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

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

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Arnaud LETORT

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

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

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

Soutenue le 28 octobre 2013
Thèse n° : 2013EMNA0113

Passage à l'échelle pour les contraintes d'ordonnancement multi-ressources

JURY

Rapporteurs : **M. Philippe BAPTISTE**, Directeur de la stratégie, Ministère de l'Enseignement supérieur et de la Recherche
M. Patrice BOIZUMAULT, Professeur, Université de Caen Basse-Normandie

Examinateurs : **M. Claude JARD**, Professeur, Université de Nantes
M. Laurent PERRON, Ingénieur, Google
M. Éric PINSON, Professeur, Université Catholique de l'Ouest

Directeur de thèse : **M. Nicolas BELDICEANU**, Professeur, École des Mines de Nantes

Remerciements

Avant d'en venir au sujet je me dois de remercier ceux qui ont contribué, de quelque sorte que ce soit, à ce que ces trois années de travail aboutissent.

Je tiens tout d'abord à remercier sincèrement Nicolas Beldiceanu, mon directeur de thèse, pour ses commentaires toujours pertinents et sa connaissance étendue du domaine.

Un grand merci à Mats Carlsson, que je n'aurais finalement pas rencontré physiquement durant ces trois ans, pour ses précieux conseils et l'aide qu'il m'a apportés.

Merci à Philippe Baptiste et Patrice Boizumault, rapporteurs de cette thèse, pour leurs commentaires et remarques sur ce manuscrit. Merci également à Claude Jard, Laurent Perron et Éric Pinson d'avoir accepté de faire partie du jury.

Merci à l'ensemble des membres de l'équipe TASC pour tous ces moments d'échange sur des sujets plus ou moins sérieux, leur bonne humeur permanente et leur disponibilité. Un petit mot en particulier pour Philippe et Xavier pour le café du matin permettant de bien démarrer la journée.

Merci à Alexis, avec qui j'ai partagé le même bureau durant mes deux premières années, toujours de bonne humeur et qui s'est montré être un adversaire redoutable au jeu du lancé d'objets dans la poubelle ! Merci à Alban, qui a remplacé Alexis durant cette dernière année, pour tous ces bons moments à discuter balayage et poker.

Merci à l'ensemble de mes proches pour leur soutien et leurs encouragements tout au long de cette expérience.

Enfin, merci à Virginie pour son soutien permanent et pour s'être arraché les cheveux à la relecture du manuscrit !

Table des matières

1 Introduction **7**

I État de l'art **11**

2 La programmation par contraintes **13**
- 2.1 Modélisation d'un problème de contraintes 13
- 2.2 Résolution d'un problème de contraintes 14

3 Ordonnancement cumulatif en programmation par contraintes **15**
- 3.1 La contrainte *cumulative* 16
- 3.2 Différents filtrages pour la contrainte *cumulative* 16
- 3.3 Algorithme de balayage de 2001 pour la contrainte *cumulative* 18
  - 3.3.1 Types d'évènements 19
  - 3.3.2 État de la droite de balayage 20
  - 3.3.3 Algorithme de filtrage 20
  - 3.3.4 Complexité d'une phase de balayage 23
- 3.4 Synthèse critique 24

II Contribution : vers un algorithme de filtrage intégrant une conjonction de contraintes *cumulative*, *cumulative colorée* et *précédence* **25**

4 Motivation et démarche : passage à l'échelle en maîtrisant la convergence au point fixe **27**
- 4.1 Faiblesses de l'algorithme de balayage de 2001 27
- 4.2 Solutions préconisées 28

5 Algorithme de balayage dynamique pour une seule contrainte *cumulative* **31**
- 5.1 Propriété 31
- 5.2 Types d'évènements 32
- 5.3 État de la droite de balayage 33
- 5.4 Algorithme de filtrage 34
  - 5.4.1 Algorithme principal 34
  - 5.4.2 Filtrage 35
  - 5.4.3 Resynchronisation des évènements 37
- 5.5 Correction et propriété vérifiée par sweep_min 41
- 5.6 Complexité 43

6 Balayage synchrone pour plusieurs contraintes *cumulative* avec ou sans précédences **45**
6.1 Une première approche sans contrainte de précédence 49
6.1.1 Types d'évènements 49
6.1.2 État de la droite de balayage 49
6.1.3 Algorithme de filtrage 50
6.1.4 Complexité 54
6.2 Une seconde approche intégrant les contraintes de précédence 55
6.2.1 Types d'évènements 58
6.2.2 État de la droite de balayage 59
6.2.3 Algorithme de filtrage 60
6.2.4 Complexité 67
6.3 Cumulative colorée : ou comment remplacer la somme par le nombre de valeurs distinctes 74
6.3.1 Définition de la contrainte *multiSumColorPrecCumulative* 74
6.3.2 Schémas classiques d'apparition de la contrainte *multiSumColorPrecCumulative* 75
6.3.3 Reformulation quadratique de la contrainte cumulative colorée . . . 77
6.3.4 Intégration au balayage synchrone avec précédences 78

7 Synthèse des nouveaux algorithmes de balayage **81**
7.1 Algorithme de balayage dynamique 81
7.2 Algorithme de balayage synchrone sans précédence 82
7.3 Algorithme de balayage synchrone avec précédences 82

III Améliorations pratiques et évaluation **85**

8 Améliorations pratiques **87**
8.1 Mieux gérer les matrices creuses de consommations 87
8.2 Agrégation des parties obligatoires 88
8.3 Mode glouton 88
8.3.1 Types d'évènements 88
8.3.2 État de la droite de balayage 89
8.3.3 Algorithme glouton 90

9 Évaluation **101**
9.1 Présentation des problèmes sélectionnés 101
9.2 Instances aléatoires 102
9.3 Instances de la PSPLib 103
9.4 Redéploiement d'un réseau d'informations financières à grande échelle . . 107

10 Conclusion **111**

A Annexe **113**

1

Introduction

La programmation par contraintes est un paradigme ayant montré son intérêt dans la résolution de nombreux problèmes combinatoires provenant de domaines d'applications variés ; les problèmes de tournées de véhicules, de création d'emplois du temps, de configuration, d'allocation de ressources en sont des exemples classiques et très étudiés par la communauté contrainte.

Lors de la table ronde sur le futur de la programmation par contraintes qui a eu lieu lors la conférence CP 2011, un des défis identifiés a été de traiter des problèmes de grandes tailles [51]. Les problèmes de bin-packing multi-dimensionnels ont notamment été cités comme des exemples particulièrement pertinents dans le contexte omniprésent du cloud computing. L'importance des problèmes de grande taille et plus particulièrement du bin-packing ont été récemment soulignés dans [58] et ont fait l'objet du Challenge Roadef 2012 [59].

Jusqu'à maintenant, la tendance est d'utiliser des algorithmes dédiés et des métaheuristiques [66] pour traiter les problèmes de grandes tailles. En programmation par contraintes, les algorithmes de filtrage généralement développés pour une contrainte se concentrent plus sur le pouvoir déductif que sur la problématique de passage à l'échelle. Cela explique probablement pourquoi ils se focalisent généralement sur des problèmes n'impliquant que peu de variables. En ordonnancement par exemple, bien qu'il existe une certaine variété dans les raisonnements proposés pour la contrainte *cumulative*, comme Edge-Finding [49, 48, 50, 46, 71, 37], Energetic Reasoning [25, 45, 44, 4], Not First / Not Last [65, 64], Timetabling [41, 8], la grande majorité des algorithmes de filtrage implémentant ces méthodes sont conçus pour des problèmes ne dépassant pas quelques milliers de tâches.

Comme ce qui a déjà été fait pour la contrainte de placement *geost* [13], qui peut traiter jusqu'à deux millions d'objets, notre objectif dans cette thèse est de proposer des algorithmes de filtrage légers pour les problèmes cumulatifs aussi bien mono-ressource que multi-ressources et même colorés, où la somme est remplacée par le nombre de valeurs distinctes. Pour atteindre cet objectif nous nous focalisons dans un premier temps sur la contrainte *cumulative* impliquant une seule ressource et sur l'algorithme de balayage introduit en 2001 dans [8]. En amenant un aspect *dynamique* à cet algorithme, dans le sens où les déductions effectuées durant le balayage sont aussitôt réutilisées, ainsi que la possibilité d'utiliser le même algorithme en mode glouton, nous parvenons à une amélioration significative de la vitesse de convergence au point fixe. Dans un second temps, nous empruntons

l'idée de *balayage synchronisé* introduit dans [14] pour proposer un algorithme prenant en compte une conjonction de contraintes *cumulative*, *cumulative colorée* et *précédence* simultanément.

Dans l'optique d'augmenter encore la taille des problèmes traités nous dérivons de chacun des algorithmes présentés un mode glouton qui entrelace la propagation de contraintes et le fait de fixer les tâches : pour avoir le plus de chances de trouver une solution, le mode glouton s'appuie complètement sur la propagation pour, à chaque étape, restreindre les origines des tâches avant de les fixer. Cette approche, qui a montré son efficacité pour la contrainte *geost*, permet de contourner le goulot d'étranglement classique des solveurs de contraintes concernant l'utilisation importante de la mémoire par le système de trail [1] ou de recopie des structures de données [61] qui permettent le retour arrière dans le cadre d'une recherche arborescente.

Plan de la thèse

Ce document s'articule autour de trois grandes parties :

- État de l'art Dans cette première partie nous présentons tout d'abord un état de l'art orienté vers les problèmes d'ordonnancement dans le cadre de la programmation par contraintes.

- Contribution : vers un algorithme de filtrage intégrant une conjonction de contraintes *cumulative*, *cumulative colorée* et *précédence* Dans cette deuxième partie nous proposons un ensemble d'algorithmes de filtrage permettant d'améliorer le passage à l'échelle pour des problèmes d'ordonnancement cumulatif et coloré.

- Améliorations pratiques et évaluation Enfin, dans cette troisième partie, nous présentons quelques améliorations apportées aux nouveaux algorithmes présentés dans cette thèse puis nous effectuons un ensemble de tests sur des problèmes variés.

Nous présentons maintenant les différents chapitres composant ces parties.

État de l'art

Chapitre 2 : La programmation par contraintes

Dans le chapitre 2 nous présentons brièvement le paradigme de la programmation par contraintes en définissant ce qu'est un réseau de contraintes, puis en présentant les mécanismes mis en jeu lors de sa résolution.

Chapitre 3 : Ordonnancement cumulatif en programmation par contraintes

Dans le chapitre 3 après avoir présenté la contrainte *cumulative* introduite par Aggoun et Beldiceanu dans [2], nous présentons les principaux algorithmes de filtrage pour cette contrainte. Nous nous attardons plus particulièrement sur l'algorithme de balayage (ou sweep) introduit en 2001 dans [8].

Contribution : vers un algorithme de filtrage intégrant une conjonction de contraintes *cumulative*, *cumulative colorée* et *précédence*

Chapitre 4 : Motivation et démarche : passage à l'échelle en maîtrisant la convergence au point fixe

Dans ce chapitre nous proposons une analyse critique de l'algorithme de balayage de 2001 en identifiant ses faiblesses en vue du passage à l'échelle. Pour chacune de ces faiblesses, nous apportons ensuite une réponse qui sera appliquée aux nouveaux algorithmes que nous introduisons dans cette thèse.

Chapitre 5 : Algorithme de balayage dynamique pour une seule contrainte *cumulative*

Nous présentons un premier algorithme de balayage dynamique pour la contrainte *cumulative*, mettant en place les différentes réponses proposées au chapitre précédent.

Chapitre 6 : Balayage synchrone pour plusieurs contraintes *cumulative* avec ou sans précédences

Dans un premier temps, nous proposons un algorithme de balayage synchrone permettant de traiter simultanément une conjonction de contraintes *cumulative*. Dans un deuxième temps, nous étendons cette idée en introduisant un algorithme de balayage synchrone permettant de traiter une conjonction de contraintes *cumulative* et des précédences. Enfin, dans un troisième temps, nous étendons ce dernier algorithme pour y intégrer la contrainte *cumulative colorée* présente naturellement dans un certain nombre de problèmes d'ordonnancement.

Chapitre 7 : Synthèse des nouveaux algorithmes de balayage

Nous apportons dans ce chapitre une vision synthétique des nouveaux algorithmes de filtrage présentés dans cette thèse.

Améliorations pratiques et évaluation

Chapitre 8 : Améliorations pratiques

Dans ce chapitre, nous mettons en avant plusieurs améliorations pouvant être apportées aux algorithmes présentés dans cette thèse. Nous nous étendons plus particulièrement sur le mode glouton dérivant directement du dernier algorithme de propagation du chapitre 6.

Chapitre 9 : Évaluation

Nous effectuons dans ce chapitre une évaluation des différents algorithmes présentés dans cette thèse sur trois problèmes différents ; des instances aléatoires, des instances issues de la PSPLib [56] et enfin des instances issues d'un problème de redéploiement de réseau d'informations financières.

État de l'art

2

La programmation par contraintes

Sommaire

2.1 Modélisation d'un problème de contraintes 13
2.2 Résolution d'un problème de contraintes 14

La programmation par contraintes (PPC) est un paradigme dont l'objectif est de résoudre des problèmes combinatoires. Dans un premier temps, l'utilisateur décrit son problème dans un langage déclaratif, puis dans un second temps, la machine le résout[1]. Nous présentons brièvement dans ce chapitre les principes de la programmation par contraintes en commençant par la partie modélisation puis en enchainant par la partie résolution.

2.1 Modélisation d'un problème de contraintes

Un problème de satisfaction de contraintes est décrit pas un réseau de contraintes. Dans le cadre de cette thèse, nous nous limitons aux problèmes discrets, c'est-à-dire que les domaines des variables ne contiennent que des valeurs entières.

Définition 2.1. (Réseau de contraintes). *Un réseau de contraintes est défini par un triplet* $R = \langle X, D, C \rangle$ *où :*

- X *est un ensemble fini de variables* $\{x_0, x_1, \ldots, x_{n-1}\}$*,*
- D *est un ensemble fini de domaines* $\{D_0, D_1, \ldots, D_{n-1}\}$*. Chaque domaine* D_i *(*$0 \leq i < n$*) définit l'ensemble des valeurs que peut prendre la variable* x_i*. Par la suite, nous noterons* $\underline{x_i}$ *(respectivement* $\overline{x_i}$*) la valeur minimale (respectivement maximale) que peut prendre la variable* x_i*.*
- C *est un ensemble fini de contraintes* $\{c_0, c_1, \ldots, c_{m-1}\}$*. Une contrainte* C_j *(*$0 \leq j < m$*) est une relation impliquant un sous-ensemble des variables de* X*.*

[1] « Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming : the user states the problem, the computer solves it. » Eugene C. Freuder [26]

L'utilisateur final doit simplement déclarer l'ensemble des variables définissant son problème ainsi que l'ensemble des relations les unissant.

Une solution à un problème de satisfaction de contraintes est une instanciation complète et cohérente de toutes les variables, c'est-à-dire, une affectation de toutes les variables à l'une des valeurs de leur domaine respectif satisfaisant l'ensemble des contraintes.

2.2 Résolution d'un problème de contraintes

La résolution d'un problème de satisfaction de contraintes s'appuie traditionnellement sur la combinaison de la *recherche* et de la *propagation*. Les deux paragraphes suivants décrivent brièvement ces deux points. Notons cependant que d'autres techniques de résolution telles que la recherche locale [47, 53], la programmation linéaire [36] ou SAT [22] peuvent aussi être utilisées ou combinées en prenant en entrée une même conjonction de contraintes.

Recherche La recherche consiste en un parcours de l'espace de recherche par un jeu d'affectations et de retours arrières. Souvent représentée par un arbre, la recherche énumère toutes les instanciations possibles jusqu'à trouver une solution, c'est-à-dire, une instanciation complète cohérente, ou conclure qu'il n'existe pas de solution. Utiliser un tel algorithme de recherche pour résoudre un problème de satisfaction est en pratique inexploitable. En effet, ce dernier a une complexité temporelle exponentielle par rapport au nombre de variables.

Propagation La propagation vient assister la recherche en essayant de déduire de nouvelles informations à partir de l'état courant des domaines. A partir d'une instanciation partielle, le mécanisme de propagation tente de supprimer des valeurs impossibles des domaines des variables, réduisant ainsi la taille de l'espace de recherche. L'algorithme chargé de déterminer quelles valeurs supprimer par rapport à une contrainte donnée est appelé *algorithme de filtrage*. Malheureusement, supprimer l'ensemble des valeurs inconsistantes est bien souvent un problème NP-difficile. En programmation par contraintes, il est nécessaire de trouver l'équilibre entre la puissance du filtrage effectué, c'est-à-dire le nombre de valeurs retirées, et le temps de calcul nécessaire.

Contrainte globale Un bon moyen de modéliser concisément un problème et de renforcer la propagation est d'utiliser les contraintes globales [9]. Une contrainte globale permet de modéliser un sous problème entre un nombre non fixé de variables, pour laquelle on peut définir, un algorithme de filtrage efficace par rapport à sa décomposition en contraintes primitives dans le cadre de la programmation par contraintes, un coût de violation dans le cas de la recherche locale, une reformulation dans le cas de la programmation linéaire ou SAT, une estimation du nombre de solutions dans le cas d'heuristiques dédiées [54]. Dans [17], Bessière et Van Hentenryck donnent une définition du terme contrainte globale basée sur trois axes, sémantique, opérationnel et algorithmique. La contrainte globale de différences $alldifferent(x_0, x_1, \dots, x_{n-1})$ [57] qui exprime que la valeur prise par chacune des variables x_i $(0 \leq i < n-1)$ est différente de celle des autres variables est certainement la contrainte globale la plus connue. Au delà du caractère « pratique » qu'elle apporte pour la modélisation, comparativement à la pose d'un nombre quadratique de contraintes de différence binaire, elle permet de définir un algorithme de filtrage raisonnant directement sur l'ensemble des variables.

3

Ordonnancement cumulatif en programmation par contraintes

Sommaire

3.1 La contrainte *cumulative* **16**

3.2 Différents filtrages pour la contrainte *cumulative* **16**

3.3 Algorithme de balayage de 2001 pour la contrainte *cumulative* **18**

3.3.1 Types d'évènements 19

3.3.2 État de la droite de balayage 20

3.3.3 Algorithme de filtrage 20

3.3.4 Complexité d'une phase de balayage 23

3.4 Synthèse critique **24**

En 1974, Baker définit dans [3] un problème d'ordonnancement comme l'allocation de ressources à des activités à travers le temps. La résolution d'un problème d'ordonnancement consiste à déterminer les dates de début et de fin d'un ensemble de tâches soumises à différents types de contraintes (e.g. contraintes de ressource, d'enchainement, de délai). Il existe une grande variété de problèmes d'ordonnancement comme les problèmes disjonctifs, cumulatifs, avec ou sans préemption... On trouve une classification bien connue en ordonnancement qui est celle de Graham [32]. Dans cette thèse, nous nous intéresserons aux problèmes d'ordonnancement cumulatifs sans préemption. Dans cette configuration, une tâche est définie par une date de début, une durée, une date de fin, et une hauteur représentant sa consommation à chaque point de temps de son exécution. La résolution d'un problème d'ordonnancement cumulatif consiste à déterminer les dates de début et de fin des tâches sans que la hauteur cumulée des tâches s'exécutant à chaque point de temps ne dépasse la quantité de ressource disponible.

Nous présentons dans un premier temps la contrainte *cumulative* servant à modéliser les problèmes d'ordonnancement cumulatifs. Dans un second temps, nous présenterons les principaux algorithmes de filtrage pour cette contrainte en nous attardant plus particulièrement sur l'algorithme de balayage qui, comme nous le verrons, semble le plus adapté à notre problématique de passage à l'échelle.

3.1 La contrainte *cumulative*

En programmation par contraintes, la contrainte *cumulative* a été introduite par Aggoun et Beldiceanu dans [2] pour modéliser des problèmes d'ordonnancement cumulatifs.

Définition 3.1. (Contrainte *cumulative*). *Étant donné n tâches et une ressource d'une capacité maximale $limit$, où chaque tâche t ($0 \leq t < n$) est définie par une variable de début s_t, une durée fixée d_t, une variable de fin e_t et consommation de ressource fixée h_t, la contrainte* cumulative *avec les deux arguments :*

- $\langle\langle s_0, d_0, e_0, h_0\rangle, \ldots, \langle s_{n-1}, d_{n-1}, e_{n-1}, h_{n-1}\rangle\rangle$
- $limit$

est satisfaite si et seulement si les conditions (3.1) et (3.2) sont vérifiées :

$$\forall t \in [0, n-1] \ : s_t + d_t = e_t \tag{3.1}$$

$$\forall i \in \mathbb{Z} \ : \sum_{\substack{t \in [0,n-1], \\ i \in [s_t, e_t)}} h_t \leq limit \tag{3.2}$$

Exprimée de manière plus informelle, la contrainte *cumulative* est satisfaite si la somme des hauteurs des tâches s'exécutant à chaque point de temps est inférieure ou égale à la capacité maximale de la ressource.

Plusieurs types de raisonnements ont été développés pour écrire des algorithmes de filtrage pour cette contrainte, comme la méthode de Timetabling présentée par Le Pape dans [41], l'Edge-Finding introduit et développé dans [49, 48, 50, 46, 71, 37], l'Energetic Reasoning [25, 45, 44, 4], Not First / Not Last [65, 64], Task Intervals [20, 21], Overload Checking [73], ainsi que des méthodes basées sur les problèmes de sac à dos [12, 10].

Les raisonnements de Timetabling et d'Edge-Finding effectuent des filtrages différents qui semblent être incomparables. En effet, Vilim propose en 2011 une méthode très efficace issue de la combinaison de ces deux méthodes dans [72]. Cet algorithme a été repris et expliqué[1] en 2013 par Schutt, Feydy et Stuckey dans [63] et est actuellement la méthode la plus performante sur les instances de la PSPLib [56].

3.2 Différents filtrages pour la contrainte *cumulative*

Nous présentons maintenant, de manière brève, les principaux algorithmes de filtrage pour la contrainte *cumulative* basés sur la notion d'énergie. Nous précisons également pour chacun d'eux les principales raisons nous laissant penser qu'ils sont ou ne sont pas les meilleures approches pour traiter des instances de très grande taille.

Edge-Finding L'Edge-Finding est une technique permettant de déduire des relations de précédence entre les tâches au regard de leur fenêtre temporelle, leur énergie et de l'énergie disponible sur certains intervalles. Grâce à ces précédences, il est alors possible de filtrer les dates de début et de fin des tâches. D'abord développés pour les problèmes d'ordonnancement disjonctifs [18], les algorithmes d'Edge-Finding ont ensuite été adaptés au cas

[1] Le mot « expliqué » fait ici référence au mécanisme d'apprentissage de nogood présent dans certains solveurs de contraintes.

cumulatif [48, 4]. Pour la contrainte *cumulative*, l'algorithme d'Edge-Finding le plus performant est celui de Kameugne et al. [37] dont la complexité temporelle dans le pire des cas est $O(n^2)$, où n désigne le nombre de tâches.

Dans l'optique du passage à l'échelle, les algorithmes d'Edge-Finding ne nous semblent pas être les meilleurs candidats. En effet la complexité temporelle dans le pire des cas de l'algorithme de Kameugne et al., i.e. $O(n^2)$ est toujours atteinte.

Timetable Edge-Finding En 2011, Vilim propose un nouvel algorithme [72] utilisant à la fois les raisonnements d'Edge-Finding et de Timetabling. Tout en conservant la même complexité que l'algorithme d'Edge-Finding de Kameugne et al. (i.e. $O(n^2)$), ce nouvel algorithme apporte un filtrage plus puissant au niveau déductif.

Pour les mêmes raisons que celles évoquées pour l'Edge-Finding de Kameugne et al., cet algorithme ne nous semble pas être le meilleur candidat pour traiter des problèmes cumulatifs de très grande taille.

Energetic Reasoning L'Energetic Reasoning est un raisonnement introduit et développé dans [25, 45, 44, 4] dominant l'Edge-Finding. L'idée est de déterminer dans un premier temps la demande énergétique minimale des tâches sur un intervalle de temps donné, puis dans un second temps, de filtrer les tâches en observant la différence entre la quantité d'énergie offerte par l'intervalle et cette demande. Dans [4], Baptiste et al. précisent que le nombre d'intervalles d'intérêt est de $O(n^2)$ et que l'on peut ajuster les n tâches sur chacun de ces intervalles, et donc que la complexité de leur algorithme est $O(n^3)$.

Une complexité temporelle de $O(n^3)$ semble rédhibitoire en vue du passage à l'échelle.

Timetabling Les premiers algorithmes de filtrage pour la contrainte *cumulative* sont basés sur la notion de partie obligatoire, initialement introduite par Lahrichi en 1982 [40]. Ces algorithmes construisent un profil des parties obligatoires (PPO) des tâches et l'utilisent pour filtrer les dates de début des tâches afin de ne pas dépasser la capacité de la ressource.

Pour définir la notion de PPO nous rappelons tout d'abord la définition d'une instance réalisable d'une tâche et la définition de partie obligatoire d'une tâche, initialement introduite par Lahrichi [40].

Définition 3.2. (Instance réalisable). *L'instance réalisable d'une tâche t est une tâche où les variables de début s_t et fin e_t sont fixées à une valeur de leur domaine respectif et respectent la condition 3.1 page 16.*

Définition 3.3. (Partie obligatoire). *La partie obligatoire d'une tâche t est l'intersection de toutes ses positions réalisables. La hauteur de la partie obligatoire d'une tâche t à un point de temps donné i est définie par h_t si $i \in [\overline{s_t}, \underline{e_t}[$, et 0 sinon.*

Définition 3.4. (PPO). *Étant donné un ensemble de tâches $\mathcal{T}$, le PPO de l'ensemble $\mathcal{T}$ est l'agrégation de toutes les parties obligatoires des tâches de $\mathcal{T}$. La hauteur du PPO à un point de temps donné i est défini par $\sum_{\substack{t \in \mathcal{T}, \\ i \in [\overline{s_t}, \underline{e_t})}} h_t$.*

Bien qu'il soit nécessaire d'avoir un profil de parties obligatoire non nul pour effectuer les premiers ajustements, la méthode de Timetabling est largement utilisée et semble en pratique disposée à résoudre des instances relativement grandes. Bien que d'autres algorithmes de filtrage comme ceux utilisant la méthode d'Edge-Finding sont capables de filtrer les domaines des variables plus tôt dans l'arbre de recherche, ils ne parviennent pas à capturer toutes les déductions effectuées à partir du profil de parties obligatoires. Pour ces raisons, nous pensons que la méthode de Timtabling est adaptée au passage à l'échelle.

3.3 Algorithme de balayage de 2001 pour la contrainte *cumulative*

L'algorithme de balayage pour la contrainte *cumulative* proposé dans [8] en 2001 repose sur un principe largement utilisé en géométrie algorithmique appelé balayage, ou sweep [24]. En programmation par contraintes, ce principe a déjà été utilisé pour implémenter un algorithme de filtrage pour la contrainte de *non-overlapping* ainsi que d'autres contraintes géométriques [7].

Dans un espace à deux dimensions, l'algorithme de balayage [24] résout le problème en déplaçant une droite verticale, appelée droite de balayage, de gauche à droite et en utilisant deux éléments :

- L'*état de la droite de balayage*, qui contient les informations relatives à la position courante δ de la droite de balayage.
- La *série d'évènements*, qui contient l'ensemble des évènements à traiter, ordonnée par ordre croissant suivant l'axe du temps.

L'algorithme commence par initialiser l'état de la droite de balayage à la position initiale. Ensuite, la droite « saute » d'évènement en évènement, chacun d'entre eux étant traité puis inséré ou retiré de l'état de la droite de balayage de manière incrémentale.

Dans le contexte de la contrainte *cumulative*, la droite de balayage parcourt l'axe du temps pour construire le *profil des parties obligatoires* (PPO), effectuer des vérifications de non dépassement du plafond de ressource et filtrer les domaines des variables selon ce profil et la capacité de la ressource. L'algorithme de balayage de 2001 est une implémentation de la méthode de Timetabling présentée dans [41].

Nous introduisons maintenant un exemple qui sera utilisé et étendu par la suite pour illustrer les différents algorithmes de balayage présentés dans cette thèse.

Exemple 3.1. *Considérons cinq tâches t_0,t_1,...,t_4 ayant les débuts, durées, fins et hauteurs suivantes :*

- t_0 : $s_0 \in [1,1]$, $d_0 = 1$, $e_0 \in [2,2]$, $h_0 = 2$,
- t_1 : $s_1 \in [0,3]$, $d_1 = 2$, $e_1 \in [2,5]$, $h_1 = 2$,
- t_2 : $s_2 \in [0,5]$, $d_2 = 2$, $e_2 \in [2,7]$, $h_2 = 1$,
- t_3 : $s_3 \in [0,9]$, $d_3 = 1$, $e_3 \in [1,10]$, $h_3 = 1$,
- t_4 : $s_4 \in [0,7]$, $d_4 = 3$, $e_4 \in [3,10]$, $h_4 = 2$,

impliquées dans la contrainte cumulative *:*

$$cumulative(\langle\langle s_0,d_0,e_0,h_0\rangle,\langle s_1,d_1,e_1,h_1\rangle,\langle s_2,d_2,e_2,h_2\rangle,\langle s_3,d_3,e_3,h_3\rangle,\langle s_4,d_4,e_4,h_4\rangle\rangle,3)$$

Cette instance est illustrée par la partie (A) de la figure 3.1. Comme la tâche t_0 débute au point de temps 1 et que la tâche t_1 ne peut pas se superposer à t_0 sans dépasser la capacité de la ressource, le début au plus tôt de la tâche t_1 est ajusté à 2. De la même manière, la tâche t_4 ne peut pas se superposer à t_0, entrainant l'ajustement de son début au plus tôt de la tâche t_4 à 2 (partie (B)). Puisque la tâche t_1 a désormais une partie obligatoire sur l'intervalle

$[3, 4[$ et puisque la tâche t_4 ne peut pas se superposer à t_1 sans dépasser la capacité de la ressource, le début au plus tôt de t_4 est ajusté à 4 (partie (C)).

L'objectif de l'algorithme de balayage est d'effectuer ce filtrage de manière efficace.

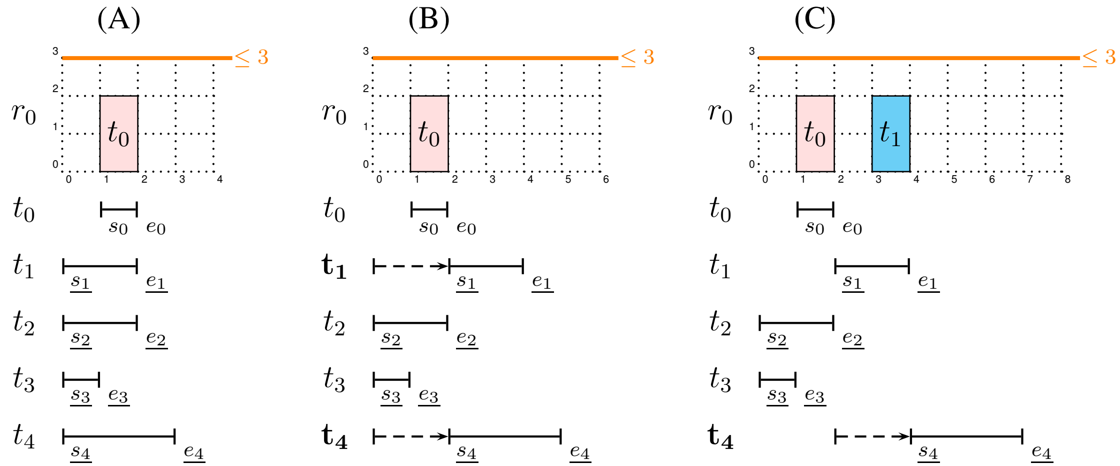


FIGURE 3.1 – Les parties (A), (B) et (C) représentent respectivement la position au plus tôt de chacune des tâches ainsi que le PPO, du problème initial décrit dans l'exemple 3.1, après un premier balayage et après un second balayage.

3.3.1 Types d'évènements

Pour construire le PPO et supprimer des valeurs des domaines des variables de début et de fin des tâches, l'algorithme de balayage considère les types d'évènements suivants :

- Les *évènements de profil*, qui permettent de construire le PPO. Ils correspondent aux dates de début au plus tard et de fin au plus tôt des tâches pour lesquelles la partie obligatoire n'est pas vide. C'est-à-dire pour l'ensemble des tâches pour lesquelles la date de début au plus tard est strictement inférieure à la date de fin au plus tôt.

- Les *évènements d'activation*, qui permettent d'enregistrer les tâches candidates au filtrage. Ces évènements correspondent au début au plus tôt de chacune des tâches non fixées. En effet, ce n'est qu'à partir du moment où la droite de balayage passe sur le début au plus tôt d'une tâche que l'on peut éventuellement commencer à filtrer la date de début de la tâche.

La table 3.1 décrit les différents types d'évènements, où chaque évènement correspond à un quadruplet $\langle type, tâche, date, hauteur \rangle$. Le dernier attribut, $hauteur$, n'est pertinent que pour les évènements de profil et donne la variation de ressource disponible. Ces évènements sont triés par ordre croissant suivant leur attribut $date$.

Exemple 3.2. (Évènements générés). *A partir des domaines initiaux des tâches de l'exemple 3.1, les évènements suivants sont générés et ordonnés selon leur date :* $\langle PR, 1, 0, 0 \rangle$ $\langle PR, 2, 0, 0 \rangle$ $\langle PR, 3, 0, 0 \rangle$ $\langle PR, 4, 0, 0 \rangle$ $\langle SCP, 0, 1, -2 \rangle$ $\langle ECP, 0, 2, 2 \rangle$.

Évènements généré	Conditions
$\langle SCP, t, \overline{s_t}, -h_t \rangle$	$\overline{s_t} < \underline{e_t}$
$\langle ECP, t, \underline{e_t}, +h_t \rangle$	$\overline{s_t} < \underline{e_t}$
$\langle PR, t, \underline{s_t}, 0 \rangle$	$\underline{s_t} \neq \overline{s_t}$

TABLE 3.1 – Les types d'évènements de l'algorithme de balayage avec la condition nécessaire à leur génération. Le dernier attribut, *hauteur*, n'est pertinent que pour les évènements du type SCP et ECP.

3.3.2 État de la droite de balayage

La droite de balayage maintient les trois informations suivantes :

- La position courante δ de la droite de balayage, initialement placée à la date du premier évènement.
- La quantité de ressource disponible au point de temps δ, notée gap, i.e. la différence entre la capacité maximale de la ressource $limit$ et la hauteur du PPO au point de temps δ.
- Une liste de tâches $\mathcal{T}_{prune}$, contenant toutes les tâches qui peuvent potentiellement couper le point de temps δ, i.e. les tâches pour lesquelles la variable de début peut être filtrée au regard du point de temps δ et de la hauteur de ressource disponible gap.

3.3.3 Algorithme de filtrage

L'algorithme de balayage de 2001 commence par créer et trier les évènements en fonction de leur date. Puis, la droite de balayage avance d'un évènement au suivant, en mettant à jour gap et $\mathcal{T}_{prune}$. Une fois que tous les évènements associés à la position δ ont été traités, l'algorithme tente de filtrer les tâches présentes dans $\mathcal{T}_{prune}$ en observant la quantité de ressource disponible gap et l'intervalle $[\delta, \delta_{next}[$, où δ_{next} est la prochaine position de la droite de balayage, i.e. la date du prochain évènement. Plus précisément, étant donné une tâche $t \in \mathcal{T}_{prune}$ n'ayant pas de partie obligatoire dans l'intervalle $[\delta, \delta_{next}[$ et telle que $h_t > gap$, l'intervalle $[\delta - d_t + 1, \delta_{next}[$ est supprimé du domaine de sa variable de début s_t.

Dans le pseudo code décrivant l'algorithme sweep_2001 nous utiliserons la fonction $\mathrm{sort}(l)$ qui trie la liste d'évènements l par ordre croissant suivant leur date. La fonction $\mathrm{remove_interval_var}(v, x, y)$ supprime l'intervalle de valeurs $[x, y]$ du domaine de la variable v et retourne **false** si le domaine est vide, **true** sinon.

Algorithme principal

L'algorithme sweep_2001 page 22 est composé des parties suivantes :

- `[INITIALISATION]` (lignes 2 à 7). Les évènements sont générés selon les conditions données dans la table 3.1 et triés par ordre croissant suivant leur date dans la liste l_events. Le premier évènement est extrait ce qui permet d'initialiser la position de la droite de balayage. Enfin, la hauteur de ressource disponible gap est initialisée à la capacité maximale de la ressource, $limit$.
- `[BOUCLE PRINCIPALE]` (lignes 9 à 22). À chaque tour de la boucle principale un évènement est lu et traité, et l'algorithme de filtrage filter_2001 est appelé si la droite de balayage doit se déplacer. La boucle principale se décompose en quatre étapes :

 – `[DÉPLACEMENT DE LA DROITE DE BALAYAGE]` (lignes 12 à 15). Lorsque la date du prochain évènement est différente du précédent, ce qui signifie que tous les évènements associés à la date courante ont été lus, la droite de balayage doit se déplacer vers ce prochain évènement. A ce stade, on sait que la hauteur de ressource restante sur l'intervalle $[\delta, \delta_{next}[$ est fixe et égale à gap. On vérifie dans un premier temps qu'il n'y a pas de dépassement de ressource (ligne 13), puis l'algorithme 2 est appelé pour tenter de filtrer le domaine des variables des tâches actives, i.e. les tâches contenues dans $\mathcal{T}_{prune}$.
 – `[TRAITER UN ÉVÈNEMENT DE PROFIL (SCP ou SCP)]` (ligne 17). Lorsqu'un évènement de début ou de fin de partie obligatoire est lu, la hauteur de ressource disponible gap est mise à jour grâce au quatrième attribut de l'évènement, noté inc.
 – `[TRAITER UN ÉVÈNEMENT DE DÉBUT AU PLUS TÔT (PR)]` (ligne 20). Lorsqu'un évènement correspondant au début au plus tôt d'une tâche est lu, la tâche est ajoutée à la liste des tâches actives $\mathcal{T}_{prune}$.
 – `[RÉCUPÉRER LE PROCHAIN ÉVÈNEMENT]` (ligne 22). On extrait le prochain évènement, ce qui nous permet de déterminer la prochaine position de la droite de balayage, δ_{next}.

Filtrage

Lorsque tous les évènements relatifs à la position courante δ de la droite de balayage ont été lus, l'algorithme 2 tente de filtrer le domaine des variables des tâches contenues dans $\mathcal{T}_{prune}$ par rapport à l'intervalle de balayage courant $[\delta, \delta_{next}[$ et à la quantité de ressource disponible gap.

- `[PARCOURS DES TÂCHES ACTIVES]` (lignes 2 à 10). A chaque appel de l'algorithme 2, l'ensemble de tâches pouvant intersecter l'intervalle de balayage courant est parcouru, i.e. les tâches présentes dans $\mathcal{T}_{prune}$.

 – `[FILTRAGE DE LA TÂCHE]` (lignes 4 à 7). Si la hauteur de la tâche t est supérieure à l'espace disponible gap et qu'elle n'a pas de partie obligatoire comprise dans l'intervalle $[\delta, \delta_{next}[$ alors ses variables de début et fin sont filtrées. Pour la variable de début de la tâche t, notée s_t, on retire les valeurs $[\delta - d_t + 1, \delta_{next} + 1]$, empêchant la tâche t de couper l'intervalle de balayage. On effectue un retrait analogue pour la variable de fin e_t afin de maintenir la cohérence entre les deux variables, i.e. pour maintenir la relation $s_t + d_t = e_t$.
 – `[DÉSACTIVATION DE LA TÂCHE]` (lignes 9 à 10). Si la condition $\delta_{next} > \overline{e_t}$ est évaluée à vrai, alors la tâche t ne coupera plus jamais l'intervalle de balayage et peut être retirée de $\mathcal{T}_{prune}$.

Exemple 3.3. (Illustration de l'algorithme de balayage). *Nous détaillons maintenant pas à pas le déroulement de l'algorithme de balayage de 2001 sur l'instance donnée dans l'exemple 3.1 page 18. Après la génération et le tri des évènements (voir exemple 3.2 page 19),* $\langle PR, 1, 0, 0\rangle$ *est extrait de l_events, la position* δ *de la droite de balayage et l'espace disponible* gap *sont respectivement initialisés à 0 et à 3.*

ALGORITHM sweep_2001() : boolean

```
1: [INITIALISATION]
2: l_events ← generation of events wrt n, s_t, s̄_t, d_t, e_t and h and Table 3.1.
3: sort(l_events)
4: extract ⟨type, task, δ_next, inc⟩ from l_events
5: δ ← δ_next
6: gap ← limit
7: T_prune ← ∅
8: [BOUCLE PRINCIPALE]
9: while ⟨type, task, δ_next, inc⟩ ≠ null do
10:    if type ≠ PR then
11:       [DÉPLACEMENT DE LA DROITE DE BALAYAGE]
12:       if δ ≠ δ_next then
13:          if gap < 0 then return false ;
14:          if ¬filter_2001() then return false ;
15:          δ ← δ_next
16:       [TRAITER UN ÉVÈNEMENT DE PROFIL (SCP ou SCP)]
17:       gap ← gap + inc
18:    else
19:       [TRAITER UN ÉVÈNEMENT DE DÉBUT AU PLUS TÔT (PR)]
20:       T_prune ← T_prune ∪ {task}
21:    [RÉCUPÉRER LE PROCHAIN ÉVÈNEMENT]
22:    extract ⟨type, task, δ_next, inc⟩ from l_events
23: if gap < 0 then return false ;
24: if ¬filter_2001() then return false ;
25: return true
```

Algorithme 1: Retourne **false** si un dépassement de ressource est détecté ou si le domaine d'une variable devient vide durant le balayage, **true** sinon.

Lecture des évènements associés à l'instant 0. *Le premier évènement étant du type PR, la tâche associée t_1 est ajoutée dans la liste des tâches actives $\mathcal{T}_{prune}$ (algorithme 1 ligne 20). Les trois prochains évènements de la liste, $\langle PR, 2, 0, 0\rangle$, $\langle PR, 3, 0, 0\rangle$ et $\langle PR, 4, 0, 0\rangle$ étant placés à la même date et étant du même type conduisent à l'ajout de tâches t_2, t_3 et t_4 dans $\mathcal{T}_{prune}$.*

Lecture de l'évènement associé à l'instant 1. *Ensuite l'évènement $\langle SCP, 0, 1, -2\rangle$ correspondant au début de partie obligatoire de la tâche t_0 est extrait (ligne 22) et δ_{next} prend la valeur 1. La position courante δ et la prochaine position δ_{next} de la droite de balayage sont maintenant différentes, ce qui signifie que sur l'intervalle de balayage $[\delta, \delta_{next}[$ la hauteur de ressource disponible est égale à gap, i.e. 3. L'algorithme 2 est alors appelé pour tenter de filtrer le domaine des variables des tâches contenues dans $\mathcal{T}_{prune}$ par rapport à cet intervalle et à l'espace disponible.*

Filtrage par rapport à l'intervalle de balayage courant $[0, 1[$ et déplacement de la droite de balayage. *La hauteur de chacune des tâches t_1, t_2, t_3 et t_4 étant strictement inférieure à gap aucun filtrage ne peut être effectué (algorithme 2, ligne 4). La fin au plus tard de chacune de ces tâches étant strictement supérieure à δ_{next} aucune d'elles n'est retirée de $\mathcal{T}_{prune}$ (ligne 9). Puis la droite de balayage avance au point de temps 1 (algorithme 1, ligne 15) et le début de la partie obligatoire de la tâche t_0 réduit l'espace disponible qui vaut désormais 1*

```
ALGORITHM filter_2001() : boolean
 1: [PARCOURS DES TÂCHES ACTIVES]
 2: for all t ∈ T_prune do
 3:   [FILTRAGE DE LA TÂCHE]
 4:   if h_t > gap then
 5:     if (s̄_t ≥ e_t) ∨ (e_t ≤ δ ∨ s̄_t ≥ δ_next) then // pas de partie obligatoire présente dans l'intervalle [δ, δ_next[
 6:       if ¬remove_interval_var(s_t, δ − d_t + 1, δ_next − 1) return false
 7:       if ¬remove_interval_var(e_t, δ + 1, δ_next + d_t − 1) return false
 8:   [DÉSACTIVATION DE LA TÂCHE]
 9:   if δ_next > ē_t then
10:     remove t from T_prune
11: return true
```

Algorithme 2: Retourne **false** si le domaine d'une variable devient vide, **true** sinon.

