycle est -1=-3+2-2+2.

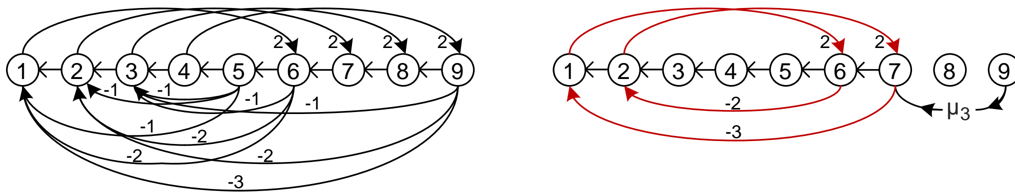


Figure 3.4: Le graphe d'ordonnancement du problème initial à gauche et le cycle négatif du graphe transformé avec $\mu_3 = 7$ à droite.

On pourrait construire un algorithme de force brute qui réduit de façon incrémental les bornes supérieures jusqu'à ce qu'un cycle négatif soit créé. Cet algorithme établirait la cohérence de bornes pour la contrainte Multi-Inter-Distance. La complexité d'un tel algorithme serait fonction de l'espace de recherche $\mu_{max} - r_{min}$. Les théorêmes et propriétés de la prochaine section permettent d'optimiser le processus et d'obtenir un propagateur polynomial.

3.3.2 Le propagateur

Nous appelons *graphe modifié* et notons G_i^v le graphe d'ordonnancement résultant de la diminution de la borne supérieure du domaine de la variable X_i tel que $\text{dom}(X_i) = [est_i, v)$. Si G_i^v a un cycle négatif, alors la contrainte ne peut être satisfaite et le propagateur conclut que le filtrage qui s'ensuit, $\text{dom}(X_i) = [v, \mu_i)$, est correct.

Théorème 8. *Soit X_i et X_j, deux variables dont les domaines satisfont la relation $\mu_i \leq \mu_j$. Si toutes les valeurs entières incluses dans $[est_i, v)$ n'ont pas de support d'intervalle dans $dom(X_i)$, alors elles n'ont pas de support d'intervalle dans $dom(X_j)$.*

Démonstration. Nous prouvons la converse : s'il existe une valeur $a \in [est_i, v)$ qui a un support d'intervalle dans $\text{dom}(X_j)$ alors il existe une valeur $b \in [est_i, v)$ qui a un support d'intervalle dans $\text{dom}(X_i)$. Supposons qu'il existe un support d'intervalle t tel que $t[j] \in [est_i, v)$, nous voulons prouver qu'il existe un support d'intervalle t' tel que $t'[i] \in [est_i, v)$. Nous distinguons deux cas. Si $t[i] \leq t[j]$ alors les inégalités $est_i \leq t[i] \leq t[j] \leq \mu_i \leq \mu_j$ sont satisfaites et $t[i] \in [est_i, v)$. Le théorème est satisfait pour $t' = t$. Par ailleurs, Si $t[i] > t[j]$, les inégalités $est_j \leq t[j] \leq t[i] \leq \mu_i \leq \mu_j$ sont satisfaites. Nous permutons les valeurs $t[i]$ et $t[j]$ pour obtenir le support t'. Nous avons $t'[j] = t[i] \in [est_i, v)$. Ainsi t' est un support valide. □

Le Théorème 8 nous assure qu'en traitant les tâches en ordre non-décroissant des temps limites, l'algorithme n'a pas à découvrir plus d'une fois la même région tabou.

Exemple 5. *Posons le problème visant à ordonner $n = 5$ tâches de durée $p = 3$ sur $m = 2$ machines identiques. Nous avons les domaines suivants :*

$$X_1 \in [7, 9) \quad X_2 \in [2, 4) \quad X_3 \in [4, 7) \quad X_4 \in [2, 7) \quad X_5 \in [3, 5)$$

...et la matrice des poids des arêtes du graphe d'ordonnancement associé au problème :

a\b	2	3	4	5	6	7	8	9
2				2				
3	0				2			
4	-1	0				2		
5	**-2**	**-1**	**0**				2	
6				0				2
7	-4	-2	-1		0			
8						0		
9	-5	-3	-2			-1	0	

a\b	2	3	4	5	6	7	8	9
2				2				
3	0				2			
4	-1	0				2		
5	**-3**	**-2**	**-2**				2	
6				0				2
7	-4	-2	-1		0			
8						0		
9	-5	-3	-2			-1	0	

Les nombres indiquent le poids de l'arête allant de a vers b. L'absence de nombre indique qu'il n'y a pas d'arête (a, b). *À gauche, la matrice du problème original. À droite, la matrice résultant de la diminution du domaine de* X_3.

Si l'on diminue le domaine de la variable X_3 de l'exemple 5 à $[4, 5)$, nous obtenons un cycle négatif formé des arêtes (5,2) et (2,5). Ce qui indique que les valeurs comprises dans $[4, 5)$ n'ont pas de support de solution et que l'intervalle $[4, 5)$ est une région tabou pour la variable X_3. Sur la matrice des poids de l'exemple 5, les arêtes dont le poids subit une modification sont indiquées en gras. Le poids de l'arête (5,2) devient -3. Le théorème 8 stipule que l'intervalle $[4, 5)$ est aussi une région tabou pour toute variable X_j telle que $\mu_j \geq \mu_3$. Par exemple, en réduisant le domaine de X_4 à $[2, 5)$, le poids des mêmes arêtes est reduit de une unité ce qui conduit au même cycle négatif. L'intervalle $[2, 5)$ est une région tabou pour la variable X_4. Ce qui confirme que $[4, 5) \subseteq [2, 5)$ est une région tabou pour la variable X_4.

La *matrice des distances* D_G d'un graphe G est une matrice carrée dont les entrées $D_G[a, b]$ indiquent la plus courte distance du noeud a au noeud b dans le graphe G. La matrice existe seulement s'il n'y a pas de cycle négatif dans le graphe.

Lemme 9. *Soit* G *le graphe d'ordonnancement d'un problème dont les domaines des variables* X_i *sont donnés par* $[est_i, \mu_i)$ *pour* $1 \leq i \leq n$. *Soit* G' *le graphe d'ordonnancement du même problème avec le domaine de* X_i *modifié à* $[est'_i, \mu'_i)$ *où* $est_i \leq est'_i < \mu'_i \leq \mu_i$. *Alors la relation,* $D_G[a, b] \geq D_{G'}[a, b]$ *est satisfaite pour toute paire de noeuds* (a, b).

Démonstration. Le poids d'une arête descendante (μ_j, est_k) est l'opposé du nombre de variables dont le domaine est entièrement inclus dans $[est_k, \mu_j)$. Lorsque le domaine d'une variable est diminué, le poids des arêtes descendantes du graphe d'ordonnancement est soit diminué de une unité ou il demeure inchangé. Or, des arêtes au poids «plus négatif» ne peuvent mener à de plus longues distances. Par ailleurs, il peut survenir que la diminution du domaine de X_i à $[est'_i, \mu'_i)$ fasse en sorte d'éliminer des arêtes incidentes à est_i ou μ_i dans G. En pareilles circonstances, la présence des arêtes nulles permet toujours de leur substituer une chaîne équivalente. Par exemple, pour un graphe G associé à un problème à une tâche, l'arête (μ_i, est_i) dans G est équivalente à la chaîne $(\mu_i, \mu_i - 1) +, \ldots, +(\mu'_i, est'_i) +, \ldots, +(est_i + 1, est_i)$ dans G'. □

Soit $U = \{\mu_i \mid i \in 1, \ldots, n\}$ l'ensemble des temps limites et soit $\mu^* = \min\{\mu \in U \mid \mu > est_i\}$ le plus petit temps limite supérieur à est_i. Le graphe modifié $G_i^{\mu^*}$ informe sur la façon de filtrer correctement et suffisamment la borne inférieure du domaine de X_i. Si l'on détecte un cycle négatif dans $G_i^{\mu^*}$, alors $X_i \geq \mu^*$ et la borne inférieure de son domaine peut être ajustée à μ^*. Si on ne détecte aucun cycle négatif dans $G_i^{\mu^*}$, alors les théorèmes 10 et 11 indiquent le filtrage à effectuer sur $\text{dom}(X_i)$.