(ligne 17).

Lecture de l'évènement associé à l'instant 2. *Enfin, le dernier évènement $\langle ECP, 0, 2, 2\rangle$ correspondant à la fin de la partie obligatoire de la tâche t_0 est extrait. Puisque la position courante δ et la prochaine position δ_{next} de la droite de balayage sont différentes, on vérifie qu'il n'y a pas de dépassement de ressource et l'algorithme 2 est appelé.*

Filtrage par rapport à l'intervalle de balayage courant $[1, 2[$ et déplacement de la droite de balayage. *Seules les tâches t_1 et t_4 parmi celles candidates au filtrage ont une hauteur strictement supérieure à l'espace disponible. La tâche t_1 ne possédant pas de partie obligatoire, l'intervalle de valeurs $[0, 1]$ est retiré du domaine de sa variable de début et l'intervalle $[2, 3]$ est retiré de sa variable de fin. Comme pour t_1, la tâche t_4 ne possède pas de partie obligatoire et l'intervalle de valeurs $[-1, 1]$ est donc retiré du domaine de sa variable de début et l'intervalle $[2, 4]$ est retiré du domaine de sa variable de fin. La fin au plus tard des tâches présentent dans $\mathcal{T}_{prune}$ étant toutes strictement supérieures à δ_{next}, aucune de ces tâches n'est retirée de $\mathcal{T}_{prune}$. Ensuite, la droite de balayage avance au point de temps 2 (algorithme 1, ligne 15) et la fin de la partie obligatoire de la tâche t_0 augmente l'espace disponible qui vaut désormais 3 (ligne 17). La liste des évènements l_events étant vide, l'algorithme 1 termine et renvoie **true**.*

Lors de ce premier balayage, le début au plus tôt des tâches t_1 et t_4 a été ajusté à 2 à cause de la tâche t_0 qui est fixée. La partie (B) de la figure 3.1 page 19 illustre l'état des variables après ce premier balayage. La fin au plus tôt de la tâche t_1 est maintenant strictement supérieure à son début au plus tard, i.e. la tâche t_1 possède une partie obligatoire. Pour prendre en compte cette nouvelle partie obligatoire l'algorithme de balayage doit être relancé une deuxième fois. Pendant ce deuxième appel, illustré par la partie (C) de la figure 3.1, le début au plus tôt de la tâche t_4 est repoussé à 4. L'algorithme est relancé une troisième et dernière fois et aucun filtrage n'est effectué, le point fixe est atteint.

3.3.4 Complexité d'une phase de balayage

Étant donné une contrainte *cumulative* impliquant n tâches, la complexité temporelle dans le pire des cas de l'algorithme de balayage de 2001 est $O(n^2)$.

3.4 Synthèse critique

Une limite fondamentale des algorithmes de filtrage pour la contrainte *cumulative* réside dans le fait qu'au delà d'un certain nombre de tâches, toutes les relaxations supposent que toute la marge (i.e. la différence entre l'espace disponible et l'espace nécessaire) peut se concentrer à un seul endroit (e.g. sur une colonne ou sur un intervalle donné) alors qu'elle se répartit en pratique. En cela, toutes les relaxations induisent un effet d'horizon. En effet, si elles aident significativement jusqu'à une certaine taille de problèmes ou un certain profil de données, elles deviennent « aveugles » et trop lourdes ensuite.

Mis à part les travaux récents sur les explications et les nogoods, les contributions se concentrent généralement sur le traitement d'une seule contrainte. Entrelacer l'exécution d'algorithmes de filtrage lourds associés à des contraintes globales est de plus en plus problématique par rapport à la convergence au point fixe dès lors que le nombre de contraintes globales augmente.

Contribution : vers un algorithme de filtrage intégrant une conjonction de contraintes *cumulative*, *cumulative colorée* et *précédence*

4

Motivation et démarche : passage à l'échelle en maîtrisant la convergence au point fixe

Sommaire

4.1 Faiblesses de l'algorithme de balayage de 2001 27

4.2 Solutions préconisées 28

Dans ce chapitre nous identifions dans un premier temps les faiblesses de l'algorithme de balayage de 2001 proposé dans [8] en vue d'un meilleur passage à l'échelle. Dans un deuxième temps, nous proposons pour chacune des faiblesses identifiées une ligne de conduite qui sera adoptée pour la conception des nouveaux algorithmes de balayage présentés dans cette thèse. Finalement, dans un troisième temps nous montrons comment l'idée d'utiliser un tri topologique pour propager en deux phases un ensemble de contraintes de précédence s'étend pour traiter conjointement un ensemble de contraintes de précédence et de contraintes *cumulative*.

4.1 Faiblesses de l'algorithme de balayage de 2001

En revisitant l'algorithme de balayage proposé dans [8] pour le filtrage de la contrainte *cumulative* et en réexaminant son implémentation dans les solveurs Choco [69] et SICStus [19], nous avons identifié les cinq faiblesses suivantes d'un point de vue passage à l'échelle.

① `[Trop statique]` L'augmentation potentielle du PPO durant un balayage n'est pas dynamiquement pris en compte. En d'autres termes, les créations et extensions des parties obligatoires durant une phase de balayage ne sont pas immédiatement utilisées pour effectuer plus de filtrage durant cette même phase de balayage. L'exemple 3.3 page 21 illustre ce point par le fait que, déjà sur un tout petit nombre de tâches, l'algorithme a besoin d'être relancé trois fois avant d'atteindre son point fixe.

② [Atteint souvent sa complexité temporelle maximale] La complexité temporelle dans le pire des cas de l'algorithme de balayage est $O(n^2)$ où n est le nombre de tâches. Cette complexité est souvent atteinte en pratique lorsque la plupart des tâches peuvent être placées n'importe où sur l'axe du temps. La raison étant que l'algorithme a besoin de reparcourir systématiquement, à chaque position δ de la droite de balayage, toutes les tâches intersectant δ. En profilant l'exécution de l'algorithme de balayage disponible dans les solveurs de contraintes Choco et SICStus, on constate qu'il passe plus de 50% de son temps à parcourir la liste des tâches candidates au filtrage.

③ [Crée des trous dans les domaines] L'algorithme de balayage supprime des intervalles de valeurs consécutives des domaines des variables. C'est un point faible qui empêche de traiter des instances avec un découpage fin du temps puisque les domaines des variables ne peuvent pas être représentés de manière compacte en mémoire par leurs valeurs minimale et maximale respectives (i.e. on est obligé de gérer les trous dans les domaines dans un contexte où il faut restaurer l'état du domaine d'une variable).

④ [Ne profite pas du bin-packing] Lorsque toutes les tâches ont une durée égale à un, on est en présence d'un problème de bin-packing, la complexité temporelle dans le pire des cas de l'algorithme, $O(n^2)$, reste inchangée.

⑤ [Trop local] Avoir dans un même problème plusieurs contraintes *cumulative* qui partagent des variables conduit à la source d'inefficacité suivante. Dans une configuration classique chaque contrainte est propagée indépendamment, et à cause des variables partagées l'algorithme de balayage associé à chaque *cumulative* devra être relancé plusieurs fois avant d'atteindre le point fixe global à l'ensemble des contraintes. Notons que la modification de la date de début au plus tôt ou de la date de fin au plus tard d'une seule tâche provoquera le réveil de toutes les autres contraintes dans lesquelles la tâche est impliquée.

4.2 Solutions préconisées

Éviter le point ① [Trop statique]. Comme l'illustre l'exemple 3.3 page 21, l'algorithme de balayage doit être relancé plusieurs fois pour atteindre son point fixe. Cela est dû au fait que durant une étape de balayage, la restriction des domaines des variables de début et fin des tâches n'est pas directement pris en compte. Les nouveaux algorithmes de balayage présentés dans cette thèse filtrent les variables de début des tâches en deux phases successives. Une première, dénommée *sweep_min*, tente d'ajuster le début au plus tôt des tâches en effectuant un balayage de la gauche vers la droite, tandis que la seconde, nommée *sweep_max*, tente d'ajuster la fin au plus tard des tâches en effectuant un balayage de la droite vers la gauche. Il est important de noter que ce propagateur doit malgré tout répéter ces deux phases pour atteindre son point fixe. Supposons que *sweep_min* est terminé son exécution et *sweep_max* augmente le PPO. Dans ce cas, *sweep_min* ne sera plus à son point fixe et devra être relancé et ainsi de suite. Notons que les parties *sweep_min* et *sweep_max* sont complètement symétriques. Dans tous les nouveaux algorithmes de balayage que nous présenterons dans cette thèse, *sweep_min* (et donc *sweep_max*) profite dynamiquement des déductions qu'il effectue pour atteindre son point fixe en un seul balayage (i.e. *sweep_min* prend directement en compte tous les ajustements de début au plus tôt qu'il fait au cours d'un balayage).

Pour traiter cet aspect, nous introduirons au chapitre suivant le concept d'*évènement conditionnel*, i.e. un évènement qui est créée durant le balayage, et d'*évènement dynamique*, i.e. un évènement qui peut éventuellement se déplacer sur l'axe temporel.

Éviter le point ② `[Atteint souvent sa complexité temporelle maximale]`. Pour éviter partiellement le point ② dû au reparcours systématique des tâches qui intersectent la position courante de la droite de balayage pour essayer de mettre à jour leur date de début au plus tôt, nous introduirons plusieurs structures de données dédiées dans nos algorithmes. L'intuition à l'origine de ces structures de données est la suivante : si une tâche de hauteur h ne peut pas couper la position courante de la droite de balayage entrainant un ajustement de son début au plus tôt, alors toutes les tâches similaires possédant une hauteur supérieure ou égale à h doivent aussi être ajustées ; et symétriquement, si une tâche de hauteur h peut couper la position courante de la droite de balayage, alors toutes les tâches de hauteur inférieure ou égale à h n'ont pas besoin d'être ajustées.

Éviter le point ③ `[Crée des trous dans les domaines]`. Les nouveaux algorithmes que nous présentons dans cette thèse ne traitent que les bornes des domaines, ce qui est un bon moyen de réduire la consommation mémoire due à la représentation des domaines des variables dans un contexte où il faut les restaurer.

Éviter le point ④ `[Ne profite pas du bin-packing]`. Les structures de données dédiées que nous introduisons pour éviter le point ② nous permettrons également de réduire la complexité temporelle dans le pire des cas pour les problèmes de bin-packing. L'intuition étant que lorsque l'on a trouvé un point de temps pour accueillir le tout début de la tâche, on n'a pas besoin de test supplémentaire lorsque sa durée vaut 1. Ce point sera détaillé au prochain chapitre, dans l'analyse de la complexité de l'algorithme de balayage dynamique.

Éviter le point ⑤ `[Trop local]`. Pour traiter ce problème nous avons dans un premier temps conçu un algorithme de filtrage qui gère plusieurs ressources en parallèle au sein d'une seule et même contrainte nommée k*-dimensional cumulative*. La différence principale étant que l'on ajuste directement le début au plus tôt des tâches en considérant toutes les ressources simultanément plutôt que successivement, évitant ainsi de repropager chaque ressource indépendamment les unes des autres. Dans un second temps, nous avons prolongé cette idée et conçu un autre algorithme permettant de traiter plusieurs ressources en parallèle ainsi qu'un ensemble de contraintes de précédence dans une seule contrainte que nous avons nommée k*-dimensional cumulative with precedences*.

Traiter un ensemble de précédences. Étant donné un ensemble de tâches $\mathcal{T}$ où chacune d'elles est définie par sa durée fixe, son début au plus tôt et sa fin au plus tard, et un ensemble de précédences, où chacune d'elles indique qu'une tâche appartenant à $\mathcal{T}$ doit se terminer avant le début d'une autre tâche de $\mathcal{T}$, ajuster le début au plus tôt et la fin au plus tard de chaque tâche peut être fait en deux phases successives. En représentant ces relations de précédence par un graphe orienté où chaque tâche est un sommet et chaque arc est une relation de précédence entre deux tâches :

① La première phase ajuste successivement le début au plus tôt de chaque tâche en sélectionnant une source, i.e. une tâche sans prédécesseur, en la supprimant du graphe et en mettant à jour le début au plus tôt de ses successeurs directs.

② De manière similaire, la deuxième phase ajuste successivement la fin au plus tard de chaque tâche en sélectionnant un puits, i.e. une tâche sans successeur, en la supprimant du graphe et en mettant à jour la fin au plus tard de ses prédécesseurs directs.

Intégration de l'idée du tri topologique. Dès lors que des contraintes de ressource entrent en jeu, la méthode décrite ci-dessus n'est plus considérée et les contraintes de précédence sont propagées indépendamment jusqu'au point fixe. L'idée clé de cette thèse est de réutiliser autant que possible le principe de la méthode que nous venons de voir. Dans une première phase, on sélectionne la tâche qui a le plus petit début au plus tôt et on l'ajuste en prenant en compte l'*ensemble* des contraintes où elle est impliquée, i.e. les contraintes de précédence mais aussi les contraintes de ressource. Pour atteindre cet objectif, nous revisitons la manière dont les contraintes de ressource et de précédence sont propagées en les considérant de manière synchrone plutôt qu'indépendamment. Sans pour autant augmenter le pouvoir de déduction on augmente dans la pratique la vitesse de convergence au point fixe d'une conjonction de contraintes de précédence et de ressource.

5

Algorithme de balayage dynamique pour une seule contrainte *cumulative*

Sommaire

5.1 Propriété **31**

5.2 Types d'évènements **32**

5.3 État de la droite de balayage **33**

5.4 Algorithme de filtrage **34**

5.4.1 Algorithme principal 34

5.4.2 Filtrage 35

5.4.3 Resynchronisation des évènements 37

5.5 Correction et propriété vérifiée par sweep_min **41**

5.6 Complexité **43**

Nous présentons dans ce chapitre un nouvel algorithme de balayage dynamique pour la contrainte *cumulative* dont l'objectif principal est de permettre un meilleur passage à l'échelle. Ce nouvel algorithme s'appuie donc logiquement sur les décisions retenues et énoncées au chapitre précédent. Dans un premier temps nous donnerons la propriété vérifiée par notre algorithme à son point fixe. Dans un deuxième temps, nous ferons la description des nouveaux évènements et de l'état de la droite de balayage qui suivra le même schéma que celui de l'algorithme de balayage présenté dans l'état de l'art. Enfin, dans un troisième et quatrième temps, nous prouverons que la propriété énoncée est bien vérifiée au point fixe, et nous analyserons la complexité de l'algorithme présenté.

5.1 Propriété

Notre algorithme de balayage dynamique sweep_min page 36 pour la contrainte *cumulative* vérifie la propriété suivante.

Propriété 5.1. *Étant donné une contrainte* cumulative *portant sur un ensemble de tâches* $\mathcal{T}$ *et une ressource de capacité maximale* $limit$*, l'algorithme* sweep_min *vérifie que :*

$$\forall t \in \mathcal{T}, \forall i \in [\underline{s_t}, \underline{e_t}) \ : \ h_t + \sum_{\substack{t' \in \mathcal{T} \setminus \{t\}, \\ i \in [\overline{s_{t'}}, \underline{e_{t'}})}} h_{t'} \leq limit \tag{5.1}$$

La propriété 5.1 vérifie que, pour n'importe quelle tâche t de la contrainte *cumulative*, on peut fixer t à son début au plus tôt sans dépasser la capacité maximale de la ressource en considérant le PPO des tâches de $\mathcal{T} \setminus \{t\}$. Notons que de cette propriété nous pouvons construire une solution relâchée de la contrainte *cumulative* :

- en fixant à 0 la consommation des tâches qui n'ont pas de partie de obligatoire,
- en fixant la durée des tâches ayant une partie obligatoire à la taille de celle-ci, i.e. $\underline{e_t} - \overline{s_t}$,
- en fixant le début de chaque tâche à sa date de début au plus tôt.

Une telle construction permet de pouvoir caractériser un algorithme de filtrage incomplet en fonction de la relaxation qu'il effectue. L'objectif de cette caractérisation est de décrire la quantité de filtrage effectuée par rapport à la relaxation de la contrainte [55].

5.2 Types d'évènements

Pour résoudre le problème soulevé par le point ① `[Trop statique]` du chapitre précédent, sweep_min doit être capable de gérer à la volée la création et l'extension des parties obligatoires causées par les ajustements des débuts au plus tôt des tâches. Nous devons par conséquent modifier les évènements introduit dans la table 3.1 du chapitre 3. La nouvelle table 5.1 présente les évènements utilisés par sweep_min ainsi que leur correspondance avec ceux de l'algorithme de balayage originel.

- L'évènement $\langle SCP, t, \overline{s_t}, -h_t \rangle$ correspondant au début de la partie obligatoire de la tâche t reste inchangé. Remarquons que, puisque sweep_min ne modifie que la date de début au plus tôt des tâches, le début de la partie obligatoire (qui correspond au début au plus tard) ne peut jamais être repoussé.
- L'évènement $\langle ECP, t, \underline{e_t}, h_t \rangle$ correspondant à la fin de la partie obligatoire de la tâche t est converti en $\langle ECPD, t, \underline{e_t}, h_t \rangle$ où le D dénote le caractère *dynamique* de l'évènement. En effet, cet évènement est associé à la date de fin au plus tôt de t et peut donc être repoussé si l'on ajuste le début au plus tôt de la tâche t.
- Un nouvel évènement, $\langle CCP, t, \overline{s_t}, 0 \rangle$, où CCP signifie *conditional compulsory part*, est initialement créé pour chaque tâche t n'ayant pas de partie obligatoire. Au plus tard, lorsque la droite de balayage atteint la position $\overline{s_t}$, elle ajuste le début au plus tôt de la tâche t, ce qui permet de savoir si une partie obligatoire apparait ou pas. Si tel est le cas, cet évènement se transforme en deux autres évènements SCP et $ECPD$ reflétant la création de la nouvelle partie obligatoire. Par simplicité nous appellerons par la suite ce nouveau type d'évènement *évènement conditionnel*.
- L'évènement $\langle PR, t, \underline{s_t}, 0 \rangle$ correspondant au début au plus tôt de la tâche t reste inchangé. Il est nécessaire pour ajouter la tâche t à la liste des tâches qui peuvent potentiellement intersecter la position courante de la droite de balayage.

Nouveaux évènements	Anciens évènements	Conditions
$\langle SCP, t, \overline{s_t}, -h_t\rangle$	$\langle SCP, t, \overline{s_t}, -h_t\rangle$	$\overline{s_t} < \underline{e_t}$
$\langle ECPD, t, \underline{e_t}, +h_t\rangle$	$\langle ECP, t, \underline{e_t}, +h_t\rangle$	$\overline{s_t} < \underline{e_t}$
$\langle CCP, t, \overline{s_t}, 0\rangle$		$\overline{s_t} \geq \underline{e_t}$
$\langle PR, t, \underline{s_t}, 0\rangle$	$\langle PR, t, \underline{s_t}, 0\rangle$	$\underline{s_t} \neq \overline{s_t}$

TABLE 5.1 – Liste des différents types d'évènements avec la condition nécessaire à leur génération. Le dernier attribut, *hauteur*, n'est pertinent que pour les évènements du type SCP, ECP et $ECPD$.

D'une part, certains évènements peuvent voir leur date modifiée pendant le balayage (cf. $ECPD$). D'autre part, certains évènements peuvent être créés pendant le balayage (cf. CCP). C'est pourquoi, à la place de simplement trier les évènements générés initialement, on les insère dans un tas nommé h_events, ce qui nous permet d'en ajouter ou de les mettre à jour dynamiquement en cours de balayage.

Exemple 5.1. (Évènements générés). *En reprenant l'instance de l'exemple 3.1 page 18 introduit pour illustrer l'algorithme de balayage de 2001, les évènements suivants sont générés et insérés dans le tas des évènement h_events (les nouveaux évènements apparaissent en gras) :* $\langle PR, 1, 0, 0\rangle$, $\langle PR, 2, 0, 0\rangle$, $\langle PR, 3, 0, 0\rangle$, $\langle PR, 4, 0, 0\rangle$, $\langle SCP, 0, 1, -2\rangle$, $\langle \mathbf{ECPD}, \mathbf{0}, \mathbf{2}, \mathbf{2}\rangle$, $\langle \mathbf{CCP}, \mathbf{1}, \mathbf{3}, \mathbf{0}\rangle$, $\langle \mathbf{CCP}, \mathbf{2}, \mathbf{5}, \mathbf{0}\rangle$, $\langle \mathbf{CCP}, \mathbf{4}, \mathbf{7}, \mathbf{0}\rangle$, $\langle \mathbf{CCP}, \mathbf{3}, \mathbf{9}, \mathbf{0}\rangle$. *L'évènement $\langle \mathbf{ECPD}, \mathbf{0}, \mathbf{2}, \mathbf{2}\rangle$ dénote la fin de la partie obligatoire de la tâche t_0. Dans notre exemple, puisque la tâche t_0 est déjà fixée, cet évènement ne sera en fait jamais repoussé sur l'axe temporel. L'évènement $\langle \mathbf{CCP}, \mathbf{1}, \mathbf{3}, \mathbf{0}\rangle$ indique le point de temps où la partie obligatoire de la tâche t_1 peut apparaitre, si et seulement si sa date de début au plus tôt est suffisamment ajustée, i.e. si $\underline{s_t} + d_t > \overline{s_t}$.*

5.3 État de la droite de balayage

La droite de balayage maintient les informations suivantes :

- La position courante δ de la droite de balayage, initialement positionnée à la date du premier évènement (i.e. la plus petite date).
- La quantité de ressource disponible au point de temps δ, noté gap, i.e. la différence entre la capacité maximale de la ressource $limit$ et la hauteur du PPO au point de temps δ.
- Deux tas, $h_conflict$ et h_check, pour éviter partiellement le point ② évoqué au chapitre précédent, c'est-à-dire éviter de reparcourir entièrement toute la liste des tâches qui peuvent couper la position courante de la droite de balayage chaque fois qu'elle se déplace. Supposons que la droite de balayage soit à sa position initiale et qu'un évènement de type PR associé à la tâche t soit lu. Alors, suivant la hauteur de la tâche t on observe les deux cas de figure suivants :
  - D'une part, si la hauteur de la tâche t est strictement plus grande que la ressource disponible au point de temps courant δ, on peut en conclure que le début au plus tôt de la tâche t devra être ajusté. Pour éviter de re-vérifier à chaque fois que la droite de balayage se déplace si oui ou non l'espace disponible est suffisant pour t (i.e. si $gap \geq h_t$), on place la tâche en $conflict$ par rapport à δ. Pour cela, on

l'insère dans le tas $h_conflict$ qui enregistre toutes les tâches qui sont en conflit avec la position courante δ de la droite de balayage, ordonnées par ordre croissant de hauteur, i.e. le premier élément du tas est la tâche qui possède la plus petite hauteur. Cet ordre est induit par le fait que, si on doit ajuster le début au plus tôt de la tâche t, alors on doit également ajuster celui de toutes les autres tâches qui ont une hauteur supérieure ou égale à h_t.

- D'autre part, si la hauteur de la tâche t est inférieure ou égale à la ressource disponible au point de temps courante δ, on sait que le début au plus tôt de la tâche t peut être égale à δ. Mais, pour en être sûr, nous devons vérifier la propriété 5.1 pour la tâche t, i.e. vérifier qu'il n'y a aucun conflit sur l'intervalle $[\delta, \delta{+}d_t[$. Dans ce but, on insère la tâche t dans le tas h_check qui enregistre toutes les tâches pour lesquelles on s'occupe de vérifier la propriété 5.1. La tâche t reste donc dans le tas h_check jusqu'à ce qu'un conflit soit détecté ou que la droite de balayage ait atteint le point de temps $\delta{+}d_t$ sans détecter de conflit. Dans le dernier cas, on a trouvé la première position possible pour la tâche t par rapport à la propriété 5.1. Dans le tas h_check les tâches sont ordonnées par ordre décroissant de hauteur, i.e. le premier élément du tas est la tâche qui possède la hauteur la plus grande. De manière symétrique au tas $h_conflict$, si la tâche t au sommet du tas n'est pas en conflit au point de temps δ, alors toutes les autres tâches dans le tas h_check avec une hauteur plus petite ou égale à h_t ne le sont pas non plus.

5.4 Algorithme de filtrage

L'algorithme sweep_min effectue un unique balayage sur la série d'évènements pour ajuster le début au plus tôt des tâches et respecter à la fin du balayage la propriété 5.1. Il est composé d'un algorithme principal, d'une partie dédié au filtrage et d'une autre partie s'occupant de la synchronisation. Cette dernière est requise pour gérer les créations et extensions des parties obligatoires produites par les ajustements de débuts au plus tôt durant le balayage.

Pour la description de l'algorithme nous utiliserons la fonction $\mathrm{empty}(h)$ qui retourne **true** si le tas h est vide, **false** sinon. La fonction $\mathrm{get_top_key}(h)$ retourne la clé du premier élément du tas h. La fonction $\mathrm{adjust_min_var}(var, val)$ ajuste la valeur minimale de la variable var à val. Nous introduisons un tableau d'entiers $mins$ qui, pour chaque tâche t présente dans le tas h_check, conserve la valeur de δ lorsque t a été ajoutée dans h_check. Un tableau de booléens $evup$ indique pour chaque tâche t si les évènements relatifs à sa partie obligatoire sont à jour ou pas. L'entrée correspondante à la tâche t dans $evup$ est à **true** dès que l'on a trouvé la valeur finale pour le début au plus tôt de la tâche t et que les évènements relatifs à sa partie obligatoire, si elle existe, sont à jour dans le tas h_events. Enfin, nous introduisons une liste $newActiveTasks$ qui contient toutes les tâches dont l'évènement PR a été lu sur la position courante de la droite de balayage δ, i.e. les nouvelles tâches actives. Cette liste servira de tampon pour accumuler toutes les nouvelles tâches actives avant de les traiter.

5.4.1 Algorithme principal

L'algorithme sweep_min page 36 est composé des parties suivantes :

- `[INITIALISATION]` (lignes 2 à 6). Les évènements sont générés et ajoutés dans le tas h_events selon les conditions spécifiées dans la table 5.1. Les tas h_check et $h_conflict$ sont initialisés à vide. Le booléen $evup_t$ est initialisé à **true** si et seulement

si la tâche t est fixée (i.e. $\underline{s_t} = \overline{s_t}$). L'entier $mins_t$ est initialisé à la date de début au plus tôt de la tâche t. La liste $newActiveTasks$ est initialisée à vide. La droite de balayage est positionnée sur la date du premier évènement, i.e. la date de l'évènement au sommet de h_events.

- [BOUCLE PRINCIPALE] (lignes 8 à 25). Pour chaque date δ la boucle principale lit et traite tous les évènements correspondant. Elle comporte les étapes suivantes :
  - [DÉPLACEMENT DE LA DROITE DE BALAYAGE] (lignes 10 à 17). A chaque fois que la droite de balayage se déplace, son état est mis à jour par rapport aux nouvelles tâches actives, i.e. l'ensemble des tâches pour lesquelles le début au plus tôt est égal à δ. Toutes les nouvelles tâches actives qui sont en conflit à la date δ sont insérées dans le tas $h_conflict$ (ligne 13). Pour chaque tâche t qui n'est pas en conflit, on vérifie si l'intervalle de balayage $[\delta, \delta_{next}[$ est suffisamment grand pour accueillir la tâche sur toute sa durée. Les tâches pour lesquelles l'intervalle de balayage est trop petit sont ajoutées dans le tas h_check (ligne 14). Puis, pour prendre en compte la variation de ressource disponible au point de temps δ, l'algorithme filter_min page 38 est appelé pour mettre à jour les tas h_check et $h_conflict$ et ajuster le début au plus tôt des tâches pour lesquelles la première position réalisable par rapport à la propriété 5.1 a été trouvée.
  - [TRAITER L'ÉVÈNEMENT COURANT] (lignes 19 à 22). Tout d'abord, l'algorithme synchronize appelé ligne 19 (1) converti les évènements conditionnels CCP en évènements SCP et $ECPD$, ou les ignore suivant que la tâche correspondante ait ou non une partie obligatoire, (2) il repousse les évènements dynamiques ($ECPD$) à leur véritable position, i.e. il modifie la date associé à l'évènement. Ensuite, le premier évènement du tas h_events est extrait. Selon son type (i.e. SCP ou $ECPD$), la ressource disponible est mise à jour, ou (i.e. PR), la tâche associé à l'évènement courant est ajoutée dans la liste des nouvelles tâches actives (ligne 22).
  - [RÉCUPÉRER LE PROCHAIN ÉVÈNEMENT] (lignes 24 à 25). S'il n'y a plus d'évènement dans h_events, l'algorithme filter_min est appelé dans le but de vider le tas h_check, ce qui peut générer de nouveaux évènements de parties obligatoires. En effet, même s'il n'y a plus d'évènement dans le tas h_events, les tâches présentes dans le tas h_check peuvent être en attente d'un ajustement de leur début au plus tôt. C'est cet ajustement qui peut engendrer la création de nouvelles parties obligatoires et ainsi provoquer de nouveaux ajustements.

5.4.2 Filtrage

Lorsque tous les évènements dont la date est égale à la position courante de la droite de balayage, δ, ont été traités, l'algorithme filter_min page 38 prend en compte la modification du PPO, i.e. la modification de l'espace disponible en hauteur, gap. Il tente d'ajuster le début au plus tôt des tâches se trouvant dans les tas h_check et $h_conflict$ par rapport à la propriété 5.1. Pour éviter les répétitions de code, nous utilisons aux lignes 8, 16 et 20, après l'ajustement du début au plus tôt de la tâche t, l'instruction « update events of the compulsory part of t ». Cette instruction met à jour et ajoute les évènements relatifs à la tâche t dans le tas h_events, découlant de cet ajustement. Si la tâche t avait une partie obligatoire avant l'ajustement, la date de l'évènement $ECPD$ est ajustée à la nouvelle fin au plus tôt de la

ALGORITHM sweep_min() : boolean
1: [INITIALISATION]
2: $h_events \leftarrow$ generation of events wrt n, $\underline{s_t}$, $\overline{s_t}$, d_t, $\underline{e_t}$ and h and Table 5.1.
3: h_check, $h_conflict \leftarrow \emptyset$; $newActiveTasks \leftarrow \emptyset$
4: **for** $t = 0$ **to** $n-1$ **do**
5: $evup_t \leftarrow (\underline{s_t} = \overline{s_t})$; $mins_t \leftarrow \underline{s_t}$
6: $\delta \leftarrow$ get_top_key(h_events) ; $\delta_{next} \leftarrow \delta$; $gap \leftarrow limit$
7: [BOUCLE PRINCIPALE]
8: **while** $\neg$empty(h_events) **do**
9: [DÉPLACEMENT DE LA DROITE DE BALAYAGE]
10: **if** $\delta \neq \delta_{next}$ **then**
11: **while** $\neg$empty($newActiveTasks$) **do**
12: extract first task t from $newActiveTasks$
13: **if** $h_t > gap$ **then** add $\langle h_t, t\rangle$ into $h_conflict$
14: **else if** $d_t > \delta_{next} - \delta$ **then** {add $\langle h_t, t\rangle$ into h_check ; $mins_t \leftarrow \delta$;}
15: **else** $evup_t \leftarrow$ **true**
16: **if** $\neg$filter_min(δ, δ_{next}) **then return false**
17: $\delta \leftarrow \delta_{next}$
18: [TRAITER L'ÉVÈNEMENT COURANT]
19: $\delta \leftarrow$ synchronize(δ)
20: extract $\langle type, t, \delta, dec\rangle$ from h_events
21: **if** $type = SCP \vee type = ECPD$ **then** $gap \leftarrow gap + dec$
22: **else if** $type = PR$ **then** $newActiveTasks \leftarrow newActiveTasks \cup \{t\}$
23: [RÉCUPÉRER LE PROCHAIN ÉVÈNEMENT]
24: **if** empty(h_events) $\wedge$ $\neg$filter_min($\delta, +\infty$) **then return false**
25: $\delta_{next} \leftarrow$ synchronize(δ)
26: **return true**

Algorithme 3: Retourne **false** si un dépassement de ressource est détecté durant le balayage. Si **true**, vérifie que le début au plus tôt de chaque tâche est ajusté selon la propriété 5.1

tâche t. Sinon, si cet ajustement provoque l'apparition d'une nouvelle partie obligatoire, l'évènement CCP est converti en un évènement SCP et un évènement $ECPD$ associé à la nouvelle fin au plus tôt de la tâche est créé et ajouté au tas h_events. Il est principalement composé des parties suivantes :

- [VÉRIFIER LE DÉPASSEMENT DE RESSOURCE] (ligne 2). Si la quantité de ressource disponible gap est négative sur l'intervalle de balayage courant $[\delta, \delta_{next}[$ l'algorithme filter_min retourne **false**, signifiant que la capacité maximale de la ressource est dépassée sur cet intervalle.
- [MISE À JOUR DES TÂCHES DU TAS h_check] (lignes 4 à 10). Toutes les tâches dans h_check dont la hauteur est supérieure à l'espace disponible gap sont extraites et traitées de la manière suivante.
  - Premier cas : la tâche t est restée suffisamment longtemps dans le tas h_check, i.e. $\delta - mins_t \geq d_t$, ligne 6, signifiant que la tâche n'est pas en conflit sur l'intervalle $[mins_t, \delta[$, dont la taille est supérieure ou égale à la durée d_t de la tâche t. Le début au plus tôt de la tâche doit donc être ajustée à la valeur $mins_t$. Rappelons que $mins_t$ correspond à la dernière position de la droite de balayage où la tâche t a été insérée dans le tas h_check.

– Deuxième cas : la droite de balayage a dépassé le début au plus tard de la tâche t, i.e. $\delta \geq \overline{s_t}$, ligne 6. Cela signifie que la tâche t n'a pas été en conflit sur l'intervalle $[mins_t, \delta[$, et que par conséquent on peut ajuster son début au plus tôt à la valeur de $mins_t$.

– Troisième cas : le tas des évènements h_events est vide, i.e. $\text{empty}(h_events)$, ligne 6. Dans ce cas, la hauteur du PPO est nécessairement égale à 0 et on doit vider le tas h_check, ce qui peut re-créer de nouveaux évènements.

– Dans tous les autres cas, puisque la hauteur de la tâche t est supérieure à la ressource disponible, la tâche est simplement mise en conflit, i.e. elle est insérée dans le tas $h_conflict$, ligne 10.

- [MISE À JOUR DES TÂCHES DU TAS $h_conflict$] (lignes 13 à 23). Toutes les tâches de $h_conflict$ qui ne sont plus en conflit, i.e. dont la hauteur est inférieure ou égale à l'espace disponible gap, sont extraites du tas $h_conflict$ et traitées de la manière suivante :

  – Premier cas : la position courante δ de la droite de balayage ne se situe pas avant le début au plus tard de la tâche t (ligne 14). Dans ce cas, on peut en déduire que la tâche t ne peut pas être positionnée ailleurs qu'à sa position au plus tard, et par conséquent son début au plus tôt est ajusté à la valeur de son début au plus tard.

  – Dans tous les autres cas (ligne 17) : on compare la durée d_t de la tâche t avec la taille de l'intervalle courant de balayage $[\delta, \delta_{next}[$; si d_t est inférieure ou égale, alors le début au plus tôt de la tâche est ajustée à δ, sinon, la tâche est insérée dans le tas h_check.

5.4.3 Resynchronisation des évènements

Pour traiter les évènements dynamiques et conditionnels, l'algorithme synchronize page 39 vérifie et met potentiellement à jour le premier évènement du tas h_events avant qu'il ne soit effectivement traité par sweep_min. L'algorithme synchronize est composé des parties suivantes :

- [MISE À JOUR DES PREMIERS ÉVÈNEMENTS] (lignes 2 à 17). Les évènements dynamiques et conditionnels requièrent d'être vérifiés avant d'être extraits et traités par l'algorithme sweep_min. La boucle débutant à la ligne 2 et se terminant à la ligne 17 déplace le premier évènement du tas h_events jusqu'à ce que l'évènement situé en tête du tas soit à jour. Notons que la boucle de saturation est nécessaire puisqu'après une mise à jour, l'évènement peut-être repoussé sur l'axe temporel et ne sera alors peut-être plus le premier élément du tas h_events.

- [TRAITER UN ÉVÈNEMENT DYNAMIQUE (ECPD)] (lignes 6 à 9). Un évènement du type $ECPD$ doit être mis à jour si la tâche associée à cet évènement est dans l'un des deux tas h_check ou $h_conflict$. Si la tâche t est dans le tas h_check cela signifie que son début au plus tôt peut être ajusté à $mins_t$. Son évènement $ECPD$ est donc mis à jour à la date $mins_t + d_t$ (ligne 7). Si la tâche t est dans le tas $h_conflict$, cela signifie que t ne peut pas commencer avant sa date de début au plus tard $\overline{s_t}$. Son évènement $ECPD$ est donc repoussé à la date $\overline{s_t} + d_t$ (ligne 8).

```
ALGORITHM filter_min(δ, δ_next) : boolean
 1: [VÉRIFIER LE DÉPASSEMENT DE RESSOURCE]
 2: if gap < 0 then return false
 3: [MISE À JOUR DES TÂCHES DU TAS h_check]
 4: while ¬empty(h_check) ∧ (empty(h_events) ∨ get_top_key(h_check) > gap) do
 5:    extract ⟨h_t, t⟩ from h_check
 6:    if δ ≥ s̄_t ∨ δ − mins_t ≥ d_t ∨ empty(h_events) then
 7:       if ¬adjust_min_var(s_t, mins_t) ∨ ¬adjust_min_var(e_t, mins_t + d_t) then re-
          turn false
 8:       if ¬evup_t then {update events of the compulsory part of t ; evup_t ← true ;}
 9:    else
10:       add ⟨h_t, t⟩ into h_conflict
11: [MISE À JOUR DES TÂCHES DU TAS h_conflict]
12: while ¬empty(h_conflict) ∧ get_top_key(h_conflict) ≤ gap do
13:    extract ⟨h_t, t⟩ from h_conflict
14:    if δ ≥ s̄_t then
15:       if ¬adjust_min_var(s_t, s̄_t) ∨ ¬adjust_min_var(e_t, ē_t) then return false
16:       if ¬evup_t then {update events of the compulsory part of t ; evup_t ← true ;}
17:    else
18:       if δ_next − δ ≥ d_t then
19:          if ¬adjust_min_var(s_t, δ) ∨ ¬adjust_min_var(e_t, δ + d_t) then return false
20:          if ¬evup_t then {update events of the compulsory part of t ; evup_t ← true ;}
21:       else
22:          add ⟨h_t, t⟩ into h_check ; mins_t ← δ ;
23: return true
```

Algorithme 4: Essaye d'ajuster le début au plus tôt des tâches dans h_check et $h_conflict$ par rapport à l'intervalle de balayage $\mathcal{I} = [\delta, \delta_{next}[$ et à la ressource disponible gap sur $\mathcal{I}$. Retourne **false** si un dépassement de ressource est constaté, **true** sinon.

- [TRAITER UN ÉVÈNEMENT CONDITIONNEL (CCP)] (lignes 11 à 16). Lorsque la droite de balayage atteint l'évènement CCP d'une tâche t, on doit savoir si une partie obligatoire existe ou pas pour la tâche t. Puisque le booléen $evup_t$ est fixé à **false**, on sait que la tâche t est soit dans le tas h_check soit dans le tas $h_conflict$. Si t est dans h_check, une partie obligatoire est créée si et seulement si $mins_t + d_t > \delta$ (lignes 12 et 13). Sinon, si t est dans $h_conflict$ la tâche est fixée à sa date de début au plus tard et les évènements associés à sa partie obligatoire sont ajoutés dans h_events (ligne 15).

Exemple 5.2. (Illustration du balayage dynamique). *Détaillons maintenant le déroulement de l'algorithme de balayage dynamique en reprenant l'instance de l'exemple 3.1 page 18 introduit pour illustrer l'algorithme de balayage de 2001. La liste des évènements générés est donnée dans l'exemple 5.1 page 33.*

Initialisation. *L'algorithme de balayage dynamique commence par initialiser la position courante de la droite de balayage à 0, i.e. la date du premier évènement, et l'espace disponible à 3, i.e. la capacité maximale de la ressource.*

Lecture des évènements associés à l'instant 0 et filtrage par rapport à l'intervalle de balayage courant $[0, 1[$**.** *L'algorithme lit les quatre évènements PR associés aux tâches*

ALGORITHM synchronize(δ) : integer
1: [MISE À JOUR DES PREMIERS ÉVÈNEMENTS]
2: **repeat**
3: **if** empty(h_events) **then return** $-\infty$
4: $sync \leftarrow$ **true** ; $\langle date, t, type, dec \rangle \leftarrow$ consult top event of h_events ;
5: [TRAITER UN ÉVÈNEMENT DYNAMIQUE (ECPD)]
6: **if** $type = ECPD \ \wedge \ \neg evup_t$ **then**
7: **if** $t \in h_check$ **then** update event date to $mins_t + d_t$
8: **else** update event date to $\overline{s_t} + d_t$
9: $evup_t \leftarrow$ **true** ; $sync \leftarrow$ **false** ;
10: [TRAITER UN ÉVÈNEMENT CONDITIONNEL (CCP)]
11: **else if** $type = CCP \ \wedge \ \neg evup_t \ \wedge \ date = \delta$ **then**
12: **if** $t \in h_check \wedge mins_t + d_t > \delta$ **then**
13: add $\langle SCP, t, \delta, -h_t \rangle$ and $\langle ECPD, t, mins_t + d_t, h_t \rangle$ into h_events
14: **else if** $t \in h_conflict$ **then**
15: add $\langle SCP, t, \delta, -h_t \rangle$ and $\langle ECPD, t, \overline{e_t}, h_t \rangle$ into h_events
16: $evup_t \leftarrow$ **true** ; $sync \leftarrow$ **false** ;
17: **until** $sync$
18: **return** $date$

Algorithme 5: Met à jour l'évènement au sommet du tas h_events et retourne la date du prochain évènement ou $-\infty$ si h_events est vide.

t_1, t_2, t_3 et t_4. Puisque la hauteur de t_1, t_2 et t_4 est inférieure ou égale à l'espace disponible et que leur durée est strictement supérieure à la taille de l'intervalle de balayage, ces tâches sont insérées dans le tas h_check (algorithme 3, ligne 14). La tâche t_3 n'est pas insérée dans le tas h_check car sa durée est égale à la taille de l'intervalle (algorithme 3, ligne 14), i.e. le début au plus tôt de t_3 ne peut pas être ajustée. L'état de la droite de balayage à ce stade est illustré par la figure 5.1.

Lecture de l'évènement associé à l'instant 1 et filtrage par rapport à l'intervalle de balayage courant $[1, 2[$. *Ensuite la droite de balayage est positionnée au point de temps 1 où se trouve le prochain évènement à traiter. L'évènement $\langle SCP, 0, 1, -2 \rangle$ est lu et l'espace libre gap est fixé à 1 (i.e. $3 - 2$). L'appel de* filter_min *avec $\delta = 1$, $\delta_{next} = 2$ et $gap = 1$ fait sortir les tâches t_1 et t_4 de h_check pour les transférer dans le tas $h_conflict$ (algorithme 4, ligne 10). En effet, vu la leur hauteur elles ne peuvent pas commencer à l'instant 1. L'état de la droite de balayage à ce stade est illustré par la figure 5.2.*

Lecture de l'évènement associé à l'instant 2 et filtrage par rapport à l'intervalle de balayage courant $[2, 3[$. *Ensuite la droite de balayage avance au point de temps 2 où l'évènement $\langle ECPD, 0, 2, 2 \rangle$ est traité et l'espace disponible fixé à 3. L'appel de* filter_min *avec $\delta = 2$, $\delta_{next} = 3$ et $gap = 3$ fait sortir les tâches t_1 et t_4 de $h_conflict$ pour les transférer dans h_check (algorithme 4, ligne 22). L'état de la droite de balayage à ce stade est illustré par la figure 5.3.*

Lecture des évènements associés à l'instant 3 et filtrage par rapport à l'intervalle de balayage courant $[3, 4[$. *Puis la droite de balayage avance au point de temps 3 et l'évènement conditionnel $\langle CCP, 1, 3, 0 \rangle$ est lu. Comme la tâche t_1 est dans le tas h_check et que sa date de fin au plus tôt est strictement supérieure à δ (algorithme 5, ligne 12) l'évènement CCP*

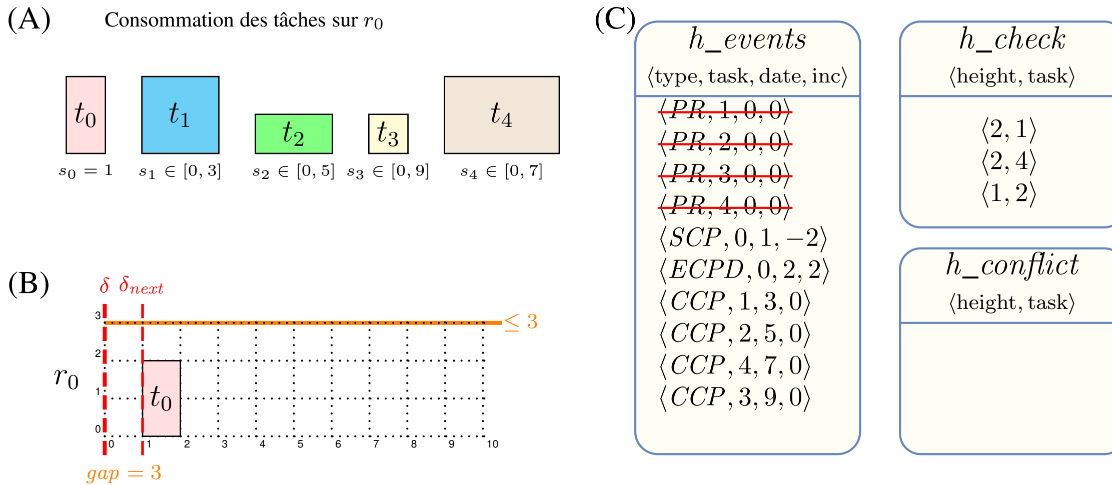


FIGURE 5.1 – État des tâches, de la droite de balayage et des tas après la lecture des évènements $\langle PR, 1, 0, 0\rangle$, $\langle PR, 2, 0, 0\rangle$, $\langle PR, 3, 0, 0\rangle$ et $\langle PR, 4, 0\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 16) - (A) date de début et consommations de chaque tâche, (B) le PPO et la position de la droite de balayage, (C) la liste des évènements restant à traiter ainsi que les tas h_check et $h_conflict$.

de t_1 est converti en deux évènements $\langle SCP, 1, 3, -2\rangle$ et $\langle ECPD, 1, 4, +2\rangle$ représentant la création de la partie obligatoire sur l'intervalle $[3, 4)$. Notons que la partie obligatoire créée pour la tâche t_1 est bien apparue après la position courante δ de la droite de balayage, ce qui est un point clé pour garantir la propriété 5.1. Le nouvel évènement SCP de t_1 est lu et l'espace disponible gap est fixé à 1. L'appel de filter_min *avec $\delta = 3$, $\delta_{next} = 4$ et $gap = 1$ retire la tâche t_4 du tas h_check et l'insère dans le tas $h_conflict$. L'état de la droite de balayage à ce stade est illustré par la figure 5.4.*

Lecture de l'évènement associé à l'instant 4 et filtrage par rapport à l'intervalle de balayage courant $[4, 5[$. *Enfin, la droite de balayage est déplacé au point de temps 4, l'évènement $\langle ECPD, 1, 4, +2\rangle$ est lu et l'espace disponible est fixé à 3. La tâche t_4 repasse dans le tas h_check. L'état de la droite de balayage à ce stade est illustré par la figure 5.5.*

La fin de l'exécution se résume à la lecture des trois derniers évènements CCP associés aux tâches t_2, t_4 et t_3, qui donnerons lieu à l'ajustement de la date de début au plus tôt de t_4 en 4 sans pour autant créer de partie obligatoire. Le point fixe est donc atteint.

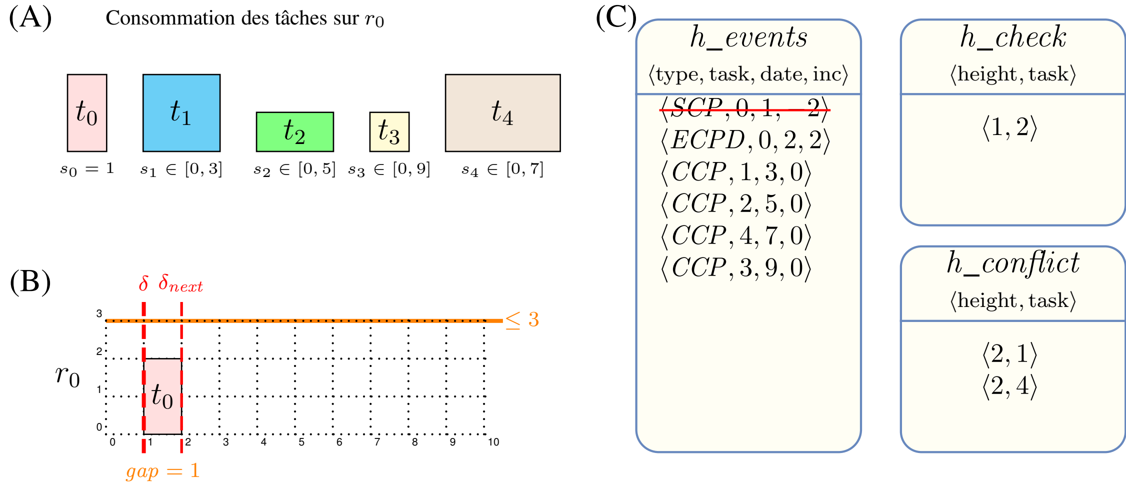


FIGURE 5.2 – État des tâches, de la droite de balayage et des tas après la lecture de l'évènement $\langle SCP, 0, 1, -2\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 16) - (A) date de début et consommations de chaque tâche, (B) le PPO et la position de la droite de balayage, (C) la liste des évènements restant à traiter ainsi que les tas h_check et $h_conflict$.

5.5 Correction et propriété vérifiée par sweep_min

Prouvons maintenant qu'après la terminaison de l'algorithme sweep_min (page 36), la propriété 5.1 est bien atteinte. Dans ce but, nous commençons par introduire le lemme suivant.

Lemme 5.1. *À tout point de temps de son exécution,* sweep_min *ne peut pas générer de nouvelles parties obligatoires se trouvant avant la position courante δ de la droite de balayage.*

Preuve du lemme 5.1 Puisque le début de la partie obligatoire d'une tâche t correspond à sa date de début au plus tard $\overline{s_t}$, qui est mémorisée par la date de son évènement CCP ou SCP, et puisque sweep_min n'ajuste que le début au plus tôt des tâches, la partie obligatoire de t ne peut pas débuter avant la date associé à cet évènement. Par conséquent, la dernière position δ de la droite de balayage pour savoir si une partie obligatoire apparait pour t est l'instant $\overline{s_t}$. Ce cas est traité par l'algorithme 5 aux lignes 11 à 16.

La fin de la partie obligatoire d'une tâche t correspond à sa date de fin au plus tôt $\underline{e_t}$ et est indiquée par son évènement $ECPD$. Pour traiter sa potentielle extension vers la droite, le début au plus tôt de la tâche t doit être ajusté à sa position finale avant que l'algorithme de balayage n'ait extrait son évènement $ECPD$. Ce cas est traité par l'algorithme 5 aux lignes 6 à 9.

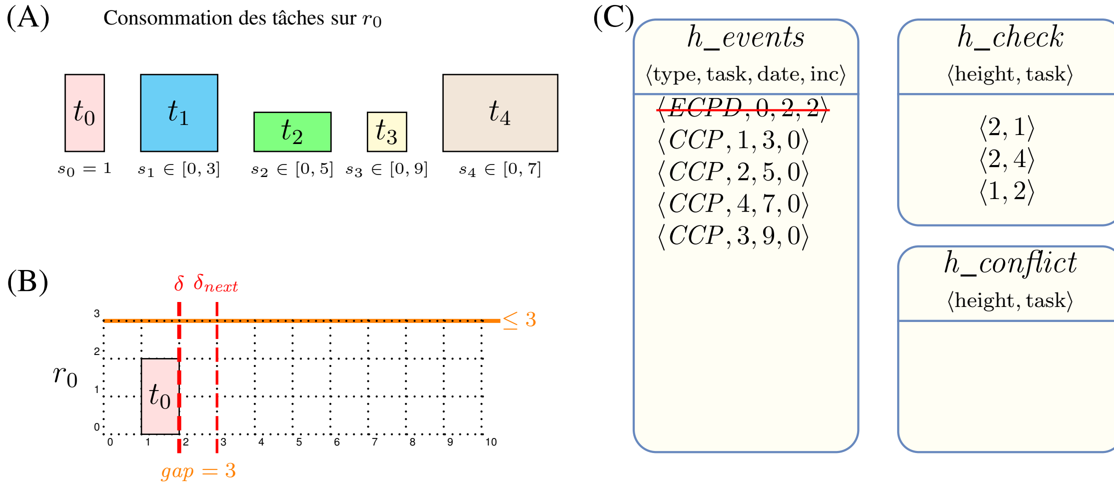


FIGURE 5.3 – État des tâches, de la droite de balayage et des tas après la lecture de l'évènement $\langle ECPD, 0, 2, 2\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 16) - (A) date de début et consommations de chaque tâche, (B) le PPO et la position de la droite de balayage, (C) la liste des évènements restant à traiter ainsi que les tas h_check et $h_conflict$.

Preuve de la propriété 5.1 Étant donné une tâche t, δ_t and min_t représentent respectivement la position de la droite de balayage au moment où le début au plus tôt de la tâche t est ajusté par sweep_min, et la nouvelle valeur du début au plus tôt de la tâche t. Nous montrons successivement les point suivants :

① Lorsque la droite de balayage est positionnée au point de temps δ_t on peut fixer le début de la tâche t à min_t sans dépasser la capacité maximale de la ressource $limit$, i.e.

$$\forall t' \in \mathcal{T}\backslash\{t\}, \forall i \in [min_t, \delta_t) : \; h_t + \sum_{\substack{t' \in \mathcal{T}\backslash\{t\},\\ i \in [\overline{s_{t'}}, \underline{e_{t'}})}} h_{t'} \leq limit$$

L'ajustement du début au plus tôt de la tâche t à la valeur min_t implique que t n'est pas en conflit sur l'intervalle $[min_t, \delta_t[$ par rapport au PPO. La condition get_top_key$(h_check) > gap$ (algorithme 4, ligne 4) garantit que l'ajustement fait à la ligne 7 n'induit pas un dépassement de la ressource sur l'intervalle $[min_t, \delta_t[$, sinon t aurait été ajoutée dans $h_conflict$. La condition get_top_key$(h_conflict) \leq gap$ (algorithme 4, ligne 12) implique que la tâche t est en conflit jusqu'à la position courante de la droite de balayage δ. Si $\delta \geq \overline{s_t}$ (ligne 14), le conflit sur l'intervalle $[\overline{s_t}, \delta_t)$ n'est pas « réel » puisque la partie obligatoire de la tâche t est déjà prise en compte dans le PPO. Enfin, à la ligne 19 de l'algorithme 4, le début au plus tôt de la tâche t est ajusté à la position courante de la droite de balayage, et donc l'intervalle $[min_t, \delta_t)$ est vide.

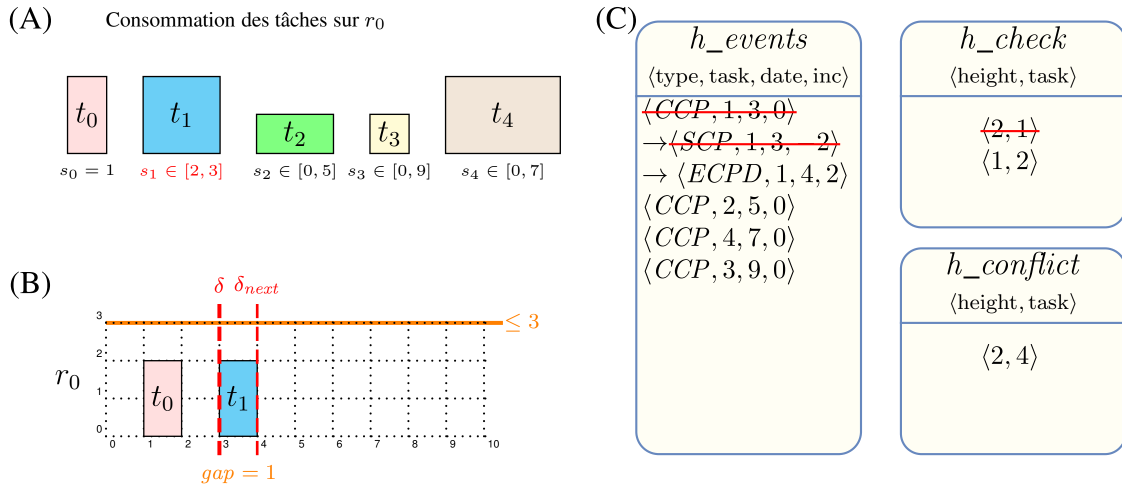


FIGURE 5.4 – État des tâches, de la droite de balayage et des tas après la lecture des évènements $\langle CCP, 1, 3, 0\rangle$ et $\langle SCP, 1, 3, -2\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 16) - (A) date de début et consommations de chaque tâche, (B) le PPO et la position de la droite de balayage, (C) la liste des évènements restant à traiter ainsi que les tas h_check et $h_conflict$.

② Pour chaque valeur de δ supérieure ou égale à δ_t, sweep_min ne peut pas créer de partie obligatoire avant le point de temps δ_t. Ceci est impliqué par le lemme 5.1, qui garantit que sweep_min ne peut pas créer de partie obligatoire avant δ.

Lorsque l'algorithme sweep_min est terminé, n'importe quelle tâche peut donc être fixée à sa date de début au plus tôt sans pour autant créer un PPO dépassant la capacité maximale de la ressource $limit$.

5.6 Complexité

Nous donnons maintenant la complexité temporelle dans le pire des cas de notre algorithme de balayage dynamique ainsi que sa preuve.

Théorème 5.1. *Étant donné une contrainte* cumulative *impliquant n tâches, la complexité temporelle dans le pire des cas de l'algorithme de balayage dynamique* sweep_min *est* $O(n^2 \log\ n)$.

Preuve du théorème 5.1 Notons tout d'abord que la complexité dans le pire des cas de l'algorithme synchronize lors d'une exécution complète du balayage est $O(n \log\ n)$ puisque les évènements dynamiques et conditionnels ne sont mis à jour qu'au plus une seule fois.

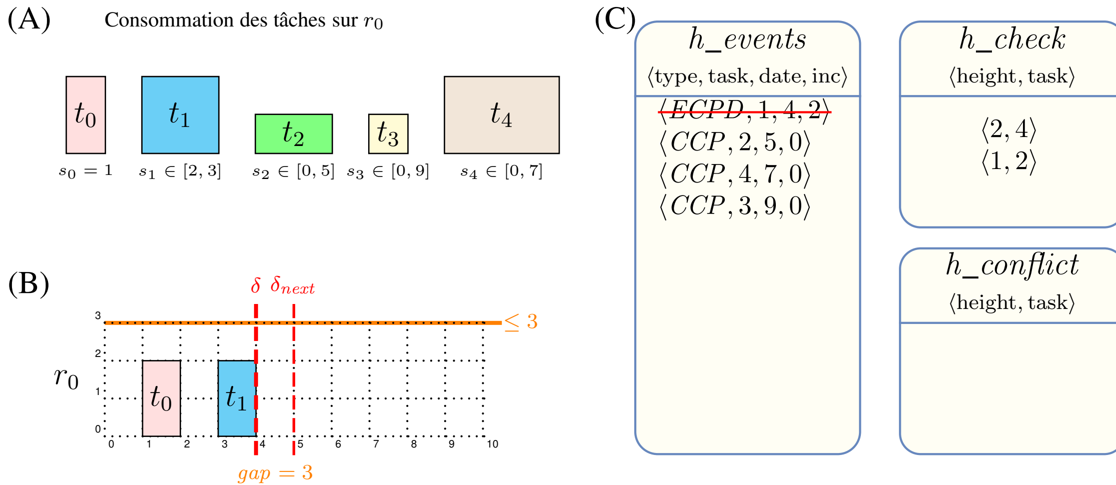


FIGURE 5.5 – État des tâches, de la droite de balayage et des tas après la lecture de l'évènement $\langle ECPD, 1, 4, 2\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 16) - (A) date de début et consommations de chaque tâche, (B) le PPO et la position de la droite de balayage, (C) la liste des évènements restant à traiter ainsi que les tas h_check et $h_conflict$.

Le pire des cas $O(n^2 \log\ n)$ peut être atteint dans le cas particulier où le PPO consiste en une succession de hauts pics et de vallées profondes et étroites. Supposons qu'il y ait $O(n)$ pics, $O(n)$ vallées, et $O(n)$ tâches à transférer entre les tas h_check et $h_conflict$ à chaque fois. Un ajout ou une insertion dans notre structure de tas coûte $O(n \log\ n)$. La complexité temporelle dans le pire des cas est donc $O(n^2 \log\ n)$. □

Dans le cas du bin-packing, les tas h_check et $h_conflict$ nous permettent de réduire la complexité à $O(n \log\ n)$. En effet, le début au plus tôt des tâches de durée un sortant de $h_conflict$ est directement ajusté, sans passer par h_check. Dans le cas du bin-packing le tas h_check n'est jamais utilisé.

6

Balayage synchrone pour plusieurs contraintes *cumulative* avec ou sans précédences

Sommaire

6.1 Une première approche sans contrainte de précédence 49
- 6.1.1 Types d'évènements 49
- 6.1.2 État de la droite de balayage 49
- 6.1.3 Algorithme de filtrage 50
- 6.1.4 Complexité 54

6.2 Une seconde approche intégrant les contraintes de précédence 55
- 6.2.1 Types d'évènements 58
- 6.2.2 État de la droite de balayage 59
- 6.2.3 Algorithme de filtrage 60
- 6.2.4 Complexité 67

6.3 Cumulative colorée : ou comment remplacer la somme par le nombre de valeurs distinctes 74
- 6.3.1 Définition de la contrainte *multiSumColorPrecCumulative* 74
- 6.3.2 Schémas classiques d'apparition de la contrainte *multiSumColor-PrecCumulative* 75
- 6.3.3 Reformulation quadratique de la contrainte cumulative colorée . . 77
- 6.3.4 Intégration au balayage synchrone avec précédences 78

Ce chapitre présente successivement deux nouveaux algorithmes de balayage synchrones qui traitent plusieurs ressources cumulatives en un unique balayage. Dans cette nouvelle configuration, chaque tâche consomme simultanément sur l'ensemble ou sur un sous-ensemble des ressources cumulatives. L'objectif principal est de proposer une approche permettant un passage à l'échelle aussi bien par rapport au nombre de tâches que par rapport au nombre de

ressources. Soulignons que le nombre de ressources peut être important dans de nombreuses situations :

- Le challenge Roadef 2012 [59] propose par exemple des instances de bin-packing issues d'un problème d'assignation de machine virtuelles sur un ensemble de machines physiques impliquant jusqu'à 12 ressources en parallèle.
- Une nouvelle ressource peut également être introduite pour modéliser le fait qu'un sous-ensemble des tâches est sujet à une contrainte *cumulative* ou *disjunctive*. Les tâches n'appartenant pas à ce sous-ensemble ont une consommation nulle sur cette nouvelle ressource. Nous retrouverons ce cas dans la section d'évaluation de cette thèse avec l'application industrielle [67] dans laquelle on a jusqu'à 8 ressources différentes. Puisque l'on peut potentiellement avoir un grand nombre de ce type de contrainte, cela peut conduire à un nombre élevé de ressources.

Avoir au sein d'un même problème plusieurs contraintes *cumulative* partageant systématiquement les mêmes tâches amène au problème d'efficacité décrit au chapitre 4, point ⑤ `[Trop local]`. Les deux nouveaux algorithmes de balayage synchrones que nous présentons dans ce chapitre sont des versions k dimensionnelles de la méthode de Timetabling faisant exactement le même filtrage que k instances de la version à une dimension rapportée au chapitre 5, en convergeant plus vite au point fixe.

Étant donné k ressources et n tâches, où chaque ressource r $(0 \leq r < k)$ est décrite par sa capacité maximale $limit_r$, et chaque tâche t $(0 \leq t < n)$ est décrite par sa date de début s_t, sa durée fixée d_t $(d_t \geq 0)$, sa date de fin e_t et ses consommations de ressource $h_{t,0}, \ldots, h_{t,k-1}$ $(h_{t,i} \geq 0, i \in [0, k-1])$ sur chacune des k ressources, la contrainte k*-dimensional cumulative* avec les deux arguments

- $\langle\langle s_0, d_0, e_0, \langle h_{0,0}, \ldots, h_{0,k-1}\rangle\rangle, \ldots, \langle s_{n-1}, d_{n-1}, e_{n-1}, \langle h_{n-1,0}, \ldots, h_{n-1,k-1}\rangle\rangle\rangle$,
- $\langle limit_0, \ldots, limit_{k-1}\rangle$

est vérifiée si et seulement si les conditions (6.1) et (6.2) sont vérifiées :

$$\forall t \in [0, n-1] \ : s_t + d_t = e_t \tag{6.1}$$

$$\forall r \in [0, k-1], \ \forall i \in \mathbb{Z} \ : \sum_{\substack{t \in [0,n-1], \\ i \in [s_t, e_t)}} h_{t,r} \leq limit_r \tag{6.2}$$

Afin d'illustrer notre propos nous étendons l'exemple 3.1 en ajoutant une deuxième ressource cumulative r_1.

Exemple 6.1. *Considérons deux ressources cumulatives* r_0, r_1 *(i.e.* $k = 2$*) dont les capacités maximales sont* $limit_0 = 3$ *et* $limit_1 = 2$ *ainsi que cinq tâches* t_0, t_1,…,t_4 *ayant les dates de début, les durées, les dates de fin et les consommations de ressource suivantes :*

- t_0 : $s_0 \in [1, 1]$, $d_0 = 1$, $e_0 \in [2, 2]$, $h_{0,0} = 2$, $h_{0,1} = 1$
- t_1 : $s_1 \in [0, 3]$, $d_1 = 2$, $e_1 \in [2, 5]$, $h_{1,0} = 2$, $h_{1,1} = 1$
- t_2 : $s_2 \in [0, 5]$, $d_2 = 2$, $e_2 \in [2, 7]$, $h_{2,0} = 1$, $h_{2,1} = 2$
- t_3 : $s_3 \in [0, 9]$, $d_3 = 1$, $e_3 \in [1, 10]$, $h_{3,0} = 1$, $h_{3,1} = 1$
- t_4 : $s_4 \in [0, 7]$, $d_4 = 3$, $e_4 \in [3, 10]$, $h_{4,0} = 2$, $h_{4,1} = 0$

L'application de la méthode de Timetabling sur cette instance est illustrée par la figure 6.1. Puisque la tâche t_1 ne peut pas se superposer à la tâche t_0 sans provoquer un dépassement de consommation sur la ressource r_0, le début au plus tôt de la tâche t_1 est ajusté à 2. Puisque la tâche t_1 possède maintenant une partie obligatoire sur l'intervalle $[3, 4[$ et comme, sur la ressource r_0, la tâche t_4 ne peut pas se superposer à t_1, le début au plus tôt de la tâche t_4 est ajusté à 4. Sur la ressource r_1, comme la tâche t_2 ne peut pas se superposer à la tâche t_1, son début au plus tôt est ajusté à 4. L'état atteint au point fixe de cet exemple est illustré par la partie (B) de la figure 6.1.

L'objectif des nouveaux algorithmes de balayage synchrones de ce chapitre est d'effectuer précisément ce filtrage de manière efficace, i.e. en un seul balayage.

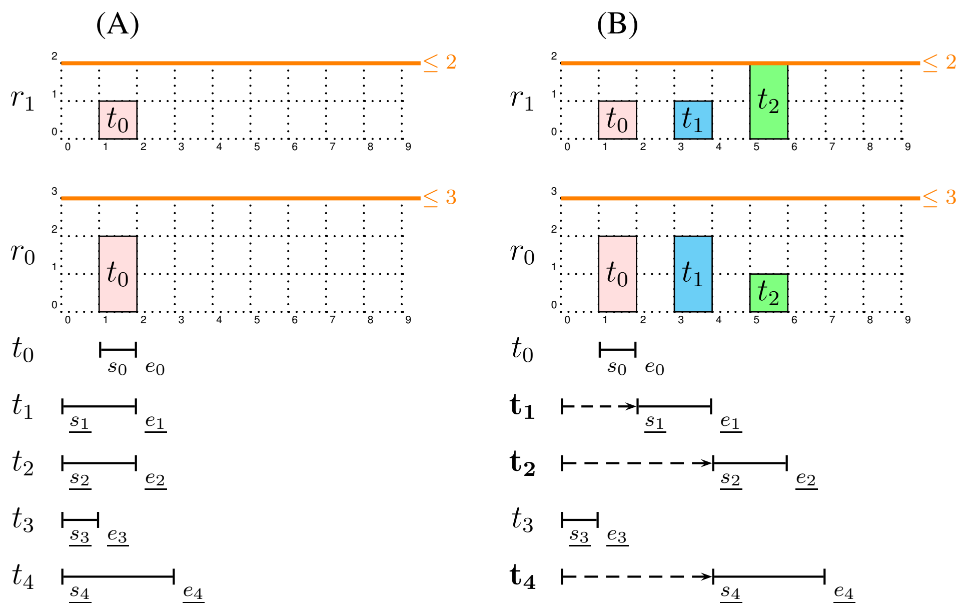


FIGURE 6.1 – Les parties (A) et (B) représentent respectivement la position au plus tôt de chacune des tâches ainsi que le PPO sur les ressources r_0 et r_1, (A) du problème initial décrit dans l'exemple 6.1, (B) une fois le point fixe atteint.

Nous montrons maintenant dans l'exemple 6.2 comment la décomposition de cet exemple en une conjonction de deux contraintes *cumulative*, mène à un ping-pong entre ces deux contraintes pour atteindre le point fixe.

Exemple 6.2. *L'instance donnée dans l'exemple 6.1 peut être naturellement décomposée avec les deux contraintes* cumulative *suivantes :*

- c_0 : $cumulative(\langle \langle s_0, d_0, e_0, h_{0,0}\rangle, \langle s_1, d_1, e_1, h_{1,0}\rangle, \langle s_2, d_2, e_2, h_{2,0}\rangle, \langle s_3, d_3, e_3, h_{3,0}\rangle, \langle s_4, d_4, e_4, h_{4,0}\rangle\rangle, limit_0)$,
- c_1 : $cumulative(\langle \langle s_0, d_0, e_0, h_{0,1}\rangle, \langle s_1, d_1, e_1, h_{1,1}\rangle, \langle s_2, d_2, e_2, h_{2,1}\rangle, \langle s_3, d_3, e_3, h_{3,1}\rangle\rangle, limit_1)$.

Notons que la tâche t_4 n'apparait pas dans la contrainte c_1 puisque sa hauteur sur la ressource r_1 est nulle.

- *Lors du le premier balayage par rapport à la contrainte c_0 (partie (A) de la figure 6.2), la partie obligatoire de la tâche t_0 sur la ressource r_0 sur l'intervalle $[1, 2[$ permet d'ajuster le début au plus tôt de la tâche t_1 à 2 puisque l'espace disponible est strictement inférieur à la consommation de ressource de t_1 sur r_0. La tâche t_1 a désormais une partie obligatoire sur l'intervalle $[3, 4[$. Cette nouvelle partie obligatoire permet d'ajuster le début au plus tôt de la tâche t_4 à 4.*

- *Un deuxième balayage par rapport à la contrainte c_1 (partie (B) de la figure 6.2) ajuste la date de début au plus tôt de la tâche t_2 car elle ne peut se superposer ni à la tâche t_0, ni à la partie obligatoire de la tâche t_1. La tâche t_2 possède maintenant une partie obligatoire sur l'intervalle $[5, 6[$.*

- *Pour terminer, un troisième balayage par rapport à la contrainte c_0 est effectué pour découvrir que rien de plus ne peut être déduit grâce aux parties obligatoires, signifiant que le point fixe est atteint.*

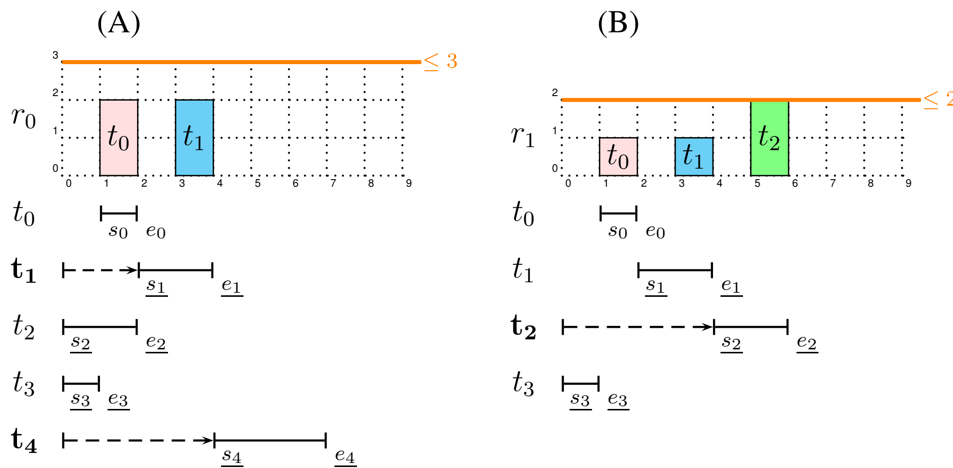


FIGURE 6.2 – Les parties (A) et (B) représentent respectivement la position au plus tôt de chacune des tâches et le PPO, (A) après un premier balayage sur la ressource r_0, (B) après un second balayage sur la ressource r_1.

Les deux algorithmes de balayage que nous proposons dans ce chapitre atteignent ce même point fixe en une seule étape. Précisons que le deuxième algorithme étend le premier puisqu'il est capable de gérer un ensemble de contraintes de précédence en plus des différentes ressources. Donnons maintenant la propriété vérifiée au point fixe par l'algorithme sweep_min page 51, traitant la contrainte k*-dimensional cumulative*.

Propriété 6.1. *Étant donné une contrainte* k-dimensional cumulative *impliquant n tâches et k ressources, l'algorithme* sweep_min *page 51 garantit :*

$$\forall r \in [0, k-1], \forall t \in [0, n-1], \forall i \in [\underline{s_t}, \underline{e_t}) \ : \ h_{t,r} + \sum_{\substack{t' \neq t, \\ i \in [\overline{s_{t'}}, \underline{e_{t'}})}} h_{t',r} \leq limit_r \tag{6.3}$$

La propriété 6.1 vérifie que pour chaque tâche t de la contrainte k-*dimensional cumulative*, on peut fixer t à sa date de début au plus tôt, sans dépasser la capacité maximale de chacune des ressources r $(0 \leq r < k)$ en ne considérant que les différents PPO des tâches $\mathcal{T}\backslash\{t\}$.

6.1 Une première approche sans contrainte de précédence

Nous présentons dans cette section un algorithme de filtrage synchrone pour la contrainte k-*dimensional cumulative* vérifiant la propriété 6.1. Comme pour la description de l'algorithme de balayage dynamique du chapitre 5, nous commençons par la série d'évènements puis nous détaillons les données attachées à la droite de balayage. Enfin, après avoir présenté l'algorithme nous en donnons sa complexité.

6.1.1 Types d'évènements

Dans le contexte des algorithmes de balayage, les évènements sont des points de temps spécifiques où l'état de la droite de balayage est modifié. Comme les évènements ne sont liés qu'à l'aspect temporel et qu'ils ne dépendent pas du nombre de ressources présentes dans le problème, nous pouvons donc les factoriser par rapport à l'ensemble des ressources.

Le quatrième attribut d'un évènement tel qu'il est décrit au chapitre 5 page 32 enregistre la modification de l'espace disponible à appliquer lorsque l'évènement est traité. Dans notre nouvelle configuration, i.e. le multi-ressources, nous faisons le choix de le supprimer plutôt que de l'étendre à une liste de hauteurs correspondant à la consommation de la tâche sur chacune des ressources. Lorsqu'un évènement de partie obligatoire d'une tâche t sera lu, nous accèderons directement aux hauteurs de la tâche via son attribut $h_{t,r}$ $(0 \leq r < k)$.

Exemple 6.3. (Évènements générés). *Pour l'instance introduite dans l'exemple 6.1 page 46, les évènements générés et insérés dans le tas des évènements* h_events *sont les mêmes que pour la version mono-dimensionnelle de l'exemple 5.1, à la différence près qu'ils n'ont plus maintenant que trois attributs :* $\langle PR, 1, 0\rangle$, $\langle PR, 2, 0\rangle$, $\langle PR, 3, 0\rangle$, $\langle PR, 4, 0\rangle$, $\langle SCP, 0, 1\rangle$, $\langle ECPD, 0, 2\rangle$, $\langle CCP, 1, 3\rangle$, $\langle CCP, 2, 5\rangle$, $\langle CCP, 4, 7\rangle$, $\langle CCP, 3, 9\rangle$.

6.1.2 État de la droite de balayage

Pour construire le PPO de chaque ressource et ajuster le début au plus tôt de chaque tâche, la droite de balayage avance d'évènement en évènement (par date d'évènement croissante) et maintient les informations suivantes :

- La position courante δ de la droite de balayage, initialement postionnée à la date du premier évènement.

- Pour chaque ressource $r \in [0, k-1]$, la quantité de ressource disponible au point de temps δ est donnée par gap_r (i.e. la différence entre la capacité maximale de la ressource $limit_r$ et la hauteur du PPO sur la ressource r au point de temps δ) et sa valeur précédente par gap'_r. Comme nous le verrons par la suite, conserver la valeur précédente à gap_r nous permet de ne parcourir qu'un sous ensemble des tâches lors du filtrage.

- Pour chaque ressource $r \in [0, k-1]$, un tableau a_check_r qui enregistre les tâches qui ont été rencontrées par la droite de balayage mais dont le début au plus tôt n'a pas encore été déterminé, i.e. toutes les tâches qui peuvent potentiellement couper la position courante de la droite de balayage. Ces tâches seront par la suite appelées *tâches actives*. La $i^{ème}$ entrée de ce tableau contient le sous-ensemble des tâches actives t pour lesquelles $h_{t,r} = i$. Cette structure de données nous permet d'accéder directement aux tâches actives ayant une hauteur donnée sur une ressource donnée.

- Pour chaque tâche t, le nombre de conflits au point de temps δ est stocké dans $conflicts_t$. On dit qu'une tâche est *en conflit par rapport à une ressource* r si et seulement si $t \in a_check_r \wedge h_{t,r} > gap_r$. Une tâche t ne peut pas couper la position courante de la droite de balayage si le nombre de conflits associé à cette tâche est supérieur à 0.

Comme pour la version mono-dimensionnelle, l'algorithme commence par créer et insérer les évènements dans le tas h_events. Puis, la droite de balayage se déplace d'un évènement à un autre et met à jour la quantité de ressource disponible sur chaque ressource (i.e. gap_r, $0 \leq r < k$) ainsi que la liste des tâches actives. Une fois que tous les évènements associés à la date δ ont été traités, la droite de balayage essaye d'ajuster le début au plus tôt des tâches actives par rapport à l'espace disponible sur chacune des ressources sur l'intervalle $[\delta, \delta_{next}[$. Pour mettre à jour le nombre de conflits des tâches, pour chaque ressource r l'algorithme parcourt les tâches présentes dans a_check_r et pour lesquelles la hauteur sur r est comprise entre gap_r et gap'_r, i.e. les tâches qui viennent juste de passer en conflit sur la ressource r, ou les tâches qui ne sont plus en conflit sur la ressource r.

En plus des informations maintenues par la droite de balayage nous définissons deux fonctions $\text{pred}_r : \mathbb{N} \rightarrow \mathbb{N}$ et $\text{succ}_r : \mathbb{N} \rightarrow \mathbb{N}$ qui nous permettent de parcourir de manière efficace les tâches ayant une hauteur comprise entre gap_r et gap'_r ($0 \leq r < k$) :

$$\text{pred}_r(h) = \begin{cases} \max_{0 \leq t < n}\{h_{t,r} \mid h_{t,r} < h\} & \text{, si un tel } h_{t,r} \text{ existe} \\ 0 & \text{, sinon} \end{cases}$$

$$\text{succ}_r(h) = \begin{cases} \min_{0 \leq t < n}\{h_{t,r} \mid h_{t,r} > h\} & \text{, si un tel } h_{t,r} \text{ existe} \\ +\infty & \text{, sinon} \end{cases}$$

6.1.3 Algorithme de filtrage

Tout comme la version mono-dimensionnelle, sweep_min page 51 effectue un unique balayage de gauche à droite pour ajuster le début au plus tôt des tâches et vérifier la propriété 6.1. Il consiste principalement en une boucle composée d'une partie dédiée au traitement des évènements et d'une autre partie dédiée au filtrage. La partie dédiée au traitement des évènements enregistre les nouvelles tâches actives, met à jour la ressource disponible à la position courante de la droite de balayage sur chacune des ressources et gère la création et l'extension des parties obligatoires. La partie dédiée au filtrage ajuste le début au plus tôt des tâches sur l'intervalle $[\delta, \delta_{next}[$ par rapport à l'espace disponible sur chacune des ressources.

Algorithme principal

L'algorithme sweep_min page 51 est composé des parties suivantes :

- `[CRÉATION DES ÉVÈNEMENTS]` (ligne 2). Les évènements sont générés et insérés dans le tas h_events selon les conditions spécifiées dans la table 5.1.

ALGORITHM sweep_min() : boolean
1: `[CRÉATION DES ÉVÈNEMENTS]`
2: $h_events \leftarrow$ generation of events wrt n, $\underline{s_t}$, $\overline{s_t}$, d_t, $\underline{e_t}$ and Table 5.1.
3: `[INITIALISATION]`
4: **for** $r = 0$ **to** $k - 1$ **do**
5: $\quad gap_r \leftarrow limit_r$
6: $\quad gap'_r \leftarrow \text{pred}_r(gap_r + 1)$
7: $\quad a_check_r \leftarrow$ empty array of size equal to the number of distinct heights on resource r

8: `[BOUCLE PRINCIPALE]`
9: **while** $\neg\text{empty}(h_events)$ **do**
10: $\quad \langle \delta, \delta_{next} \rangle \leftarrow$ process_events()
11: $\quad$ **if** $\neg\text{filter_min}(\delta, \delta_{next})$ **then return false**
12: **return true**

Algorithme 6: Retourne **false** si et seulement si un dépassement de la capacité d'une ressource est détecté, **true** sinon.

- `[INITIALISATION]` (lignes 4 à 7). Les tableaux a_check_r ($0 \leq r < k$) sont initialisés à vide. gap_r et gap'_r sont respectivement initialisés à la capacité maximale de la ressource r et à la hauteur maximale consommée par une tâche sur la ressource r.

- `[BOUCLE PRINCIPALE]` (lignes 9 à 11). Pour chaque date, la boucle principale traite tous les évènements correspondant à la date en question. La boucle principale est composée des deux étapes suivantes :

  – La première étape (ligne 10) traite d'abord tous les évènements de h_events dont la date est égale à la position courante δ de la droite de balayage, et détermine ensuite sa prochaine position δ_{next}.

  – La deuxième étape (ligne 11) détecte si un dépassement de ressource a lieu et traite les tâches actives dans le but d'ajuster leur début au plus tôt par rapport aux parties obligatoires sur l'intervalle $[\delta, \delta_{next}[$.

Traitement des évènements

Pour mettre à jour l'état de droite de balayage, l'algorithme process_events page 53 lit et traite tous les évènements relatifs à la position courante δ de la droite de balayage et détermine l'intervalle $[\delta, \delta_{next}[$, dans lequel δ_{next} désigne la plus petite date strictement supérieure à δ d'un évènement pas encore traité. Cet algorithme est composé des parties suivantes :

- `[TRAITER LES ÉVÈNEMENTS DE DÉBUT DE PARTIE OBLIGATOIRE (SCP)]` (lignes 3 à 4). Pour chaque ressource r l'espace disponible gap_r, sur l'intervalle de balayage courant est mis à jour.

- `[TRAITER LES ÉVÈNEMENTS DYNAMIQUES (ECPD)]` (lignes 6 à 9). Lorsque la droite de balayage atteint la fin de partie obligatoire d'une tâche t, i.e. sa date de fin au plus tôt, on doit déterminer si oui ou non il s'agit de sa position finale, i.e. si oui ou non il est encore possible d'ajuster le début au plus tôt de t. Cela requièrt de considérer les cas suivants :

  - Si la tâche t est encore en conflit sur au moins une ressource (i.e. $conflicts_t > 0$, ligne 7), alors le début au plus tôt de la tâche t ne peut pas être ajustée autre part qu'à sa position au plus tard.
  - Si le début au plus tôt de la tâche t a déjà été ajusté depuis la création de l'évènement, alors cet évènement $ECPD$ est repoussé à sa bonne date (i.e. $\underline{e_t}$) dans le tas h_events (ligne 8). Sinon, pour chaque ressource r l'espace disponible gap_r est mis à jour.

- `[TRAITER LES ÉVÈNEMENTS CONDITIONNELS (CCP)]` (lignes 11 à 15). Lorsque la droite de balayage atteint le début au plus tard d'une tâche t initialement sans partie obligatoire, il est nécessaire de savoir si une partie obligatoire est apparue. Pour cela on considère les cas suivants :

  - Si la tâche t est en conflit sur au moins une ressource, alors t ne peut pas débuter avant sa position au plus tard (ligne 12).
  - Si le début au plus tôt de la tâche t a déjà été mis à jour et que sa fin au plus tôt est supérieure à la position courante de la droite de balayage, signifiant qu'une partie obligatoire a été créée, alors l'évènement relatif à la fin de partie obligatoire de t est ajouté dans h_events (ligne 14) et les espaces disponibles gap_r $(0 \leq r < k)$ sont mis à jour afin de prendre directement en compte cette création de partie obligatoire.

- `[DÉTERMINER LA DATE DU PROCHAIN ÉVÈNEMENT]` (ligne 17). Pour procéder au traitement des évènements du type PR il est nécessaire de connaitre la prochaine position δ_{next} de la droite de balayage.

- `[TRAITER LES ÉVÈNEMENTS DE DÉBUT AU PLUS TÔT (PR)]` (lignes 19 à 23). Si la taille de l'intervalle de balayage courant est trop petit par rapport à la durée de la tâche t ou s'il existe au moins un conflit (i.e. $d_t > \delta_{next} - \delta \vee \exists r \mid h_{t,r} > gap_r$, ligne 20), alors, pour chaque ressource r $(0 \leq r < k)$ la tâche t est ajoutée dans a_check_r et son nombre de conflits est mis à jour.