Théorème 10. *Soit X_i une variable soumise à une contrainte* Multi-Inter-Distance. *Soit μ^* le plus petit temps limite plus grand que est_i parmi l'ensemble des variables soumises à la contrainte et soit $G_i^{\mu^*}$, le graphe modifié. Si $G_i^{\mu^*}$ n'a pas de cycle négatif, alors soit t, le plus grand entier tel que $D_{G_i^{\mu^*}}[est_i, t] = 0$. Toutes les valeurs inférieures à t dans le domaine de X_i n'ont pas de support d'intervalle.*

Démonstration. Puisque $G_i^{\mu^*}$ n'a pas de cycles négatifs, il existe au moins une solution avec $X_i \in [est_i, \mu^*)$ où $\mu^* = \min\{\mu_k \mid \mu_k > est_i\}$. Soit t le plus grand noeud dans $G_i^{\mu^*}$ pour lequel la plus courte distance à partir de est_i est nulle. Nous prouvons par contradiction que $t < \mu^*$. Supposons $t \geq \mu^*$. Le poids de l'arête (μ^*, est_i) est au plus -1 puisque ses noeuds incidents incluent le domaine diminué de X_i. Conséquemment il y a un cycle négatif dans $G_i^{\mu^*}$ qui va de est_i à t au coût de 0 et qui revient en empruntant les arêtes nulles de t à μ^* et ensuite l'arête (μ^*, est_i) de coût -1. Alors $t < \mu^*$.

Sachant que $t < \mu^*$, nous construisons le graphe modifié G_i^t. Mis à part X_i dont le domaine original est diminué à $[est_i, t)$, le domaine des variables dans G_i^t est le même que dans G. L'arête (t, est_i) a un poids négatif dans G_i^t puisque l'intervalle $[est_i, t)$ inclut complétement le domaine de X_i. La distance de est_i à t est nulle dans $G_i^{\mu^*}$ et, par le lemme 9, plus petite ou égale à 0 dans G_i^t. Ainsi, il y a un cycle négatif dans G_i^t passant par l'arête (t, est_i) ce qui implique qu'aucune valeur dans $[est_i, t)$ n'a de support d'intervalle dans $\text{dom}(X_i)$. □

Pour illustrer le Théorème 10, considérons la variable X_1 de l'exemple 5. Le plus petit temps limite qui soit plus grand que $est_1 = 7$ est 9. Le graphe modifié G_1^9 est identique au graphe G. De la matrice des poids de l'exemple 10, on constate que le chaîne (7,2,5,8) a un poids total $w = 0$ et que 8 est le plus grand noeud accessible à partir du noeud 7 à une distance de 0. Nous avons $t = 8 = \max(t \mid D_{G_i^{\mu^*}}[est_i, t] = 0)$. Alors le Théorème 10 stipule qu'aucune valeur inférieure à $t = 8$ n'a de support dans le domaine de X_1.

Le théorème suivant démontre que $X_1 = t = 8$ a un support d'intervalle. En effet, de l'exemple 10, on constate qu'il existe un support d'intervalle pour $X_1 = 8$. Il est donné par la solution : $X_1 = 8, X_2 = 2, X_3 = 5, X_4 = 6$ et $X_5 = 3$.

Théorème 11. *Soit X_i une variable soumise à une contrainte* Multi-Inter-Distance. *Soit μ^* le plus petit temps limite plus grand que est_i parmi l'ensemble des variables soumises à la contrainte et soit $G_i^{\mu^*}$, le graphe modifié. Si $G_i^{\mu^*}$ n'a pas de cycle négatif, alors soit t, le plus grand entier tel que $D_{G_i^{\mu^*}}[est_i, t] = 0$. La valeur t a un support d'intervalle dans le domaine de X_i.*

Démonstration. Nous construisons le graphe modifié G_i^{t+1}. Mis à part X_i dont le domaine original est diminué à $[est_i, t+1)$, le domaine des variables dans G_i^{t+1} est le même que dans G. Si l'on parvient à montrer que G_i^{t+1} n'a pas de cycles négatifs, alors nous prouvons que $X_i = t$

a un support d'intervalle. Du théorème 10, nous savons que $X_i \geq t$, alors $X_i = t$ demeure la seule valeur de $[est_i, t+1)$ qui satisfasse la contrainte. Le graphe modifié G_i^{t+1} n'a pas de cycles négatifs lorsque $t+1 = \mu^*$. Dans ce cas particulier, les graphes modifiés G_i^{t+1} et $G_i^{\mu^*}$ sont identiques. De la démonstration du théorème 10, nous savons que $t \leq \mu^*$. Il nous reste alors à démontrer que G_i^{t+1} n'a pas de cycles négatifs lorsque $t+1 < \mu^*$.
Soit μ_q, le plus grand temps limite strictement plus petit que μ^*. Par construction, il n'y a pas de temps limite plus grand que est_i et plus petit que μ^*, ainsi $\mu_q < est_i$. La présence des arêtes nulles nous assure que $D_{G_i^{t+1}}[est_i, \mu_q] \leq 0$. Pour n'importe quelle borne inférieure est_p plus petite ou égale à est_i, le domaine modifié de $X_i = [est_i, t+1)$ est le seul domaine qui est inclus dans $[est_p, t+1)$ mais pas dans $[est_p, \mu_q)$, ainsi $-w(\mu_q, est_p) = -w(t+1, est_p) - 1$. Le graphe G_i^{t+1} diffère de $G_i^{\mu^*}$ seulement par l'addition d'arêtes descendantes $(t+1, est_q)$ pour toutes les bornes inférieures $est_q \leq est_i$. Puisqu'il n'y a pas de cycles négatifs dans $G_i^{\mu^*}$, il est suffisant de démontrer que tous les cycles passant par ces arêtes supplémentaires $(t+1, est_q)$ ne sont pas de poids négatifs. Puisque l'addition des arêtes descendantes partant de $t+1$ n'affecte pas les chemins menant à $t+1$, nous avons $D_{G_i^{t+1}}[est_i, t+1] = D_{G_i^{\mu^*}}[est_i, t+1]$. Par la définition de t nous obtenons $0 < D_{G_i^{t+1}}[est_i, t+1] = D_{G_i^{\mu^*}}[est_i, t+1]$. Conséquemment, tout chemin partant de est_i et aboutissant à $t+1$ est de poids positif. Ce qui inclus nécessairement tous les chemins passant par l'arête (μ_q, est_p). Nous avons donc :

$$0 < D_{G_i^{t+1}}[est_i, \mu_q] + w(\mu_q, est_p) + D_{G_i^{t+1}}[est_p, t+1]$$

En substituant $w(\mu_q, est_p) = w(t+1, est_p) + 1$ et $D_{G_i^{t+1}}[est_i, \mu_q] \leq 0$ dans l'équation ci-haut nous obtenons :

$$0 < 0 + w(t+1, est_p) + 1 + D_{G_i^{t+1}}[est_p, t+1]$$
$$0 \leq w(t+1, est_p) + D_{G_i^{t+1}}[est_p, t+1]$$

Donc aucun cycle passant par l'arête $(t+1, est_p)$ n'est négatif et par conséquent, il n'y a aucun cycle négatif dans G_i^{t+1}. □

L'algorithme 12 filtre les bornes inférieures d'un ensemble de tâches régi par une contrainte Multi-Inter-Distance. Une seconde exécution de l'algorithme sur l'opposé mathématique du problème filtre les bornes supérieures et permet d'obtenir la cohérence de bornes. L'algorithme 12 est donc un propagateur idempotent de la contrainte Multi-Inter-Distance. L'algorithme débute par un appel à l'algorithme 9 (ligne 1) sur les données brutes du problème. S'il y a un cycle négatif dans le graphe d'ordonnancement original, l'algorithme 9 le détecte et met fin à l'exécution. Le contraire nous assure que chaque variable dispose d'un support de solution. Par conséquent, l'algorithme 12 ne procède au filtrage des tâches que si la contrainte peut être satisfaite.