Filtrage

L'algorithme filter_min page 54 traite les tâches actives, i.e. les tâches présentes dans a_check_r $(0 \leq r < k)$, pour ajuster leur début au plus tôt. Il est composé des parties suivantes :

- `[VÉRIFIER LE DÉPASSEMENT DE RESSOURCE]` (ligne 2). Si l'un des espaces disponibles gap_r $(0 \leq r < k)$ est négatif sur l'intervalle de balayage courant, l'algorithme 8 retourne **false** indiquant ainsi un échec de la contrainte.

- `[TÂCHES ENTRANT EN CONFLIT]` (lignes 4 à 13). On parcourt chaque ressource pour laquelle l'espace disponible est strictement inférieur à l'espace disponible à la position précédente de la droite de balayage (i.e $gap'_r > gap_r$, ligne 5). On considère chaque tâche $t \in a_check_r$ telle que $gap'_r \geq h_{t,r} > gap_r$, i.e. les tâches qui n'étaient pas en conflit à la position précédente de la droite de balayage mais qui le sont devenues à la position courante δ, par rapport à la ressource r. Si le début au plus tôt d'une tâche t sans partie obligatoire est valide, i.e. $conflicts_t = 0 \wedge \delta \geq \underline{e_t}$, alors, la tâche t est retirée de a_check_r. Pour une tâche t avec une partie obligatoire, il n'est

ALGORITHM process_events() : $\langle \delta, \delta_{next} \rangle$
1: $\langle \delta, \mathcal{E} \rangle \leftarrow$ extract_min(h_events)
2: [TRAITER LES ÉVÈNEMENTS DE DÉBUT DE PARTIE OBLIG. (SCP)]
3: **for all** tasks t that belong to an event of type SCP in $\mathcal{E}$ **do**
4: **for** $r = 0$ **to** $k - 1$ **do** $gap_r \leftarrow gap_r - h_{t,r}$
5: [TRAITER LES ÉVÈNEMENTS DYNAMIQUES (ECPD)]
6: **for all** tasks t that belong to an event of type ECPD in $\mathcal{E}$ **do**
7: **if** $conflicts_t > 0$ **then** $(\underline{s_t}, \underline{e_t}) \leftarrow (\overline{s_t}, \overline{e_t})$
8: **if** $\underline{e_t} > \delta$ **then** add $\langle \text{ECPD}, t, \underline{e_t} \rangle$ to h_events
9: **else for** $r = 0$ **to** $k - 1$ **do** $gap_r \leftarrow gap_r + h_{t,r}$
10: [TRAITER LES ÉVÈNEMENTS CONDITIONNELS (CCP)]
11: **for all** tasks t that belong to an event of type CCP in $\mathcal{E}$ **do**
12: **if** $conflicts_t > 0$ **then** $(\underline{s_t}, \underline{e_t}) \leftarrow (\overline{s_t}, \overline{e_t})$
13: **if** $\underline{e_t} > \delta$ **then**
14: add $\langle \text{ECPD}, t, \underline{e_t} \rangle$ to h_events
15: **for** $r = 0$ **to** $k - 1$ **do** $gap_r \leftarrow gap_r - h_{t,r}$
16: [DÉTERMINER LA DATE DU PROCHAIN ÉVÈNEMENT]
17: $\delta_{next} \leftarrow$ peek_key(h_events)
18: [TRAITER LES ÉVÈNEMENTS DE DÉBUT AU PLUS TÔT (PR)]
19: **for all** tasks t that belong to an event of type PR in $\mathcal{E}$ **do**
20: **if** $d_t > \delta_{next} - \delta \vee \exists r$ such that $h_{t,r} > gap_r$ **then**
21: **for** $r = 0$ **to** $k - 1$ **do**
22: $a_check_r[h_{t,r}] \leftarrow a_check_r[h_{t,r}] \cup \{t\}$
23: **if** $h_{t,r} > gap'_r$ **then** $conflicts_t \leftarrow conflicts_t + 1$
24: **return** $\langle \delta, \delta_{next} \rangle$

Algorithme 7: Appelé à chaque fois que la droite de balayage se déplace. Extrait et traite tous les évènements au point de temps courant δ, et retourne la position courante δ ainsi que la position future δ_{next} de la droite de balayage.

pas nécessaire de la conserver dans a_check_r une fois que la droite de balayage a atteint le début de sa partie obligatoire. En effet, sa contribution au consommation sera directement comptabilisée dans le PPO grâce à ses évènements SCP et $ECPD$.

- [TÂCHES SORTANT DE CONFLIT] (lignes 15 à 29). On parcourt chaque ressource dont l'espace disponible est strictement supérieur à l'espace disponible à la position précédente de la droite de balayage (i.e. $\text{succ}_r(gap'_r) < gap_r$, ligne 16). Pour chaque ressource r, on parcourt les tâches $t \in a_check_r$ telles que $gap'_r < h_{t,r} \leq gap_r$, i.e. les tâches qui étaient en conflit à la position précédente de la droite de balayage mais qui ne le sont plus à la position courante δ, par rapport à r. On distingue les trois cas suivants :
  - Si δ a dépassé le début au plus tard de la tâche t (i.e. $\delta \geq \overline{s_t}$, ligne 22) cela signifie qu'il n'existe aucune autre position possible pour t que sa position au plus tard.
  - Sinon, si l'intervalle de balayage $[\delta, \delta_{next}[$ est suffisamment grand par rapport à la durée de la tâche et s'il n'y a pas de conflit sur les autres ressources (i.e. $conflicts_t = 0 \wedge \delta_{next} - \delta \geq d_t$, ligne 24), le début au plus tôt de la tâche t est ajustée à δ. Notons que cette condition ne dépend pas de la ressource r.
  - Sinon (ligne 26), le début au plus tôt de la tâche est ajusté à δ et la tâche est laissée dans a_check_r pour vérifier que cette nouvelle position est valide sur

toute sa durée.

```
ALGORITHM filter_min(δ, δ_next) : boolean
 1: [VÉRIFIER LE DÉPASSEMENT DE RESSOURCE]
 2: for r = 0 to k − 1 do if gap_r < 0 then return false
 3: [TÂCHES ENTRANT EN CONFLIT]
 4: for r = 0 to k − 1 do
 5:    while gap'_r > gap_r do
 6:       T ← a_check_r[gap'_r]
 7:       U ← ∅
 8:       for all t ∈ T do
 9:          if ¬(conflicts_t = 0 ∧ (s̄_t ≤ δ ∨ e_t ≤ δ)) then
10:             conflicts_t ← conflicts_t + 1
11:             U ← U ∪ {t}
12:       a_check_r[gap'_r] ← U
13:       gap'_r ← pred_r(gap'_r)
14: [TÂCHES SORTANT DE CONFLIT]
15: for r = 0 to k − 1 do
16:    while succ_r(gap'_r) ≤ gap_r do
17:       gap'_r ← succ_r(gap'_r)
18:       T ← a_check_r[gap'_r]
19:       U ← ∅
20:       for all t ∈ T do
21:          conflicts_t ← conflicts_t − 1
22:          if δ ≥ s̄_t then
23:             (s_t, e_t) ← (s̄_t, ē_t)
24:          else if conflicts_t = 0 ∧ δ_next − δ ≥ d_t then
25:             (s_t, e_t) ← (δ, δ + d_t)
26:          else
27:             (s_t, e_t) ← (δ, δ + d_t)
28:             U ← U ∪ {t}
29:       a_check_r[gap'_r] ← U
30: return true
```

Algorithme 8: Appelé à chaque fois que la droite de balayage se déplace de la position δ à la position δ_{next} pour ajuster le début au plus tôt des tâches par rapport à l'espace disponible sur chacune des ressources.

6.1.4 Complexité

Nous présentons maintenant la complexité temporelle dans le pire des cas de cet algorithme de balayage synchrone pour une contrainte impliquant n tâches et k ressources. Pour cela, nous introduisons tout d'abord le lemme suivant.

Lemme 6.1. *Durant une exécution complète de l'algorithme 6 page 51 au plus quatre évènements par tâche sont générés.*

Preuve du lemme 6.1 Initialement, au plus trois évènements sont générés par tâche. Au plus, un seul évènement dynamique $ECPD$ par tâche peut être généré pendant le balayage

comme nous le montrons ci-après. On a deux cas possibles, dans le premier, la tâche dispose d'une partie obligatoire, dans le second, elle n'en possède pas.

① Supposons que la tâche t ait initialement un évènement $ECPD$, que la ligne 8 de l'algorithme 7 soit atteinte et que sa condition soit évaluée à **true**. Un évènement $ECPD$ est alors généré et ajouté dans h_events. D'après la ligne 7, on a $\underline{e_t} = \overline{e_t}$ ou $conflicts_t = 0$.

② Supposons que la tâche t ait initialement un évènement CCP, que la ligne 13 de l'algorithme 7 soit atteinte et que sa condition soit évaluée à **true**. Un évènement $ECPD$ est alors généré et ajouté dans h_events. D'après la ligne 12, on a $\underline{e_t} = \overline{e_t}$ ou $conflicts_t = 0$.

Dans le cas où $\underline{e_t} = \overline{e_t}$ alors $\underline{e_t}$ ne peut en aucun cas être mis à jour une nouvelle fois et il n'est plus possible de créer un second évènement dynamique $ECPD$. Supposons maintenant que $\underline{e_t} < \overline{e_t}$. Alors l'égalité $conflicts_t = 0$ doit être vérifiée. L'unique possibilité pour ajuster $\underline{e_t}$ avant de traiter l'évènement dynamique $ECPD$ est d'atteindre la ligne 20 de l'algorithme 8. Mais pour cela, il faut d'abord atteindre la ligne 9 de l'algorithme 8. La première fois que l'on atteint cette ligne, la condition $\overline{s_t} \leq \delta$ est vérifiée, sinon nous n'aurions pas rencontré l'évènement $CCP/ECPD$. Par conséquent la condition ligne 9 sera fausse, la tâche t sera retirée du tableau et la ligne 20 ne sera jamais atteinte. Dans tous les cas, au plus un évènement dynamique $ECPD$ est généré. □

Théorème 6.1. *Étant donné une contrainte k-*dimensional cumulative *impliquant n tâches et k ressources, la complexité temporelle dans le pire des cas de l'algorithme de balayage synchrone* sweep_min *est $O(kn^2)$.*

Preuve du théorème 6.1 Le traitement d'un évènement par l'algorithme process_events coûte $O(k + \log n)$. Chaque position de la droite de balayage appelle l'algorithme filter_min dont la complexité temporelle dans le pire des cas est $O(kn)$. D'après le lemme 6.1 on génère $O(n)$ évènements, la complexité temporelle dans le pire des cas de l'algorithme sweep_min est donc $O(kn^2)$. □

6.2 Une seconde approche intégrant les contraintes de précédence

Nous présentons dans cette section une version revisité de l'algorithme précédent, et nous l'étendons de manière à prendre en compte un ensemble de contraintes de précédence entre les tâches, pendant le balayage. Dans ce contexte, une contrainte de précédence entre une tâche t et une tâche t' signifie que t doit être terminée avant que t' ne puisse commencer, i.e. $s_t + d_t \leq e'_t$. Notre objectif ici est de fournir un algorithme passant à l'échelle même lorsque le nombre de contraintes de précédence est grand, ce qui est généralement une source d'inefficacité dans les solveurs de contraintes (point ⑤ du chapitre 4), tout du moins lorsque l'ordre de réveil des contraintes de précédence n'est pas contrôlé finement par le noyau réveillant les contraintes [39].

Étant donné k ressources et n tâches, où chaque ressource r ($0 \leq r < k$) est décrite par sa capacité maximale $limit_r$, où chaque tâche t ($0 \leq t < n$) possède une liste de successeurs

S_t et est décrite par sa date de début s_t, sa durée fixée d_t ($d_t \geq 0$), sa date de fin e_t et ses consommations de ressource $h_{t,0}, \ldots, h_{t,k-1}$ ($h_{t,i} \geq 0$, $i \in [0, k-1]$) sur les k ressources, la contrainte *k-dimensional cumulative with precedences* avec les trois arguments

- $\langle\langle s_0, d_0, e_0, \langle h_{0,0}, \ldots, h_{0,k-1}\rangle\rangle, \ldots, \langle s_{n-1}, d_{n-1}, e_{n-1}, \langle h_{n-1,0}, \ldots, h_{n-1,k-1}\rangle\rangle\rangle$,
- $\langle limit_0, \ldots, limit_{k-1}\rangle$
- $\langle S_0, \ldots, S_{n-1}\rangle$

est vérifiée si et seulement si les conditions (6.4), (6.5) et (6.6) sont respectées :

$$\forall t \in [0, n-1] \ : s_t + d_t = e_t \tag{6.4}$$

$$\forall r \in [0, k-1], \ \forall i \in \mathbb{Z} \ : \sum_{\substack{t \in [0,n-1], \\ i \in [s_t, e_t)}} h_{t,r} \leq limit_r \tag{6.5}$$

$$\forall t \in [0, n-1], \ \forall t' \in S_t \ : e_t \leq s_{t'} \tag{6.6}$$

Notons que le graphe de précédences est supposé acyclique et les durées des tâches non nulles.

Exemple 6.4. *Considérons deux ressources cumulatives r_0, r_1 ($k = 2$) dont les capacités maximales respectives sont $limit_0 = 3$ et $limit_1 = 2$ ainsi que cinq tâches t_0, t_1,…, t_4 ayant les dates de début, les durées, les dates de fin et les consommations de ressource suivantes :*

- t_0 : $s_0 \in [1, 1]$, $d_0 = 1$, $e_0 \in [2, 2]$, $h_{0,0} = 2$, $h_{0,1} = 1$
- t_1 : $s_1 \in [0, 3]$, $d_1 = 2$, $e_1 \in [2, 5]$, $h_{1,0} = 2$, $h_{1,1} = 1$
- t_2 : $s_2 \in [0, 5]$, $d_2 = 2$, $e_2 \in [2, 7]$, $h_{2,0} = 1$, $h_{2,1} = 2$
- t_3 : $s_3 \in [0, 9]$, $d_3 = 1$, $e_3 \in [1, 10]$, $h_{3,0} = 1$, $h_{3,1} = 1$
- t_4 : $s_4 \in [0, 7]$, $d_4 = 3$, $e_4 \in [3, 10]$, $h_{4,0} = 2$, $h_{4,1} = 0$

Considérons également l'ensemble des contraintes de précédence suivantes :

- $e_0 \leq s_3$, *signifiant que la tâche t_0 doit être terminée avant que t_3 ne débute,*
- $e_1 \leq s_3$, *signifiant que la tâche t_1 doit être terminée avant que t_3 ne débute,*
- $e_2 \leq s_4$, *signifiant que la tâche t_2 doit être terminée avant que t_4 ne débute.*

Cette instance est illustrée par la partie (A) de la figure 6.3. Dans un premier temps, si nous ignorons les contraintes de précédence nous avons exactement la même instance que dans l'exemple 6.1, et nous pouvons donc faire les mêmes déductions, i.e. à cause des capacités maximales des ressources r_0 et r_1, le début au plus tôt de la tâche t_1 est ajusté à 2, les débuts au plus tôt de t_2 et t_4 sont ajustés à 4 (partie (B) de la figure 6.1). Dans un second temps, si nous considérons aussi les contraintes de précédence, nous pouvons effectuer les ajustements supplémentaires suivants :

- *le début au plus tôt de la tâche t_3 est ajusté à 4 puisque la fin au plus tôt de t_1 est égale à 4,*

- *le début au plus tôt de la tâche* t_4 *est ajusté à* 6 *puisque la fin au plus tôt de* t_2 *est égale à* 6 *(partie (B) de la figure 6.3).*

L'objectif du balayage synchrone étendu pour les contraintes de précédence est d'effectuer ce filtrage en un seul balayage.

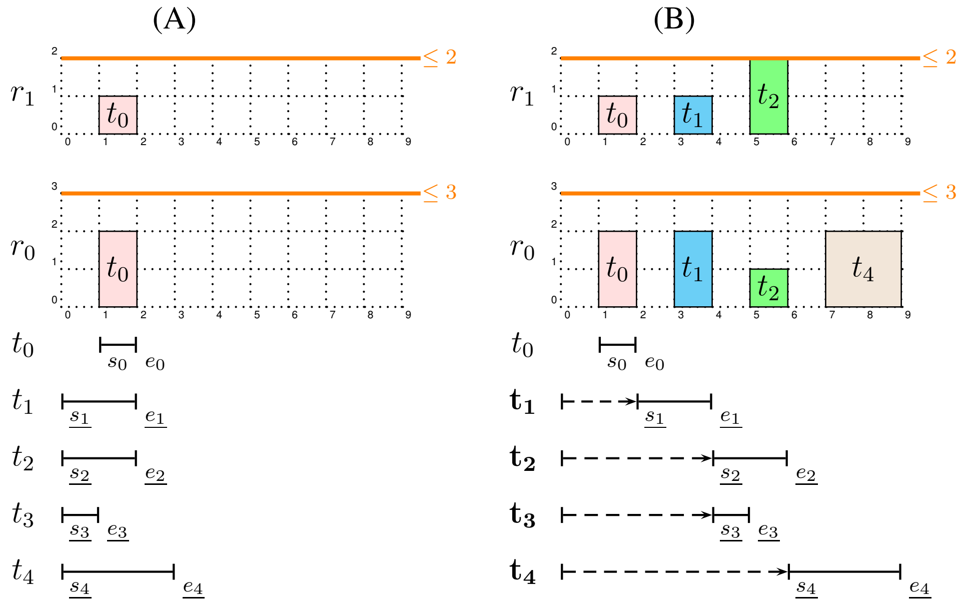


FIGURE 6.3 – Les parties (A) et (B) représentent respectivement la position au plus tôt de chacune des tâches ainsi que le PPO des ressources r_0 et r_1, (A) du problème initial décrit dans l'exemple 6.4, (B) une fois le point fixe atteint.

Montrons maintenant dans l'exemple 6.5 comment la décomposition de la contrainte 2-*dimensional cumulative with precedences* de l'exemple 6.4 atteint ce même point fixe.

Exemple 6.5. *L'instance donnée dans l'exemple 6.4 peut naturellement être décomposée en deux contraintes* cumulative *et trois contraintes d'inégalité :*

- $c_0 : cumulative(\langle \langle s_0, d_0, e_0, h_{0,0}\rangle, \langle s_1, d_1, e_1, h_{1,0}\rangle, \langle s_2, d_2, e_2, h_{2,0}\rangle, \langle s_3, d_3, e_3, h_{3,0}\rangle, \langle s_4, d_4, e_4, h_{4,0}\rangle\rangle, limit_0)$.
- $c_1 : cumulative(\langle \langle s_0, d_0, e_0, h_{0,1}\rangle, \langle s_1, d_1, e_1, h_{1,1}\rangle, \langle s_2, d_2, e_2, h_{2,1}\rangle, \langle s_3, d_3, e_3, h_{3,1}\rangle\rangle, limit_1)$.
- $c_3 : e_0 \leq s_3$.
- $c_4 : e_1 \leq s_3$.
- $c_5 : e_2 \leq s_4$.

Généralement, à chaque noeud de l'arbre de recherche, un solveur de contraintes traite d'abord les contraintes les plus légères [62, 38], i.e. c_3*,* c_4*,* c_5 *et atteint le point fixe sur ce sous-ensemble. Puis, il va traiter une contrainte* cumulative*. Ces deux phases vont se répéter jusqu'à ce que le point fixe sur l'ensemble des contraintes soit atteint. Le problème d'efficacité vient du fait que, lorsqu'une contrainte de précédence ajuste une borne d'une variable, il est nécessaire de réexécuter completement le filtrage de toutes les contraintes* cumulative *impliquant la variable en question.*

La propriété vérifiée par l'algorithme sweep_min page 60 à son point fixe est une extension de la propriété 6.1 qui maintenant considère aussi les contraintes de précédence.

Propriété 6.2. *Étant donné une contrainte* k-dimensional cumulative with precedences *avec* n *tâches et* k *ressources,* sweep_min *vérifie que :*

$$\forall r \in [0, k-1], \forall t \in [0, n-1], \forall i \in [\underline{s_t}, \underline{e_t}) \ : \ h_{t,r} + \sum_{\substack{t' \neq t, \\ i \in [\overline{s_{t'}}, \underline{e_{t'}})}} h_{t',r} \leq limit_r \tag{6.7}$$

$$\forall t \in [0, n-1], \forall t' \in P_t \ : \ \underline{e_t} \leq \underline{s_{t'}} \tag{6.8}$$

La propriété 6.2 vérifie que pour chaque tâche t de la contrainte k*-dimensional cumulative with precedences*, on peut fixer t à son début au plus tôt sans dépasser la capacité maximale de chaque ressource r ($0 \leq r < k$) par rapport au PPO et que tous ses successeurs immédiats ne peuvent pas commencer avant sa fin au plus tôt.

6.2.1 Types d'évènements

Pour construire le PPO des ressources, on a besoin des évènements introduits dans la section 6.1. On notera cependant une différence puisque l'évènement de type CCP sera fusionné avec l'évènement SCP. Ceci est possible puisque ces deux évènements sont associés au même point de temps, i.e. le début au plus tard d'une tâche. Notons que cette fusion aurait pu être réalisée pour les algorithmes précédents mais que nous tenions à les conserver tels qu'ils ont été présentés dans [42, 43], et ainsi conserver cette simplification pour ce nouvel algorithme.

- L'évènement $\langle SCP, t, \overline{s_t} \rangle$ correspond au début de la partie obligatoire de la tâche t, i.e. la date de début au plus tard de t. Cet évènement est généré pour toutes les tâches. Si la tâche n'a pas de partie obligatoire, il sera simplement ignoré au moment où ladroite de balayage le lira.
- L'évènement $\langle ECPD, t, \underline{e_t} \rangle$ correspond à la fin de la partie obligatoire de la tâche t, i.e. la date de fin au plus tôt de t. Cet évènement peut être repoussé sur l'axe temporel du fait de l'ajustement du début au plus tôt de t. Cet évènement est généré si et seulement si la tâche t possède déjà une partie obligatoire lors de la création des évènements, i.e. si $\overline{s_t} < \underline{e_t}$.
- L'évènement $\langle PR, t, \underline{s_t} \rangle$ correspond à la date de début au plus tôt de la tâche t. Cet évènement n'est généré que si la tâche n'est pas complètement fixée, i.e. si $\underline{s_t} \neq \overline{s_t}$.

Pour garantir la condition (6.8) de la propriété 6.2, les évènements des tâches ayant au moins un prédécesseur ne sont initialement pas ajoutés au tas h_events. Une tâche ne sera ajoutée à h_events uniquement lorsque le début au plus tôt de tous ses prédécesseurs immédiats seront ajustés à leur position finale par rapport à la propriété 6.2. Pour connaître l'instant où les évènements d'une tâche sont ajoutés nous introduisons le type d'évènement suivant :

- L'évènement $\langle RS, t, \underline{e_t} \rangle$ correspond à la fin au plus tôt de la tâche t. Cet évènement est généré pour toutes les tâches ayant au moins un successeur. Il est nécessaire pour empêcher le début au plus tôt d'un successeur d'être ajusté, avant que le début au plus tôt final de t ne soit lui-même connu.

Exemple 6.6. (Évènements générés). *Pour l'instance donnée dans l'exemple 6.4 page 56, les évènements suivants sont générés et insérés dans le tas h_events : $\langle PR, 1, 0\rangle$, $\langle PR, 2, 0\rangle$, $\langle SCP, 0, 1\rangle$, $\langle ECPD, 0, 2\rangle$, $\langle \mathbf{RS}, \mathbf{0}, \mathbf{2}\rangle$, $\langle \mathbf{RS}, \mathbf{1}, \mathbf{2}\rangle$, $\langle \mathbf{RS}, \mathbf{2}, \mathbf{2}\rangle$, $\langle SCP, 1, 3\rangle$, $\langle SCP, 2, 5\rangle$. D'un part, puisque les tâches t_0, t_1 et t_2 ont au moins un successeur, on génère un évènement de type RS (en gras) pour chacune d'elles. D'autre part, puisque t_3 et t_4 n'ont aucun successeur, on ne leur génère pas d'évènement RS.*

6.2.2 État de la droite de balayage

Pour construire le PPO de chaque ressource et ajuster le début au plus tôt de chaque tâche, la droite de balayage avance d'évènement en évènement et maintient les informations suivantes.

- La position courante δ de la droite de balayage, initialement positionnée à la date du premier évènement.
- Pour chaque ressource $r \in [0, k-1]$, la quantité de ressource disponible au point de temps δ est donnée par gap_r (i.e. la différence entre la capacité maximale de la ressource $limit_r$ et la hauteur du PPO sur la ressource r au point de temps δ) et sa valeur précédente par gap'_r.
- Pour chaque tâche $t \in [0, n-1]$, $ring_t$ enregistre son statut, et est égal à :
  - $none$ si et seulement si la droite de balayage n'a pas encore lu l'évènement PR de la tâche t,
  - $ready$ si et seulement si le début au plus tôt de la tâche t a été ajusté à sa valeur finale,
  - $check$ si et seulement si $\delta \in [\underline{s_t}, \overline{s_t})$ et $\forall r \in [0, k-1] : h_{t,r} \leq gap_r$, i.e. pour toutes les ressources, la consommation de la tâche t ne dépasse pas la quantité de ressource disponible par rapport au PPO,
  - $conflict_r$ si et seulement si $\delta \in [\underline{s_t}, \overline{s_t})$ and $\exists r \in [0, k-1) : h_{t,r} > gap_r$, i.e. il existe au moins une ressource r où la tâche t est en conflit. Notons que l'on enregistre que la première ressource où l'on detecte un conflit.

  Toutes les tâches t pour lesquelles $ring_t = check$ ou $ring_t = conflict_r$ seront appelées *tâches actives* par la suite. D'un point de vue implémentation, le statut des tâches actives est enregistré dans des anneaux, i.e. des listes circulaires doublement chainées, ce qui nous permet d'itérer rapidement sur toutes les tâches ayant comme statut $check$ ou $conflict$ et également de passer une tâche de $check$ à $conflict$ et vice versa en temps constant. Pour la suite, $conflict_\star$ sera utilisé pour indiquer qu'une tâche est en conflit sur une ressource dont on n'a pas besoin de connaitre l'identifiant.
- Pour chaque tâche $t \in [0, n-1]$, $nbpred_t$ maintient le nombre de prédécesseurs de la tâche t pour lesquels la date de début au plus tôt n'a pas encore été trouvée à la position courante de la droite de balayage.

L'algorithme de balayage synchrone avec précédences commence par créer et trier les évènements des tâches n'ayant pas de prédécesseur. Puis, la droite de balayage avance d'un évènement à un autre, mettant à jour, la quantité de ressource disponible sur chaque ressource et le statut des tâches. Une fois que le début au plus tôt de tous les prédécesseurs d'une tâche t ont été trouvés, i.e. $nbpred_t = 0$, les évènements relatifs à la tâche t sont générés et ajoutés dans le tas h_events.

6.2.3 Algorithme de filtrage

L'algorithme sweep_min du balayage synchrone avec précédences consiste en une boucle principale composée d'une partie dédiée au traitement des évènements et d'une partie dédiée au filtrage. La partie de traitement des évènements appelle l'algorithme release_task page 116 qui ajoute une tâche dans le système, i.e. qui génère et ajoute ses évènements dans h_events seulement lorsque tous les débuts au plus tôt de ses prédécesseurs ont été ajustés à leur valeur finale.

Algorithme principal

L'algorithme principale sweep_min page 60 est composé des parties suivantes :

- `[CRÉATION DES ÉVÈNEMENTS]` (ligne 2). Les évènements sont générés et insérés dans le tas h_events.
- `[INITIALISATION]` (lignes 4 à 7). Pour chaque ressource r ($0 \leq r < k$) la ressource disponible gap_r et sa précédente valeur gap'_r sont initialisées à la capacité maximale correspondante $limit_r$. Pour chaque tâche t, son statut est initialisé à $none$ si t n'est pas fixée, $ready$ dans le cas contraire.
- `[BOUCLE PRINCIPALE]` (lignes 9 à 14). Pour chaque position de la droite de balayage la boucle principale traite les évènements associés et met à jour l'état de la droite de balayage. L'algorithme sweep_min retourne **false** si un dépassement de la capacité maximale d'une des ressources est détecté ou si une tâche t ne peut plus être introduite dans sa fenêtre temporelle à cause de ses prédécesseurs.

ALGORITHM sweep_min() : boolean

1: `[CRÉATION DES ÉVÈNEMENTS]`
2: $h_events \leftarrow$ generation of events wrt n, $\underline{s_t}$, $\overline{s_t}$, d_t, $\underline{e_t}$ and the precedence constraints.
3: `[INITIALISATION]`
4: **for** $r = 0$ **to** $k - 1$ **do**
5: $\quad gap_r, gap'_r \leftarrow limit_r$
6: **for** $t = 0$ **to** $n - 1$ **do**
7: $\quad$ **if** $\underline{s_t} = \overline{s_t}$ **then** $ring_t \leftarrow$ ready **else** $ring_t \leftarrow$ none
8: `[BOUCLE PRINCIPALE]`
9: **while** $\neg$empty(h_events) **do**
10: $\quad \langle \delta, \delta_{next}, success \rangle \leftarrow$ process_events()
11: $\quad$ **if** $\neg success$ **then**
12: $\quad\quad$ **return false**
13: $\quad$ **if** $\neg$filter_min(δ, δ_{next}) **then**
14: $\quad\quad$ **return false**
15: **return true**

Algorithme 9: Retourne **false** si un dépassement de ressource est détecté ou si une contrainte de précédence ne peut pas être satisfaite. Retourne **true** sinon. La propriété 6.2 est vérifiée dans ce dernier cas.

Traitement des évènements

Pour mettre à jour le statut des tâches, l'algorithme process_events page 70 lit et traite tous les évènements se trouvant à la date δ et détermine l'intervalle de balayage $[\delta, \delta_{next}[$. L'algorithme process_events est composé des partie suivantes :

- `[TRAITER LES ÉVÈNEMENTS DE DÉBUT DE PARTIE OBLIG. (SCP)]` (lignes 3 à 14). Lorsque la droite de balayage atteint le début au plus tard d'une tâche t, on doit savoir si oui ou non son début au plus tôt peut encore être ajusté. Cela requiert de considérer les cas suivants :
  - Si la tâche t est en conflit ($ring_t = conflict_\star$, ligne 5), alors t ne peut pas être ajustée ailleurs qu'à sa position au plus tard.
  - Si le statut de la tâche t est $check$ (ligne 8), signifiant qu'il n'y a pas de conflit sur l'intervalle $[\underline{s_t}, \overline{s_t})$, alors le début au plus tôt de la tâche t ne peut pas être ajusté. Pour garantir la propriété 6.2, la consommation de la tâche t sur l'intervalle $[\overline{s_t}, \underline{e_t})$, qui est vide si la tâche n'a pas de partie obligatoire, est prise en compte dans le PPO.
- Une fois que le début et la fin au plus tôt de la tâche t sont à jour, il est nécessaire de savoir si une partie obligatoire a été créée (i.e. si oui ou non $\delta = \overline{s_t}$ est strictement supérieur à $\underline{e_t}$, ligne 10). Si une partie obligatoire est apparue, l'espace disponible sur chaque ressource gap_r ($0 \leq r < k$) est diminué selon la consommations de la tâche t sur la ressource r et un évènement $ECPD$ est ajouté à h_events.
- `[TRAITER LES ÉVÈNEMENTS DYNAMIQUES (ECPD)]` (lignes 16 à 21). Lorsque la droite de balayage atteint l'évènement $ECPD$ d'une tâche t on vérifie d'abord que la date de cet évènement n'est pas désynchronisée par rapport à la fin au plus tôt de la tâche t. Si la date est désynchronisée ($\underline{e_t} > \delta$, ligne 17), l'évènement $ECPD$ est replacé dans le tas h_events à sa bonne date. Si l'évènement est bien placé, la quantité de ressource disponible sur chacune des ressources est mise à jour (lignes 20 à 21) pour refléter l'apparition de partie obligatoire.
- `[TRAITER LES ÉVÈNEMENTS DE FIN AU PLUS TÔT (RS)]` (lignes 23 à 32). Lorsque la droite de balayage atteint l'évènement RS d'une tâche t, on doit d'abord savoir si cet évènement se trouve à sa position finale, i.e. si oui ou non la fin au plus tôt de la tâche t peut être modifiée. Cela requiert de considérer les cas suivants :
  - Si le statut de t est à $conflict$, l'évènement RS est repoussé à la première position possible, i.e. $\delta + d_t$ (ligne 25). Cette position considère que le début au plus tôt de la tâche t sera ajusté au moins à δ.
  - Sinon, si la position de l'évènement RS ne correspond pas à la fin au plus tôt de la tâche t, signifiant que celle-ci a été ajustée depuis la création de l'évènement RS, on repousse simplement l'évènement à la bonne position e_t dans le tas des évènements à traiter (ligne 27).

  Si l'évènement RS est à sa position finale, signifiant que le début au plus tôt de la tâche t ne peut plus être ajusté, les successeurs de t sont parcourus. Pour chaque successeur t' de la tâche t, le nombre de tâches restant à ajuster par rapport à la propriété 6.2 (i.e. $nbpred_{t'}$) est décrémenté (ligne 30). Si le début au plus tôt de tous les prédécesseurs de la tâche t' sont à jour par rapport à cette propriété 6.2, i.e. $nbpred_t = 0$, alors

les évènements relatifs à la tâche t' sont générés et insérés dans le tas des évènements à traiter. Cette dernière partie est effectuée dans l'algorithme release_task page 116.

- [DÉTERMINER LA DATE DU PROCHAIN ÉVÈNEMENT] (ligne 34). Pour pouvoir traiter les évènements de type PR, il est nécessaire de connaitre la prochaine position δ_{next} de la droite de balayage.

- [TRAITER LES ÉVÈNEMENTS DE DÉBUT AU PLUS TÔT (PR)] (lignes 36 à 42). Si un conflit est détecté (i.e. $\exists r \mid h_{t,r} > gap_r$, ligne 37), le statut de la tâche t est passé à $conflict$. Sinon, si l'intervalle de balayage est trop petit par rapport à la durée de la tâche t, son statut est positionné à $check$. Sinon on sait que le début au plus tôt de t ne pourra pas être ajusté lors de ce balayage.

Ajout d'un successeur

Une fois que les débuts au plus tôt de tous les prédécesseurs d'une tâche t ont été ajustés par rapport à la propriété 6.2, l'algorithme release_task page 116 génère et ajoute les évènement de la tâche t dans le tas h_events ou directement dans la liste des évènements venant juste d'être extraits $\mathcal{E}$. L'algorithme est composé des parties suivantes :

- [VÉRIFIER LE NOUVEAU DÉBUT AU PLUS TÔT] (lignes 2 à 3). Cette partie supprime du domaine de la variable de début (respectivement de fin) de la tâche t toutes les valeurs strictement inférieures à δ (respectivement strictement inférieures à $\delta + d_t$). Ce filtrage est imposé par la date de fin au plus tôt maximale des prédécesseurs de la tâche t. Retourne **false** si l'un des domaines devient vide.

- [LE DÉBUT AU PLUS TÔT DE LA TÂCHE t EST AJOUTÉ À δ] (lignes 5 à 16). On considère tout d'abord le cas où le début au plus tôt de la tâche t est égal à δ. Si la date de début de la tâche t est fixée (i.e. $\underline{s_t} = \overline{s_t}$, ligne 6), alors, l'espace disponible sur chacune des ressources est décrémenté par rapport aux consommations de t. Sinon, un évènement PR est ajouté dans la liste des évènements à traiter à la position courante de la droite de balayage (ligne 11). En effet, les évènements associés à la position courante de la droite de balayage ayant déjà été extraits le nouvel évènement PR doit directement être ajouté à la liste $\mathcal{E}$. Comme les évènements SCP et $ECPD$ ne peuvent pas être associés à la courante position δ de la droite de balayage, ils sont ajoutés dans le tas des évènements h_events. Enfin, si la tâche t a au moins un successeur, un évènement RS est également généré pour signaler aux successeurs de t l'instant où l'on aura trouvé un début au plus tôt pour la tâche t.

- [LE DÉBUT AU PLUS TÔT DE LA TÂCHE t EST AJOUTÉ APRÈS δ] (lignes 18 à 27). On considère le cas où le début au plus tôt de la tâche t est strictement plus grand que δ. Dans ce cas, les évènements sont générés comme lors de l'initialisation de sweep_min et insérés dans le tas h_events.

Filtrage

L'algorithme filter_min page 72 prend en compte les variations de l'espace disponible sur chaque ressource ainsi que la position de la droite de balayage pour traiter les tâches dont le statut est $conflict$ ou $check$, et éventuellement ajuster leur début au plus tôt. L'algorithme est composé des trois parties suivantes :

- `[VÉRIFIER LE DÉPASSEMENT DE RESSOURCE]` (lignes 2 à 3). Si la ressource disponible est négative sur l'intervalle de balayage $[\delta, \delta_{next}[$ sur au moins une des ressources, l'algorithme $\mathrm{filter_min}$ retourne **false**, indiquant que la contrainte n'a pas de solution.

- `[TÂCHES SORTANT DE CHECK]` (lignes 5 à 9). On parcourt chaque ressource r où la quantité de ressource disponible sur l'intervalle de balayage courant est inférieure à la quantité de ressource disponible sur l'intervalle de balayage précédent (i.e. $gap'_r > gap_r$, ligne 6). On considère chaque tâche t dont le statut est à $check$ telle que la hauteur de t est supérieure à la ressource disponible (ligne 7), i.e. les tâches dont le statut est à $check$ va nécessairement changer. Si la droite de balayage a dépassé la fin au plus tôt de la tâche t, signifiant qu'il n'y a pas de conflit sur l'intervalle $[\underline{s_t}, \underline{e_t})$, alors son statut est positionné à $ready$. Sinon, le statut de la tâche t est positionné à $conflict_r$.

- `[TÂCHES SORTANT DE CONFLIT SUR LA RESSOURCE` r`]` (lignes 11 à 22). On parcourt chaque ressource r où la quantité de ressource disponible sur l'intervalle de balayage courant est strictement supérieure à la quantité disponible sur l'intervalle précédent. On considère chaque tâche t dont le statut est à $conflict_r$ telle que la hauteur est inférieure ou égale à la quantité de ressource disponible sur l'intervalle de balayage courant (ligne 13), i.e. les tâches qui ne sont plus en conflit par rapport à la ressource r. On observe les deux cas suivants :

  - Si la tâche t est en conflit sur une autre ressource r' ($\exists r'.h_{t,r'} > gap_{r'}$, ligne 14) son statut est positionné à $conflict_{r'}$.
  - Sinon, le début au plus tôt de la tâche t est mis à jour (ligne 18) à la position courante de la droite de balayage et un évènement $ECPD$ est ajouté si une partie obligatoire apparaît (lignes 20 à 21).

Exemple 6.7. (Illustration du balayage synchrone avec précédences). *Détaillons maintenant le déroulement de l'algorithme de balayage synchrone avec précédences en reprenant l'instance de l'exemple 6.4 page 56. La liste des évènements générés est donnée dans l'exemple 6.6 page 59.*

Initialisation. *L'algorithme de balayage synchrone avec précédences commence par initialiser la position courante de la droite de balayage à 0, correspondant à la date du premier évènement, gap_0 à 3 et gap_1 à 2, correspondant à la capacité de la ressource r_0 et r_1. Puisque la tâche t_0 est fixée son statut est positionné à $ready$, celui des autres tâches est pour le moment positionné à $none$.*

Lecture des évènements associés à l'instant 0 et filtrage par rapport à l'intervalle de balayage courant $[0, 1[$. *La droite de balayage lit les évènements PR des tâches t_1 et t_2 et fixe leur statut à $check$ puisque leurs hauteurs sont inférieures à l'espace disponible sur chacune des ressources et que leur durée est supérieure à la taille de l'intervalle de balayage courant $[0, 1[$ (algorithme 10, ligne 40). Rien ne peut être déduit lors du premier appel de $\mathrm{filter_min}$ puisque pour toute ressource r ($0 \leq r < k$), $gap_r = gap_{r'}$. L'état du système à ce stade est illustré par la figure 6.4.*

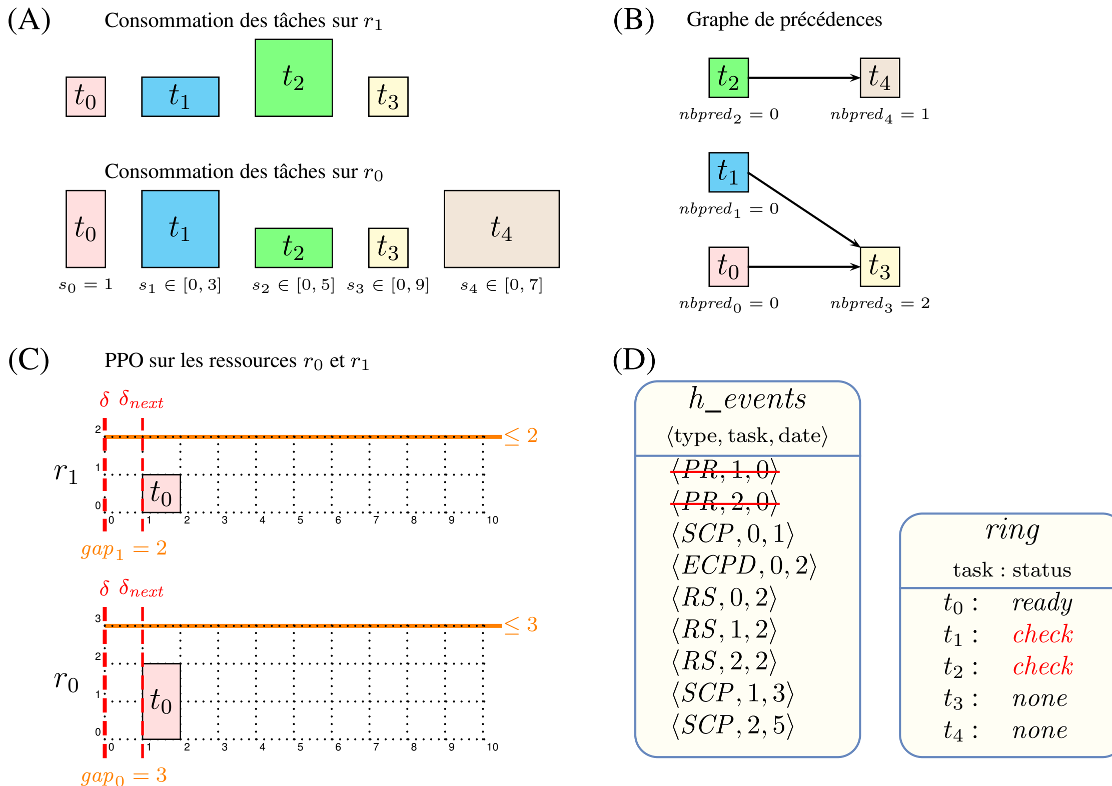


FIGURE 6.4 – État des tâches et des informations maintenues par la droite de balayage après la lecture des évènements $\langle PR, 1, 0\rangle$ et $\langle PR, 2, 0\rangle$, et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches.

Lecture de l'évènement associé à l'instant 1 et filtrage par rapport à l'intervalle de balayage courant $[1, 2[$**.** *Puis la droite de balayage avance à la position 1, lit l'évènement de début de partie obligatoire* SCP *de la tâche* t_0 *et modifie l'espace disponible* gap_0 *et* gap_1 *à 1. L'appel de* filter_min *change le statut de la tâche* t_1 *à* $conflict$ *en raison de sa hauteur sur la ressource* r_0*, et le statut de la tâche* t_2*, en raison de sa hauteur sur la ressource* r_1*. L'état du système à ce stade est illustré par la figure 6.5.*

Lecture des évènements associés à l'instant 2 et filtrage par rapport à l'intervalle de balayage courant $[2, 3[$**.** *La droite de balayage avance à la position 2, lit l'évènement* $ECPD$ *de la tâche* t_0*, fixe l'espace disponible* gap_0 *à 3 et* gap_1 *à 2 et lit les trois évènements* RS *des tâches* t_0*,* t_1 *et* t_2*. Puisque la tâche* t_0 *est initialement fixée, son évènement* RS *est nécessairement bien placé et on peut donc mettre à jour son unique successeur, la tâche* t_3 *(algorithme 10, ligne 29). Par conséquent* $nbpred_3$ *est maintenant égal à 1, signifiant que le début au plus tôt d'un prédécesseur de la tâche* t_3 *n'a pas encore été ajusté à sa valeur finale. Puisque le statut des tâches* t_1 *et* t_2 *est à* $conflict$*, leur évènement* RS *est repoussé à l'instant 4 (algorithme 10, ligne 25). L'appel de* filter_min *modifie le statut des tâches* t_1 *et* t_2 *à* $check$ *et ajuste leur début au plus tôt à 2. L'évènement* $\langle ECPD, 1, 4\rangle$ *est généré, reflétant la création de la partie obligatoire de la tâche* t_1*. L'état du système à ce stade est illustré par la figure 6.6.*

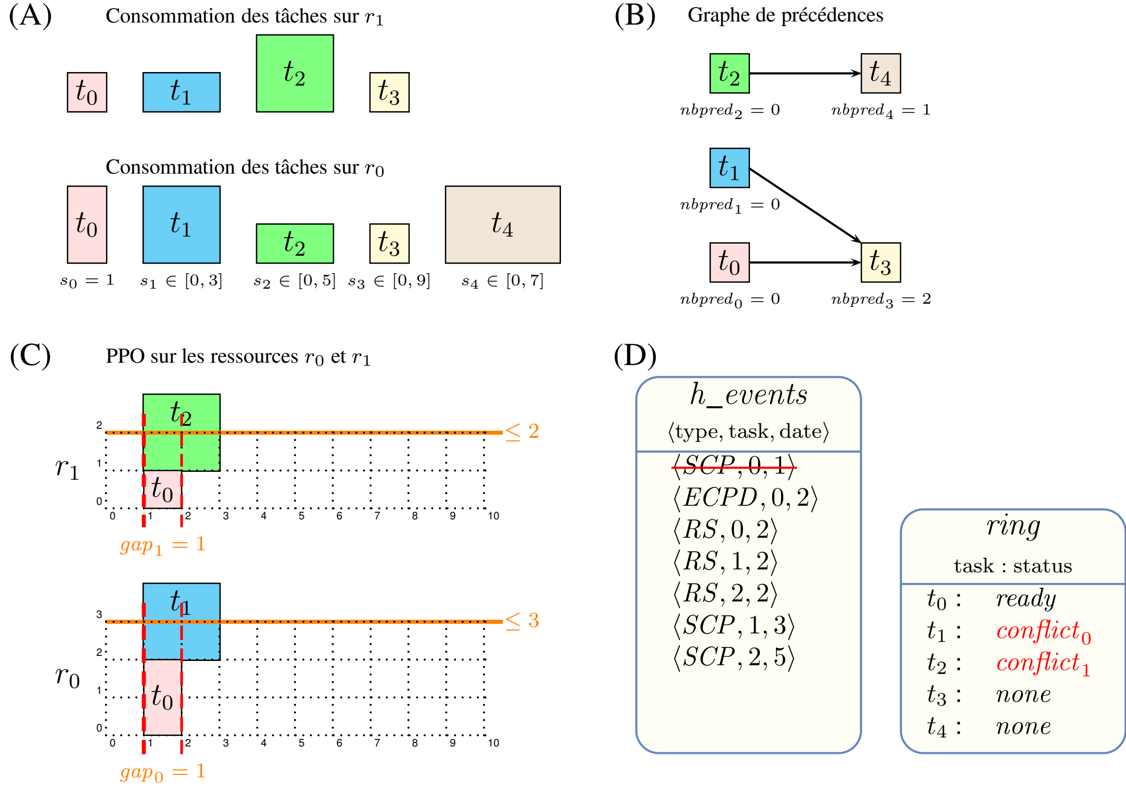


FIGURE 6.5 – État des tâches et des informations maintenues par la droite de balayage après la lecture de l'évènement $\langle SCP, 0, 1\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches.

Lecture de l'évènement associé à l'instant 3 et filtrage par rapport à l'intervalle de balayage courant $[3, 4[$**.** *Ensuite, la droite de balayage avance à la position 3, lit l'évènement SCP correspondant au début de la partie obligatoire de la tâche t_1 et ajuste l'espace disponible gap_0 à 1 et gap_1 à 1. Le statut de la tâche t_1 est maintenant à $ready$ (algorithme 10, ligne 9). L'appel de* filter_min *modifie le statut de la tâche t_2 à $conflict_1$ à cause de sa hauteur sur la ressource r_1. L'état du système à ce stade est illustré par la figure 6.7.*

Lecture des évènements associés à l'instant 4 et filtrage par rapport à l'intervalle de balayage courant $[4, 5[$**.** *La droite de balayage avance à la position 4, met à jour l'espace disponible gap_0 à 3 et gap_1 à 2 à cause de la fin de partie obligatoire de la tâche t_1. La droite de balayage lit l'évènement RS de t_1 qui est maintenant à sa position finale. Ainsi $nbpred_3$ vaut maintenant 0, signifiant que tous les prédécesseurs de la tâche t_3 ont atteint leur point fixe, et que les évènements de t_3 peuvent être maintenant générés et ajoutés à h_events. Cette partie est effectuée par* release_task *qui est appelé ligne 32 de l'algorithme 10. Dans* release_task*, le début au plus tôt de la tâche t_3 est ajusté à la position courante de la droite de balayage, i.e. 4. Puisque la tâche t_3 n'est pas fixée et qu'elle n'a pas de partie obligatoire, les évènements suivants sont générés : $\langle PR, 3, 4\rangle$, $\langle SCP, 3, 9\rangle$ (algorithme 21, lignes 11 à 12). Puis l'évènement PR est immédiatement traité et le statut de t_3 est positionné à $ready$ (algorithme 10, ligne 42). L'appel de* filter_min *modifie le statut de la tâche t_2 à $check$, ajuste son début au plus tôt à 4 et génère l'évènement $\langle ECPD, 2, 6\rangle$ reflétant la création*

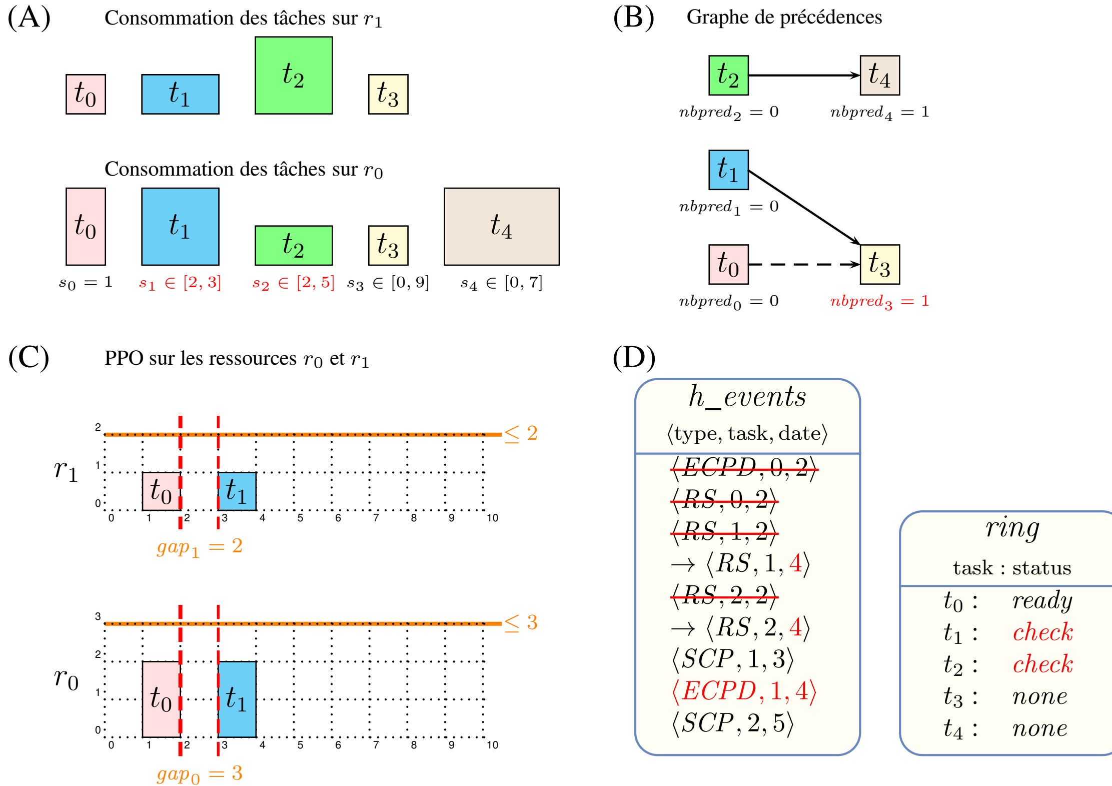


FIGURE 6.6 – État des tâches et des informations maintenues par la droite de balayage après la lecture des évènements $\langle ECPD, 0, 2\rangle$, $\langle RS, 0, 2\rangle$, $\langle RS, 1, 2\rangle$ et $\langle RS, 2, 2\rangle$, et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches.

d'une partie obligatoire. L'état du système à ce stade est illustré par la figure 6.8.

Lecture de l'évènement associé à l'instant 5 et filtrage par rapport à l'intervalle de balayage courant $[5, 6[$. *La droite de balayage avance à la position 5, lit l'évènement SCP correspondant au début de la partie obligatoire de la tâche t_2 et ajuste l'espace disponible gap_0 à 2 et gap_1 à 0. Le statut de t_2 est mis à ready (algorithme 10, ligne 9). Rien ne peut être déduit par l'appel de* filter_min. *L'état du système à ce stade est illustré par la figure 6.9.*

Lecture des évènements associés à l'instant 6 et filtrage par rapport à l'intervalle de balayage courant $[6, 7[$. *La droite de balayage avance à la position 6, lit l'évènement ECPD correspondant à la fin de la partie obligatoire de la tâche t_2 et ajuste l'espace disponible gap_0 à 3 et gap_1 à 2. Elle lit également l'évènement RS de la tâche t_2 qui est à sa position finale. Par conséquent, $nbpred_4$ vaut maintenant 0 (algorithme 10, ligne 30), signifiant que le début au plus tôt de tous les prédécesseurs de la tâche t_4 ont été ajustés à leur valeur finale par rapport à la propriété 6.2. L'appel de* release_task *génère les évènements $\langle PR, 4, 6\rangle$, $\langle SCP, 4, 7\rangle$ et $\langle ECPD, 4, 9\rangle$. L'état du système à ce stade est illustré par la figure 6.10.*

Enfin, la droite de balayage avance successivement aux positions 7 et 9, correspondant au début et à la fin de la partie obligatoire de la tâche t_4, et vérifie que la capacité maximale

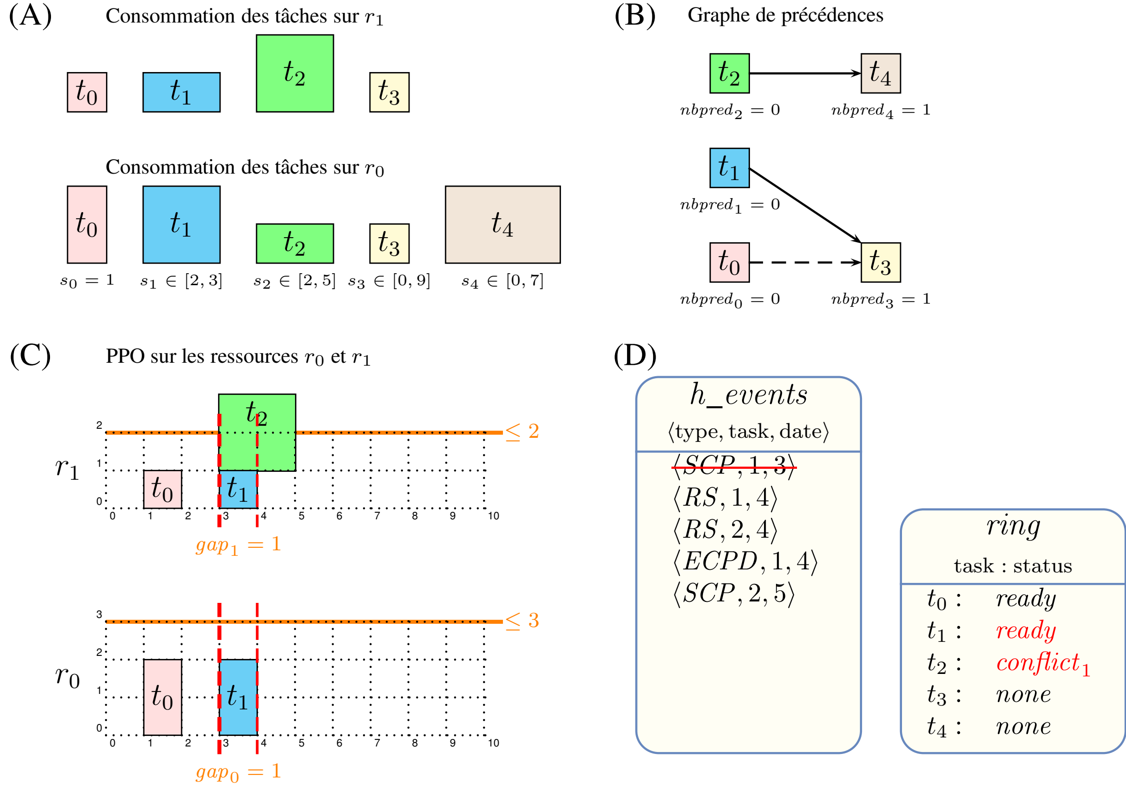


FIGURE 6.7 – État des tâches et des informations maintenues par la droite de balayage après la lecture de l'évènement $\langle SCP, 1, 3\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches.

des ressources n'a jamais été dépassée.

6.2.4 Complexité

Nous donnons maintenant la complexité temporelle dans le pire des cas de notre algorithme de balayage synchrone avec précédences ainsi que sa preuve.

Théorème 6.2. *Étant donné une contrainte k-*dimensional cumulative with precedences *impliquant n tâches, la complexité temporelle dans le pire des cas de l'algorithme synchrone avec précédences est $O(kn^2 + nX(k + \log n) + p)$, où X désigne le nombre maximum de fois qu'un évènement RS peut être repoussé sur l'axe temporel et p est le nombre d'arcs dans le graphe des précédences.*

Preuve du théorème 6.2 Dans le pire des cas, pour une tâche t, l'évènement RS peut être repoussé $\lceil (\overline{e_t} - \underline{e_t})/d_t \rceil$ fois (algorithme 10, ligne 25). Sur une exécution complète de l'algorithme sweep_min, la complexité temporelle dans le pire de cas de l'algorithme 10 est $O(kn + n\log n + nX \log n + p)$. La partie $nX \log n$ qui n'est pas présente dans la complexité de l'algorithme de balayage synchrone sans précédence s'explique par le fait que l'on doit traiter les $O(nX)$ RS évènements. Sur une exécution complète, la complexité temporelle dans le pire des cas de l'algorithme 12 est $O(kn^2 + nXk)$. En effet, à cause des $O(nX)$ évènements RS, l'algorithme 12 peut être appelé (nX) fois avec $\forall r \in [0..k-1] : gap_r = gap'_r$.

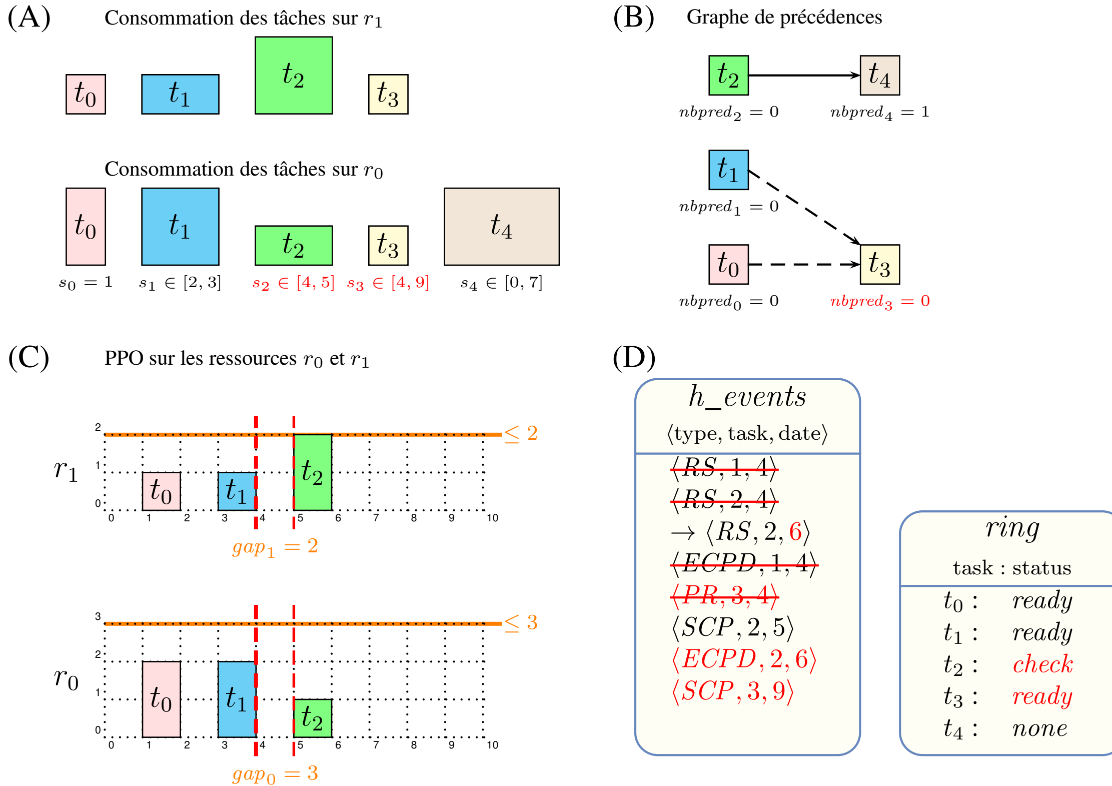


FIGURE 6.8 – État des tâches et des informations maintenues par la droite de balayage après la lecture des évènements $\langle ECPD, 1, 4\rangle$, $\langle RS, 1, 4\rangle$, $\langle RS, 2, 4\rangle$ et $\langle PR, 3, 4\rangle$, et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches.

Dans un tel cas, la complexité de l'algorithme 12 se limite à $O(k)$ (lignes 6 et 12). □

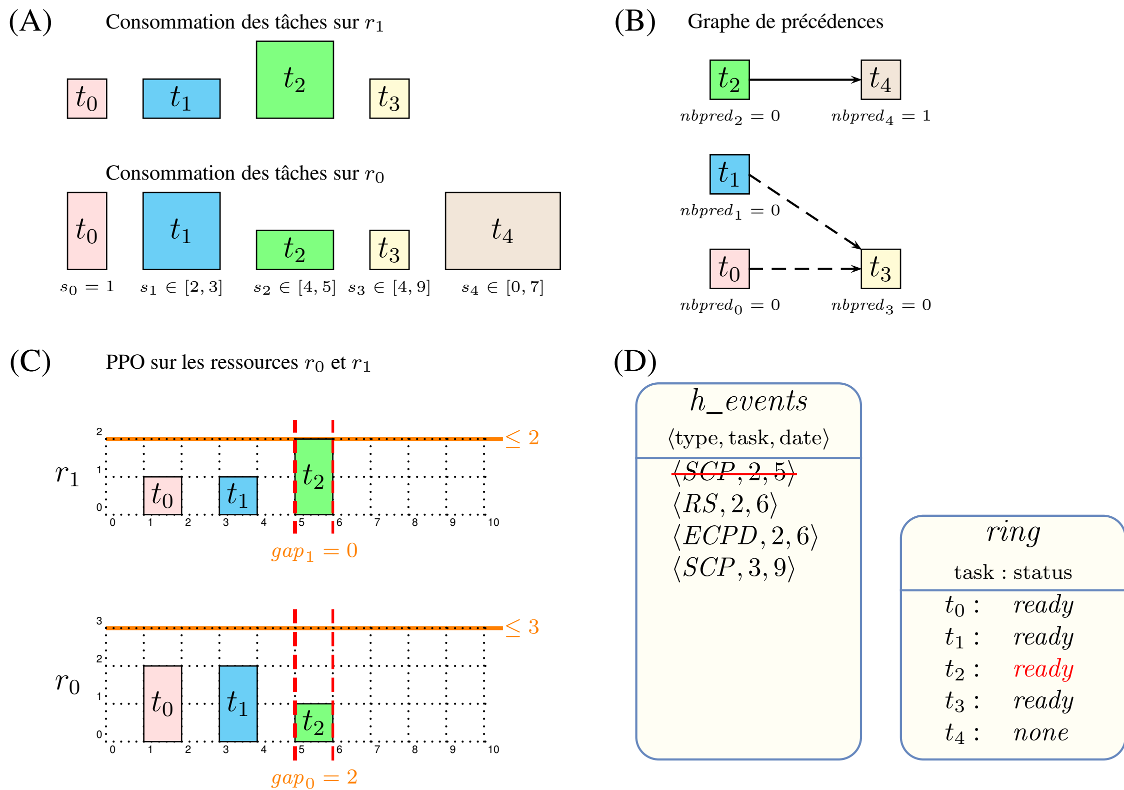


FIGURE 6.9 – État des tâches et des informations maintenues par la droite de balayage après la lecture de l'évènement $\langle SCP, 2, 5\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches.

ALGORITHM process_events() : $\langle$integer, integer$\rangle$

1: $\langle \delta, \mathcal{E} \rangle \leftarrow$ extract and record in $\mathcal{E}$ all the events in h_events related to the minimal date δ
2: [TRAITER LES ÉVÈNEMENTS DE DÉBUT DE PARTIE OBLIG. (SCP)]
3: **for all** events of type $\langle SCP, t, \overline{s_t} \rangle$ in $\mathcal{E}$ **do**
4: $\quad ecp' \leftarrow \underline{e_t}$
5: $\quad$ **if** $ring_t = \text{conflict}_\star$ **then**
6: $\quad\quad$ adjust_min_var$(s_t, \overline{s_t})$; adjust_min_var$(e_t, \overline{e_t})$
7: $\quad\quad ring_t \leftarrow$ ready
8: $\quad$ **else if** $ring_t =$ check **then**
9: $\quad\quad ring_t \leftarrow$ ready
10: $\quad$ **if** $\delta < \underline{e_t}$ **then**
11: $\quad\quad$ **for** $r = 0$ **to** $k-1$ **do**
12: $\quad\quad\quad gap_r \leftarrow gap_r - h_{t,r}$
13: $\quad\quad$ **if** $ecp' \leq \delta$ **then**
14: $\quad\quad\quad$ add $\langle ECPD, t, \underline{e_t} \rangle$ to h_events
15: [TRAITER LES ÉVÈNEMENTS DYNAMIQUES (ECPD)]
16: **for all** events of type $\langle ECPD, t, \underline{e_t} \rangle$ in $\mathcal{E}$ **do**
17: $\quad$ **if** $\underline{e_t} > \delta$ **then**
18: $\quad\quad$ add $\langle ECPD, t, \underline{e_t} \rangle$ to h_events
19: $\quad$ **else**
20: $\quad\quad$ **for** $r = 0$ **to** $k-1$ **do**
21: $\quad\quad\quad gap_r \leftarrow gap_r + h_{t,r}$
22: [TRAITER LES ÉVÈNEMENTS DE FIN AU PLUS TÔT (RS)]
23: **for all** events of type $\langle RS, t, \underline{e_t} \rangle$ in $\mathcal{E}$ **do**
24: $\quad$ **if** $ring_t = \text{conflict}_\star$ **then**
25: $\quad\quad$ add $\langle RS, t, \delta + d_t \rangle$ to h_events
26: $\quad$ **else if** $\delta \neq \underline{e_t}$ **then**
27: $\quad\quad$ add $\langle RS, t, \underline{e_t} \rangle$ to h_events
28: $\quad$ **else**
29: $\quad\quad$ **for all** $t' \in successors_t$ **do**
30: $\quad\quad\quad nbpred_{t'} \leftarrow nbpred_{t'} - 1$
31: $\quad\quad\quad$ **if** $nbpred_{t'} = 0$ **then**
32: $\quad\quad\quad\quad$ **if** $\neg$release_task$(t', \delta, \mathcal{E})$ **then return false**
33: [DÉTERMINER LA DATE DU PROCHAIN ÉVÈNEMENT]
34: $\delta_{next} \leftarrow$ get_top_key(h_events)
35: [TRAITER LES ÉVÈNEMENTS DE DÉBUT AU PLUS TÔT (PR)]
36: **for all** events of type $\langle PR, t, \underline{s_t} \rangle$ in $\mathcal{E}$ **do**
37: $\quad$ **if** $\exists r \mid h_{t,r} > gap_r$ **then**
38: $\quad\quad ring_t \leftarrow \text{conflict}_r$
39: $\quad$ **else if** $\underline{e_t} > \delta_{next}$ **then**
40: $\quad\quad ring_t \leftarrow$ check
41: $\quad$ **else**
42: $\quad\quad ring_t \leftarrow$ ready
43: **return** $\langle \delta, \delta_{next} \rangle$

Algorithme 10: Appelé à chaque fois que la droite de balayage se déplace. Extrait et traite tous les évènements associés au point de temps δ. Retourne la position courante δ, la prochaine position δ_{next} de la droite de balayage et un booléen indiquant si une erreur est détectée.

ALGORITHM release_task$(t, \delta, \mathcal{E})$: boolean

1: [VÉRIFIER LE NOUVEAU DÉBUT AU PLUS TÔT]
2: **if** $\neg$adjust_min_var$(s_t, \delta) \vee \neg$adjust_min_var$(e_t, \delta + d_t)$ **then**
3:     **return false**
4: [LE DÉBUT AU PLUS TÔT DE LA TÂCHE t EST AJOUTÉ À δ]
5: **if** $\underline{s_t} = \delta$ **then**
6:     **if** $\underline{s_t} = \overline{s_t}$ **then**
7:         **for** $r = 0$ **to** $k - 1$ **do**
8:             $gap_r \leftarrow gap_r - h_{t,r}$
9:         $ring_t \leftarrow$ ready
10:     **else**
11:         add $\langle PR, t, \underline{s_t} \rangle$ to $\mathcal{E}$
12:         add $\langle SCP, t, \overline{s_t} \rangle$ to h_events
13:     **if** $\overline{s_t} < \underline{e_t}$ **then**
14:         add $\langle ECPD, t, \underline{e_t} \rangle$ to h_events
15:     **if** task t has a least one successor **then**
16:         add $\langle RS, t, \underline{e_t} \rangle$ to h_events
17: [LE DÉBUT AU PLUS TÔT DE LA TÂCHE t EST AJOUTÉ APRÈS δ]
18: **else**
19:     add $\langle SCP, t, \overline{s_t} \rangle$ to h_events
20:     **if** $\overline{s_t} < \underline{e_t}$ **then**
21:         add $\langle ECPD, t, \underline{e_t} \rangle$ to h_events
22:     **if** $\underline{s_t} < \overline{s_t}$ **then**
23:         add $\langle PR, t, \underline{s_t} \rangle$ to h_events
24:     **else**
25:         $ring_t \leftarrow$ ready
26:     **if** task t has at least one successor **then**
27:         add $\langle RS, t, \underline{e_t} \rangle$ to h_events
28: **return true**

Algorithme 11: Génère et ajoute les évènement associés à la tâche t, signifiant que tous ses prédécesseurs ont atteint leur point fixe. Retourne **false** si δ a dépassé la date de début au plus tard de la tâche t.

```
ALGORITHM filter_min(δ, δ_next) : boolean
 1: [VÉRIFIER LE DÉPASSEMENT DE RESSOURCE]
 2: for r = 0 to k − 1 do
 3:    if gap_r < 0 then return false
 4: [TÂCHES SORTANT DE CHECK]
 5: for r = 0 to k − 1 do
 6:    if gap'_r > gap_r then
 7:       for all t | ring_t = check ∧ h_{t,r} > gap_r do
 8:          ring_t ← if e̲_t > δ then conflict_r else ready
 9:       gap'_r ← gap_r
10: [TÂCHES SORTANT DE CONFLIT SUR LA RESSOURCE r]
11: for r = 0 to k − 1 do
12:    if gap'_r < gap_r then
13:       for all t | ring_t = conflict_r ∧ h_{t,r} ≤ gap_r do
14:          if ∃r'.h_{t,r'} > gap_{r'} then
15:             ring_t ← conflict_{r'}
16:          else
17:             ecp' ← e̲_t
18:             adjust_min_var(s_t, δ) ; adjust_min_var(e_t, δ + d_t) ;
19:             ring_t ← if e̲_t > δ_next then check else ready
20:             if s̄_t ≥ ecp' ∧ s̄_t < e̲_t then
21:                add ⟨ECPD, t, e̲_t⟩ to h_events
22:       gap'_r ← gap_r
23: return true
```

Algorithme 12: Appelé à chaque fois que la droite de balayage se déplace de δ à δ_{next} pour ajuster le début au plus tôt des tâches par rapport à l'espace disponible sur chaque ressource.

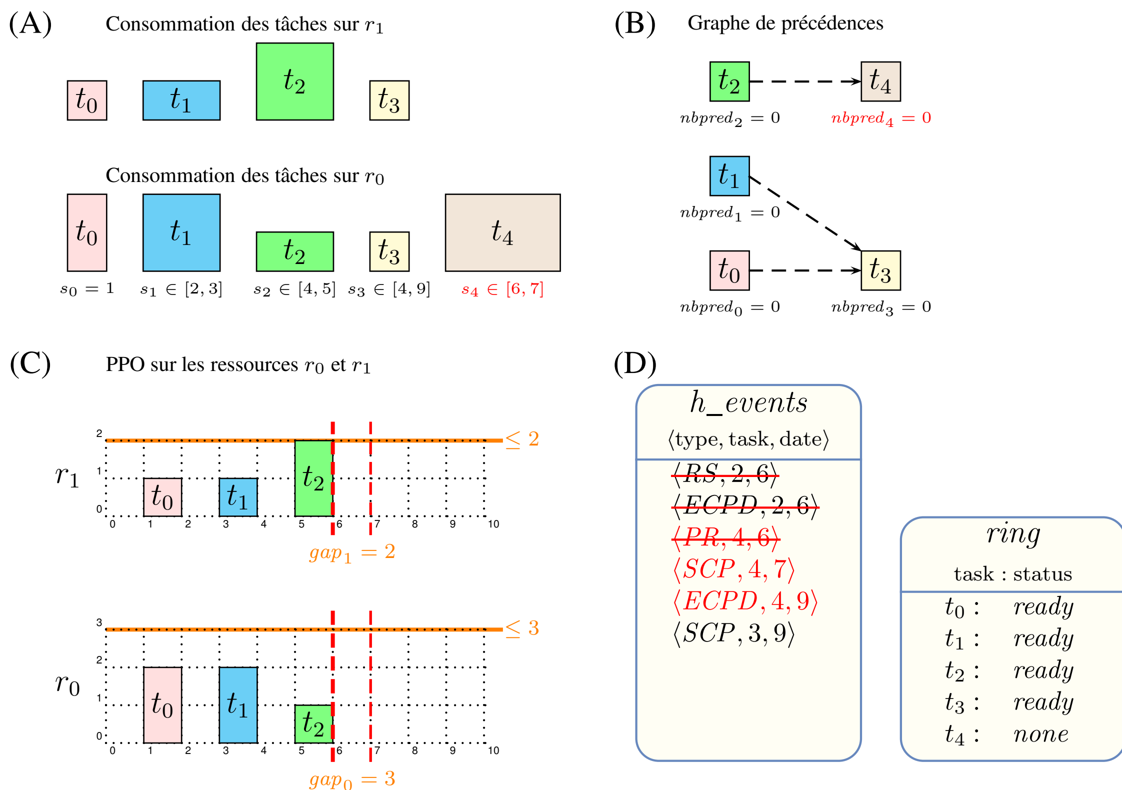


FIGURE 6.10 – État des tâches et des informations maintenues par la droite de balayage après la lecture des évènements $\langle RS, 2, 6\rangle$, $\langle ECPD, 2, 6\rangle$ et $\langle PR, 4, 6\rangle$, et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches.

6.3 Cumulative colorée : ou comment remplacer la somme par le nombre de valeurs distinctes

Alors que les contraintes de somme et de comptage sont omniprésentes en programmation par contraintes, seule la contrainte de somme a été considérée de manière approfondie jusqu'à présent dans le cadre de l'ordonnancement pour définir la contrainte *cumulative*. Étant donné une tâche ayant une date de début et une durée, on lui associe une consommation de ressource et on vérifie que la somme des hauteurs des tâches coupant chaque point de temps ne dépasse pas la capacité maximale de la ressource. En remplaçant cette contrainte de somme par la contrainte *nvalue* [52, 6, 15] et la consommation de ressource d'une tâche par une couleur, la première version du catalogue de contraintes globales [5, page 83] introduit la contrainte *coloured_cumulative*, qui à chaque point de temps s'assure que le nombre de couleurs distinctes affectées aux tâches coupant ce point de temps n'excède pas une limite donnée. Bien que la contrainte *coloured_cumulative* apparaît naturellement dans un certain nombre de problèmes d'ordonnancement et de bin-packing, et généralement avec d'autres contraintes telles que des *cumulative* et des précédences, aucun algorithme de filtrage n'a été publié pour cette contrainte. Notons cependant que dans le cadre du bin-packing une telle contrainte a été introduite dans [23]. Dans cette section, nous montrons comment intégrer la contrainte *coloured_cumulative* au sein de notre algorithme de balayage synchrone avec précédences pour obtenir un algorithme passant à l'échelle pour des problèmes mixant ce type de contraintes. L'idée clé du balayage synchrone, qui est de propager l'ensemble des contraintes (*cumulative*, *coloured_cumulative*, précédences) simultanément pour converger plus rapidement au point fixe est évidemment conservée.

Après avoir définit la contrainte *multiSumColorPrecCumulative*, nous donnons plusieurs cas réels où cette contrainte apparait naturellement. Ensuite, nous en proposerons une décomposition en un nombre quadratique de contraintes. Enfin, nous expliquerons comment adapter notre dernier algorithme de balayage synchrone pour cette contrainte.

6.3.1 Définition de la contrainte *multiSumColorPrecCumulative*

Étant donné n tâches, où chaque tâche t ($0 \leq t < n$) est décrite par sa date de début s_t, sa durée fixe d_t ($d_t \geq 0$), sa date de fin e_t et une couleur donnée r_t, la contrainte *coloured_cumulative* avec les deux arguments :

- $\langle\langle s_0, d_0, e_0, r_0\rangle, \ldots, \langle s_n, d_n, e_n, r_n\rangle\rangle$,
- $limit$,

est vérifiée si et seulement si les deux conditions (6.9) and (6.10) sont vraies :

$$\forall t \in [0, n-1] \ : s_t + d_t = e_t \tag{6.9}$$

$$\forall j \in \mathbb{Z} \ : |\{r_t : t \in \mathcal{T}_j\} - \{0\}| \leq limit \text{ avec } \mathcal{T}_j = \{t \in [0, n-1] : s_t \leq j < e_t\} \tag{6.10}$$

Dans la condition (6.10), la valeur 0 n'est pas comptée comme une couleur car $r_t = 0$ indique que la tâche t n'utilise aucune couleur. La figure 6.11 illustre cette contrainte à travers un exemple impliquant cinq tâches et une ressource de capacité égale à deux.

Remarquons que, (a) en attribuant une couleur différente à chaque tâche de la contrainte *coloured_cumulative* et (b) en fixant la limite de la ressource à 1, on se rapporte à un problème d'ordonnancement disjonctif, pour lequel le problème de satisfiabilité correspondant est NP-hard [29].

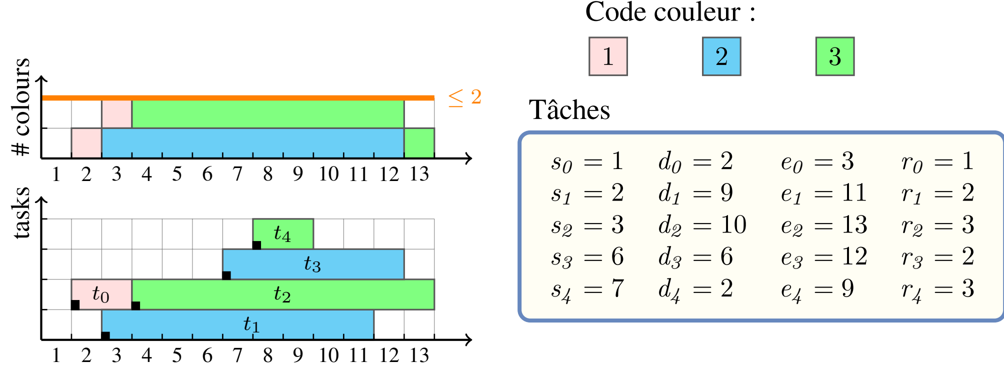


FIGURE 6.11 – Illustration d'une solution impliquant cinq tâches et une ressource de capacité deux dans une contrainte cumulative colorée.

L'introduction d'une couleur neutre nous permet de modéliser des problèmes impliquant plusieurs contraintes *coloured_cumulative* portant chacune sur un sous-ensemble différent de tâches. En pratique beaucoup de problèmes d'ordonnancement embarquent simultanément des contraintes *cumulative*, *coloured_cumulative* et des précédences. Pour conserver le passage à l'échelle de notre algorithme de balayage synchrone avec précédences nous introduisons la contrainte *multiSumColorPrecCumulative*.

Étant donné k ressources et n tâches, où chaque tâche t ($0 \leq t < n$) est décrite par un ensemble de successeurs S_t, sa date de début s_t, sa durée fixe d_t ($d_t \geq 0$), sa date de fin e_t et ses consommations de ressource $r_{t,i}$ ($0 \leq i < k$, $r_{t,i} \geq 0$), et chaque ressource i ($0 \leq i < k$) est décrite par son type $type_i$ ($type_i \in \{sum, colour\}$) et sa limite $limit_i$, la contrainte *multiSumColorPrecCumulative* avec les deux arguments :

- $\langle\langle \mathcal{S}_0, s_0, d_0, e_0, \langle r_{0,0}, \ldots, r_{0,k-1}\rangle\rangle, \ldots, \langle \mathcal{S}_{n-1}, s_{n-1}, d_{n-1}, e_{n-1}, \langle r_{n-1,0}, \ldots, r_{n-1,k-1}\rangle\rangle\rangle$,
- $\langle\langle type_0, limit_0\rangle, \ldots, \langle type_{k-1}, limit_{k-1}\rangle\rangle$

est vérifiée si et seulement si les conditions sont vraies :

$$\forall t \in [0, n-1],\ \forall t' \in \mathcal{S}_t\ : e_t \leq s_{t'} \tag{6.11}$$

$$\forall t \in [0, n-1]\ : s_t + d_t = e_t \tag{6.12}$$

$$\begin{gathered}\forall i \in [0, k-1] | type_i = sum,\ \forall j \in \mathbb{Z} : \\ \sum_{t \in \mathcal{T}_j} r_{t,i} \leq limit_i \text{ avec } \mathcal{T}_j = \{t \in [0, n-1] : s_t \leq j < e_t\}\end{gathered} \tag{6.13}$$

$$\begin{gathered}\forall i \in [0, k-1] | type_i = colour,\ \forall j \in \mathbb{Z} : \\ |\{r_{t,i} : t \in \mathcal{T}_j\} - \{0\}| \leq limit_i \text{ avec } \mathcal{T}_j = \{t \in [0, n-1] : s_t \leq j < e_t\}\end{gathered} \tag{6.14}$$

Par la suite, une *ressource cumulative* et une *ressource colorée* indiquerons respectivement une ressource pour laquelle le type associé est à sum et $colour$.

6.3.2 Schémas classiques d'apparition de la contrainte *multiSumColorPrecCumulative*

Nous illustrons dans cette partie l'utilisation de la contrainte *multiSumColorPrecCumulative* dans des problèmes d'ordonnancement et de bin-packing multi-dimensionnels mixant ressources cumulatives et colorées. Pour cela nous considérons un problème de placement issu des centres de données virtualisés, ainsi qu'un problème d'ordonnancement issu de l'industrie.

Centre de données virtualisés

Étant donné un ensemble de machines virtuelles (VM), où chaque VM consomme une certaine quantité de mémoire vive (RAM), de ressource CPU, et s'exécute sur un système d'exploitation, et un ensemble de serveurs où chaque serveur dispose d'une quantité de mémoire donnée, d'une capacité maximale de CPU, l'objectif est d'affecter les VM sur les serveurs en respectant deux types de contraintes :

① Contraintes de capacité

- `[RAM]` Une première contrainte *cumulative* garantit que la capacité mémoire de chaque serveur n'est jamais dépassée.
- `[CPU]` Une seconde contrainte *cumulative* garantit que la capacité CPU de chaque serveur n'est jamais dépassée.

② Contraintes de placement

- `[Un système d'exploitation par serveur]` La consommation de mémoire dans les centres de données virtualisés est un goulot d'étranglement. La partage de page mémoire entre VM est une méthode qui permet de réduire la mémoire consommée par l'ensemble des VM hébergées sur un serveur donné. Pour être efficace, il a été montré dans [74, 68] que les VM hébergées sur un même serveur devraient s'exécuter sur le même système d'exploitation. Une contrainte *coloured_cumulative* permet donc de garantir que toutes les VM hébergées sur un même serveur utilise le même système d'exploitation : à chaque système d'exploitation on fait correspondre une couleur et on fixe la limite de la ressource à 1. Cette contrainte est nommée *split* dans [34]. Remarquons qu'il est possible d'utiliser plusieurs contraintes *split* pour modéliser les contraintes de placement similaires.
- `[Isolation de VM]` Pour des raisons de sécurité, un client peut vouloir isoler ses VM sur un sous-ensemble de serveurs. On modélise cette contrainte en ajoutant une ressource colorée dont la limite est égale à 1, où chaque tâche/VM de ce client utilise la couleur 1, et toutes les autres tâches/VM utilisent la couleur 2. Cette contrainte est nommée *lonely* dans [34]. Un point important pour le passage à l'échelle est de remarquer que l'on peut modéliser toutes les contraintes *lonely* d'un centre de données avec une seule ressource colorée de limite 1. Pour cela, on affecte une même couleur aux VM d'un client, différente de toutes les autres VM. Toutes les VM qui ne sont pas impliquées dans une contrainte *lonely* utilisent la même couleur, différente de toutes les autres.
- `[Tolérance aux pannes]` Certaines VM assurent un service qui se doit d'être continu, tel qu'un service web par exemple. Pour augmenter la tolérance aux pannes matérielles, ces VM sont généralement répliquées, et les répliques sont hébergés sur des serveurs différents. Il est possible de modéliser cette contrainte avec une ressource cumulative de limite 1, où chaque réplique de la VM d'origine consomme 1 et les autres tâches/VM consomment 0. Cette contrainte est nommée *spread* dans [34].

Considérons le cas où nous avons a contraintes *split*, b contraintes *lonely*, et c contraintes *spread*. Alors, en rassemblant l'ensemble des ressources, nous avons une seule contrainte *multiSumColorPrecCumulative* avec $c + 2$ ressources cumulatives et $a + 1$ ressources colorées.

Redéploiement d'un réseau d'informations financières à grande échelle

Le problème consiste à déployer un réseau fournissant un accès rapide aux centres financiers pour des opérations d'achats/ventes automatiques. L'objectif est un redéploiement rapide du réseau pour offrir ce nouveau service (plus coûteux) au client. Le problème met en jeu plus de 150 pays dans le monde. Une série de jobs doivent donc être ordonnancés sur plusieurs années. Chaque job est composé de plusieurs tâches qui peuvent requérir des quantités limitées de différentes ressources. Chaque tâche doit être réalisée dans un pays donné. À chaque instant, l'ensemble des tâches en cours de réalisation ne doivent impliquer qu'au plus k pays différents, où k est une limite fixée. Enfin, il y a des dépendances dans l'enchainement des tâches se traduisant par des contraintes de précédence entre les tâches.