Algorithme 12 : FiltreBornesInférieures $(est[1,\ldots,n], lct[1,\ldots,n], m, p)$

```
1  ÉvaluationDesTempsDeDépart (est[1,...,n], lct[1,...,n], m, p)
2  F ← ∅
3  U ← {μ_1,...,μ_n}
   for j ∈ {1,...,n} do
       repeat
4          l ← min([est_j, μ_j) \ F)
5          μ* ← min((l, μ_j] ∩ U)
6          G_j^{μ*} ← le graphe modifié avec X_j ∈ [l, μ*)
7          D_{G_j^{μ*}}[l] ← ÉvaluePlusCourtesDistances (l, G_j^{μ*})
           if Détecte un cycle négatif dans G_j^{μ*} then
8              F ← F ∪ [l, μ*)
       until Il n'y a pas de cycle négatif dans G_j^{μ*}
9      est_j ← argmax_t {t | D_{G_j^{μ*}}[l][t] = 0}
```

L'algorithme 12 initialise un support (ligne 2) pour l'ensemble des régions tabous F. Le propagateur tire profit du résultat (théorème 8) en vertu duquel une région tabou définie pour une tâche j devient région tabou pour toutes les tâches subséquentes dont le temps limite n'est pas inférieur à μ_j. Les tâches sont donc triées en ordre non-décroissant de leur temps limite (ligne 3) et la liste ordonnée U est ainsi composée. Pour le filtrage (lignes 4 à 9), l'algorithme itère sur chaque tâche dans l'ordre du rang de leur temps limite dans la liste U. La ligne 4 trouve le premier noeud non-inférieur au temps de sortie de la tâche en traitement se situant hors des régions tabous. L'algorithme note ce noeud l. De même, la ligne 5 trouve dans la liste U le premier temps limite, noté μ^*, suivant la valeur l. Les valeurs l et μ^* sont substituées aux valeurs originales de temps de sortie et de temps limite de la tâche en traitement pour constituer le graphe d'ordonnancement modifié (ligne 6). Ces substitutions sont suffisantes pour s'assurer qu'une même région tabou ne soit pas découverte plus d'une fois [26]. Si un cycle négatif est détecté dans le graphe modifié (ligne 8), la région tabou $[l, \mu^*)$ est ajoutée à l'ensemble F en vertu du théorème 10. Dans le cas contraire, la ligne 9 applique le filtrage définitif à la borne inférieure de la tâche tel que stipulé par le théorème 11. Le filtrage requiert de trouver le plus grand noeud du graphe modifié dont la plus courte distance du noeud l est nulle. Pour ce faire, l'algorithme 12 fait appel, à la ligne 7, à une version modifiée de l'algorithme López-Ortiz-Quimper [27], l'algorithme *ÉvaluePlusCourtesDistances*. Tout comme l'algorithme dont il est une généralisation, *ÉvaluePlusCourtesDistances* détecte les cycles négatifs et sa complexité est dans la famille $O(n^2)$. La section 3.3.3 traite des modifications apportées à l'algorithme 9 pour obtenir les plus courtes distances à partir d'un noeud quelconque l. Un survol de l'implémentation de l'algorithme 12 y est aussi présentée.

26. En choisissant μ^* le plus petit possible comme dans l'algorithme 12, on découvre les régions tabous le plus tôt possible. Le résultat démontré par le théorème 8 est ainsi exploité de façon optimale.

Calculer les plus courtes distances dans le graphe modifié mène à l'une ou l'autre des éventualités suivantes : la détection d'un cycle négatif dans le graphe ou la certitude qu'un horaire est réalisable, auquel cas la boucle *repeat* se termine. La terminaison de la boucle se produit exactement n fois, une fois pour chaque variable. Si le graphe modifié contient un cycle négatif, le propagateur boucle tout en créant la région tabou $[l, \mu^*)$. La valeur μ^* est nécessairement la borne supérieure du domaine de l'une des variables. À chaque fois qu'une nouvelle région tabou est créée deux situations peuvent se produire. La première est que la valeur μ^* devienne pour la première fois la borne supérieure ouverte d'une région tabou. La seconde est qu'une valeur μ^* soit incluse dans une région tabou. Ces deux situations surviennent une fois chacune au maximum pour toutes les n valeurs possibles de μ^* à l'exception de μ_{max} qui ne peut techniquement s'inclure dans un intervalle. Ainsi, le propagateur crée un maximum de $2n-1$ régions tabous. Au total, en incluant l'appel initial, le propagateur lance l'algorithme de López-Ortiz et Quimper (algorithme 9) ou sa version étendue (*ÉvaluePlusCourtesDistances*) un maximum de $3n$ fois ce qui place sa complexité dans la famille $O(n^3)$.

3.3.3 L'implémentation

La ligne 7 de l'algorithme 12 fait un appel à l'algorithme *ÉvaluePlusCourtesDistances* qui retourne les plus courtes distances d'un noeud quelconque l à tous les autres noeuds du graphe d'ordonnacement. Nous montrons comment modifier l'algorithme 9 pour obtenir ce résultat.

Dans l'algorithme 9, le vecteur d fait la jonction entre la relaxation des arêtes ascendantes et des arêtes descendantes dans lesquelles il est constamment mis à jour. Au terme de son exécution, il contient pour chaque position $a \in \{0, \ldots, n\}$, le plus grand des noeuds accessibles à une plus courte distance d'au plus $-a$ du noeud horizon μ_{max}. La présence des arêtes nulles assure que les noeuds à une égale distance sont consécutifs dans le graphe d'ordonnancement. Ainsi, il suffit de ne conserver que le plus grand noeud dans le vecteur d pour connaître la plus courte distance de tous les noeuds du graphe. Toutefois, l'optimisation appliquée à l'algorithme de Bellman-Ford, rendue possible par les propriétés particulières au graphe d'ordonnancement (voir section 3.2.2), fait en sorte que les routines de relaxation des arêtes (algorithmes 11 et 10) ignorent plusieurs arêtes et plusieurs noeuds. Il importe donc d'initialiser le vecteur d en s'assurant d'affecter à chacune de ses positions, un noeud que le noeud horizon est certain d'accèder à la plus courte distance correspondante. Dans l'algorithme 9, on l'initialise comme suit : $d = [\mu_{max}, r_{min}, \ldots, r_{min}]$ soit le noeud horizon (μ_{max}) en position 0 et le plus petit noeud du graphe d'ordonnancement (r_{min}) à chacune des n positions correspondant aux distances négatives. Puisqu'il y a nécessairement une arête de poids $-n$ reliant μ_{max} à r_{min} dans le graphe d'ordonnancement, l'affectation r_{min} est correcte pour chacune des positions du vecteur correspondant aux distances négatives $\{-1, \ldots, -n\}$.

Pour obtenir les plus courtes distances d'un noeud quelconque l à tous les autres noeuds t du graphe modifié, on doit rallonger le vecteur d pour permettre de traiter les plus courtes distances positives, $\delta(l,t) \in \{1,\ldots,n\}$ [27]. Le vecteur d résultant comporte donc $2n+1$ positions. Par commodité, on choisit de le représenter en ordre croissant des distances, de $-n$ à n comme sur la figure 3.5. Le vecteur d de l'algorithme *ÉvaluePlusCourtesDistances* contient le plus grand noeud accessible du noeud l à la plus courte distance correspondant à sa position dans le vecteur, de $-n$ à n. Pour le filtrage, l'algorithme 12 nécessite (ligne 9) de connaître le plus grand noeud t accessible à une distance nulle de l. Cette valeur est contenu dans la position centrale du vecteur d au terme de l'exécution. S'il y a un cycle négatif dans le graphe modifié, il est détecté lorsque le noeud l ou un noeud plus grand se retrouve à l'une des positions du vecteur d qui correspond à une plus courte distance négative. Pour l'initialisation du vecteur d, il importe de procéder avec les mêmes réserves que dans l'algorithme 9, c'est-à-dire affecter aux positions du vecteur un noeud que le noeud source l est certain d'atteindre à la plus courte distance correspondante. S'il s'avère que le noeud l puisse atteindre un noeud plus grand, l'algorithme le trouve nécessairement lors de son exécution.