Le problème de steel mill slab

Le steel mill slab est un problème connu de la communauté contrainte et présent dans la liste de la CSPLib [31] (problème 38). Dans une aciérie disposant de barres d'acier d'une taille donnée, on souhaite découper ces barres suivant un ensemble de commandes à satisfaire. A chaque commande est affectée une couleur, correspondant au chemin suivi par la commande dans l'aciérie et une taille, donnant la taille de la barre demandée. Une même barre d'acier ne peut pas être utilisée pour des commandes requérant plus de deux couleurs distinctes. Dans sa version d'optimisation, on cherche à minimiser les pertes, c'est-à-dire, minimiser la somme des hauteurs des barres d'aciers utilisées dans la solution.

Des propositions pour la modélisation de ce problème ont d'abord été données dans [27, 28, 35], et plus récemment, des modèles plus sophistiqués et compétitifs sont présentés dans [30, 70, 60, 33]. Les travaux antérieurs de Dawande et al. [23] sur un problème de bin-packing avec des hauteurs variables et une dimension colorée ont également été motivés par des problèmes issus de l'industrie de l'acier.

6.3.3 Reformulation quadratique de la contrainte cumulative colorée

La contrainte *coloured_cumulative* avec n tâches peut être exprimée en $O(n^2)$ contraintes réifiées [11] et $O(n)$ contraintes *atmost_nvalue* [16], une complexité spatiale dans le pire des cas qui dépend uniquement du nombre de tâches et pas du nombre de points de temps :

① Soit $\mathcal{T}^+$ l'ensemble des tâches t ($t \in [0, n-1]$) possédant une couleur non neutre, i.e. différente de 0.

② Pour chaque paire de tâches i, j $(i, j \in \mathcal{T}^+)$, on crée une variable $C_{i,j}$ qui prend la couleur de la tâche j si la tâche j coupe le début de la tâche i, et la couleur de la tâche i sinon.

③ Pour chaque tâche i $(i \in \mathcal{T}^+)$, on crée la contrainte *atmost_nvalue* suivante qui restreint le nombre maximum de couleurs distinctes associées aux tâches qui coupent le début de la tâche i afin de ne pas dépasser la limite autorisée à chaque point de temps :

$$C_{i,j} = \begin{cases} r_j, & \text{si } s_j \leq s_i \wedge e_j > s_i \\ r_i, & \text{sinon} \end{cases} \tag{6.15}$$

$$atmost_nvalue(limit, \{C_{i,j} \mid j \in \mathcal{T}^+\}) \tag{6.16}$$

6.3.4 Intégration au balayage synchrone avec précédences

Nous présentons maintenant comment intégrer à notre algorithme de balayage synchrone avec précédences (algorithme sweep_min page 60) la notion de ressource colorée. Dans ce nouveau contexte, la droite de balayage parcourt l'axe du temps, construit le PPO ainsi que le *profil de parties obligatoires colorées* (PPOC), effectue des vérifications et filtre les domaines des variables selon ces profils. Définissons maintenant ce qu'est le profil de parties obligatoires colorées.

Définition 6.1. (PPOC). *Étant donné un ensemble de tâches $\mathcal{T}$ et une ressource i, le PPOC de la ressource i de l'ensemble $\mathcal{T}$ consiste en l'agrégation des couleurs attachées aux parties obligatoires de tâches de $\mathcal{T}$. La hauteur du PPOC pour la ressource i à un point de temps donné est* $|\{r_{t,i} : t \in \mathcal{T} \wedge r_{t,i} \neq 0 \wedge j \in [\overline{s_t}, \underline{e_t})\}|$.

La propriété vérifiée par l'algorithme de balayage que nous décrivons dans ce chapitre étend la propriété 6.2 page 58 pour intégrer les dimensions colorées.

Propriété 6.3. *Étant donnée une contrainte* multiSumColorPrecCumulative *impliquant n tâches et k ressources, l'algorithme de propagation associé vérifie que :*

$$\forall i \in [0, k-1] | type_i = sum,\ \forall t \in [0, n-1],\ \forall j \in [\underline{s_t}, \underline{e_t}) : \quad r_{t,i} + \sum_{\substack{t' \in [0,n-1] \wedge \\ t' \neq t \wedge \\ j \in [\overline{s_{t'}}, \underline{e_{t'}})}} r_{t',i} \leq limit_i \tag{6.17}$$

$$\forall i \in [0, k-1] | type_i = colour,\ \forall t \in [0, n-1],\ \forall j \in [\underline{s_t}, \underline{e_t}) : \quad \left|\left\{\{r_{t,i}\} \cup \{r_{t',i} : t' \in [0, n-1] \ \wedge\ t' \neq t \ \wedge\ j \in [\overline{s_{t'}}, \underline{e_{t'}})\}\right\} - \{0\}\right| \leq limit_i \tag{6.18}$$

$$\forall t \in [0, n-1],\ \forall t' \in \mathcal{S}_t :\ \underline{e_t} \leq \underline{s_{t'}} \tag{6.19}$$

La propriété 6.3 assure que pour chaque tâche t de la contrainte *multiSumColorPrecCumulative*, on peut fixer t à sa date de début au plus tôt :

① sans dépasser pour chacune des ressources cumulatives i ($0 \leq i < k$) sa limite par rapport au PPO de la ressource i des tâches $t' \in [0, n-1], t' \neq t$,

② sans dépasser pour chacune des ressources colorées i ($0 \leq i < k$) sa limite par rapport au PPOC de la ressource i des tâches $t' \in [0, n-1], t' \neq t$,

③ et que tous ses successeurs immédiats ne peuvent pas débuter avant sa fin au plus tôt.

Types d'évènements

Puisque le PPO et le PPOC sont basés sur le début et fin des parties obligatoires des tâches, on conserve à l'identique l'ensemble des évènements utilisés pour l'algorithme de balayage synchrone avec précédences.

État de la de droite de balayage

Pour construire le PPO des ressources cumulatives et ajuster le début au plus tôt des tâches par rapport à ces ressources, on réutilise l'ensemble des structures introduite pour l'algorithme de balayage synchrone avec précédences, que nous rappelons rapidement ici :

- La position courante δ de la droite de balayage, initialisée à la date du premier évènement.

- Pour chaque ressource cumulative, la quantité de ressource disponible au point de temps δ.
- Pour chaque tâche $t \in [0, n-1]$, son statut est :
  - $none$ si et seulement si la droite de balayage n'a pas encore atteint le début au plus tôt de la tâche t,
  - $ready$ si et seulement si le début au plus tôt de la tâche t a été ajusté à sa valeur finale (i.e. elle ne sera plus modifiée par le balayage en cours).
  - $conflict_i$ si le statut de la tâche t n'est ni $none$ ni $ready$, et qu'il existe au moins une ressource cumulative i où la tâche t est en conflit. Une tâche t est en conflit sur une ressource cumulative i si et seulement si sa consommation de ressource dépasse la quantité de ressource disponible (i.e. la différence entre la limite de la ressource et la hauteur du PPO).
  - $check$ si aucun des autres cas n'est vérifié.
- Pour chaque tâche $t \in [0, n-1]$, $nbpred_t$ donne le nombre de ses prédécesseurs dont le statut n'est pas encore égal à $ready$.

Dans un premier temps, pour construire le PPOC des ressources colorées, on ajoute les structures de données suivantes. Pour chaque ressource colorée $i \in [0, k-1]$:

- $nbDistinctColors_i$ donne le nombre de couleurs distinctes qui sont utilisées au point de temps δ.
- $countColors_{i,c}$ donne le nombre de tâches utilisant la couleur c au point de temps δ.

Remarquons que pour n'importe quel évènement de profil, i.e. début ou fin de partie obligatoire, la mise à jour du PPOC, i.e. $nbDistinctColors_i$ et $countColors_{i,c}$ est faite en temps constant.

Dans un second temps, on adapte le sens donné au statut $conflict$ pour les ressources colorées de la manière suivante :

- Une tâche $t \in [0, n-1]$ est en conflit sur une ressource colorée i si et seulement si $nbDistinctColors_i = limit_i \wedge countColors_{i,r_{t,i}} = 0$, i.e. le nombre maximal de couleurs distinctes utilisées sur la ressource i est atteint et ces couleurs utilisées sont toutes différentes de la couleur de la tâche t.

Nous avons présenté dans cette section l'ensemble des modifications permettant d'adapter notre algorithme de balayage synchrone avec précédences décrit à la section 6.2 pour la contrainte *multiSumColorPrecCumulative*. La structure générale de l'algorithme n'étant pas remise en cause par cette adaptation nous ne le décrirons donc pas mais le lecteur pourra le consulter en annexe page 113.

7

Synthèse des nouveaux algorithmes de balayage

Sommaire

7.1 Algorithme de balayage dynamique 81
7.2 Algorithme de balayage synchrone sans précédence 82
7.3 Algorithme de balayage synchrone avec précédences 82

Nous présentons dans ce chapitre une vue synthétique des différents algorithmes de balayage introduits aux chapitres précédents. Pour chacun d'eux, nous rappelons les points clés concernant les évènements générés et traités, les informations maintenues par la droite de balayage et leur complexité.

7.1 Algorithme de balayage dynamique

Commençons par le premier algorithme de balayage présenté dans cette thèse au chapitre 5.

- `[ÉVÈNEMENTS]` Au plus quatre évènements par tâche sont générés et ajoutés au tas des évènements. Lorsqu'une tâche n'est initialement pas fixée (i.e. $\underline{s_t} = \overline{s_t}$), un évènement PR associé à sa date de début au plus tôt ainsi qu'un évènement CCP associé à sa date de début au plus tard sont générés. Puis, l'évènement CCP peut être converti en deux autres évènements, SCP et $ECPD$, seulement si une partie obligatoire apparaît. Les évènements *conditionnels* CCP et *dynamiques* $ECPD$ sont des points clés de l'algorithme puisqu'ils permettent de prendre en compte l'extension du PPO en un même balayage.

- `[ÉTAT DE LA DROITE DE BALAYAGE]` Les principales structures de données sont les deux tas h_check et $h_conflict$ qui enregistrent le statut des tâches. L'utilisation de ces deux tas permet d'éviter de reparcourir systématiquement l'ensemble des tâches actives à chaque nouvelle position de la droite de balayage.

- [COMPLEXITÉ] La complexité temporelle dans le pire des cas est de $O(n^2 \log n)$. Elle peut être réduite à $O(n^2)$ en remplaçant les deux tas h_check et $h_conflict$ par une liste enregistrant le statut des tâches, mais avec une telle structure, la complexité $O(n^2)$ est plus souvent atteinte en pratique.

7.2 Algorithme de balayage synchrone sans précédence

Continuons avec l'algorithme de balayage synchrone ne prenant pas en compte les contraintes de précédence, présenté au chapitre 6, section 6.1.

- [ÉVÈNEMENTS] Comme pour l'algorithme de balayage dynamique mono-dimensionnel, au plus quatre évènements sont générés par tâche. Pour une tâche initialement non fixée, un évènement de type PR associé à sa date de début au plus tôt est généré. Si la tâche possède une partie obligatoire alors les évènements SCP et $ECPD$ sont générés, sinon, l'évènement *conditionnel* CCP associé à sa date de début au plus tard est généré. Les évènements *conditionnels* CCP et dynamiques $ECPD$ restent des points clés pour que l'algorithme puisse prendre en compte l'extension du PPO à la volée.

- [ÉTAT DE LA DROITE DE BALAYAGE] Le tableau a_check_r $(0 \leq r < k)$ associé à chaque ressource ainsi que les deux fonctions $pred$ et $succ$ permettant d'itérer uniquement sur les tâches pouvant potentiellement changer de statut constituent des points clés de l'algorithme. En effet, ces deux éléments permettent d'éviter de reparcourir systématiquement l'ensemble des tâches actives à chaque nouvelle position de la droite de balayage. Contrairement à l'algorithme de balayage dynamique pour la contrainte cumulative mono-ressource, nous n'utilisons plus la structure de tas pour enregistrer le statut des tâches. L'avantage donné par les tas h_check et $h_conflict$ sur la version mono-ressource vient du fait qu'une tâche se trouve soit dans le tas h_check, soit dans le tas $h_conflict$. Pour la version multi-ressources, nous aurions dû créer ces deux tas pour chaque ressource, et pour chaque tâche, nous aurions dû la dupliquer dans tous les tas h_check pour signaler qu'elle n'est pas en conflit.

- [COMPLEXITÉ] La complexité temporelle dans le pire des cas de cet algorithme est $O(kn^2)$.

7.3 Algorithme de balayage synchrone avec précédences

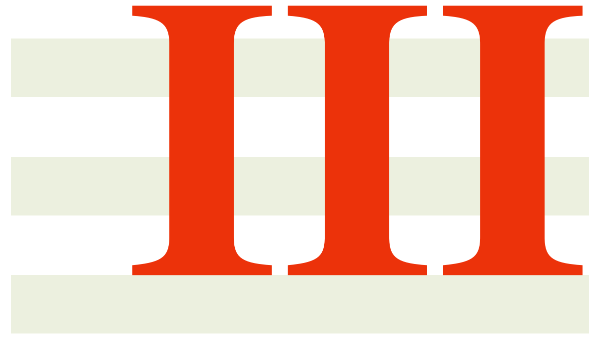
Terminons par l'algorithme de balayage synchrone avec précédences présenté au chapitre 6, section 6.2.

- [ÉVÈNEMENTS] Initialement seules les tâches n'ayant pas de prédécesseurs ont leurs évènements générés et insérés dans le tas des évènements. Les évènements sont les mêmes que pour l'algorithme précédent, plus un évènement RS associé à la date de fin au plus tôt des tâches ayant au moins un successeur. Dans le pire des cas, chaque évènement RS d'une tâche t peut être repoussé sur l'axe temporel $\lceil (\overline{e_t} - \underline{e_t})/d_t \rceil$ fois.

- [ÉTAT DE LA DROITE DE BALAYAGE] Les principales structures de données de cet algorithme sont les listes doublement chainées circulaires enregistrant le statut des tâches, qui nous permettent de modifier le statut d'une tâche (i.e. passer la tâche

d'une liste à une autre) en temps constant. Pour traiter les précédences nous avons ajouté un entier $nbpred_t$ associé à chaque tâche t, qui enregistre le nombre de prédécesseurs pour lesquels la date de début au plus tôt n'a pas encore été trouvée à la position courante de la droite de balayage.

- `[COMPLEXITÉ]` Dans le pire des cas, la complexité temporelle de l'algorithme de balayage synchrone avec précédences est $O(kn^2 + nX(k + \log n) + p)$ où X est le nombre maximal de fois qu'un évènement RS peut être repoussé, i.e. $\max_{t \in [0..n-1]}(\lceil(\overline{e_t} - \underline{e_t})/d_t\rceil)$ et p est le nombre d'arcs dans le graphe des précédences.



Améliorations pratiques et évaluation

8

Améliorations pratiques

Sommaire

8.1 Mieux gérer les matrices creuses de consommations 87

8.2 Agrégation des parties obligatoires 88

8.3 Mode glouton 88

8.3.1 Types d'évènements 88

8.3.2 État de la droite de balayage 89

8.3.3 Algorithme glouton 90

8.1 Mieux gérer les matrices creuses de consommations

Lors la modélisation d'un problème d'ordonnancement cumulatif, un utilisateur peut être amené à ajouter des ressources « virtuelles » au problème pour modéliser des contraintes (e.g. des disjonctions) entre un sous-ensemble de tâches données. Les tâches non concernées par ces contraintes voient donc leurs hauteurs sur ces ressources mises à zéro. La matrice des consommations des tâches sur les différentes ressources est alors creuse. Comme nous le verrons lors de l'évaluation des différents algorithmes de cette thèse, c'est le cas pour beaucoup d'instances de la PSPLib [56].

Dans les deux algorithmes de balayage synchrone que nous avons présenté au chapitre 6, nous ne tirons pas partie de cette observation. En effet, lors de la lecture d'un évènement de début de partie obligatoire par exemple, la mise à jour du PPO est effectuée par une boucle parcourant l'ensemble des ressources du problème, même celles où la tâche a une consommation nulle.

Une amélioration simple consiste alors a créer un tableau de listes indiquant pour chaque tâche les ressources pour lesquelles sa hauteur n'est pas nulle. Une amélioration similaire dans le cas coloré concerne les tâches utilisant une réellement une couleur (i.e. une couleur non neutre).

8.2 Agrégation des parties obligatoires

À chaque noeud de l'arbre de recherche, les algorithmes de balayage dynamique et synchrones que nous avons introduits aux chapitres 5 et 6 de cette thèse opèrent en deux phases successives, *sweep_min* et *sweep_max*, qui se répètent jusqu'à ce le point fixe soit atteint.

Afin de ne pas avoir à recalculer entièrement le profil des parties obligatoires (PPO) des tâches à chaque appel de *sweep_min* et *sweep_max*, nous choisissons à chaque noeud de l'arbre de recherche de calculer un profil correspondant à l'agrégation des parties obligatoires des tâches entièrement fixées (i.e. les tâches pour lesquelles $\underline{s_t} = \overline{s_t}$). Ce profil est ensuite passé successivement à *sweep_min* et *sweep_max* par le biais des évènements associés aux instants où la hauteur du profil cumulé des parties obligatoires des tâches fixées varie.

Dans le cas d'une contrainte *cumulative* (mono-dimensionel), ce profil est calculé en une complexité de $O(n \log\ n)$ et apporte un gain non négligeable d'environ 20% sur le temps complet de résolution.

8.3 Mode glouton

À partir de chaque algorithme de balayage décrit aux chapitres 5 et 6, nous avons conçu un mode glouton permettant d'augmenter sensiblement la taille des instances traitées ainsi que le temps de résolution. A chaque noeud de l'arbre de recherche, l'algorithme glouton tente de trouver une solution au problème. Si il échoue, l'algorithme de propagation classique est alors exécuté et la recherche continue. Le mode glouton n'est pas composable, c'est-à-dire que pour être utilisé, le problème donné au solveur doit être entièrement modélisé dans une unique contrainte.

Nous décrivons dans cette section le mode glouton dérivé du dernier algorithme présenté, le balayage synchrone avec précédences. Le mode glouton réutilise l'algorithme $\mathrm{sweep_min}$ de propagation classique. En effet, la droite de balayage parcourt de gauche à droite l'axe du temps, puis, lorsque le début au plus tôt d'une tâche est trouvé, celle-ci y est directement fixée. Si cette date est antérieure à la position courante de la droite de balayage, l'état de la droite de balayage est réinitialisée à cette date pour prendre en compte le changement du PPO. L'algorithme glouton reprenant les principes de l'algorithme de propagation classique, nous ne réutiliserons pas l'exemple 6.4 mais un exemple plus simple n'utilisant qu'une ressource et trois tâches pour illustrer le plus clairement possible le mécanisme de réinitialisation de la droite de balayage.

Exemple 8.1. *Considérons une ressource de capacité maximale 3 ainsi que trois tâches t_0, t_1 et t_2 ayant les débuts, durées, fins et hauteurs suivants :*

- t_0 : $s_0 \in [2, 2]$, $d_0 = 2$, $e_0 \in [4, 4]$, $h_0 = 2$,
- t_1 : $s_1 \in [0, 5]$, $d_1 = 3$, $e_1 \in [3, 8]$, $h_1 = 1$,
- t_2 : $s_2 \in [0, 5]$, $d_2 = 3$, $e_2 \in [3, 8]$, $h_2 = 1$.

Cette instance est illustrée par la partie (A) de la figure 8.1.

8.3.1 Types d'évènements

Nous conservons les deux types d'évènement PR et SCP auxquels nous ajoutons les deux évènements suivants :

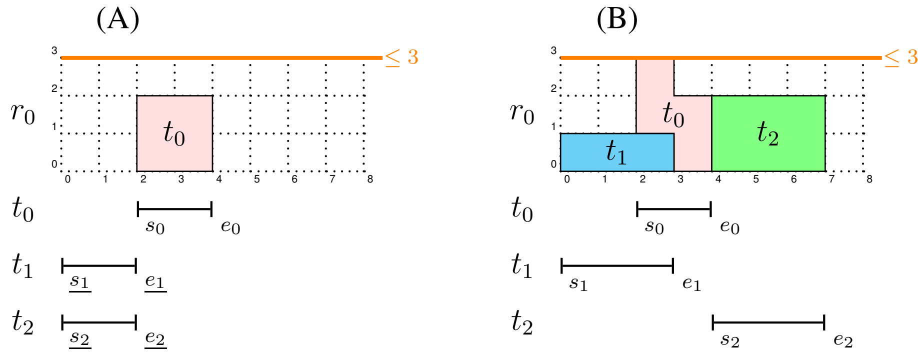


FIGURE 8.1 – Les parties (A) et (B) représentent respectivement la position au plus tôt de chacune des tâches ainsi que le PPO, (A) du problème initial décrit dans l'exemple 8.3, (B) la solution obtenue par l'algorithme glouton.

- L'évènement $\langle SFP, t, \underline{s_t} \rangle$ correspond au début d'une tâche fixée (i.e. une tâche pour laquelle $\underline{s_t} = \overline{s_t}$). Cet évènement est généré pour toutes les tâches fixées, initialement ou pendant le balayage. L'évènement de type SCP disparaît donc pour les tâches fixées.
- L'évènement $\langle EFP, t, \underline{s_t} \rangle$ correspond à la fin d'une tâche fixée. Cet évènement est généré pour toutes les tâches fixées, initialement ou pendant le balayage.

L'évènement de type RS associé à la date de fin au plus tôt d'une tâche t et servant à insérer ses successeurs n'est pas conservé. Le mode glouton fixant toutes les tâches en un seul balayage, le nouvel évènement EFP associé à la date de fin d'une tâche fixée remplace l'évènement RS. De plus, le type $ECPD$ n'est plus utile. En effet, comme nous le verrons par la suite, la droite de balayage ne pourra jamais atteindre la fin d'une partie obligatoire d'une tâche non fixée.

Exemple 8.2. (Évènements générés). *Pour l'instance introduite à l'exemple 8.1 page 88, les évènements suivants sont générés et insérés dans le tas h_events :* $\langle PR, 1, 0 \rangle$, $\langle PR, 2, 0 \rangle$, $\langle \mathbf{SFP}, \mathbf{0}, \mathbf{2} \rangle$, $\langle \mathbf{EFP}, \mathbf{0}, \mathbf{4} \rangle$, $\langle SCP, 1, 5 \rangle$, $\langle SCP, 2, 5 \rangle$. *Initialement, seule la tâche t_0 est fixée, les évènements SFP et EFP sont donc générés.*

8.3.2 État de la droite de balayage

Pour construire le PPO de chaque ressource et fixer les tâches, la droite de balayage maintient l'ensemble des données présentes dans l'algorithme de propagation classique, que nous rappelons brièvement ici :

- La position courante δ de la droite de balayage, initialement positionnée à la date du premier évènement.
- Pour chaque ressource $r \in [0, k-1]$, la quantité de ressource disponible au point de temps δ est donnée par gap_r et sa valeur précédente par gap'_r.
- Pour chaque tâche $t \in [0, n-1]$, $ring_t$ enregistre son statut. Notons que la signification des statuts $none$ et $ready$ est différente de celle de l'algorithme de propagation classique. Une tâche t a pour statut :
  - $none$ si et seulement si la droite de balayage n'a pas encore lu l'évènement PR de la tâche t ou si la tâche t est fixée.

  - $ready$ si la tâche t n'est pas encore fixée, mais on connaît la date à laquelle elle doit l'être,
  - $check$ si la tâche n'est pas dans l'un des états précédement donné et qu'il n'y a aucun conflit sur l'ensemble des ressources, i.e. pour chaque ressource, la consommation de la tâche t ne dépasse pas la quantité de ressource disponible par rapport au PPO,
  - $conflict_r$ si le statut de la tâche est différent de $none$ et $ready$ et qu'elle est en conflit sur la ressource r, i.e. la quantité de ressource gap_r est inférieure à la quantité requise par la tâche t sur la ressource r.

- Pour chaque tâche $t \in [0, n-1]$, $nbpred_t$ maintient le nombre de prédécesseurs de la tâche t pour lesquels la date de début au plus tôt n'a pas encore été trouvée à la position courante de la droite de balayage.

Pour gérer les retours arrières de la droite de balayage sur l'axe temporel lorsqu'une tâche est fixée à une date antérieure à sa position courante, on ajoute à l'état de la droite de balayage les éléments suivants :

- Une pile $trail$ qui stocke les informations nécessaires pour rétablir l'état de la droite de balayage à une date antérieure. Une information est présente dans la pile sous forme d'un couple $\langle tag, data\rangle$ où tag est l'un des quatre types d'évènement, i.e. PR, SCP, SFP et EFP, et $data$ est l'identifiant de la tâche, ou, $tag = DELTA$ et $data$ est dans ce cas un point de temps.
- Un ensemble $\mathcal{U}$ qui stocke l'ensemble des tâches restantes à fixer.
- Pour chaque tâche t, le booléen $efpread_t$ indique si l'évènement EFP associé à t a été lu. À cause des retours arrières de la droite de balayage, un même évènement EFP peut-être lu plusieurs fois pendant le déroulement de l'algorithme. Comme nous l'avons expliqué dans la description des évènements, l'évènement EFP d'une tâche t reprend le rôle tenu par le type RS dans l'algorithme de propagation classique, pour rappel, la ré-introduction de ses successeurs. Le booléen $efpread_t$ nous permet d'éviter de ré-introduire plusieurs fois les successeurs d'une même tâche.

L'algorithme glouton commence par générer les évènements des tâches n'ayant pas de prédécesseurs et les insère dans le tas h_events. Comme pour la version de propagation classique la droite de balayage avance d'évènement en évènement, mettant à jour les quantités de ressource disponible. Lorsque le début au plus tôt d'une tâche est trouvé, la droite de balayage est ré-initialisée à cette date grâce à la pile $trail$ et la tâche y est fixée. Lors de la réinitialisation de la droite de balayage d'une position donnée à une position antérieure, aucun ajustement ou filtrage effectué jusque là n'est remis en cause, les évènements lus entre ces deux positions sont ré-insérés dans le tas h_events et les quantités de ressource disponible sont rétablies. L'algorithme s'arrête lorsqu'un dépassement de ressource est constaté ou lorsque toutes les tâches ont été fixées, signifiant qu'une solution valide a été trouvée.

8.3.3 Algorithme glouton

L'algorithme greedy consiste en une boucle principale composée d'une partie chargée de réinitialiser la droite de balayage à un point de temps antérieur et fixer une tâche (algorithme assign), d'une autre pour le traitement des évènements (algorithme $\mathrm{process_events}$) et enfin, d'une partie dédiée au traitement des tâches actives (algorithme $\mathrm{process_active_tasks}$).

Algorithme principal

L'algorithme principal greedy page 91 est composé des parties suivantes :

- `[CRÉATION DES ÉVÈNEMENTS]` (ligne 2). Les évènements des tâches n'ayant aucun prédécesseur sont générés et insérés dans le tas h_events.
- `[INITIALISATION]` (lignes 4 à 10). Pour chaque ressource r $(0 \leq r < k)$ la ressource disponible gap_r et sa précédente valeur gap'_r sont initialisées à la capacité maximale correspondante $limit_r$. Qu'une tâche soit fixée ou non, son statut est initialisé à $none$. On ajoute à la pile $trail$ l'information $\langle DELTA, -\infty\rangle$ (ligne 10) indiquant le début de l'algorithme.
- `[BOUCLE PRINCIPALE]` (lignes 12 à 21). A chaque position, la droite de balayage observe si le début au plus tôt d'une tâche a été trouvé, signifiant que cette tâche est prête à être fixée (ligne 13). Sinon, les évènements associés à la position courante de la droite de balayage sont lus et traités par process_events, retournant **false** si une tâche ne peut plus être introduite dans sa fenêtre temporelle à cause de ses prédécesseurs. Enfin, l'appel de l'algorithme process_active_tasks traite les tâches dont le statut est $check$ ou à $conflict$ et retourne **false** si un dépassement de ressource est détecté.

ALGORITHM greedy() : boolean
1: `[CRÉATION DES ÉVÈNEMENTS]`
2: $h_events \leftarrow$ generation of events wrt $n,\ \underline{s_t},\ \overline{s_t},\ d_t,\ \underline{e_t}$ and the precedence constraints.
3: `[INITIALISATION]`
4: **for** $r = 0$ **to** $k-1$ **do**
5: $\quad gap_r, gap'_r \leftarrow limit_r$
6: **for** $t = 0$ **to** $n-1$ **do**
7: $\quad ring_t \leftarrow none$
8: $\quad$ **if** $\underline{s_t} \neq \overline{s_t}$ **then** $\mathcal{U} \leftarrow \mathcal{U} \cup \{t\}$
9: $\quad efpread_t \leftarrow$ **false**
10: add $\langle$DELTA$, -\infty\rangle$ to $trail$
11: `[BOUCLE PRINCIPALE]`
12: **while** $\neg$empty$(h_events) \vee \exists t.ring_t = ready$ **do**
13: $\quad$ **if** $\exists t.ring_t = ready$ **then**
14: $\quad\quad \langle \delta, \delta_{next}\rangle \leftarrow$ assign()
15: $\quad$ **else**
16: $\quad\quad \langle \delta, \delta_{next}, success\rangle \leftarrow$ process_events()
17: $\quad\quad$ **if** $\neg success$ **then**
18: $\quad\quad\quad$ **return false**
19: $\quad$ **if** $\neg$process_active_tasks(δ, δ_{next}) **then**
20: $\quad\quad$ **return false**
21: $\quad$ add $\langle$DELTA$, \delta\rangle$ to $trail$
22: **return true**

Algorithme 13: Retourne **true** si une solution a été trouvée par l'algorithme glouton, **false** sinon.

Traitement des évènements

Pour mettre à jour le statut des tâches, l'algorithme process_events page 97 lit et traite tous les évènements se trouvant à la date δ et détermine l'intervalle de balayage $[\delta, \delta_{next}[$. Lorsqu'un évènement est lu, une information est enregistrée dans la pile $trail$ pour permettre dans le cas d'un retour arrière de la droite de balayage de recalculer la capacité disponible sur chaque ressource. L'algorithme process_events est composé des partie suivantes :

- `[TRAITER LES ÉVÈNEMENTS DE DÉBUT DE TÂCHE (SFP)]` (lignes 3 à 5). Lorsqu'un évènement indiquant le début d'une tâche t est lu, on enregistre l'information correspondante $\langle SFP, t\rangle$ dans la pile $trail$ pour pouvoir rétablir l'état de la droite de balayage dans le cas d'un retour à une position antérieure. Puis l'espace disponible sur chaque ressource est diminué en fonction de la consommation de la tâche.
- `[TRAITER LES ÉVÈNEMENTS DE FIN DE TÂCHE (EFP)]` (lignes 7 à 17). Comme pour l'évènement SFP, l'information correspondante est enregistrée et l'espace disponible de chaque ressource diminué. Si cet évènement est lu pour la première fois ($\neg efpread_t$, ligne 10) la liste de successeurs est parcourue (ligne 12) et ceux pour lesquels tous les évènements de fin de leurs prédécesseurs ont été lus sont ajoutés au système (ligne 17).
- `[TRAITER LES ÉVÈNEMENTS DE DÉBUT DE PARTIE OBLIG. (SCP)]` (lignes 19 à 28). Lorsque l'évènement SCP d'une tâche non fixée t est lu par la droite de balayage, le statut de t est soit égal à $check$ soit $conflict$.
  - Si la tâche t est en conflit, alors la seule position réalisable est sa position au plus tard, débutant à la position courante de la droite de balayage. L'algorithme fix page 99 est alors appelé pour fixer la tâche, ajouter l'évènement SFP à la pile $trail$, ajouter l'évènement EFP au tas h_events et diminuer les espaces disponibles. Il n'y a plus rien à faire pour cet évènement, on traite donc le suivant.
  - Si la tâche t a pour statut $check$, l'appel de l'algorithme fix va modifier son statut à $ready$ indiquant que sa date de début est connue. Dans ce cas précis, la date de début de la tâche t est nécessairement antérieure à la position courante de la droite de balayage. La tâche n'est donc pas directement fixée puisqu'un retour arrière à cette position sera indispensable, elle est simplement étiquetée grâce à son nouveau statut $ready$ comme prête à être fixée.

  L'évènement SCP est enregistré dans la pile $trail$, puis si une partie obligatoire existe, l'espace disponible sur chaque ressource est mis à jour.
- `[DÉTERMINER LA DATE DU PROCHAIN ÉVÈNEMENT]` (ligne 30). Pour pouvoir traiter les évènements de type PR, il est nécessaire de connaitre la prochaine position de la droite de balayage.
- `[TRAITER LES ÉVÈNEMENTS DE DÉBUT AU PLUS TÔT (PR)]` (lignes 32 à 37). Pendant un premier parcours des évènements de début au plus tôt (lignes 32 à 33), on fixe les tâches pouvant se placer dans l'intervalle $[\delta, \delta_{next}[$ par rapport aux différents espaces disponibles sur les ressources. Pour les tâches n'ayant pas pu être fixées durant ce premier parcours, on enregistre l'information $\langle PR, t\rangle$ dans la pile $trail$ puis on détermine leur statut.

Traitement des tâches actives

L'algorithme process_active_tasks page 98 traite l'ensemble des tâches dont le statut est à $check$ ou à $conflict$ par rapport à l'intervalle de balayage courant $[\delta, \delta_{next}[$ et aux ressources disponibles gap_r $(0 < r \leq k)$. À la différence de l'algorithme filter_min page 72 de la version de propagation classique (i.e. non gloutonne), une première boucle tente de fixer des tâches directement sur l'intervalle courant.

- `[VÉRIFIER LE DÉPASSEMENT DE RESSOURCE]` (lignes 2 à 3). Si l'espace disponible sur une des ressources est négatif, l'algorithme retourne **false**, signifiant un échec de l'algorithme glouton.

- `[TÂCHES À FIXER SUR L'INTERVALLE COURANT]` (lignes 5 à 12). On parcourt chaque ressource r où la quantité de ressource disponible sur l'intervalle de balayage courant est inférieure à la quantité de ressource disponible sur l'intervalle précédent (i.e. $gap'_r < gap_r$, ligne 6). On considère chaque tâche t dont le statut est à $conflict$ et telle que sa consommation sur r est inférieure à l'espace disponible sur r, et telle que sa durée est inférieure à la taille de l'intervalle de balayage (ligne 7). Si la tâche est en conflit sur une des autres ressources r' (i.e. $\exists r'.h_{tr'} > gap_{r'}$, ligne 8), le statut de la tâche t est alors ajusté à $conflict_{r'}$, sinon, la tâche t est fixée sur l'intervalle courant (ligne 11 et 12). Dans ce dernier cas, l'algorithme fix est appelé pour fixer directement la tâche et diminuer les ressources disponibles en conséquence.

- `[TÂCHES SORTANT DE CONFLIT SUR LA RESSOURCE` r`]` (lignes 14 à 23). On parcourt une nouvelle fois chaque ressource r où la quantité de ressource disponible a augmenté par rapport au précédent intervalle de balayage. On considère chaque tâche t dont le statut est à $conflict_r$ telle que la hauteur est maintenant inférieure à la quantité de ressource disponible sur r.

  - S'il existe au moins une ressource r' sur laquelle la tâche t consomme plus que l'espace disponible correspondant, i.e. $\exists r'.h_{tr'} > gap_{r'}$, alors le statut de t est mis à $conflict_{r'}$.
  - Sinon, le début au plus tôt de la tâche est ajusté à la position courante de la droite de balayage, et son statut est mis à $check$. Pour ne pas perdre cet ajustement dans le cas d'un retour arrière de la droite de balayage, on enregistre l'information $\langle PR, t \rangle$ dans le $trail$.

- `[TÂCHES SORTANT DE CHECK]` (lignes 25 à 30). Pour terminer, on parcourt chaque ressource r où l'espace disponible a diminué entre la position précédente et la position courante de la droite de balayage. On considère les tâches dont le statut est à $check$ et dont la hauteur sur cette ressource r est maintenant supérieure à l'espace disponible.

  - Si la tâche considérée t est restée suffisamment longtemps avec le statut $check$, i.e. aucun conflit n'a été détecté sur l'intervalle $[\underline{s_t}, \underline{e_t}[$, l'algorithme fix va modifier son statut à $ready$, signifiant que la tâche est prête à être fixée.
  - Sinon, la tâche t est remise en conflit par rapport à la ressource r.

Réinitialisation du statut de la droite de balayage

Tant qu'il reste des tâches prêtes à être fixées, c'est-à-dire des tâches dont le statut est à $ready$ et nécessitant un retour arrière de la droite de balayage, cette dernière ne peut plus avancer sur l'axe du temps (algorithme 13, ligne 13). L'algorithme assign page 99 est appelé et procède en trois phases consécutives, (a) sélectionner une tâche t' à fixer à sa date $\underline{s_{t'}}$ parmi les candidates, (b) restaurer le statut de la droite de balayage à cette date et (c) fixer la tâche t'.

- `[SÉLECTIONNER LA TÂCHE À FIXER]` (ligne 2). On commence par sélectionner la tâche t' à fixer parmi l'ensemble des tâches candidates, i.e. les tâches t telles que $ring_t = ready$. On choisit de sélectionner la tâche ayant la date de début la plus petite.
- `[RESTAURER LES INFORMATIONS ENREGISTRÉES]` (lignes 4 à 23). Tant que la droite de balayage ne se trouve pas à la date de début de la tâche que l'on souhaite fixer, on continue de dépiler les informations enregistrées dans la pile $trail$. Suivant l'information extraite $\langle tag, t\rangle$ ligne 5 on applique les traitements suivants :
  - Si $tag = SFP$ (ligne 6) on réintroduit l'évènement correspondant $\langle SFP, t, \delta\rangle$ dans le tas h_events et on applique l'opération inverse de celle effectuée sur les différents gap_r lorsque cet évènement a été lu par la droite de balayage.
  - Si $tag = EFP$ (ligne 10), de manière analogue au cas précédent, on réintroduit l'évènement EFP dans h_events et on applique l'opération inverse sur les gap_r à celle effectuée lorsque cet évènement a été lu par la droite de balayage.
  - Si $tag = SCP$ (ligne 14) et que la tâche t à l'origine de cette information n'est pas fixée, on réintroduit l'évènement dans le tas h_events, et seulement si elle possède une partie obligatoire on met à jour les différents gap_r. Si la tâche est fixée alors cette information est ignorée.
  - Si $tag = PR$ (ligne 19), que la tâche t n'est pas encore fixée et que son statut est différent de $none$, l'évènement $\langle PR, t, \underline{s_t}\rangle$ est ajouté dans h_events et le statut de la tâche est passé à $none$. Lors du balayage une tâche t peut être ajustée plusieurs fois par l'algorithme process_active_tasks, ligne 21, ce qui ajoute dans ce cas plusieurs fois l'information $\langle PR, t\rangle$ au $trail$. Vérifier que le statut de la tâche t est différent de $none$ permet de ne conserver que le dernier ajustement fait sur le début au plus tôt de t.
  - Si $tag = DELTA$ (ligne 22), alors toutes les informations associées à la position courante δ de la droite de balayage ont été dépilées et traitées. La prochaine position δ_{next} de la droite de balayage est alors placée à sa position courante δ, et la position courante est fixée à la valeur t.
- `[FIXER LA TÂCHE` t'`]` (lignes 25 à 27). Une fois que les évènements ont été restaurés dans le tas h_events, que l'espace disponible sur chaque ressource et que la droite de balayage ont été ré-initialisés au point de temps $\underline{s_{t'}}$, on applique les modifications nécessaires pour fixer la tâche t', i.e. fixer ses variables de début et fin, ajouter les deux évènements correspondants dans le tas h_events et la retirer de l'ensemble des tâches non fixées.

Exemple 8.3. (Illustration de l'algorithme glouton). *Détaillons maintenant le déroulement de l'algorithme glouton sur l'exemple 8.1 page 88. La liste des évènements générés est donnée dans l'exemple 8.2 page 89.*

Initialisation. *L'algorithme glouton commence par initialiser la position courante de la droite de balayage à 0, gap_0 à 3. A part la tâche t_0 qui est fixée, toutes les tâches sont ajoutées à l'ensemble $\mathcal{U}$ (l'ensemble des tâches restant à fixer). Une première information est ajoutée a la pile $trail$, $\langle \mathrm{DELTA}, -\infty \rangle$, indiquant le début de la boucle principale.*

Lecture des évènements associés à l'instant 0 et filtrage par rapport à l'intervalle de balayage courant $[0, 2[$. *Le statut de chaque tâche étant à $none$, l'algorithme 14 (*process_events*) est appelé et lit les deux premiers évènements $\langle PR, 1, 0 \rangle$ et $\langle PR, 2, 0 \rangle$. L'intervalle de balayage étant $[0, 2[$, aucune des tâches t_1 et t_2 ne peut être directement fixées dans cet intervalle (algorithme 14, ligne 33). Les deux évènements PR sont ajoutés à la pile $trail$, le statut des tâches t_1 et t_2 est ajusté à $check$. Puisque $gap'_0 = gap_0$, l'algorithme 15 (*process_active_tasks*) chargé de traiter les tâches actives retourne simplement **true**. Enfin, l'information $\langle \mathrm{DELTA}, 0 \rangle$ est empilée dans $trail$, indiquant que tous les traitements ont été effectués sur cette position de la droite de balayage.*

Lecture de évènement associé à l'instant 2 et filtrage par rapport à l'intervalle de balayage courant $[2, 4[$. *La droite de balayage avance à la position 2, l'algorithme 14 lit l'évènement relatif au début de la tâche fixée t_0 (ligne 3), ajoute l'information $\langle SFP, 0 \rangle$ à la pile $trail$ et diminue l'espace disponible à 1. L'algorithme 15 est ensuite appelé sur l'intervalle $[2, 4[$ avec $gap_0 = 1$. La diminution de gap_0 modifie le statut de la tâche t_2 qui n'a plus assez d'espace de $check$ à $conflict_0$. Enfin, l'information $\langle \mathrm{DELTA}, 2 \rangle$ est enregistrée dans $trail$, indiquant que tous les traitements ont été effectués sur cette position de la droite de balayage.*

Lecture de l'évènement associé à l'instant 4 et filtrage par rapport à l'intervalle de balayage courant $[4, 5[$. *La droite de balayage avance à la postion 4, l'algorithme 14 lit l'évènement de fin de la tâche fixée t_0 (ligne 7), ajoute l'information $\langle EFP, 0 \rangle$ à la pile $trail$ et augmente l'espace disponible à 3. L'algorithme 15 est ensuite appelé sur l'intervalle $[4, 5[$ avec $gap_0 = 3$. L'augmentation de gap_0 provoque la modification du statut de la tâche t_2 de $conflict_0$ à $check$ (ligne 22), l'ajustement de la date de début de t_2 à la position courante de la droite de balayage (ligne 21), i.e. 4, et à l'enregistrement dans la pile $trail$ de l'information correspondante $\langle PR, 2 \rangle$ (ligne 20). Enfin l'information $\langle \mathrm{DELTA}, 4 \rangle$ est enregistrée dans $trail$.*

Lecture des évènements associés à l'instant 5. *La droite de balayage avance à la position 5, l'algorithme 14 et lit les évènements $\langle SCP, 2, 5 \rangle$ et $\langle SCP, 1, 5 \rangle$. Le statut des deux tâches t_1 et t_2 étant à $check$, l'algorithme 17 (*fix*) est appelé (ligne 25) et le statut des deux tâches est modifié à $ready$, indiquant que t_1 et t_2 sont prêtes à être fixées mais que leur date de début, antérieure à la position courante de la droite de balayage, nécessite un retour en arrière de celle-ci. Les informations $\langle SCP, 2 \rangle$ et $\langle SCP, 1 \rangle$ sont ajoutées à $trail$ et gap_0 est diminué de 2 à cause de la partie obligatoire de t_2. Enfin l'information $\langle \mathrm{DELTA}, 5 \rangle$ est enregistrée dans $trail$.*

Réinitialise la droite de balayage à la position 0 et fixe la tâche t_1. *Le statut des tâches t_1 et t_2 étant à $ready$ la droite de balayage ne progresse plus et l'algorithme 16 (*assign*) est appelé. La tâche t_1 est sélectionnée pour être fixée (algorithme 16, ligne 2 car sa date de début est inférieure à celle de t_2. La tâche t_1 devant être fixée à la date 0, i.e. la position initiale de la droite de balayage, tous les éléments de la pile $trail$ sont dépilés (ligne 4 à 23).*

Lors de ce retour arrière de la droite de balayage, les évènements suivants sont ajoutés à h_events : $\langle SFP, 0, 2\rangle$, $\langle EFP, 0, 4\rangle$, $\langle PR, 2, 4\rangle$, $\langle SCP, 1, 5\rangle$ et $\langle SCP, 2, 5\rangle$. Enfin, la tâche t_1 est fixée à sa date de début, sortie de l'ensemble des tâches restant à fixer $\mathcal{U}$ et les deux évènements $\langle SFP, 1, 0\rangle$ et $\langle EFP, 1, 3\rangle$ sont ajoutés à h_events.