Nous initialisons le vecteur $d[-n,\ldots,n]$ par morceau de la façon suivante :

Soit $w = -|\{\mu_j \mid \mu_j \le l\}|$, le poids de l'arête descendante (l, r_{min}).

$$d[-n,\ldots,w-1] = [r_{min} - (w+n)p, r_{min} - (w+n-1)p, \ldots, r_{min} - p] \tag{3.17}$$

$$d[w,\ldots,-1] = [r_{min},\ldots,r_{min}] \quad \text{si } w < 0 \tag{3.18}$$

$$d[0,\ldots,n-1] = [l,\ldots,l] \tag{3.19}$$

$$d[n] = \mu_{max} \tag{3.20}$$

Les valeurs dans $d[0,\ldots,n-1]$ sont initialisées à l car il est trivial que le noeud source l puisse être atteint de l à toute distance positive ou nulle. Quant au noeud μ_{max}, tous les noeuds du graphe, donc l, peuvent le joindre à une distance n (voir section 3.2.1). Puisque le noeud r_{min} est à une distance d'au plus w du noeud l, il est correct d'initialiser les positions w à -1 du vecteur d avec la valeur r_{min}. Par ailleurs, il est possible que le noeud source l ne puisse joindre aucun noeud à une distance inférieure à w. Par conséquent, les positions inférieures à w doivent être remplies avec des valeurs non-significatives. En procédant selon (3.17), on s'assure que l'algorithme 11 n'effectue aucune mise à jour incorrecte. La figure 3.5 illustre le processus d'initialisation.

27. Du lemme 4 nous avons que le vecteur des distances Δ est monotone croissant. Chaque unité d'accroissement dans le vecteur des distance, $\Delta[t+1] > \Delta[t]$, correspond au départ d'une tâche au temps t dans la solution (3.15). Or il peut y avoir jusqu'à n temps de départ non-inférieurs à l.

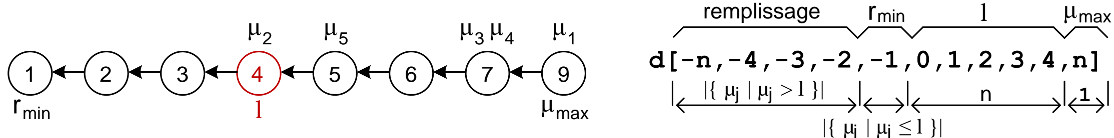


FIGURE 3.5: Illustration de l'initialisation du vecteur d de l'exemple 5 pour $l = 4$. La partie de gauche montre le graphe d'ordonnacement et le noeud source l. Les temps limite sont indiqués en haut des noeuds. La partie de droite montre le vecteur d, ses $2n+1$ positions et les quatre morceaux considérés pour l'initialisation. On peut voir en haut, les valeurs d'affectation et en bas les expressions qui permettent d'évaluer le nombre de positions d'affectation pour chacune.

Le résultat de l'initialisation est le vecteur $d = [-12, -9, -6, -3, 1, 4, 4, 4, 4, 4, 9]$. Un seul temps limite ($\mu_2$) n'est pas supérieur à $l = 4$, alors $w = -1$. Seulement la première position inférieure à 0 est initialisée avec la valeur $r_{min} = 1$. Les 4 autres positions négatives sont comblées de manière à éviter que la routine de relaxation des arêtes ascendantes, algorithme 11, produise des modifications sur le vecteur d.

La complexité du propagateur (algorithme 12) est dominée par celle de l'algorithme ÉvaluationDesTempsDeDépart (algorithme 9). Conséquemment plusieurs structures de données sont utilisables sans influencer la complexité de l'algorithme. Fort de ce luxe théorique, nous optons pour une optimisation pratique du propagateur et nous présentons un survol de sa structure de données et mécanismes principaux.

Nous formons une liste strictement croissante avec tous les représentants du problème originel, la liste β. La liste est doublement chaînée. Une sentinelle fait office de pointeur incident et détient le nombre d'eléments contenus dans β. La liste ainsi formée permet de transiter rapidement à l'exécution sur l'opposé mathématique du problème (filtrage des bornes supérieures) sans avoir à reconstruire la structure de données. Aux lignes 4 et 5, l'algorithme 12 procède à des substitutions qui peuvent mener à des modifications (ligne 6) du graphe d'ordonnancement. Jusqu'à deux éléments [28] à la fois peuvent être retirés de la liste β et ces éléments doivent être réinsérés par la suite [29]. Pour permettre la réinsertion des éléments à leur position d'origine en temps constant, nous avons recours au mécanisme appelé *danse de Knuth* [30] qui consiste notamment à ne pas détruire les pointeurs des éléments temporairement retirés.

28. Lorsque le domaine d'une tâche est modifié, il peut survenir que son représentant de temps de sortie ou son représentant de temps limite ou même les deux deviennent représentant de rien.

29. Obtenir la cohérence de bornes est un processus idempotent. Pour filtrer une tâche, il est inutile de connaître le filtrage appliquée à la tâche précédente. Ainsi, pour simplifier le processus, l'algorithme filtre les tâches une à une en conservant les valeurs originelles de temps de sortie et de temps limite pour toutes les autres tâches du problème.

30. La danse de Knuth s'applique si les éléments d'une liste doublement chaînée sont réinsérés dans l'ordre inverse de celui auquel ils ont été retirés. Elle est utilisée dans plusieurs solveurs de contraintes pour le mémorisation des domaines énumérés. Lorsque le solveur plonge dans l'arbre de recherche, le domaine d'une variable ne peut que perdre des valeurs (voir la Figure 1.1). Le solveur place sur une pile les valeurs retranchées à chaque noeud et procède à leur réinsertion par le dessus de la pile lorsqu'il remonte à un noeud parent. Outre son efficacité computationnelle, ce mécanisme permet au solveur d'économiser la mémoire disponible en ne nécessitant qu'une seule base de données par variable.

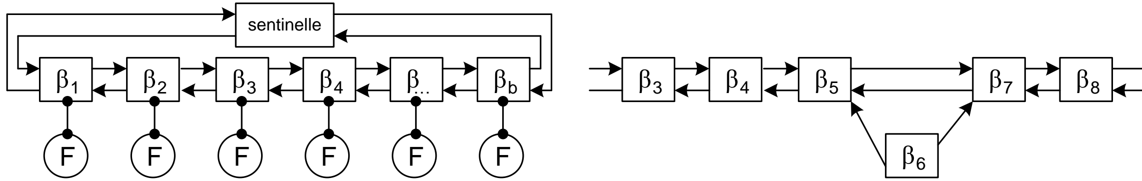


FIGURE 3.6: La partie de gauche schématise la structure de données principale. On y voit la liste doublement chaînée β et son pointeur incident, la sentinelle. La portion de droite illustre le premier pas du mécanisme appelé danse de Knuth. On vient de procéder au retrait de l'élément β_6 de la liste. En conservant les pointeurs de l'élément retiré, il est possible de le réinsérer à sa place dans la liste en temps constant.

Pour encoder les régions tabous F dans l'algorithme 12, nous avons recours à la structure de données *Union-Find*[31] initialisée avec les éléments de la liste β (schématisé par les cercles identifiés F sur la figure 3.6). Lors de la découverte d'une région tabou $[r_i, \mu_j)$, on unit tous les éléments dans l'intervalle $[r_i, \mu_j)$ dans un même ensemble. Il est ensuite possible d'interroger la structure de données à savoir quelle est la borne supérieure de la région tabou pour construire le graphe modifié (ligne 4).

3.3.4 Expérimentations

Dans cette section, nous modélisons le problème de séquençage des atterrissages d'avion (Runway Sequencing with Holding Pattern). Quand un aéronef atteint l'aire réservée aux abords d'un aéroport, il est pris en charge par le TRACON (Terminal Radar Approach CONtrol). Il lui est alloué un espace dans lequel il tourne en boucles jusqu'à sa descente finale. Seule une portion de sa boucle est un temps propice à l'amorce de l'atterrissage. Nous désignons par intervalle, la fenêtre maximale de la boucle d'attente qui est propice à l'amorce de l'atterrissage compte tenu des variations de vitesse et d'amplitude de boucle que l'appareil peut réaliser. Pour chaque appareil pris en charge, un nombre limité de boucles est envisagé en fonction de la disponibilité de l'espace d'attente alloué et de la capacité de l'appareil à attendre[32]. Aux fins de la modélisation, nous notons k_i le nombre d'intervalles disjoints pour l'aéronef i et par $s_{ij} = [a_{ij}, b_{ij}]$, le $j$ième intervalle dans lequel l'avion i peut amorcer sa descente finale (pour $1 \leq j \leq k_i$). Nous notons m le nombre de pistes d'atterrissage disponibles et p le temps minimum entre deux atterrissages successifs sur la même piste. Le problème consiste à maximiser p tout en faisant atterrir tous les appareils.

Le problème consistant à trouver une séquence réalisable pour une seule piste est abordé par Artiouchine, Baptiste et Dürr [2]. Ils considèrent d'abord une forme relaxée dans laquelle les

31. Chaque ensemble de la structure *Union-Find* est en fait un arbre dans lequel on se déplace toujours de l'un de ses noeuds vers la racine. Les propriétée utiles de l'ensemble, comme ici son élément le plus grand, sont tenues à jour à la racine de l'arbre.

32. Entres autres facteurs à considérer : la réserve de carburant et les heures de départ des transits de certains passagers.

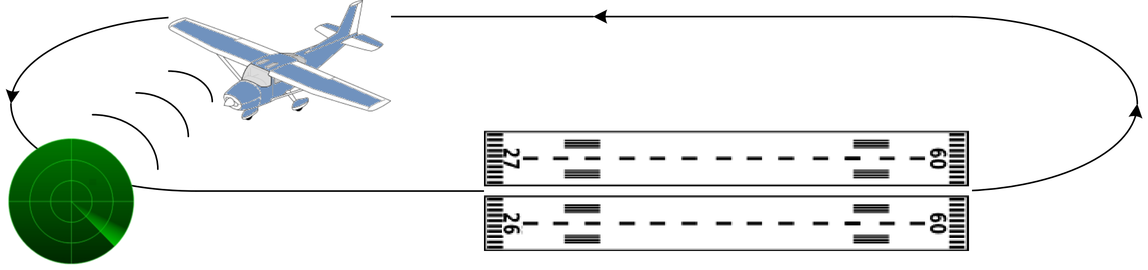


FIGURE 3.7: Illustration naïve d'une boucle d'attente.

intervalles sont de même longueur et équidistants pour tous les appareils. Ils réfèrent à cette forme simplifiée du problème par l'appellation Mono-Pattern Runway Scheduling Problem. Lorsqu'un seul intervalle ($k_i = 1, \ \forall i$) est allouée à chaque avion et que la durée p est fixe, le problème est résolu en temps polynomial par l'algorithme de Garey et al. [17] dont il est question en début de chapitre. Les auteurs affirment [2] que de déterminer si une solution existe pour le Mono-Pattern Runway Scheduling Problem est NP-Complet dès que la durée est supérieure à 2 ($p > 2$) et que le nombre d'intervalles par appareil est plus grand que 3 ($k_i > 3, \ \forall i$) [33]. Leur article traite de différentes formes de relaxation du problème qui peuvent mener à une solution polynomiale. Mentionnons au passage un algorithme dynamique qui résout en $O(n^2 s^3 T^2 p^{-2} 2^{2s\frac{T}{p}})$ où T est le temps maximal d'attente pour un appareil à partir de sa prise en charge par la TRACON et s le nombre maximum d'intervalles alloué à un même appareil. Ce qui constitue une solution polynomiale si $s\frac{T}{p}$ est fixe ! Dans le problème qui nous occupe, soit maximiser la valeur de p, ni s pas plus que T ne sont des valeurs fixes. Par conséquent, le problème est NP-Difficile.

Artiouchine et Baptiste [2, 3] mettent à profit un solveur de contraintes pour s'attaquer au problème à une seule piste. Bien que l'approche ne promette pas de solution à coup sûr, elle présente l'avantage de procurer la meilleure solution trouvée dans un temps de calcul donné. Les instances sont construites comme suit [34]. Pour chacun des n appareils, de un à cinq intervalles disjoints sont obtenus aléatoirement. Les intervalles répondent aux paramètres suivants : l'*horizon* qui fixe la limite supérieure du début du premier intervalle, l'*amplitude* qui restreint la longueur maximale d'un intervalle et la *rupture* qui est la distance maximale entre la fin d'un intervalle et le début de celui qui suit. L'échantillon contient 384 instances regroupées en six ensembles de 64 instances en fonction du nombre d'appareils noté n, $n \in \{15, 30, 45, 60, 75, 90\}$. Pour chaque ensemble, les paramètres varient de façon à produire des instances plausibles parmi les valeurs $\{75, 450, 825, 900, 1200, 1500, 1650, 2400\}$ pour l'*amplitude* et la *rupture* et $\{1125, 2250, 3375, 4500, 5625, 6750, 9000, 11250, 13500\}$ pour l'*horizon*.

33. Pour $p = 2$ et $s_i = 3$, le Mono-Pattern Runway Scheduling devient un cas particulier du problème *Scheduling short tasks with few starting times* pour lequel une solution polynomiale est connue.

34. Nous nous servons du deuxième ensemble de données disponible pour téléchargement sur le site de [3] au http//www.lix.polytechnique.fr/baptiste/flight-scheduling-data.zip.

```
nomFichier = R_15_1
<Si=4> 244 292 322 340 365 429 481 502
<Si=5> 934 954 975 1027 1030 1046 1096 1116 1149 1216
<Si=3> 335 347 406 475 548 595
<Si=1> 519 586
<Si=2> 182 235 237 305
<Si=5> 114 156 165 216 246 288 296 320 324 356
<Si=5> 542 543 554 588 659 699 731 772 789 835
<Si=2> 982 1011 1036 1070
<Si=3> 430 434 490 512 586 631
<Si=4> 460 481 504 534 597 639 687 699
<Si=5> 398 465 486 528 543 606 659 683 723 781
<Si=4> 1042 1111 1150 1187 1223 1241 1305 1356
<Si=5> 834 897 962 1027 1093 1126 1180 1197 1215 1236
<Si=5> 900 974 1041 1066 1086 1118 1126 1199 1240 1262
<Si=2> 604 611 681 722
n = 15
horizon = 1125
amplitudeMax = 75
rupture = 75
```

FIGURE 3.8: Exemple des données d'une instance à 15 avions. La balise S_i au début des i lignes indique le nombre d'intervalles k_i pour l'avion i. Les nombres qui suivent sont les bornes inférieures (a_{ij}) et supérieures (b_{ij}) des j intervalles disjoints s_{ij}.

Le modèle nécessite deux variables par avion. La première, B_i, désigne l'intervalle (B pour boucle) pendant lequel l'appareil amorce sa descente finale. Son domaine énuméré est formé des entiers 1 à k_i. La seconde variable, X_i, est le temps d'amorce et son domaine est borné par les valeurs correspondantes des bornes de l'intervalle s_{ij} pour $B_i = j$. Le radar prend en charge n avions. Pour $1 \leq i \leq n$ nous avons :

$$\text{dom}(B_i) = \{1, \ldots, k_i\} \qquad \text{et} \qquad \text{dom}(X_i) = [a_{i1}, b_{ik_i}]$$

Pour lier les deux variables, nous utilisons des contraintes d'implication.

$$B_i \leq j \Rightarrow X_i \leq b_{ij} \quad \forall i, \forall\ 1 \leq j < k_i$$
$$B_i \geq j \Rightarrow X_i \geq a_{ij} \quad \forall i, \forall\ 2 \leq j \leq k_i$$

On doit déterminer si une solution existe pour une valeur donnée de p compte tenu de tous les intervalles s_{ij} disponibles. Le solveur branche d'abord sur les B_i et procède au filtrage des bornes du domaine des S_i correspondant. Si aucun horaire est réalisable pour une valeur de $B_i = v$, la valeur v est retirée du domaine de B_i. Lorsque le solveur dispose d'une affectation cohérente pour tous les B_i, il tente d'établir un horaire précis en affectant des valeurs aux variables X_i [35].

35. Pour affecter une valeur à une variable définie comme ayant un *domaine borné*, le solveur procède en réduisant le domaine à cette seule valeur, $\text{dom}(X_i) = [v, v]$.

Pour maximiser p, on initialise par la méthode dite du galop. On assigne la valeur 1 à p. On itère en doublant la valeur de p tant qu'une solution est trouvée. Lorsque le solveur conclut à l'infaisabilité, on fixe $p_{max} = p$ et $p_{min} = \frac{p_{max}}{2}$. Ensuite on boucle de manière dichotomique en cherchant un horaire réalisable pour la valeur médiane p.

$$p = \left\lfloor \frac{p_{min} + p_{max}}{2} \right\rfloor$$

Si un horaire est trouvé, on affecte p à la valeur minimale ($p_{min} = p$) sinon p devient la borne supérieure d'infaisabilité ($p_{max} = p$). On termine si $p_{min} + 1 = p_{max}$ ou lorsque la limite de temps de calcul est atteinte. L'optimum cherché est donné par p_{min}.