Lecture de l'évènement associé à l'instant 0 et filtrage par rapport à l'intervalle de balayage courant $[0, 2[$. *La droite de balayage est maintenant replacée en 0 et l'algorithme 14 lit l'évènement $\langle SFP, 1, 0\rangle$. L'espace disponible est réduit de 2 et $\langle SFP, 1\rangle$ est empilé dans $trail$. Puis l'algorithme 15 est appelé sur l'intervalle de balayage $[0, 2[$ où rien ne peut être fait puisqu'il n'y a plus de tâche active.*

Lecture de l'évènement associé à l'instant 2 et filtrage par rapport à l'intervalle de balayage courant $[2, 3[$. *La droite de balayage avance à la position 2, l'algorithme 14 lit l'évènement de début de la tâche 2 $\langle SFP, 0, 2\rangle$, empile $\langle SFP, 2\rangle$ dans $trail$ et diminue l'espace disponible à 0. Il n'y a toujours pas de tâche active, l'algorithme 15 ne peut donc rien déduire.*

Lecture de l'évènement associé à l'instant 3 et filtrage par rapport à l'intervalle de balayage courant $[3, 4[$. *La droite de balayage avance à la position 3 où l'évènement de fin de la tâche t_1 est lu, $\langle EFP, 2\rangle$ est empilé dans $trail$ et l'espace disponible est augmenté à 2. Il n'y a toujours pas de tâche active.*

Lecture des évènements associés à l'instant 4 et filtrage par rapport à l'intervalle de balayage courant $[4, 5[$. *La droite de balayage avance à la position 4, l'algorithme 14 lit les évènements $\langle EFP, 0, 4\rangle$ et $\langle PR, 2, 4\rangle$. L'espace disponible est alors maximal, i.e. $gap_0 = 3$, et $\langle EFP, 0\rangle$ est empilé dans $trail$. L'évènement PR permet de positionner le statut de la tâche t_2 à $check$ (algorithme 14, ligne 37). L'appel de l'algorithme 15 n'entraine aucun changement sur le statut de la tâche t_2.*

Lecture des évènements associés à l'instant 5. *La droite de balayage avance à la position 5 et lit l'évènement SCP de la tâche t_2 et t_1. La tâche t_1 étant fixée son évènement SCP est ignoré. Le statut de la tâche t_2 étant à $check$ l'algorithme 17 est appelé, ligne 25. La date de début de t_2 (i.e. 4) étant située avant la position courante de la droite de balayage, son statut est passé à $ready$.*

Réinitialise de la droite de balayage à la position 4 et fixe la tâche t_2. *L'algorithme 16 est alors appelé (algorithme 13, ligne 14) pour fixer t_2 en 4. Les éléments suivants sont extraits de la pile $trail$, $\langle \mathrm{DELTA}, 5\rangle$, $\langle SCP, 2\rangle$, $\langle \mathrm{DELTA}, 4\rangle$, $\langle PR, 2\rangle$, $\langle EFP, 0\rangle$ et $\langle \mathrm{DELTA}, 3\rangle$ (lignes 4 à 23) entrainant la réintroduction des évènements $\langle EFP, 0, 4\rangle$ et $\langle SCP, 2, 5\rangle$ dans h_events. Toutes les tâches sont maintenant fixées et leur statut est à $none$.*

La droite de balayage va ensuite lire les évènements et construire le PPO (algorithme 14) et vérifier qu'aucun dépassement de ressource n'intervient (algorithme 15). La solution obtenue par l'algorithme glouton sur cette instance est illustrée par la figure 8.1, partie (B).

```
ALGORITHM process_events() : ⟨integer, integer, boolean⟩
 1: ⟨δ, ℰ⟩ ← extract_min(h_events)
 2: [TRAITER LES ÉVÈNEMENTS DE DÉBUT DE TÂCHE (SFP)]
 3: for all events of type ⟨SFP, t, s_t⟩ in ℰ do
 4:    add ⟨SFP, t⟩ to trail
 5:    for r = 0 to k − 1 do gap_r ← gap_r − h_tr
 6: [TRAITER LES ÉVÈNEMENTS DE FIN DE TÂCHE (EFP)]
 7: for all events of type ⟨EFP, t, e_t⟩ in ℰ do
 8:    add ⟨EFP, t⟩ to trail
 9:    for r = 0 to k − 1 do gap_r ← gap_r + h_tr
10:    if ¬efpread_t then
11:       efpread_t ← true
12:       for all t' ∈ succ_t do
13:          nbpred_t' ← nbpred_t' − 1
14:          if s̲_t' < δ then
15:             if δ > s̄_t' then return false
16:             (s̲_t', e̲_t') ← (δ, δ + d_t')
17:          if nbpreds_t' = 0 then add_task(t')
18: [TRAITER LES ÉVÈNEMENTS DE DÉBUT DE PARTIE OBLIG. (SCP)]
19: for all events of type ⟨SCP, t, s̄_t⟩ in ℰ | t ∈ 𝒰 do
20:    if ring_t = conflict_⋆ then
21:       (s̲_t, e̲_t) ← (s̄_t, ē_t)
22:       fix(t)
23:       continue for loop
24:    else if ring_t = check then
25:       fix(t)
26:    add ⟨SCP, t⟩ to trail
27:    if δ < e̲_t then
28:       for r = 0 to k − 1 do gap_r ← gap_r − h_tr
29: [DÉTERMINER LA DATE DU PROCHAIN ÉVÈNEMENT]
30: δ_next ← get_top_key(h_events)
31: [TRAITER LES ÉVÈNEMENTS DE DÉBUT AU PLUS TÔT (PR)]
32: for all events of type ⟨PR, t, s̲_t⟩ in ℰ | t ∈ 𝒰 do
33:    if ∀r.h_tr ≤ gap_r ∧ e̲_t ≤ δ_next then fix(t)
34: for all events of type ⟨PR, t, s̲_t⟩ in ℰ | t ∈ 𝒰 do
35:    add ⟨PR, t⟩ to trail
36:    if ∃r.h_tr > gap_r then ring_t ← conflict_r
37:    else ring_t ← check
38: return ⟨δ, δ_next, true⟩
```

Algorithme 14: Appelé à chaque fois que la droite de balayage se déplace. Extrait et traite tous les évènements associés à la position courante δ. Retourne la position courante δ et la prochaine position δ_{next} de la droite de balayage et un booléen. Ce booléen est à **false** si une tâche ne peut pas être introduite dans sa fenêtre temporelle, indiquant un échec de l'algorithme glouton.

```
ALGORITHM process_active_tasks(δ, δ_next) : boolean
 1: [VÉRIFIER LE DÉPASSEMENT DE RESSOURCE]
 2: for r = 0 to k − 1 do
 3:    if gap_r < 0 then return false
 4: [TÂCHES À FIXER SUR L'INTERVALLE COURANT]
 5: for r = 0 to k − 1 do
 6:    if gap'_r < gap_r then
 7:       for all t | ring_t = conflict_r ∧ h_tr ≤ gap_r ∧ δ + d_t ≤ δ_next do
 8:          if ∃r'.h_tr' > gap_r' then
 9:             ring_t ← conflict_r'
10:          else
11:             (s_t, e_t) ← (δ, δ + d_t)
12:             fix(t)
13: [TÂCHES SORTANT DE CONFLIT SUR LA RESSOURCE r]
14: for r = 0 to k − 1 do
15:    if gap'_r < gap_r then
16:       for all t | ring_t = conflict_r ∧ h_tr ≤ gap_r do
17:          if ∃r'.h_tr' > gap_r' then
18:             ring_t ← conflict_r'
19:          else
20:             add ⟨PR, t⟩ in trail
21:             (s_t, e_t) ← (δ, δ + d_t)
22:             ring_t ← check
23:       gap'_r ← gap_r
24: [TÂCHES SORTANT DE CHECK]
25: for r = 0 to k − 1 do
26:    if gap'_r > gap_r then
27:       for all t | ring_t = check ∧ h_tr > gap_r do
28:          if e_t ≤ δ then fix(t)
29:          else ring_t ← conflict_r
30:       gap'_r ← gap_r
31: return true
```

Algorithme 15: Appelé à chaque fois que la droite de balayage se déplace de δ à δ' pour (a) fixer les tâches qui ne sont plus en conflit par rapport à une ressource r, (b) modifier de $conflict_r$ à $conflict_{r'}$ le statut des tâches n'étant plus en conflit par rapport à r mais qui le devienne par rapport à r', (c) modifier de $check$ à $conflict_r$ les tâches qui n'étaient pas en conflit et qui le deviennent par rapport à r. Retourne **false** si un dépassement de ressource est détecté, **true** sinon.

ALGORITHM assign() : $\langle \text{integer}, \text{integer} \rangle$

1: [SÉLECTIONNER LA TÂCHE À FIXER]
2: $t' \leftarrow t \mid ring_t = ready \wedge \underline{s_t}$ is minimal
3: [RESTAURER LES INFORMATIONS ENREGISTRÉES]
4: **while** $\delta \geq \underline{s_{t'}}$ **do**
5:    extract $\langle tag, t \rangle$ from $trail$
6:    **if** $tag = SFP$ **then**
7:       add $\langle SFP, t, \delta \rangle$ to h_events
8:       **for** $r = 0$ **to** $k - 1$ **do**
9:          $gap_r \leftarrow gap_r + h_{tr}$
10:    **else if** $tag = EFP$ **then**
11:       add $\langle EFP, t, \delta \rangle$ to h_events
12:       **for** $r = 0$ **to** $k - 1$ **do**
13:          $gap_r \leftarrow gap_r - h_{tr}$
14:    **else if** $tag = SCP \wedge t \in \mathcal{U}$ **then**
15:       add $\langle SCP, t, \delta \rangle$ to h_events
16:       **if** $\delta < \underline{e_t}$ **then**
17:          **for** $r = 0$ **to** $k - 1$ **do**
18:             $gap_r \leftarrow gap_r + h_{tr}$
19:    **else if** $tag = PR \wedge t \in \mathcal{U} \wedge ring_t \neq none$ **then**
20:       add $\langle PR, t, \underline{s_t} \rangle$ to h_events
21:       $ring_t \leftarrow none$
22:    **else if** $tag = \text{DELTA} \wedge \delta > t$ **then**
23:       $(\delta_{next}, \delta) \leftarrow (\delta, t)$
24: [FIXER LA TÂCHE t']
25: $\mathcal{U} \leftarrow \mathcal{U} \setminus \{t'\}$
26: $(\overline{s_{t'}}, \overline{e_{t'}}) \leftarrow (\underline{s_{t'}}, \underline{e_{t'}})$
27: add $\langle SFP, t', \underline{s_{t'}} \rangle$ and $\langle EFP, t'\underline{e_{t'}} \rangle$ to h_events
28: **return** $\langle \delta, \delta_{next} \rangle$

Algorithme 16: Choisit une des tâches parmi l'ensemble de celles prêtes à être fixées, i.e. $ring_t = ready$, restaure la droite de balayage à sa date de début et fixe cette tâche.

ALGORITHM fix(t) : boolean

1: **if** $\underline{s_t} < \delta$ **then**
2:    $ring_t \leftarrow ready$
3:    **return false**
4: **else**
5:    $\mathcal{U} \leftarrow \mathcal{U} \setminus \{t\}$
6:    $ring_t \leftarrow none$
7:    $(\overline{s_t}, \overline{e_t}) \leftarrow (\underline{s_t}, \underline{e_t})$
8:    add $\langle SFP, t \rangle$ to $trail$
9:    add $\langle EFP, t, \underline{e_t} \rangle$ to h_events
10:    **for** $r = 0$ **to** $k - 1$ **do**
11:       $gap_r \leftarrow gap_r - h_{tr}$
12:    **return true**

Algorithme 17: Appelé pour fixer une tâche t à la volée. Retourne **true** si la tâche a pu être fixée directement, i.e. sa date de début n'est pas antérieure à la position courante de la droite de balayage, **false** sinon.

ALGORITHM add_task(t)
1: $ring_t \leftarrow none$
2: **if** $\underline{s_t} = \delta$ **then**
3:     **if** $\underline{s_t} = \overline{s_t}$ **then**
4:         $\mathcal{U} \leftarrow \mathcal{U} \setminus \{t\}$
5:         add $\langle SFP, t \rangle$ to $trail$
6:         **for** $r = 0$ **to** $k - 1$ **do**
7:             $gap_r \leftarrow gap_r - h_{tr}$
8:         add $\langle EFP, t, \underline{e_t} \rangle$ to h_events
9:     **else**
10:         $\mathcal{U} \leftarrow \mathcal{U} \cup \{t\}$
11:         add $\langle PR, t, \underline{s_t} \rangle$ to $\mathcal{E}$
12:         add $\langle SCP, t, \overline{s_t} \rangle$ to h_events
13: **else**
14:     **if** $\underline{s_t} = \overline{s_t}$ **then**
15:         $\mathcal{U} \leftarrow \mathcal{U} \setminus \{t\}$
16:         add $\langle SFP, t, \underline{s_t} \rangle$ to h_events
17:         add $\langle EFP, t, \overline{e_t} \rangle$ to h_events
18:     **else**
19:         $\mathcal{U} \leftarrow \mathcal{U} \cup \{t\}$
20:         add $\langle PR, t, \underline{s_t} \rangle$ to h_events
21:         add $\langle SCP, t, \overline{s_t} \rangle$ to h_events

Algorithme 18: Génère et ajoute les évènements relatifs à la tâche t, signifiant que tous ses prédécesseurs ont été fixés.

9

Évaluation

Sommaire

9.1 Présentation des problèmes sélectionnés 101
9.2 Instances aléatoires 102
9.3 Instances de la PSPLib 103
9.4 Redéploiement d'un réseau d'informations financières à grande échelle 107

9.1 Présentation des problèmes sélectionnés

Les différents algorithmes proposés dans cette thèse ont été implémentés (tels qu'ils ont été décrits) dans les solveurs de contraintes Choco [69] et SICStus [19]. Les évaluations ont été effectuées sur un processeur Intel Xeon à 2.93 GHz, une mémoire limitée à 14GB et un système d'exploitation Mac OS X 64 bits. Voici la liste les différents algorithmes et configurations que nous considérerons dans ce chapitre :

S L'algorithme de balayage original tel qu'il est décrit dans le chapitre 3.

UH L'algorithme de balayage dynamique tel qu'il est décrit dans le chapitre 5.

UR L'algorithme de balayage synchrone tel qu'il est décrit dans le chapitre 6, section 6.2, mais utilisé en mode mono-ressource et sans les précédences. Dans cette configuration on pose autant de contraintes *cumulative* qu'il y a de ressources et chaque précédence est encodée par une contrainte d'inégalité.

K L'algorithme de balayage synchrone tel qu'il est décrit dans le chapitre 6, section 6.2, utilisé en mode multi-ressources mais sans les précédences. Dans cette configuration on pose une seule contrainte *cumulative* pour modéliser les différentes ressources et autant de contraintes d'inégalité que de contraintes de précédence.

P L'algorithme de balayage synchrone tel qu'il est décrit dans le chapitre 6, section 6.2, utilisé en mode multi-ressources et avec les précédences. Dans cette configuration, le problème est entièrement modélisé par la pose d'une unique contrainte.

PG Le mode glouton tel qu'il est décrit au chapitre 8, section 8.3.

Pour évaluer nos différents algorithmes nous avons sélectionné trois problèmes différents.

- Des instances aléatoires, dont l'intérêt principal est d'observer l'évolution du comportement de nos différents algorithmes en fonction du nombre de tâches, de ressources et de précédences.
- Des instances de la PSPLib [56], qui constituent un des jeux d'instances de références en ordonnancement, elles nous permettent de confirmer les résultats atteints sur les instances aléatoires.
- Des instances issues du problème de redéploiement de réseau d'informations financières provenant d'un industriel tel qu'il est décrit chapitre 6, section 6.3 impliquant plusieurs milliers de tâches, plusieurs ressources cumulatives et une colorée ainsi que des précédences.

9.2 Instances aléatoires

Nous avons générés aléatoirement des instances dont le nombre de tâches varie de 1000 à 1024000, le nombre de ressources de 1 à 64, avec et sans précédence. Le taux de remplissage des instances décroit lorsque le nombre de ressources augmente, de 90% pour les instances n'ayant qu'une seule ressource à 70% pour les instances ayant 64 ressources. Cette réduction du taux de remplissage à mesure que le nombre de ressources en parallèle augmente s'impose pour éviter de générer des instances insatisfiables ou des instances trop difficiles pour une procédure arborescente construisant une solution à partir de zéro. La table 9.1 donne plus précisément le taux de remplissage utilisé pour la génération des instances sans précédence en fonction du nombre de ressources. Notons que ce tableau ne donne pas dans l'absolu les taux de remplissage maximum pour lesquels la méthode de Timetabling est suffisamment puissante pour trouver une solution sans retour arrière puisqu'un nombre important de paramètres entre en jeu comme la diversité des hauteurs et des durées des tâches, le nombre moyen de tâches pouvant se superposer à un point de temps donné...

# ressources	1	2	4	8	16	32	64
remplissage	90%	85%	80%	75%	75%	70%	70%

TABLE 9.1 – Taux de remplissage atteignables pour des instances multi-dimensionnelles sans précédence générées aléatoirement. Pour générer ces instances nous faisons varier la durée des tâches de 1 à 10 et leurs hauteurs de 1 à 5 suivant une loi uniforme. La capacité de chaque ressource est fixée à 15. A partir du taux de remplissage donné en entrée et de l'énergie moyenne d'une tâche (i.e. la durée moyenne multipliée par la hauteur moyenne), on calcule un horizon temporel sur lequel les tâches vont pouvoir se placer. La durée et les hauteurs des tâches sont ensuite générées aléatoirement suivant une loi uniforme et une correction est appliquée en fin de génération pour veiller à ce que le remplissage spécifié soit respecté à 2% près.

Comme heuristique de recherche nous choisissons la variable ayant la plus petite valeur dans son domaine, et pour cette variable, nous l'affectons à cette valeur (branchement à gauche) ou nous la retirons de son domaine (branchement à droite, i.e. la tâche est décalée à droite sans être fixée). Soulignons que notre approche ne permet pas de retrouver des

solutions générées de manière constructive avec un taux d'occupation de ressource proche de 100% dans le cadre de problèmes de très grande taille. Nous avons pu constater ce phénomène sur les instances issues du Challenge Roadef 2012. Notons cependant que l'objet du Challenge était différent : à partir d'une solution donnée, améliorer une fonction de coût complexe pour des problèmes impliquant près de 50000 items et 12 ressources.

Ces instances sont toutes résolues sans retour arrière mais avec un nombre de noeuds très proche de celui du nombre de tâches. Nous mesurons le temps nécessaire à l'obtention d'une première solution. Pour ces instances, nous avons utilisé Choco.

Dans un premier jeu d'instances (figure 9.1), nous comparons UH, UR, K et PG sur des instances aléatoires de bin-packing sans précédence. Nous remarquons que UR est uniformément 5% plus rapide que UH, confirmant l'hypothèse que la version utilisant la structure d'anneaux ($ring$) est plus performante que la version à base de tas ($h_conflict$,h_check). Une analyse préliminaire des temps obtenus pour la résolution exprimés en fonction de n et de k suggère que UR résout les instances en approximativement $O(kn^{2.1})$, alors que K les résout en approximativement $O(k^{0.25}n^{2.25})$. En d'autres termes, on observe une accélération d'environ $O(k^{0.75})$. Le schéma pour PG est quelque peu irrégulier, mais on observe que le temps d'exécution n'augmente que très peu lorsque k augmente et qu'il est de plusieurs ordres de magnitudes plus petit que pour K. PG est capable de résoudre des instances avec plus d'un million de tâches et 64 ressources.

Dans un deuxième jeu d'instances (figure 9.2), nous comparons UH, UR, K, P et PG sur des instances aléatoires de bin-packing avec précédences. On constate que les résultats pour UH, UR et K respectent le même schéma que le pour le premier jeu. En comparant K et P, on note que P est uniformément de 15% à 50% plus rapide que K, confirmant que le traitement simultané des contraintes cumulatives et des précédences est plus efficace. Comme pour PG et K dans le premier jeu d'instances sans précédence, le temps d'exécution de PG dans ce second jeu d'instances avec précédences est plusieurs ordres de magnitudes plus petit que P.

Dans un troisième jeu d'instances (figure 9.3), nous comparons les algorithmes UH, UR, K et PG sur des instances cumulatives (non bin-packing) sans précédence. L'analyse des temps d'exécutions en fonction de n et k amène aux mêmes résultats que pour le premier jeu d'instances, mais les temps d'exécutions sont environ 50% plus long.

Dans un quatrième jeu d'instances (figure 9.4), nous comparons les algorithmes UH, UR, K, P et PG sur des instances cumulatives (non bin-packing) avec précédences. L'analyse des temps d'exécutions en fonction de n et k est similaire au deuxième jeu d'instances, mais les temps d'exécutions sont environ deux fois plus long.

9.3 Instances de la PSPLib

Pour les instances de la PSPLib nous utilisons les jeux de données J30, J60, J90 et J120, impliquant 4 ressources, respectivement 30, 60, 90 et 120 tâches ainsi que des précédences. Le nombre moyen de successeurs est seulement de 1.5 par tâche. Pour ces instances nous avons utilisé SICStus Prolog. Nous avons comparé les algorithmes S, UH, K et P et encodé les instances de la manière suivante, dépendante de l'algorithme utilisé :

S et UH Nous posons quatre contraintes *cumulative*, chacune portant sur les tâches dont la hauteur est non-nulle sur la ressource correspondante, ce qui représente généralement 50% du nombre de tâches. Chaque précédence est encodée par une inégalité entre une variable de début et une variable de fin.

K Nous posons une contrainte k*-dimensional cumulative* portant sur toutes les tâches. Pour les tâches qui n'utilisent pas une ressource donnée, la valeur 0 lui est attribuée comme

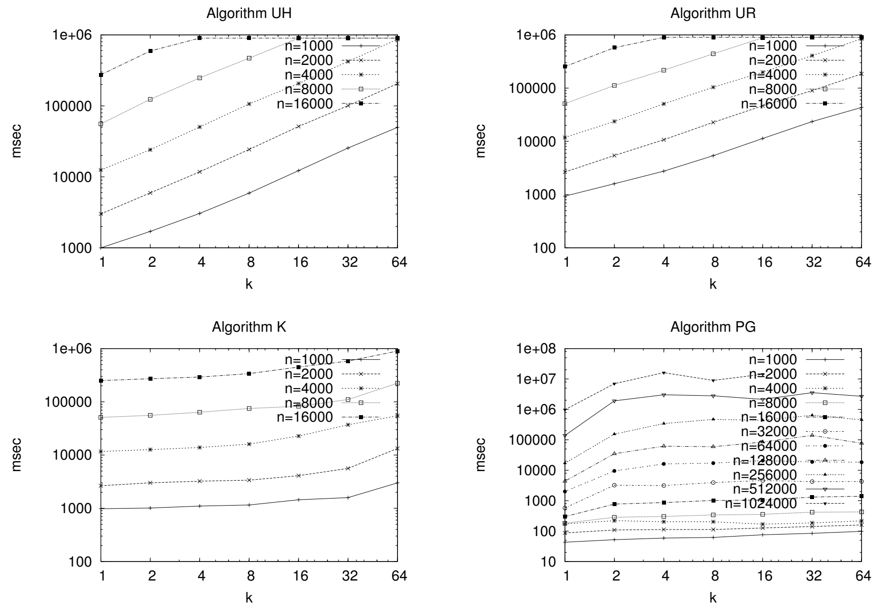


FIGURE 9.1 – Temps d'exécution en msec pour les instances aléatoires de bin-packing sans précédence.

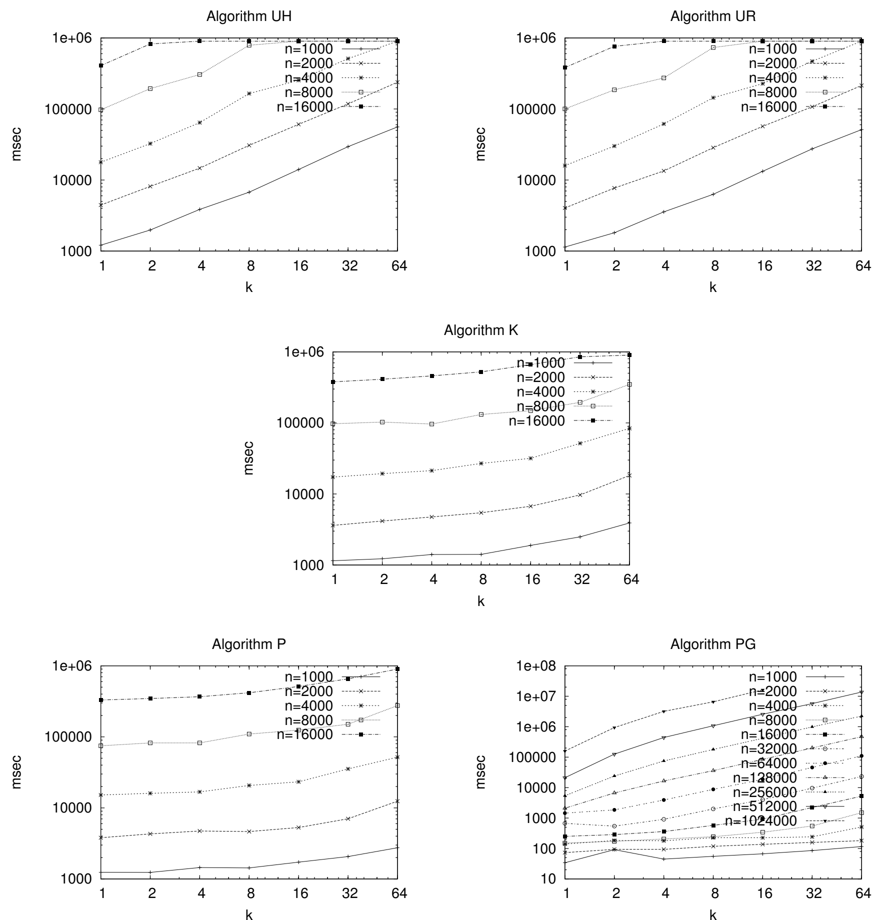


FIGURE 9.2 – Temps d'exécution en msec pour les instances aléatoires de bin-packing avec précédences.

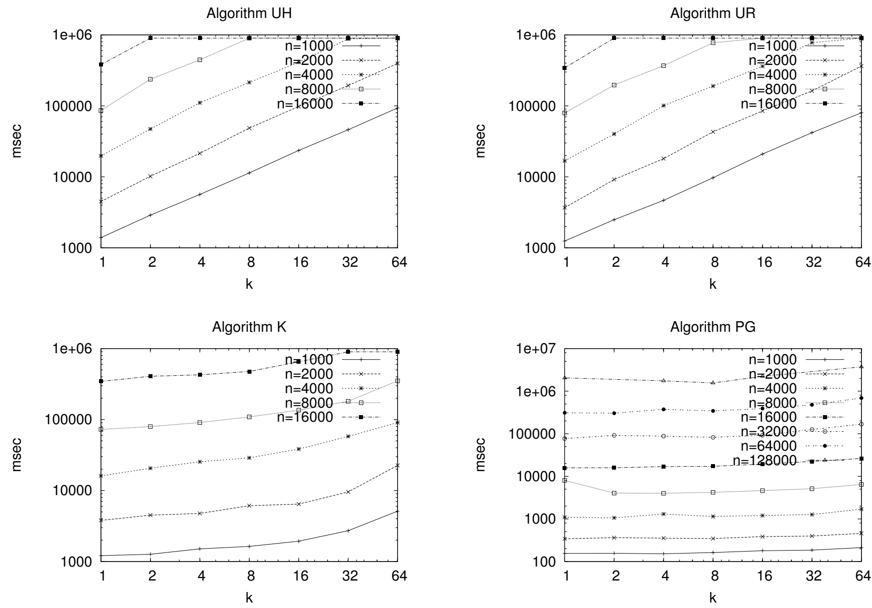


FIGURE 9.3 – Temps d'exécution en msec pour les instances cumulatives aléatoires sans précédence.

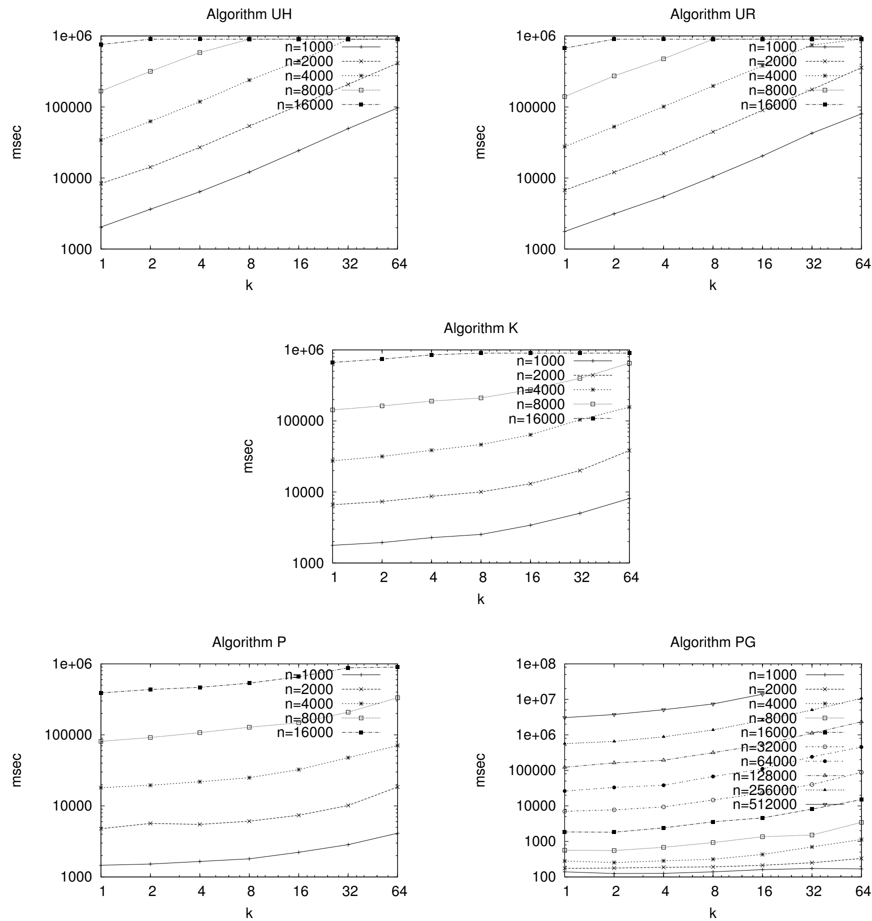


FIGURE 9.4 – Temps d'exécution en msec pour les instances cumulatives aléatoires avec précédences.

hauteur sur cette ressource. Chaque précédence est encodée par une inégalité entre une variable de début et une variable de fin.

P Nous posons une contrainte *k-dimensional cumulative with precedences* portant sur toutes les tâches et incluant les précédences.

Les domaines initiaux des variables de début correspondent à l'horizon de temps minimal si il est connu, ou à la meilleure borne supérieure sinon. Le temps est limité à 60 secondes par instance.

La procédure de recherche que nous avons utilisés se décompose en deux phases.

Phase une Les tâches sont ordonnées statiquement par ordre décroissant d'énergie, où l'énergie correspond à la durée de la tâche multipliée par la somme de ses hauteurs sur l'ensemble des ressources. La phase une sélectionne la première tâche t (i.e. celle possédant la plus grande énergie) ne possédant pas de partie obligatoire et restreint sa date de début s_t à l'intervalle $[\underline{s_t}, \underline{e_t} - 1]$. Cette restriction a pour effet de créer une partie obligatoire de longueur 1 à la tâche t et de la rendre insélectionnable par la phase une.

La phase une se termine lorsque toutes les tâches ont une partie obligatoire. La phase deux prend alors le relai.

Phase deux Jusqu'à ce que toutes les variables de début soient fixées :

① On sélectionne la tâche t ayant le plus petit début au plus tôt, les égalités sont cassées en choisissant la tâche disposant de la plus grande énergie.

② On coupe l'arbre de recherche en imposant $s_t \leq m$ dans le branchement à gauche et $s_t > m$ dans le branchement à droite, où $m = \lfloor (\underline{s_t} + \overline{s_t})/2 \rfloor$.

Gardons à l'esprit que l'arbre de recherche pour une instance donnée sera identique pour tous les algorithmes sauf S. Puisque S peut filtrer des valeurs à l'intérieur des domaines, il peut résoudre certaines instances en explorant moins de noeuds dans l'arbre de recherche.

Dans la table 9.2, nous présentons le nombre de retours arrières par seconde (bps) par jeu d'instances (J30, J60, J90 et J120) et par algorithme. Chaque ligne de la table donne, le nombre minimum, maximum, moyen et médian de bps pour les instances où ce nombre n'est pas nul et pour lesquelles le temps de résolution n'est pas trop proche de 0, ainsi que l'écart type et le nombre d'instances résolues en 60 secondes. Le nombre de bps reporté inclus aussi bien les instances résolues que non-résolues.

Des colonnes indiquant le nombre de bps moyen **moy** et le nombre de bps médian **med** on constate que l'algorithme S est plus lent que UH, lui même plus lent que K, lui même plus lent que P, bien que pour les jeux J90 et J120 il n'y ait pratiquement pas de différence entre K et P. Rappelons que la motivation de traiter les précédences directement dans l'algorithme de filtrage P était d'atteindre le point fixe plus rapidement que lorsqu'elles sont traitées en dehors. Nous supposons que c'est effectivement le cas pour J30 et J60 alors que pour J90 et J120 le gain semble plus faible.

La table 9.3 présente pour chaque jeu d'instances une comparaison par paire d'algorithmes. Chaque ligne d'un même jeu d'instances donne pour la paire x/y le nombre de bps de l'algorithme x divisé par le nombre de bps de l'algorithme y, excluant les instances dont le nombre de retours arrières ou le temps de résolution est nul. Cette table confirme les conclusions de la précédente, et montre que la plus forte hausse de performance vient du fait de traiter les k ressources cumulatives en une seule contrainte.

classe	#inst	alg	#résolues	min	max	moy	med	ecart type
J30	480	S	469	100.00	34040.00	3147.90	2827.78	3052.43
		UH	471	100.00	56733.33	4034.65	3200.00	5077.59
		K	474	1300.00	34040.00	9332.15	8212.50	5087.08
		P	475	100.00	34500.00	12437.60	11635.82	5806.21
J60	480	S	368	50.00	20600.00	1748.56	1533.91	1748.79
		UH	369	100.00	17430.77	2165.60	1914.70	1807.89
		K	374	100.00	14700.00	5650.81	5539.14	2859.14
		P	374	100.00	10622.64	5915.70	6503.00	2603.33
J90	480	S	293	50.00	18890.94	1457.08	994.43	1758.14
		UH	294	33.33	23923.77	2106.73	1437.25	2611.13
		K	296	50.00	9510.00	4184.08	4489.25	2298.52
		P	295	50.00	10568.43	4138.43	4648.68	2193.36
J120	600	S	91	50.00	12779.59	1617.27	877.87	1873.42
		UH	93	50.00	29948.11	3519.59	2117.19	4138.10
		K	95	33.33	12450.80	5239.94	5611.33	2016.51
		P	94	50.00	9455.08	5283.45	5681.04	1764.89

TABLE 9.2 – Résultats pour la PSPLib, par jeu d'instances et par algorithme (nombre de retours arrières par seconde).

Le rapports UH/K sont légèrement plus grand que le $4^{-0.75} = 0.35$ prédit par l'analyse de la figure 9.1. Nous supposons que cela est dû à l'abondance de tâches ayant une consommation nulle sur une ou plusieurs ressources dans les instances de la PSPLib, signifiant que chacune des contraintes *cumulative* mono-ressource ne traite qu'un sous-ensemble de tâches.

9.4 Redéploiement d'un réseau d'informations financières à grande échelle

Il s'agit d'un problème d'ordonnancement impliquant 8 ressources et jusqu'à 15000 tâches. La tableau d'utilisation des ressources par les tâches est très peu dense, seulement 12,5% des hauteurs sont non-nulles. Les résultats rapportés pour ce problème sont ceux obtenus sous SICStus.

Le problème correspond à celui décrit en 6.3.2, page 77. Le redéploiement du réseau consiste en une série de jobs devant être ordonnancés sur plusieurs années, chaque job étant constitué de plusieurs tâches, chacune requérant une certaines quantité de ressource. Des liens entre les tâches des différents jobs indiquent des dépendances dans le flux. La notion de couleur intervient dans ce cas pour limiter le nombre de pays impliqués à un instant donné. Les données sont des exemples de ce problème d'ordonnancement provenant d'un industriel.

Le point clé de cette application est que l'on ne souhaite pas résoudre le problème une seule fois, mais que l'on cherche à résoudre une série de scénarios où l'utilisateur doit pouvoir tester différentes possibilités. Cela signifie que la réponse au problème doit être aussi rapide que possible et par conséquent l'usage du mode glouton est crucial dans un tel cas.

Nous avons comparé les algorithmes S, UH, UR, K, P et PG dans les tables 9.4 et 9.5, en indiquant les temps d'exécutions et le nombre d'invocations des différents algorithmes. Les instances sont suffisamment simples (puisque les fenêtres temporelles sont assez large) pour être résolues sans retour arrière par tous les algorithmes. La stratégie de recherche utilisée est la même que celle décrite section 9.2.

Nous observons que S est plus lent que tous les algorithmes proposés dans cette thèse.

classes	#instances	algorithmes	min	max	moy	méd	écart type
J30	480	S/UH	0.25	2.00	0.91	0.91	0.28
		S/K	0.12	2.00	0.45	0.36	0.29
		S/P	0.08	1.80	0.32	0.28	0.23
		UH/K	0.14	2.00	0.51	0.42	0.31
		UH/P	0.09	3.00	0.39	0.30	0.35
		K/P	0.25	2.00	0.88	0.75	0.48
J60	480	S/UH	0.34	2.50	0.89	0.86	0.34
		S/K	0.10	2.55	0.40	0.32	0.29
		S/P	0.08	3.18	0.38	0.30	0.33
		UH/K	0.13	2.15	0.48	0.38	0.30
		UH/P	0.14	2.69	0.44	0.35	0.31
		K/P	0.39	2.00	0.96	0.95	0.30
J90	480	S/UH	0.25	3.00	0.78	0.70	0.36
		S/K	0.06	3.00	0.44	0.28	0.42
		S/P	0.06	2.47	0.43	0.28	0.40
		UH/K	0.12	4.00	0.57	0.38	0.56
		UH/P	0.12	4.00	0.57	0.37	0.54
		K/P	0.44	2.00	1.02	1.00	0.30
J120	600	S/UH	0.22	2.00	0.49	0.40	0.30
		S/K	0.05	3.00	0.36	0.18	0.41
		S/P	0.05	3.00	0.37	0.18	0.47
		UH/K	0.16	3.33	0.71	0.46	0.65
		UH/P	0.16	3.87	0.72	0.42	0.71
		K/P	0.33	3.00	1.01	1.00	0.23

TABLE 9.3 – Résultats pour la PSPLib, ratios par jeu d'instances et par paire d'algorithmes (nombre de retours arrières par seconde).

inst.	#tâches	#precedences	alg	temps	#invocations
A	8268	31538	S	46.01	8471
			UH	20.82	8300
			UR	14.97	8303
			K	24.23	8308
			P	30.00	8269
B	7628	26711	S	25.96	7794
			UH	14.54	7652
			UR	10.49	7655
			K	14.15	7660
			P	23.91	7629
C	7467	27055	S	24.36	7631
			UH	13.79	7493
			UR	9.79	7496
			K	13.47	7501
			P	22.63	7468
D	8024	28017	S	30.88	8196
			UH	16.45	8051
			UR	11.85	8054
			K	16.28	8059
			P	26.66	8025
E	6421	22895	S	15.41	6557
			UH	9.32	6440
			UR	6.54	6443
			K	9.55	6448
			P	16.35	6422
F	6347	22943	S	14.21	6459
			UH	9.00	6362
			UR	6.30	6365
			K	9.44	6370
			P	16.08	6348

TABLE 9.4 – Temps d'exécution en secondes pour les instances de l'application industrielle. Toutes ces instances sont résolues en moins de 2 secondes par l'algorithme PG.

UH est plus lent que UR confirmant les résultats précédents selon lesquels les structures anneaux sont dans notre cas meilleur que les tas. UR est plus rapide que K, ce qui semble dû au fait que la matrice d'utilisation des ressources par les tâches est très peu dense. Ce qui concorde également avec le nombre d'invocations des algorithmes de filtrage.

Enfin, la relative mauvaise performance de P par rapport à K peut également s'expliquer par le nombre total d'invocations. Nous pensons que la faible économie en terme de nombre d'invocations est dû au manque de densité du graphe de précédences. Cette économie est trop faible pour compenser le surcout lié au traitement des précédences dans P.

Toutes les instances présentées dans les tables 9.4 et 9.5 sont résolues en moins de 2 secondes par un unique appel de l'algorithme glouton. Nous avons également pu tester d'autres instances plus contraintes, où la combinaison de la recherche arborescente et du mode glouton sont nécessaires pour trouver une solution. Pour conclure, par la taille de ses instances et la demande de vitesse, cette application démontre l'utilité du mode glouton.

inst.	#tâches	#precedences	alg	temps	#invocations
G	14337	53218	S	115.77	14734
			UH	62.75	14403
			UR	51.65	14406
			K	57.90	14411
			P	99.83	14338
H	11354	41776	S	73.91	11638
			UH	37.56	11402
			UR	28.82	11405
			K	36.28	11410
			P	59.61	11355
I	13348	50311	S	105.46	13669
			UH	54.38	13405
			UR	42.76	13408
			K	50.45	13413
			P	85.77	13349
J	15351	59917	S	131.91	15746
			UH	73.52	15422
			UR	54.75	15425
			K	76.49	15430
			P	115.94	15352
K	14945	62541	S	121.25	15318
			UH	67.37	15008
			UR	51.53	15011
			K	69.00	15016
			P	113.85	14946

TABLE 9.5 – Temps d'exécution en secondes pour les instances de l'application industrielle. Toutes ces instances sont résolues en moins de 2 secondes par l'algorithme PG.

10

Conclusion

Au cours de cette thèse nous avons introduit de nouveaux algorithmes de filtrage pour les problèmes d'ordonnancement cumulatif dont l'objectif principal est le passage à l'échelle dans le cas de problème de grande taille. Nous avons fait le choix de partir d'une analyse des faiblesses de l'algorithme de balayage introduit en 2001 dans [8] et de proposer une solution pour chacune d'elles. En effet les algorithmes de balayage présentent l'avantage d'une mise en oeuvre simple, d'une gestion incrémentale de l'état associé à la droite de balayage permettant bien souvent de ne pas atteindre en pratique la complexité dans le pire des cas comme c'est le cas d'un certain nombre d'algorithmes plus sophistiqués. Enfin les algorithmes de balayage permettent naturellement un partage de structures de données entre les contraintes (i.e., les évènements).