Les variables X_i et le paramètre p du modèle précédent sont assimilables à une contrainte Inter-Distance($[X_1, \ldots, X_n], p$). Or l'Inter-Distance est une spécialisation de la contrainte Multi-Inter-Distance($[X_1, \ldots, X_n], m, p$) pour laquelle $m = 1$ (voir section 1.1). Le modèle et les données d'Artiouchine et Baptiste permettent de mettre en test le propagateur construit aux sections précédentes. Il est aussi possible d'étendre la modélisation aux cas d'aéroports disposant de pistes multiples ($m > 1$).

La contrainte Multi-Inter-Distance($[X_1, \ldots, X_n], m, p$) est une spécialisation (voir section 1.2) de la contrainte Cumulative($[X_1, \ldots, X_n], [h_1, \ldots, h_n], [d_1, \ldots, d_n], C$) pour laquelle $m = C$ et pour laquelle les hauteurs de toutes les tâches sont unitaires et leur durée est un paramètre constant ($h_i = 1$ et $d_i = p$ pour $1 \leq i \leq n$). Il est donc possible de substituer la contrainte Cumulative à la contrainte Multi-Inter-Distance et d'appliquer la modèle d'Artiouchine et Baptiste.

Nous comparons les résultats de notre propagateur de la contrainte Multi-Inter-Distance (MID) à ceux du propagateur de la Cumulative de Mercier et Van Hentenryck [28] (MVH) qui filtre la Cumulative par les méthodes Edge-Finding et Extended Edge-Finding (voir section 2.1) et à l'overload checking [44][49] (OC) qui effectue un test de *E-Feasibility* (voir section 2.1). Ces deux propagateurs proviennent du solveur Choco. Un problème est réputé résolu si un horaire réalisable est trouvé pour p et qu'il est démontré qu'il n'existe aucun horaire réalisable pour $p+1$. Les expérimentations ont été effectuées en décembre 2010 sur le Colosse de l'Université Laval. Les instances s'exécutent aléatoirement sur des processeurs Opteron 2375 séquencés à 2.2 GHz ou sur des Opteron 2376 séquencés à 2.3 GHz.

Pour les expérimentations portant sur $m = 1$, nous exécutons le modèle sur toutes les instances (348) de l'échantillon en allouant 20 minutes par instance. La figure 3.9 esquisse la courbe du nombre d'instances résolues en ordonnée en-deça du temps inscit en abcisse. Les échelles de comparaison sont logarithmiques pour refléter la nature exponentielle du problème.

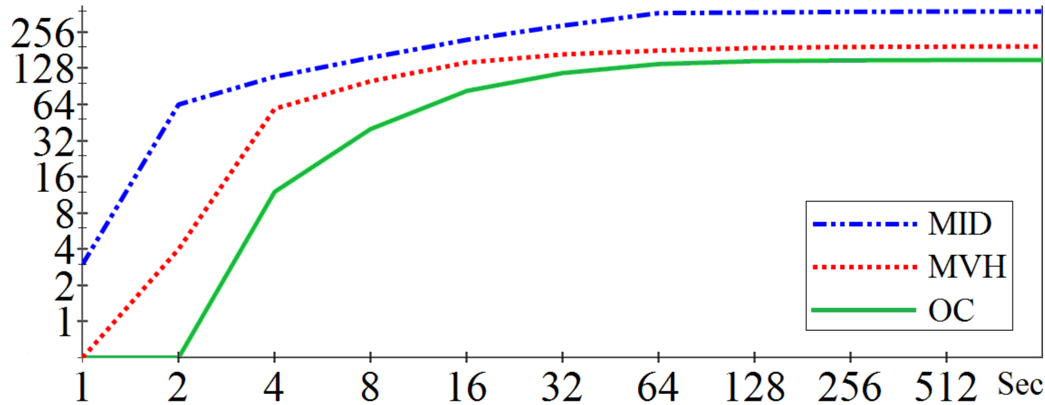


FIGURE 3.9: Nombre de problèmes résolus en fonction du temps pour $m = 1$.

Le tableau 3.1 comparent les retours-arrière. Les résultats y sont présentés séparément en fonction de la taille de l'instance.

n	OC		MVH		MID	
15	94%	999194	88%	106325	17%	22
30	95%	2755211	88%	332083	16%	333
45	97%	2402617	97%	306429	22%	1040
60	100%	1583362	97%	238024	22%	1515
75	98%	1101707	94%	152017	28%	3111
90	98%	927161	98%	106661	36%	238

TABLE 3.1: Pourcentage des instances qui génèrent des retours-arrière et nombre moyen de retours-arrière effectués pour les instances inclus dans le pourcentage de la première colonne pour $m = 1$.

Nous nous servons des mêmes données pour modéliser un aéroport à pistes multiples. Pour obtenir les instances à deux pistes ($m = 2$), nous combinons deux instances de l'échantillon originel sélectionnées aléatoirement parmi celles dont au minimum deux valeurs sont identiques pour les paramètres *amplitude*, *rupture* et *horizon*. Neuf ensembles de 16 instances pour chacune des valeurs de $n \in \{30, 45, 60, 75, 90, 105, 120, 135, 150\}$ sont ainsi créés. Pour les instances à trois pistes ($m = 3$), nous procédons en combinant 3 instances de l'échantillon originel. Nous choisissons au hasard parmi celles dont l'un des trois paramètres ci-haut mentionnés est identique et que la même valeur est partagée par au moins deux instances pour les deux autres paramètres. Neuf ensembles de 16 instances pour chacune des valeurs de $n \in \{45, 60, 75, 90, 105, 120, 135, 150, 165\}$ sont créés. Dans tous les cas de composition d'instance pour les tests sur pistes multiples, nous prenons garde de ne combiner que des instances de l'échantillon originel dont la taille diffère par au plus $n = 15$ appareils. La figure 3.10 présente les résultats combinés des 144 instances pour $m = 2$ et des 144 instances pour $m = 3$. Seuls les problèmes réputés résolus y sont cumulés.

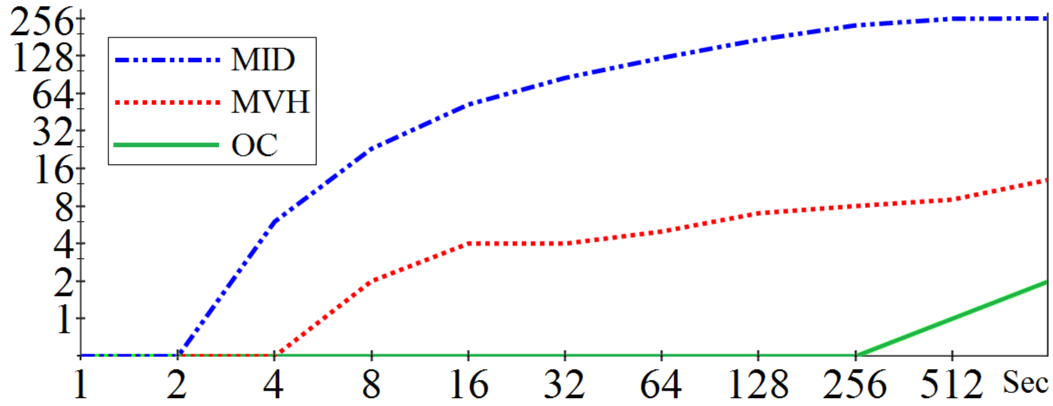


FIGURE 3.10: Nombre de problèmes résolus en fonction du temps pour $m \in \{2, 3\}$.

La tableau 3.2 présente les résultats pour les retours-arrière dans des colonnes distinctes pour $m = 2$ pistes et $m = 3$ pistes. Le propagateur *Overload Checking* n'y figure pas puisqu'il est parvenu à ne résoudre que deux instances. Par conséquent sa performance en matière de retours-arrière n'est pas significative.

	$m = 2$				$m = 3$			
n	MVH		MID		MVH		MID	
30	100%	722423	31%	177				
45	100%	322937	31%	4607	100%	376261	31%	4
60	100%	215728	15%	543	100%	213391	50%	2840
75	100%	129688	33%	1436	100%	120684	38%	128
90	100%	82496	29%	4781	100%	86733	33%	61
105	100%	66912	56%	1619	100%	60971	19%	1532
120	100%	58777	50%	785	100%	44511	25%	2583
135	100%	46525	25%	1236	100%	38623	44%	451
150	100%	29326	44%	1324	100%	27765	44%	518
165					100%	25181	47%	775

TABLE 3.2: Pourcentage des instances qui génèrent des retours-arrière et nombre moyen de retours-arrière lorsqu'au moins un retour-arrière est généré pour $m \in \{2, 3\}$.

La tableau 3.2 montre que le nombre de retours-arrière tend à diminuer avec la taille de l'instance. Pour le propagateur MVH dont chacune des instances nécessitent au moins un retour-arrière, cette tendance s'explique tout simplement parce qu'il prend plus de temps d'évaluer une instance de plus grande taille. Par conséquent, le solveur parcourt moins de noeuds ce qui génère moins de retours-arrière. Or le temps limite est fixé à 20 minutes par instance peu importe sa taille et la proportion d'instances résolues est inversement proportionnelle à la taille pour le propagateur MVH. La même raison s'applique à la MID dans une plus faible mesure car elle permet de résoudre une bonne proportion des instances en-deça des 20 minutes allouées. Il faut aussi tenir compte du fait que les instances sur pistes multiples sont issues

de la conjugaison d'instances de l'échantillon originel. Ainsi, les instances contiennent plus de disparité entre chacun des aéronefs modélisés. Par exemple pour m=3, il est impossible de combiner 3 instances de même taille parmi les tailles $n = \{15, 30, 45, 60, 75, 90\}$ et obtenir une instance de taille $n = 60$ ou $n = 120$. D'ailleurs, du tableau 3.2 on constate que les instances de taille $n = \{45, 90, 135\}$ ont tendance à générer moins de retours-arrière. Il faut interpréter le tableau 3.2 sur une base comparitive des performances des propagateurs qui y figurent.

La complexité du propagateur de Mercier et VanHentenryck pour $k = 1$ est de $O(n^2)$ alors que celle de l'Overload Checking est de $O(n \log n)$. Pourtant notre propagateur dont la complexité est de $O(n^3)$ obtient de meilleurs résultats. On pourrait tenter d'expliquer par le fait que les deux autres propagateurs ne sont pas idempotents. Par conséquent, il arrive généralement que plus d'une itération leur soit nécessaire pour l'obtention d'un point fixe à chaque appel du solveur. On devrait alors obtenir la même configuration des résultats pour $m = 1$ et $m > 1$, ce qui n'est pas le cas. De plus, le taux de réussite des propagateurs les plus rapides devraient se démarquer en proportion de la taille de l'instance en cause. Or, on observe le contraire. L'explication la plus plausible est que la quantité de filtrage prévaut dans ce type de problème. En effet, les trois propagateurs performent inversément à leur complexité et dans l'ordre de la quantité de filtrage qu'ils effectuent. Le propagateur de la Multi-Inter-Distance force la cohérence de bornes et cet avantage est sans aucun doute perçu dans le nombre de retours-arrière généré. D'ailleurs, la seule façon d'expliquer que plus de la moitié des instances à 3 pistes ($m = 3$) soit résolue sans aucun retour-arrière avec la MID, est qu'elle parvient à éliminer la majorité des valeurs incohérentes du domaine des variables B_i. Elle oriente directement l'heuristique vers l'une des solutions.

Un propagateur n'est qu'un petit maillon d'un solveur de contraintes. Sa complexité importe parce qu'il peut être appelé très souvent par le solveur. La quantité de filtrage qu'il effectue importe parce qu'un plus grand filtrage réduit l'arbre de recherche dans une plus grande mesure. Il est probable que la dialectique *rapidité / quantité de filtrage* sera toujours un vif débat au sein de la communauté de programmation par contraintes. Il est probable aussi que la nature du problème et plus particulièrement sa modélisation influent sur les caractéristiques de performance qui prévalent. Toutefois, s'il nous est permis d'émettre une opinion, nous penchons du côté de la quantité de filtrage parce qu'elle pousse dans la direction que tire le solveur... qui n'est, somme toute, qu'un outil sophistiqué pour résoudre un problème par élimination. Cependant d'aucune façon nous prétendons détenir la vérité en ce domaine et encore moins que nos expérimentations soient concluantes à cet égard. L'objectif de ce chapitre est avant-tout la conception d'un propagateur de contrainte qui applique la cohérence de bornes.

Conclusion

Dans ce mémoire nous avons principalement étudié deux contraintes : la cumulative pour laquelle trouver un horaire réalisable est NP-Difficile et la multi-inter-distance dont on peut forcer la cohérence de bornes en temps polynomial.

Dans la cas de la contrainte cumulative, on constate qu'il existe une multitude de règles de filtrage qui mènent à autant d'algorithmes. On remarque qu'aucun de ces algorithmes n'est idempotent et qu'il est nécessaire de relancer leur exécution jusqu'à l'obtention d'un point fixe. Par conséquent, on les veut les plus rapides possible. Dans cette optique, nous avons exploré deux approches : réduire le temps d'exécution par l'exploitation de structures de données appropriées et augmenter le filtrage par la conjugaison de plusieurs règles dans un même algorithme.

Pour la contrainte multi-interdistance, nous avons construit un algorithme qui applique la cohérence de bornes en exploitant les propriétés d'un graphe d'ordonnancement fortement structuré. Les expérimentations tendent à démontrer qu'il est profitable de sacrifier la performance au profit d'un filtrage plus fort pour résoudre certains problèmes.

Dans un premier temps, on peut se demander si la théorie des graphes pourrait inspirer de nouveaux algorithmes qui obtiennent un meilleur filtrage pour la contrainte cumulative. Dans un second temps, l'on pourrait se demander s'il existe une structure de données apte à abaisser la complexité de l'algorithme conçu pour la contrainte multi-inter-distance.

Annexe A

Tableau des notations

notation	signification
est_i	temps de sortie de la tâche i
lct_i	échéance de la tâche i
p_i	temps de traitement de la tâche i
h_i	hauteur de la tâche i
S_i	temps de départ de la tâche i
μ_i	temps limite de la tâche i
F_i	fenêtre de faisabilité de la tâche i
ect_i	temps minimum de terminaison de la tâche i
lst_i	temps maximum de départ de la tâche i
e_i	énergie de la tâche i
Env_i	enveloppe de la tâche i
Env_i^h	enveloppe partielle de la tâche i
C	quantité de ressources, capacité
m	nombre de machines
est_Ω	temps de sortie de l'ensemble Ω
lct_Ω	échéance de l'ensemble Ω
e_Ω	énergie de l'ensemble Ω
Env_Ω	enveloppe de l'ensemble Ω
Sl_Ω	slack de l'intervalle $[est_\Omega, lct_\Omega)$

TABLE A.1: Notations de base utilisées dans le document.

Bibliographie

[1] Aggoun, A. et N. Beldiceanu, « Extending chip in order to solve complex scheduling and placement problems », *Mathematical and Computer Modelling*, 17(7), 1993.

[2] Artiouchine, K., P. Baptiste et C. Dürr, « Runway sequencing with holding patterns », *Proceedings of 2nd international workshop on discrete optimization methods in production and logistics(abvr)*, p. 96–101, 2004.

[3] Artiouchine, K. et P. Baptiste, « Inter-distance constraint : An extension of the all-different constraint for scheduling equal length jobs », *Proceedings of the 11th international conference on principles and practice of constraint programming (cp2005)*, p. 62–76, 2005.

[4] Baptiste, P., C. L. Pape et W. Nuijten, *Satisfaiability tests and time-bound adjustments for cumulative scheduling problems*, rapport technique, Université de Technologie de Compiègne, 1998.

[5] Baptiste, P., C. L. Pape et W. Nuijten, « Satisfiability tests and time-bound adjustments for cumulative scheduling problems », *Annals of Operation Research*, 92(0), p. 305–333, 1999.

[6] Baptiste, P., C. L. Pape et W. Nuitjen, *Constraint-based programming : Appliying constraint programming to scheduling problems*, Kluwer Academic Publishers, 2001.

[7] Baptiste, P. et C. L. Pape, « Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems », *Constraints*, 5, p. 119–139, 2000.