Dans un premier temps, nous avons montré que la vitesse de convergence au point fixe de l'algorithme de balayage pour une seule contrainte *cumulative* pouvait être améliorée en y apportant un caractère *dynamique*. Considérer l'augmentation du profil des parties obligatoires à la volée durant un seul et même balayage est en effet l'élément majeur de cette amélioration.

Dans un second temps, nous exploitons l'idée de *propagation synchronisée* sur une conjonction de contraintes *cumulative*, *coloured_cumulative* et *précédence*, afin d'améliorer la vitesse de convergence au point fixe global. Cette idée tranche avec la manière « traditionnelle » qui consiste à propager chaque contrainte indépendamment des autres. L'idée n'est pas d'utiliser des algorithmes de propagation sophistiqués effectuant plus de déductions en considérant une conjonction de contraintes dans sa globalité, mais d'effectuer la propagation plus rapidement de manière à obtenir un meilleur passage à l'échelle selon le nombre de tâches et le nombre de ressources. Par exemple les évènements associés aux tâches ne sont générés et triés qu'une seule fois au lieu d'être regénérés et triés encore et encore au niveau de chaque contrainte *cumulative*. Une autre économie vient du fait que lorsqu'une tâche est en conflit par rapport à une ressource cumulative donnée on ne perd pas de temps à la traiter par rapport aux autres ressources cumulatives.

De chacun de ces nouveaux algorithmes de propagation nous avons dérivé un mode glouton qui entrelace la propagation de contraintes et le fait de fixer les tâches, ce qui nous a permis d'augmenter encore la taille des instances traitées. De plus, ce mode glouton, certes non composable avec d'autre contraintes, permet cependant de modéliser entièrement des problèmes issus d'applications réelles tels que le redéploiement de réseau d'informations

financières, le placement de machines virtuelles dans les centres de données virtualisés ou encore le problème de steel mill slab.

Nous avons pu valider nos différents algorithmes de filtrage sur plusieurs types de problèmes et montrer une réelle amélioration quant à la taille des instances traitées. Néanmoins, un certain nombre d'améliorations semblent encore possibles et cette nouvelle approche de *propagation synchronisée* explorée ici dans le cadre de problèmes d'ordonnancement ouvre quelques pistes que nous donnons ci-dessous.

- Il n'est pas rare dans certains problèmes d'ordonnancement que les tâches possèdent des durées et hauteurs variables. Il serait donc intéressant de voir si nos différents algorithmes peuvent être modifiés afin de proposer quelques règles de filtrage pour ces variables. De manière un peu plus générale, il serait intéressant de se pencher sur la contrainte *cumulatives* telle qu'elle a été décrite par Beldiceanu et Carlsson dans [8]. Cette contrainte intègre une dimension d'affectation et des hauteurs négatives pour les tâches.

- L'algorithme glouton que nous proposons dans cette thèse mêle la propagation des contraintes et la prise de décisions (lorsqu'il décide de fixer une tâche). Suivant le critère à optimiser (e.g. horizon de temps), il pourrait être judicieux pour l'utilisateur de pouvoir préciser une heuristique guidant les décisions prises par l'algorithme glouton.

- Nous pensons que le principe de *propagation synchronisée* pourrait être étendu et formalisé à un cadre plus général que celui de l'ordonnancement. Adapter les algorithmes de filtrage d'autres contraintes à ce principe permettrait certainement d'obtenir des améliorations significatives quant à la vitesse de convergence au point fixe.

Annexe

Algorithme de filtrage pour la contrainte *multiSumColorPrecCumulative*

Nous donnons dans cette section le code détaillé de l'algorithme de filtrage pour la contrainte *multiSumColorPrecCumulative* tel qu'il est décrit page 78, section 6.3.4. Il s'agit d'une extension de l'algorithme présenté page 55 section 6.2, intégrant la contrainte *cumulative colorée*. La structure des ces deux algorithmes est très proche, nous noterons cependant que l'action de mise à jour de l'espace disponible par rapport au début ou à la fin d'une partie obligatoire a été isolée dans l'algorithme 23 page 118.

ALGORITHM sweep_min() : boolean

1: [CRÉATION DES ÉVÈNEMENTS]
2: $h_events \leftarrow$ generation of events wrt $n,\ \underline{s_t},\ \overline{s_t},\ d_t,\ \underline{e_t}$ and the precedence constraints.
3: [INITIALISATION]
4: **for** $r = 0$ **to** $k-1$ **do**
5:     **if** $type_r = sum$ **then**
6:         $gap_r, gap'_r \leftarrow limit_r$
7:     **else if** $type_r = colour$ **then**
8:         $nbDistinctColors_r \leftarrow 0$
9:         $countColors_{r,0} \leftarrow 1$
10:         **for** $c = 1$ **to** $cmax_r$ **do**
11:             $countColors_{r,c} \leftarrow 0$
12: **for** $t = 0$ **to** $n-1$ **do**
13:     **if** $\underline{s_t} = \overline{s_t}$ **then** $ring_t \leftarrow$ ready **else** $ring_t \leftarrow$ none
14: [BOUCLE PRINCIPALE]
15: **while** $\neg$empty(h_events) **do**
16:     $\langle \delta, \delta_{next}, success \rangle \leftarrow$ process_events()
17:     **if** $\neg success$ **then**
18:         **return false**
19:     **if** $\neg$filter_min(δ, δ_{next}) **then**
20:         **return false**
21: **return true**

Algorithme 19: Retourne **false** si un dépassement de ressource est détecté ou si une contrainte de précédence ne peut pas être satisfaite. Retourne **true** sinon.

ALGORITHM process_events() : $\langle$integer, integer$\rangle$

1: $\langle\delta, \mathcal{E}\rangle \leftarrow$ extract and record in $\mathcal{E}$ all the events in h_events related to the minimal date δ
2: `[TRAITER LES ÉVÈNEMENTS DE DÉBUT DE PARTIE OBLIG. (SCP)]`
3: **for all** events of type $\langle SCP, t, \overline{s_t}\rangle$ in $\mathcal{E}$ **do**
4: $\quad ecp' \leftarrow \underline{e_t}$
5: $\quad$ **if** $ring_t =$ conflict$_\star$ **then**
6: $\quad\quad$ adjust_min_var$(s_t, \overline{s_t})$; adjust_min_var$(e_t, \overline{e_t})$
7: $\quad\quad ring_t \leftarrow$ ready
8: $\quad$ **else if** $ring_t =$ check **then**
9: $\quad\quad ring_t \leftarrow$ ready
10: $\quad$ **if** $\delta < \underline{e_t}$ **then**
11: $\quad\quad$ update_gaps$(t,$ **true**$)$
12: $\quad\quad$ **if** $ecp' \leq \delta$ **then**
13: $\quad\quad\quad$ add $\langle ECPD, t, \underline{e_t}\rangle$ to h_events
14: `[TRAITER LES ÉVÈNEMENTS DYNAMIQUES (ECPD)]`
15: **for all** events of type $\langle ECPD, t, \underline{e_t}\rangle$ in $\mathcal{E}$ **do**
16: $\quad$ **if** $\underline{e_t} > \delta$ **then**
17: $\quad\quad$ add $\langle ECPD, t, \underline{e_t}\rangle$ to h_events
18: $\quad$ **else**
19: $\quad\quad$ update_gaps$(t,$ **false**$)$
20: `[TRAITER LES ÉVÈNEMENTS DE FIN AU PLUS TÔT (RS)]`
21: **for all** events of type $\langle RS, t, \underline{e_t}\rangle$ in $\mathcal{E}$ **do**
22: $\quad$ **if** $ring_t =$ conflict$_\star$ **then**
23: $\quad\quad$ add $\langle RS, t, \delta + d_t\rangle$ to h_events
24: $\quad$ **else if** $\delta \neq \underline{e_t}$ **then**
25: $\quad\quad$ add $\langle RS, t, \underline{e_t}\rangle$ to h_events
26: $\quad$ **else**
27: $\quad\quad$ **for all** $t' \in successors_t$ **do**
28: $\quad\quad\quad nbpred_{t'} \leftarrow nbpred_{t'} - 1$
29: $\quad\quad\quad$ **if** $nbpred_{t'} = 0$ **then**
30: $\quad\quad\quad\quad$ **if** $\neg$release_task$(t', \delta, \mathcal{E})$ **then return false**
31: `[DÉTERMINER LA DATE DU PROCHAIN ÉVÈNEMENT]`
32: $\delta_{next} \leftarrow$ get_top_key(h_events)
33: `[TRAITER LES ÉVÈNEMENTS DE DÉBUT AU PLUS TÔT (PR)]`
34: **for all** events of type $\langle PR, t, \underline{s_t}\rangle$ in $\mathcal{E}$ **do**
35: $\quad$ **if** $(type_r = sum \wedge \exists r \mid h_{t,r} > gap_r) \vee (type_r = colour \wedge \exists r \mid (nbDistinctColors_r = limit_r \wedge countColors_{r,c_{t,r}} = 0))$ **then**
36: $\quad\quad ring_t \leftarrow$ conflict$_r$
37: $\quad$ **else if** $\underline{e_t} > \delta_{next}$ **then**
38: $\quad\quad ring_t \leftarrow$ check
39: $\quad$ **else**
40: $\quad\quad ring_t \leftarrow$ ready
41: **return** $\langle\delta, \delta_{next}\rangle$

Algorithme 20: Appelé à chaque fois que la droite de balayage se déplace. Extrait et traite tous les évènements associés au point de temps δ. Retourne la position courante δ, la prochaine position δ_{next} de la droite de balayage et un booléen indiquant si une erreur est détectée.

ALGORITHM release_task$(t, \delta, \mathcal{E})$: boolean
1: [VÉRIFIER LE NOUVEAU DÉBUT AU PLUS TÔT]
2: **if** $\neg$adjust_min_var$(s_t, \delta) \vee \neg$adjust_min_var$(e_t, \delta + d_t)$ **then**
3:    **return false**
4: [LE DÉBUT AU PLUS TÔT DE LA TÂCHE t EST AJOUTÉ À δ]
5: **if** $\underline{s_t} = \delta$ **then**
6:    **if** $\underline{s_t} = \overline{s_t}$ **then**
7:       update_gaps$(t,$ **true**$)$
8:       $ring_t \leftarrow$ ready
9:    **else**
10:       add $\langle PR, t, \underline{s_t} \rangle$ to $\mathcal{E}$
11:       add $\langle SCP, t, \overline{s_t} \rangle$ to h_events
12:    **if** $\overline{s_t} < \underline{e_t}$ **then**
13:       add $\langle ECPD, t, \underline{e_t} \rangle$ to h_events
14:    **if** task t has a least one successor **then**
15:       add $\langle RS, t, \underline{e_t} \rangle$ to h_events
16: [LE DÉBUT AU PLUS TÔT DE LA TÂCHE t EST AJOUTÉ APRÈS δ]
17: **else**
18:    add $\langle SCP, t, \overline{s_t} \rangle$ to h_events
19:    **if** $\overline{s_t} < \underline{e_t}$ **then**
20:       add $\langle ECPD, t, \underline{e_t} \rangle$ to h_events
21:    **if** $\underline{s_t} < \overline{s_t}$ **then**
22:       add $\langle PR, t, \underline{s_t} \rangle$ to h_events
23:    **else**
24:       $ring_t \leftarrow$ ready
25:    **if** task t has at least one successor **then**
26:       add $\langle RS, t, \underline{e_t} \rangle$ to h_events
27: **return true**

Algorithme 21: Génère et ajoute les évènement associés à la tâche t, signifiant que tous ses prédécesseurs ont atteint leur point fixe. Retourne **false** si δ a dépassé la date de début au plus tard de la tâche t.

ALGORITHM filter_min() : boolean

1: [VÉRIFIER LE DÉPASSEMENT DE RESSOURCE]
2: **for** $r = 0$ **to** $k - 1$ **do**
3:   **if** $type_r = sum$ **then**
4:     **if** $gap_r < 0$ **then return false**
5:   **else if** $type_r = colour$ **then**
6:     **if** $nbDistinctColors_r > limit_r$ **then return false**
7: [TÂCHES SORTANT DE CHECK]
8: **for** $r = 0$ **to** $k - 1$ **do**
9:   **if** $type_r = sum$ **then**
10:     **if** $gap'_r > gap_r$ **then**
11:       **for all** $t \mid ring_t = \text{check} \wedge h_{t,r} > gap_r$ **do**
12:         $ring_t \leftarrow$ **if** $\underline{e_t} > \delta$ **then** conflict$_r$ **else** ready
13:       $gap'_r \leftarrow gap_r$
14:   **else**
15:     **if** $nbDistinctColors_r = limit_r$ **then**
16:       **for all** $t \mid ring_t = \text{check} \wedge countColors_{r,c_{t,r}} = 0$ **do**
17:         $ring_t \leftarrow$ **if** $\underline{e_t} > \delta$ **then** conflict$_r$ **else** ready
18: [TÂCHES SORTANT DE CONFLIT SUR LA RESSOURCE r]
19: **for** $r = 0$ **to** $k - 1$ **do**
20:   **if** $type_r = sum$ **then**
21:     **if** $gap'_r < gap_r$ **then**
22:       **for all** $t \mid ring_t = \text{conflict}_r \wedge h_{t,r} \leq gap_r$ **do**
23:         **if** $\exists r'.h_{t,r'} > gap_{r'}$ **then**
24:           $ring_t \leftarrow \text{conflict}_{r'}$
25:         **else**
26:           $ecp' \leftarrow \underline{e_t}$
27:           adjust_min_var(s_t, δ) ; adjust_min_var($e_t, \delta + d_t$) ;
28:           $ring_t \leftarrow$ **if** $\underline{e_t} > \delta_{next}$ **then** check **else** ready
29:           **if** $\overline{s_t} \geq ecp' \wedge \overline{s_t} < \underline{e_t}$ **then**
30:             add $\langle ECPD, t, \underline{e_t} \rangle$ to h_events
31:       $gap'_r \leftarrow gap_r$
32:   **else**
33:     $b \leftarrow (nbDistinctColors_r < limit_r)$
34:     **for all** $t \mid ring_t = \text{conflict}_r \wedge (b \vee countColors_{r,c_{t,r}} \neq 0)$ **do**
35:       **if** $\exists r' \mid (nbDistinctColors_{r'} = limit_{r'} \wedge countColors_{r',c_{t,r'}} = 0)$ **then**
36:         $ring_t \leftarrow \text{conflict}_{r'}$
37:       **else**
38:         $ecp' \leftarrow \underline{e_t}$
39:         adjust_min_var(s_t, δ) ; adjust_min_var($e_t, \delta + d_t$) ;
40:         $ring_t \leftarrow$ **if** $\underline{e_t} > \delta_{next}$ **then** check **else** ready
41:         **if** $\overline{s_t} \geq ecp' \wedge \overline{s_t} < \underline{e_t}$ **then**
42:           add $\langle ECPD, t, \underline{e_t} \rangle$ to h_events
43: **return true**

Algorithme 22: Appelé à chaque fois que la droite de balayage se déplace de δ à δ_{next} pour ajuster le début au plus tôt des tâches par rapport à l'espace disponible sur chaque ressource.

ALGORITHM update_gaps(t, $decrease$)

1: **if** $decrease$ **then**
2: **for** $r = 0$ **to** $k - 1$ **do**
3: **if** $type_r = sum$ **then**
4: $gap_r \leftarrow gap_r - h_{t,r}$
5: **else if** $type_r = colour$ **then**
6: **if** $c_{t,r} \neq 0$ **then**
7: **if** $countColors_{r,c_{t,r}} = 0$ **then**
8: $nbDistinctColors_r \leftarrow nbDistinctColors_r + 1$
9: $countColors_{r,c_{t,r}} \leftarrow countColors_{r,c_{t,r}} + 1$
10: **else if** $\neg decrease$ **then**
11: **for** $r = 0$ **to** $k - 1$ **do**
12: **if** $type_r = sum$ **then**
13: $gap_r \leftarrow gap_r + h_{t,r}$
14: **else if** $type_r = colour$ **then**
15: **if** $c_{t,r} \neq 0$ **then**
16: **if** $countColors_{r,c_{t,r}} = 1$ **then**
17: $nbDistinctColors_r \leftarrow nbDistinctColors_r - 1$
18: $countColors_{r,c_{t,r}} \leftarrow countColors_{r,c_{t,r}} - 1$

Algorithme 23: Appelé pour mettre à jour l'espace disponible sur l'ensemble des ressources, par rapport à une tâche donnée t. Le booléen $decrease$ est à **true** s'il s'agit du début de la partie obligatoire de t, **false** sinon.

Liste des tableaux

3.1 Les types d'évènements de l'algorithme de balayage avec la condition nécessaire à leur génération. Le dernier attribut, *hauteur*, n'est pertinent que pour les évènements du type SCP et ECP. 20

5.1 Liste des différents types d'évènements avec la condition nécessaire à leur génération. Le dernier attribut, *hauteur*, n'est pertinent que pour les évènements du type SCP, ECP et $ECPD$. 33

9.1 Taux de remplissage atteignables pour des instances multi-dimensionnelles sans précédence générées aléatoirement. Pour générer ces instances nous faisons varier la durée des tâches de 1 à 10 et leurs hauteurs de 1 à 5 suivant une loi uniforme. La capacité de chaque ressource est fixée à 15. A partir du taux de remplissage donné en entrée et de l'énergie moyenne d'une tâche (i.e. la durée moyenne multipliée par la hauteur moyenne), on calcule un horizon temporel sur lequel les tâches vont pouvoir se placer. La durée et les hauteurs des tâches sont ensuite générées aléatoirement suivant une loi uniforme et une correction est appliquée en fin de génération pour veiller à ce que le remplissage spécifié soit respecté à 2% près. 102

9.2 Résultats pour la PSPLib, par jeu d'instances et par algorithme (nombre de retours arrières par seconde). 107

9.3 Résultats pour la PSPLib, ratios par jeu d'instances et par paire d'algorithmes (nombre de retours arrières par seconde). 108

9.4 Temps d'exécution en secondes pour les instances de l'application industrielle. Toutes ces instances sont résolues en moins de 2 secondes par l'algorithme PG. 109

9.5 Temps d'exécution en secondes pour les instances de l'application industrielle. Toutes ces instances sont résolues en moins de 2 secondes par l'algorithme PG. 110

Table des figures

3.1 Les parties (A), (B) et (C) représentent respectivement la position au plus tôt de chacune des tâches ainsi que le PPO, du problème initial décrit dans l'exemple 3.1, après un premier balayage et après un second balayage. . . . 19

5.1 État des tâches, de la droite de balayage et des tas après la lecture des évènements $\langle PR, 1, 0, 0\rangle$, $\langle PR, 2, 0, 0\rangle$, $\langle PR, 3, 0, 0\rangle$ et $\langle PR, 4, 0\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 16) - (A) date de début et consommations de chaque tâche, (B) le PPO et la position de la droite de balayage, (C) la liste des évènements restant à traiter ainsi que les tas h_check et $h_conflict$. 40
5.2 État des tâches, de la droite de balayage et des tas après la lecture de l'évènement $\langle SCP, 0, 1, -2\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 16) - (A) date de début et consommations de chaque tâche, (B) le PPO et la position de la droite de balayage, (C) la liste des évènements restant à traiter ainsi que les tas h_check et $h_conflict$. 41
5.3 État des tâches, de la droite de balayage et des tas après la lecture de l'évènement $\langle ECPD, 0, 2, 2\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 16) - (A) date de début et consommations de chaque tâche, (B) le PPO et la position de la droite de balayage, (C) la liste des évènements restant à traiter ainsi que les tas h_check et $h_conflict$. 42
5.4 État des tâches, de la droite de balayage et des tas après la lecture des évènements $\langle CCP, 1, 3, 0\rangle$ et $\langle SCP, 1, 3, -2\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 16) - (A) date de début et consommations de chaque tâche, (B) le PPO et la position de la droite de balayage, (C) la liste des évènements restant à traiter ainsi que les tas h_check et $h_conflict$. . . 43
5.5 État des tâches, de la droite de balayage et des tas après la lecture de l'évènement $\langle ECPD, 1, 4, 2\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 16) - (A) date de début et consommations de chaque tâche, (B) le PPO et la position de la droite de balayage, (C) la liste des évènements restant à traiter ainsi que les tas h_check et $h_conflict$. 44

6.1 Les parties (A) et (B) représentent respectivement la position au plus tôt de chacune des tâches ainsi que le PPO sur les ressources r_0 et r_1, (A) du problème initial décrit dans l'exemple 6.1, (B) une fois le point fixe atteint. 47
6.2 Les parties (A) et (B) représentent respectivement la position au plus tôt de chacune des tâches et le PPO, (A) après un premier balayage sur la ressource r_0, (B) après un second balayage sur la ressource r_1. 48
6.3 Les parties (A) et (B) représentent respectivement la position au plus tôt de chacune des tâches ainsi que le PPO des ressources r_0 et r_1, (A) du problème initial décrit dans l'exemple 6.4, (B) une fois le point fixe atteint. 57

6.4 État des tâches et des informations maintenues par la droite de balayage après la lecture des évènements $\langle PR, 1, 0\rangle$ et $\langle PR, 2, 0\rangle$, et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches. . . . 64

6.5 État des tâches et des informations maintenues par la droite de balayage après la lecture de l'évènement $\langle SCP, 0, 1\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches. 65

6.6 État des tâches et des informations maintenues par la droite de balayage après la lecture des évènements $\langle ECPD, 0, 2\rangle$, $\langle RS, 0, 2\rangle$, $\langle RS, 1, 2\rangle$ et $\langle RS, 2, 2\rangle$, et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches. 66

6.7 État des tâches et des informations maintenues par la droite de balayage après la lecture de l'évènement $\langle SCP, 1, 3\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches. 67

6.8 État des tâches et des informations maintenues par la droite de balayage après la lecture des évènements $\langle ECPD, 1, 4\rangle$, $\langle RS, 1, 4\rangle$, $\langle RS, 2, 4\rangle$ et $\langle PR, 3, 4\rangle$, et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches. 68

6.9 État des tâches et des informations maintenues par la droite de balayage après la lecture de l'évènement $\langle SCP, 2, 5\rangle$ et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches. 69

6.10 État des tâches et des informations maintenues par la droite de balayage après la lecture des évènements $\langle RS, 2, 6\rangle$, $\langle ECPD, 2, 6\rangle$ et $\langle PR, 4, 6\rangle$, et l'appel de l'algorithme filter_min (algorithme sweep_min, ligne 13) - (A) date de début et consommations de chaque tâche sur les ressources r_0 et r_1, (B) le graphe des précédences avec la valeur $nbpred_t$ associée à chaque tâche, (C) le PPO des ressources r_0 et r_1 et (D) la liste des évènements et le statut des tâches. 73

6.11 Illustration d'une solution impliquant cinq tâches et une ressource de capacité deux dans une contrainte cumulative colorée. 75

8.1 Les parties (A) et (B) représentent respectivement la position au plus tôt de chacune des tâches ainsi que le PPO, (A) du problème initial décrit dans l'exemple 8.3, (B) la solution obtenue par l'algorithme glouton. 89

9.1 Temps d'exécution en msec pour les instances aléatoires de bin-packing sans précédence. 104
9.2 Temps d'exécution en msec pour les instances aléatoires de bin-packing avec précédences. 104
9.3 Temps d'exécution en msec pour les instances cumulatives aléatoires sans précédence. 105
9.4 Temps d'exécution en msec pour les instances cumulatives aléatoires avec précédences. 105

Bibliographie

[1] Aggoun, A., Beldiceanu, N. : Time stamps techniques for the trailed data in constraint logic programming systems. In : S. Bourgault, M. Dincbas (eds.) SPLT'90, 8ème Séminaire Programmation en Logique, 16-18 mai 1990, Trégastel, France, pp. 487–510 (1990) 8

[2] Aggoun, A., Beldiceanu, N. : Extending CHIP in order to solve complex scheduling and placement problems. Mathl. Comput. Modelling **17**(7), 57–73 (1993) 8, 16

[3] Baker, K. : Introduction to sequencing and scheduling. Wiley (1974) 15

[4] Baptiste, P., Le Pape, C., Nuijten, W. : Constraint-based scheduling : applying constraint programming to scheduling problems. International Series in Operations Research and Management Science. kluwer (2001) 7, 16, 17

[5] Beldiceanu, N. : Global constraints as graph properties on structured network of elementary constraints of the same type. Tech. Rep. t2000/01, Swedish Institute of Computer Science (2000) 74

[6] Beldiceanu, N. : Pruning for the minimum constraint family and for the number of distinct values constraint family. In : T. Walsh (ed.) Principles and Practice of Constraint Programming (CP'01), 7th International Conference, *Lecture Notes in Computer Science*, vol. 2239, pp. 211–224. springer (2001) 74

[7] Beldiceanu, N., Carlsson, M. : Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In : T. Walsh (ed.) Principles and Practice of Constraint Programming (CP'01, 7th International Conference, *Lecture Notes in Computer Science*, vol. 2239, pp. 377–391. springer (2001) 18

[8] Beldiceanu, N., Carlsson, M. : A new multi-resource cumulatives constraint with negative heights. In : P. Van Hentenryck (ed.) Principles and Practice of Constraint Programming (CP'02), 8th International Conference, *Lecture Notes in Computer Science*, vol. 2470, pp. 63–79. springer (2002) 7, 8, 18, 27, 111, 112

[9] Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T. : Global constraint catalogue : Past, present, and future. Constraints **12**(1), 21–62 (2007). The current working version of the catalogue is at http://www.emn.fr/z-info/sdemasse/aux/doc/catalog.pdf 14

[10] Beldiceanu, N., Carlsson, M., Demassey, S., Poder, E. : New filtering for the *cumulative* constraint in the context of non-overlapping rectangles. Annals OR **184**(1), 27–50 (2011) 16

[11] Beldiceanu, N., Carlsson, M., Flener, P., Pearson, J. : On the reification of global constraints. Constraints **18**(1), 1–6 (2013) 77

[12] Beldiceanu, N., Carlsson, M., Poder, E. : New filtering for the cumulative constraint in the context of non-overlapping rectangles. In : L. Perron, M. Trick (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 5th International Conference, (CPAIOR'08), *Lecture Notes in Computer Science*, vol. 5015, pp. 21–35. Springer (2008) 16

[13] Beldiceanu, N., Carlsson, M., Poder, E., Sadek, R., Truchet, C. : A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In : B. Bessiere (ed.) Principles and Practice of Constraint Programming (CP'07), 13th International Conference, *Lecture Notes in Computer Science*, vol. 4741, pp. 180–194. springer (2007) 7

[14] Beldiceanu, N., Carlsson, M., Thiel, S. : Sweep synchronisation as a global propagation mechanism. Computers and Operations Research **33**(10), 2835–2851 (2006) 8

[15] Bessière, C., Hebrard, E., B., H., Kızıltan, Z., Walsh, T. : Filtering algorithms for the *nvalue* constraint. In : R. Barták, M. Milano (eds.) International conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR'05), *Lecture Notes in Computer Science*, vol. 3524, pp. 79–93. springer, Prague, Czech Republic (2005) 74

[16] Bessière, C., Hebrard, E., B., H., Kızıltan, Z., Walsh, T. : Filtering algorithms for the *nvalue* constraint. In : R. Barták, M. Milano (eds.) International conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR'05), *Lecture Notes in Computer Science*, vol. 3524, pp. 79–93. springer, Prague, Czech Republic (2005) 77

[17] Bessière, C., Van Hentenryck, P. : To be or not to be ... a global constraint. In : F. Rossi (ed.) Principles and Practice of Constraint Programming (CP'03), 9th International Conference, *Lecture Notes in Computer Science*, vol. 2833, pp. 789–794. Springer (2003) 14

[18] Carlier, J., Pinson, E. : Adjustment of heads and tails for the job-shop problem. European Journal of Operational Research **78**(2), 146–161 (1994) 16

[19] Carlsson, M., et al. : SICStus Prolog User's Manual. SICS, 4.2.3 edn. (2012). URL `http://www.sics.se/sicstus` 27, 101

[20] Caseau, Y., Laburthe, F. : Improved clp scheduling with task intervals. In : P. Van Hentenryck (ed.) Proceedings of the Eleventh International Conference on Logic Programming (ICLP'94), pp. 369–383. The MIT Press (1994) 16

[21] Caseau, Y., Laburthe, F. : Cumulative scheduling with task intervals. In : M. Maher (ed.) Proceedings of the 1996 Joint International Conference and Syposium on Logic Programming (JICSLP'96), pp. 363–377. MIT Press (1996) 16

[22] Chu, G.G. : Improving combinatorial optimization. Ph.D. thesis, The University of Melbourne (2011) 14

[23] Dawande, M., Kalagnanam, J., Sethuraman, J. : Variable sized bin packing with color constraints. Electronic Notes in Discrete Mathematics **7**, 154–157 (2001) 74, 77

[24] De Berg, M., Van Kreveld, M., Overmars, M., Schwarzkopf, O. : Computational geometry - algorithms and applications. springer (1997) 18

[25] Erschler, J., Lopez, P., Thuriot, C. : Raisonnement temporel sous contraintes de ressources et problèmes d'ordonnancement. Revue d'Intelligence Artificielle **5**(3), 7–36 (1991) 7, 16, 17

[26] Freuder, E. : In pursuit of the holy grail. Constraints **2**(1), 57–61 (1997) 13

[27] Frisch, A., Miguel, I., Walsh, T. : Modelling a steel mill slab design problem. In : Proceedings of the (IJCAI'01) Workshop on Modelling and Solving Problems with Constraints, pp. 39–45 (2001) 77

[28] Frisch, A., Miguel, I., Walsh, T. : Symmetry and implied constraints in the steel mill slab design problem. In : Proceedings of the (CP'01) Workshop on modelling and problem formulation, pp. 8–15 (2001) 77

[29] Garey, M.R., Johnson, D.S. : Computers and Intractability : A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979) 74

[30] Gargani, A., Refalo, P. : An efficient model and strategy for the steel mill slab design problem. In : B. Bessiere (ed.) Principles and Practice of Constraint Programming (CP'07), 13th International Conference, *Lecture Notes in Computer Science*, vol. 4741, pp. 77–89. springer (2007) 77

[31] Gent, I., Walsh, T. : Csplib : A benchmark library for constraints. Tech. rep., Technical report APES-09-1999 (1999). Available from http ://csplib.cs.strath.ac.uk/. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP'99) 77

[32] Graham, R.L., Lawler, E.L., Lenstra, J.K., Kan, A.H.G.R. : Optimization and approximation in deterministic sequencing and scheduling : a survey. Ann. Discrete Math. **4**, 287–326 (1979) 15

[33] Heinz, S., Schlechte, T., Stephan, R., Winkler, M. : Solving steel mill slab design problems. Constraints **17**(1), 39–50 (2012) 77

[34] Hermenier, F., Demassey, S., Lorca, X. : Bin-repacking scheduling problem in virtualized datacenters. In : J. Lee (ed.) Principles and Practice of Constraint Programming (CP'11), 17th International Conference, *Lecture Notes in Computer Science*, vol. 6876, pp. 27–41. springer (2011) 76

[35] Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T. : Hybrid modelling for robust solving. Annals of Operations Research **130**(1-4), 19–39 (2004) 77

[36] Hooker, J. : Integrated Methods for Optimization. International series in operations research & management science. Springer (2007) 14

[37] Kameugne, R., Fotso, L., Scott, J., Ngo-Kateu, Y. : A quadratic edge-finding filtering algorithm for cumulative resource constraints. In : J. Lee (ed.) Principles and Practice of Constraint Programming (CP'11), 17th International Conference, *Lecture Notes in Computer Science*, vol. 6876, pp. 478–492. springer (2011) 7, 16, 17

[38] Laburthe, F. : Choco : implementing a cp kernel. In : Proceedings of the CP'00 post conference workshop on techniques for implementing constraint programming systems (TRICS) (2000) 57

[39] Lagerkvist, M., Schulte, C. : Propagator groups. In : I. Gent (ed.) Principles and Practice of Constraint Programming (CP'09), 15th International Conference, vol. 5732, pp. 524–538. springer (2009) 55

[40] Lahrichi, A. : Ordonnancements. La notion de "parties obligatoires" et son application aux problèmes cumulatifs. RAIRO - Operations Research - Recherche Opérationnelle **16**(3), 241–262 (1982) 17

[41] Le Pape, C. : Des systèmes d'ordonnancement flexibles et opportunistes. Ph.D. thesis, Université Paris IX (1988). In French 7, 16, 18

[42] Letort, A., Beldiceanu, N., Carlsson, M. : A scalable sweep algorithm for the cumulative constraint. In : M. Milano (ed.) Principles and Practice of Constraint Programming - 18th International Conference (CP'12), *Lecture Notes in Computer Science*, vol. 7514, pp. 439–454. springer (2012) 58

[43] Letort, A., Carlsson, M., Beldiceanu, N. : A synchronized sweep algorithm for the *k-dimensional cumulative* constraint. In : C. Gomes, M. Sellmann (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, (CPAIOR'13), *Lecture Notes in Computer Science*, vol. 7874, pp. 144–159. springer (2013) 58

[44] Lopez, P. : Approche énergétique pour l'ordonnancement de tâches sous contraintes de temps et de ressources. Ph.D. thesis, Université Paul Sabatier - Toulouse III (1991). In French 7, 16, 17

[45] Lopez, P., Erschler, J., Esquirol, P. : Ordonnancement de tâches sous contraintes : une approche énergétique. Automatique, Productique, Informatique Industrielle **26**, 453–481 (1992) 7, 16, 17

[46] Mercier, L., Van Hentenryck, P. : Edge finding for cumulative scheduling. INFORMS Journal on Computing **20**(1), 143–153 (2008) 7, 16

[47] Michel, L., Van Hentenryck, P. : Comet in context. In : D. Goldin, A. Shvartsman, S. Smolka, J. Vitter, S. Zdonik (eds.) Principles of Computing & Knowledge, Paris C. Kanellakis Memorial Workshop (PCK50), pp. 95–107. ACM (2003) 14

[48] Nuijten, W. : Time and resource constrained scheduling : A constraint satisfaction approach. Ph.D. thesis, Eindhoven University of Technology (1994) 7, 16, 17

[49] Nuijten, W., Aarts, E. : Constraint satisfaction for multiple capacitated job shop scheduling. In : A. Cohn (ed.) Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94), pp. 635–639 (1994) 7, 16

[50] Nuijten, W., Aarts, E. : A computational study of constraint satisfaction for multiple capacitated job shop scheduling. European Journal of Operational Research **90**, 269–284 (1996) 7, 16

[51] O'Sullivan, B. : CP'11 panel position - the future of constraint programing. personal communication (2011) 7

[52] Pachet, F., Roy, P. : Automatic generation of music programs. In : J. Jaffar (ed.) Principles and Practice of Constraint Programming (CP'99), 5th International Conference, *Lecture Notes in Computer Science*, vol. 1713, pp. 331–345. Springer (1999) 74

[53] Perron, L., Shaw, P., Furnon, V. : Propagation guided large neighborhood search. In : M. Wallace (ed.) Principles and Practice of Constraint Programming (CP'04), 10th International Conference, *Lecture Notes in Computer Science*, vol. 3258, pp. 468–481. Springer (2004) 14

[54] Pesant, G. : Counting solutions of csps : A structural approach. In : L. Kaelbling, A. Saffiotti (eds.) Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05), pp. 260–265. Professional Book Center (2005) 14

[55] Petit, T. : Intermediary local consistencies. In : L. De Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, P. Lucas (eds.) European Conference on Artificial Intelligence ECAI'12), *Frontiers in Artificial Intelligence and Applications*, vol. 242, pp. 919–920. IOS Press (2012) 32

[56] R., K., Sprecher, A. : PSPLib – a project scheduling problem library. European journal of operational research **96**, 205–216 (1996) 9, 16, 87, 102

[57] Régin, J. : A filtering algorithm for constraints of difference in csps. In : B. Hayes-Roth, R. Korf (eds.) Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1, pp. 362–367. AAAI Press / The MIT Press (1994) 14

[58] Régin, J., Rezgui, M. : Discussion about constraint programming bin packing models. In : AI for Data Center Management and Cloud Computing, Papers from the 2011 AAAI Workshop, *AAAI Workshops*, vol. WS-11-08. AAAI (2011) 7

[59] Roadef : Challenge 2012 machine reassignment (2012). URL http://challenge.roadef.org/2012/en/index.php 7, 46

[60] Schaus, P., Monette, J., Michel, L., Van Hentenryck, P., Coffrin, C., Deville, Y. : Solving steel mill slab problems with constraint-based techniques : CP, LNS, and CBLS. Constraints **16**(2), 125–147 (2011) 77

[61] Schulte, C. : Comparing trailing and copying for constraint programming. In : D. De Schreye (ed.) Logic Programming : The 1999 International Conference (ICLP'99), pp. 275–289. The MIT Press (1999) 8

[62] Schulte, C., Stuckey, P. : Efficient constraint propagation engines. CoRR **abs/cs/0611009** (2006) 57

[63] Schutt, A., Feydy, T., P.J., S. : Explaining time-table-edge-finding propagation for the cumulative resource constraint. In : C. Gomes, M. Sellmann (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, (CPAIOR'13), *Lecture Notes in Computer Science*, vol. 7874, pp. 234–250. springer (2013) 16

[64] Schutt, A., Wolf, A. : A new $o(n^2 \log n)$ not-first/not-last pruning algorithm for cumulative resource constraints. In : D. Cohen (ed.) Principles and Practice of Constraint Programming (CP'10), 16th International Conference, *Lecture Notes in Computer Science*, vol. 6308, pp. 445–459. Springer (2010) 7, 16

[65] Schutt, A., Wolf, A., Schrader, G. : Not-first and not-last detection for cumulative scheduling in $o(n^3 \log n)$. In : M. Umeda, A. Wolf, O. Bartenstein, U. Geske, D. Seipel, O. Takata (eds.) Declarative Programming for Knowledge Management, 16th International Conference on Applications of Declarative Programming and Knowledge Management (INAP'05), *Lecture Notes in Computer Science*, vol. 4369, pp. 66–80. Springer (2005) 7, 16

[66] Shaw, P. : Using constraint programming and local search methods to solve vehicle routing problems. In : M. Maher, J. Puget (eds.) Principles and Practice of Constraint Programming (CP'98), 4th International Conference, *Lecture Notes in Computer Science*, vol. 1520, pp. 417–431. springer (1998) 7

[67] Simonis, H. : An industrial benchmark. personal communication (2013) 46

[68] Sindelar, M., Sitaraman, R., Shenoy, P. : Sharing-aware algorithms for virtual machine colocation. In : R. Rajaraman, F. Meyer auf der Heide (eds.) SPAA 2011 : Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 367–378. ACM (2011) 76

[69] Team, C. : Choco : An open source java CP library. research report 10-02-info, Ecole des Mines de Nantes (2010). URL `http://choco.emn.fr/` 27, 101

[70] Van Hentenryck, P., Michel, L. : The steel mill slab design problem revisited. In : L. Perron, M. Trick (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 5th International Conference, (CPAIOR'08), *Lecture Notes in Computer Science*, vol. 5015, pp. 377–381. Springer (2008) 77

[71] Vilím, P. : Edge finding filtering algorithm for discrete cumulative resources in $o(kn \log n)$. In : I. Gent (ed.) Principles and Practice of Constraint Programming (CP'09), 15th International Conference, *Lecture Notes in Computer Science*, vol. 5732, pp. 802–816. springer (2009) 7, 16

[72] Vilím, P. : Timetable edge finding filtering algorithm for discrete cumulative resources. In : J. Lee (ed.) Principles and Practice of Constraint Programming (CP'11), 17th International Conference, *Lecture Notes in Computer Science*, vol. 6876, pp. 230–245. springer (2011) 16, 17

[73] Wolf, A., Schrader, G. : $O(n \log n)$ overload checking for the cumulative constraint and its application. In : M. Umeda, A. Wolf, O. Bartenstein, U. Geske, D. Seipel, O. Takata (eds.) Declarative Programming for Knowledge Management, 16th International Conference on Applications of Declarative Programming and Knowledge Management (INAP'05), Lecture Notes in Computer Science, pp. 88–101. Springer (2005) 16

[74] Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E., Corner, M. : Memory buddies : exploiting page sharing for smart colocation in virtualized data centers. Operating Systems Review **43**(3), 27–36 (2009) 76

l'université nantes angers le mans

PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes

Institut Mines-Télécom

Thèse de Doctorat

Arnaud Letort

Passage à l'échelle pour les contraintes d'ordonnancement multi-ressources

Scalable multi-dimensional resources scheduling constraints

Résumé

La programmation par contraintes est une approche régulièrement utilisée pour résoudre des problèmes combinatoires d'origines diverses. Dans cette thèse nous nous focalisons sur les problèmes d'ordonnancement cumulatif. Un problème d'ordonnancement consiste à déterminer les dates de débuts et de fins d'un ensemble de tâches, tout en respectant certaines contraintes de capacité et de précédence. Les contraintes de capacité concernent aussi bien des contraintes cumulatives classiques où l'on restreint la somme des hauteurs des tâches intersectant un instant donné, que des contraintes cumulatives colorées où l'on restreint le nombre maximum de couleurs distinctes prises par les tâches.
Un des objectifs récemment identifiés pour la programmation par contraintes est de traiter des problèmes de grandes tailles, habituellement résolus à l'aide d'algorithmes dédiés et de métaheuristiques. Par exemple, l'utilisation croissante de centres de données virtualisés laisse apparaitre des problèmes d'ordonnancement et de placement multi-dimensionnels de plusieurs milliers de tâches.
Pour atteindre cet objectif, nous utilisons l'idée de balayage synchronisé considérant simultanément une conjonction de contraintes cumulative et des précédences, ce qui nous permet d'accélérer la convergence au point fixe. De plus, de ces algorithmes de filtrage nous dérivons des procédures gloutonnes qui peuvent être appelées à chaque noeud de l'arbre de recherche pour tenter de trouver plus rapidement une solution au problème. Cette approche permet de traiter des problèmes impliquant plus d'un million de tâches et 64 resources cumulatives.
Ces algorithmes ont été implémentés dans les solveurs de contraintes Choco et SICStus, et évalués sur divers problèmes de placement et d'ordonnancement.
Mots-clés : Programmation par contraintes, ordonnancement, cumulatif, passage à l'échelle, point fixe, contraintes de ressources multi-dimensionelles, balayage synchronisé.

Mots clés

Programmation par contraintes, ordonnancement, cumulatif, passage à l'échelle, point fixe, contraintes de ressources multi-dimensionelles, balayage synchronisé.

Abstract

Constraint programming is an approach often used to solve combinatorial problems in different application areas. In this thesis we focus on the cumulative scheduling problems. A scheduling problem is to determine the starting dates of a set of tasks while respecting capacity and precedence constraints. Capacity constraints affect both conventional cumulative constraints where the sum of the heights of tasks intersecting a given time point is limited, and colored cumulative constraints where the number of distinct colors assigned to the tasks intersecting a given time point is limited.
A newly identified challenge for constraint programming is to deal with large problems, usually solved by dedicated algorithms and metaheuristics. For example, the increasing use of virtualized data centers leads to multi dimensional placement problems of thousand of jobs.
Scalability is achieved by using a synchronized sweep algorithm over the different cumulative and precedence constraints that allows to speedup convergence to the fix point. In addition, from these filtering algorithms we derive greedy procedures that can be called at each node of the search tree to find a solution more quickly. This approach allows to deal with scheduling problems involving more than one million jobs and 64 cumulative resources.
These algorithms have been implemented within Choco and SICStus solvers and evaluated on a variety of placement and scheduling problems.

Key Words

Constraint programming, scheduling, cumulative, scalability, fix point, multi-dimensional resource constraints, synchronized sweep.