[8] Beldiceanu, N., M. Carlsson et E. Poder, « New filtering for the cumulative constraint in the context of non-overlapping rectangles », *Proceedings of the 5th international conference on integration of ai and or techniques in constraint programming for combinatorial optimisation problems (cpaior 2008)*, p. 21–35, 2008.

[9] Beldiceanu, N. et M. Carlsson, « A new multi-resource cumulatives constraint with negative heights », *Proceedings of the 8th international conference on principles and practice of constraint programming (cp 2002)*, p. 63–79, 2002.

[10] Bellman, R., « On a routing problem », *Quarterly Applied Mathematics*, 16, p. 87–90, 1958.

[11] Brass, P., *Advanced data structures*, Cambridge University Press, 1st edition, 2008.

[12] Brucker, P. et S. A. Kravchenko, *Scheduling jobs with equal processing times and time windows on identical parallel machines*, rapport technique 257, Fachbereich Mathematik/Informatik, universität Osnabrük, 2005.

[13] Carlier, J. et E. Pinson, « Jackson's pseudo-preemptive schedule and cumulative scheduling problems », *Discrete Applied Mathematics*, 145, p. 80–94, 2004.

[14] Cormen, T. H., C. Stein, R. L. Rivest et C. E. Leiserson, *Introduction to algorithms*, MIT Press, 3rd edition, 2009.

[15] Dürr, C. et M. Hurand, « Finding total unimodularity in optimization problems solved by linear programs », *Algorithmica*, 59(2), p. 256–268, 2011.

[16] Gabow, H. et R. Tarjan, « A linear-time algorithm for a special case of disjoint set union », *Proceedings of the 15th annual acm symposium on theory of computing*, p. 246–251, 1983.

[17] Garey, M.-R., D.-S. Johnson, B.-B. Simons et R.-E. Tarjan, « Scheduling unit-time tasks with arbitrary release times and deadlines », *SIAM Journal of Computing*, 10(2), 1981.

[18] Glover, F., « Maximum matching in a convex bipartite graph », *Naval Research Logistic Quarterly*, 14(3), p. 313–316, 1967.

[19] Hall, P., « On representatives of subsets », *Journal of the London Mathematical Society*, 10(1), p. 26–30, 1935.

[20] Jackson, J., *Scheduling a production line to minimize maximum tardiness*, rapport technique 43, University of California in Los Angeles, 1955.

[21] Kameugne, R., L. Fotso, J. Scott et Y. Ngo-Kateu, « A quadratic edge-finding filtering algorithm for cumulative resource constraints », *Proceedings of the 17th international conference on principles and practice of constraint programming (cp 2011)*, p. 478–492, 2011.

[22] Kolisch, R. et A. Sprecher, « Psplib - a project scheduling library », *European Journal of Operational Research*, 96, p. 205–216, 1996.

[23] Leconte, M., « A bounds-based reduction scheme for constraints of difference », *Proceedings of the international workshop on constraint-based reasoning*, p. 19–28, 1996.

[24] Letort, A., N. Beldiceanu et M. Carlsson, « A scalable sweep algorithm for the cumulative constraint », *Proceedings of the 18th international conference on principles and practice of constraint programming (cp 2012)*, p. 439–454, 2012.

[25] Lipski, W. et F. Preparata, « Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. », *Acta Informatica*, 15, p. 329–346, 1981.

[26] López-Ortiz, A., C.-G. Quimper, J. Tromp et P. V. Beek, « A fast and simple algorithm for bounds consistency of the alldifferent constraint », *Proceedings of the 18th international joint conference on artificial intelligence (ijcai 03)*, p. 245–250, 2003.

[27] López-Ortiz, A. et C.-G. Quimper, « A fast algorithm for multi-machine scheduling problems with jobs of equal processing times », *Proceedings of the 28th international symposium on theoretical aspects of computer science (stacs 2011)*, p. 380–391, 2011.

[28] Mercier, L. et P. V. Hentenryck, « Edge finding for cumulative scheduling », *INFORMS Journal on Computing*, 20(1), p. 143–153, 2008.

[29] Nuijten, W., *Time and resource constrained scheduling*, thèse de doctorat, Eindhoven University of Technology, 1994.

[30] Ouellet, P. et C.-G. Quimper, « The multi-inter-distance constraint », *Proceedings of the 22nd international joint conference on artificial intelligence (ijcai 2011)*, p. 629–634, 2011.

[31] Ouellet, P. et C.-G. Quimper, « Time-table-extended-edge-finding for the cumulative constraint », *Proceedings of the 19th international conference on principles and practice of constraint programming (cp2013)*, p. 562–577, 2013.

[32] Papadimitriou, C. H. et K. Steiglitz, *Combinatorial optimization*, Dover, 2nd edition, 1998.

[33] Quimper, C.-G., A. López-Ortiz et G. Pesant, « A quadratic propagator for the inter-distance constraint », *Proceedings of the 21st national conference on artificial intelligence (aaai 2006)*, p. 123–128, 2006.

[34] Quimper, C.-G., *Efficient propagators for global constraints*, thèse de doctorat, University of Waterloo, Canada, 2006.

[35] Régin, J.-C., « A filtering algorithm for constraints of difference in CSPs », *Proceedings of the 11th national conference on artificiel intelligence (aaai-1994)*, p. 362–367, 1994.

[36] Régin, J.-C., « Generalized arc consistency for global cardinality constraint », *Proceedings of the 8th annual conference on innovative applications of artificial intelligence (iaai-1996)*, p. 209–215, 1996.

[37] Schutt, A., A. Wolf et G. Schrader, « Not-first and not-last detection for cumulative scheduling in $o(n^3 logn)$ », *Proceedings of the 16th international conference on applications of declarative programming and knowledge management (inap 2005)*, p. 66–80, 2006.

[38] Schutt, A., T. Feydy et P. Stuckey, « Explaining time-table-edge-finding propagation for the cumulative resource constraint », *Proceedings of the 11th international workshop on no-good learning (cp 2012)*, 2012.

[39] Schutt, A. et A. Wolf, « A new $o(n^2 \log n)$ not-first/not-last pruning algorithm for cumulative resource constraints », *Proceedings of the 16th international conference on principles and practice of constraint programming (cp 2010)*, p. 445–459, 2010.

[40] Simons, B. et M. Warmuth, « A fast algorithm for multiprocessor scheduling of unit-length jobs », *SIAM Journal of Computing*, 18(4), p. 690–710, 1989.

[41] Simons, B., « A fast algorithm for single processor scheduling », *Proceedings of the 19th annual symposium on foundations of computer science*, p. 246–252, 1978.

[42] Simons, B., « Multiprocessor scheduling of unit length jobs with arbitrary release times and deadlines », *SIAM Journal of Computing*, 12(2), p. 294–299, 1983.

[43] Vakhania, N., « A better algorithm for sequencing with release and delivery times on identical machines », *Internatiional Journal of Computer Mathematics*, 79(6), 2002.

[44] Vilím, P., « O(n log n) filtering algorithms for unary resource constraint », *Proceedings of the 1st international conference on integration of ai and or techniques in constraint programming for combinatorial optimisation problems (cp-ai-or 2004)*, p. 335–347, 2004.

[45] Vilím, P., *Gloabal constraints in scheduling*, thèse de doctorat, Charles University in Prague, 2007.

[46] Vilím, P., « Edge finding filtering algorithm for discrete cumulative resources in o(kn log n) », *Proceedings of the 15th international conference on principles and practice of constraint programming (cp-2009)*, p. 802–816, 2009a.

[47] Vilím, P., « Max energy filtering algorithm for discrete cumulative resources », *Proceedings of the 6th international conference on integration of ai and or techniques in constraint programming for combinatorial optimization problems (cp-ai-or 2009)*, p. 294–308, 2009b.

[48] Vilím, P., « Timetable.. », *Proceedings of the 8th international conference on integration of ai and or techniques in constraint programming for combinatorial optimization problems (cp-ai-or 2011)*, p. 230–345, 2011.

[49] Wolf, A. et G. Schrader, « O(n log n) overload checking for the cumulative constraint and its application », *Proceedings of the 16th international conference on applications of declarative programming and knowledge management (inap 2005)*, p. 88–101, 2006.

[50] Yen, J., « An algorithm for finding shortest routes from all source nodes to a given destination in general network », *Quarterly Applied Mathematics*, 27, p. 526–530, 1970.