REPUBLIQUE DU CAMEROUN

REPUBLIC OF CAMEROON

UNIVERSITÉ DE YAOUNDÉ I

FACULTÉ DES SCIENCES

UNIVERSITY OF YAOUNDE I

FACULTY OF SCIENCE

DÉPARTEMENT DE MATHÉMATIQUES

DEPARTMENT OF MATHEMATICS

C*entre* de **R***echerche* et de **F***ormation* **D***octoral*

Sciences, Technologies et Géosciences

Laboratoire de Mathématiques Appliquées

TECHNIQUES DE PROPAGATION DE LA CONTRAINTE DE RESSOURCE EN ORDONNANCEMENT CUMULATIF.

THÈSE DE DOCTORAT/PHD

Discipline : Mathématiques, Spécialité : Mathématiques Appliquées

Présentée et soutenue publiquement par

KAMEUGNE Roger

Le 30 septembre 2014 à la Faculté des Sciences de l'Université de Yaoundé 1

Devant le jury ci-dessous :

Président :	ANDJIGA Nicolas Gabriel, Professeur,Université de Yaoundé 1
Rapporteur :	FOSTO Laure Pauline, Maître de Conférences,Université de Yaoundé 1
Membres :	EMVUDU Wono Yves, Maître de Conférences,Université de Yaoundé 1
	NDOUNDAM René, Maître de Conférences,Université de Yaoundé 1
	TAYOU Clémentin, Maître de Conférences,Université de Dschang
	LOPEZ Pierre, Directeur de Recherche au CNRS,LASS-CNRS de Toulouse
	OULAMARA Ammar, HDR,Université de Lorraine

Dédicace

A mes filles

Yvane

et

Hanamé

Remerciements

Je souhaite remercier toutes les personnes qui ont contribué à la finalisation de ce travail. C'est avec un grand plaisir que je les mentionne ici.

Je souhaite remercier en premier lieu le Professeur Laure Pauline FOTSO qui, malgré ses multiples occupations, a accepté de travailler avec moi. Je tiens à lui témoigner ma gratitude pour le temps, les efforts et les conseils qui m'ont permis de mieux m'orienter lorsque j'étais perdu. Enfin, je tiens à la remercier pour avoir su m'apprendre quelles sont les vraies valeurs d'une recherche scientifique de qualité. Une fois de plus merci pour tout.

Je remercie le Professeur François WAMON, Chef de Département de Mathématiques, pour toutes les facilités qu'il m'a accordées.

Je remercie l'Université de Yaoundé 1 au sein de laquelle j'ai préparé cette thèse.

Je remercie mon épouse Sévérine KAMEUGNE pour les nombreuses discussions qu'elle a bien accepté d'échanger avec moi sur les contraintes globales et la relecture du premier draft de cette thèse. Je tiens à lui témoigner ma gratitude pour le temps, les encouragements et le soutien qu'elle m'a fournis.

Je souhaite exprimer ma gratitude aux rapporteurs de cette thèse pour leur lecture attentive : Pierre LOPEZ, Guy Merlin MBAKOP, René NDOUNDAM, Thierry PETIT et Ammar OULAMARA.

Je souhaite remercier tous mes collègues de la Faculté des Sciences de Maroua, particulièrement Dr Joseph DONGHO chef de Département de Mathématiques et Informatique, Didier Alain NJAMEN NJOMEN, Michel MOUGA, Olga KENGNI, Urbain NOUTSA, KALAZAVI, Martin PENGOU et ceux de l'Ecole Normale Supérieure de Maroua, particulièrement Dr ABDOURAHMAN chef de Département de Mathématiques, Pierre WAMBO, Luc Emery FOTSO DIEKOUAM, Gilbert MANTIKA, Salifou MBOUTN-

GAM, Alexis NANGUE, Junior Max PAMBE, Sylvain ILOGA.

Mes remerciements vont également à l'endroit de l'équipe de recherche Errops du Pr FOTSO notamment mes collègues Calice Olivier PIEUME et Max Junior PAMBE.

Mes remerciements vont à l'endroit de mes co-auteurs notamment Laure Pauline FOTSO, Ervée KOUAKAM, Joseph SCOTT et Youcheu NGO-KATEU.

Je remercie le Professeur Pierre FLENER pour ses nombreux soutients, notamment ses appels personnels à l'ACP, à l'AFPC, et aux organisateurs de CP'11 pour solliciter des bourses en ma faveur.

Je remercie l'ACP (Association for Constraint Programming) qui m'a permis de participer aux conférences CP'09 au Portugal et CP'11 en Italie en prenant en charge le logement et souvent le transport.

Je remercie l'AFPC (Association Française de Programmation par Contraintes) pour le subside qu'elle m'a accordé afin que je participe à la conférence CP11 en Italie.

Je remercie mes mamans Augustine MAMDOM, Anne MATCHUENKAM, mes frères et sœurs Dr Thierry NOULAMO, Fernand KODJOU, Lili Esthère MODJOU, Dr Joseph TSEMEUGNE, Marie NEUBOU KENMEGNE, Evelyne DJUIJE.

Je n'oublie pas mes amis Bruno NTAKEU, Dr Gilbert CHENDJOU et Alex KWAYA.

Je remercie mes camarades et collègues du laboratoire MAT pour les discussions et l'ambiance qui a régné entre nous tout au long de cette thèse. Je citerais sans faire de distinguo Serge MOTO, Hypolite TAPAMO, Etienne KOUOKAM, Norbert TSOPZE, Yanik NGOKO, Donatien CHEDOM, Adamou HAMZA, Serge EBELE, Germaine TCHUENTE, Max PAMBE, Nadège MEYEMDOU, Maurice TCHOUPE, Bernard FOTSING, Vivient KAMLA, Olivier PIEUME, Mathurin SOH, Zéphyrin SOH, Anne-Marie CHANA et Christiane KAMDEM. Je n'oublie pas Josiane MAATOH, Guy Merlin NKEUMALEU, Jukes FOKOU qui n'ont jamais cessé de m'encourager.

Que tous ceux dont j'ai oublié les noms et qui d'une façon ou d'une autre ont participé à l'accomplissement de ce projet trouvent ici l'expression de toute ma reconnaissance.

Sommaire

Résumé

Le problème d'ordonnancement consiste à organiser l'exécution d'un ensemble de tâches soumises à des contraintes de temps et de ressources. C'est un problème combinatoire NP-difficile au sens fort (même lorsqu'on ne considère qu'une seule ressource). Dans cette thèse, nous utilisons l'approche par contraintes pour résoudre le problème d'ordonnancement cumulatif non-préemptif. Nous nous intéressons particulièrement à la propagation de la contrainte de ressources.

Nous décrivons et comparons tant d'un point de vue expérimental que théorique, de nouveaux algorithmes de propagation de la contrainte de ressources en ordonnancement cumulatif. Il s'agit de l'edge-finding, de l'extended edge-finding et du not-first/not-last. Chacun de ces algorithmes de filtrage étudiés effectue différents dégrés de filtrage. C'est pourquoi ils sont combinés dans la contrainte globale *cumulative* pour une propagation maximale. Toutes ces règles cherchent à déterminer la position d'une tâche i par rapport à un ensemble de tâches Ω toutes nécessitant la même ressource. Nous proposons pour chacune des règles, de nouveaux algorithmes de filtrage qui même lorsque leur complexité n'est pas toujours meilleure, s'avère très utile dans la pratique.

Pour l'edge-finding, nous proposons un algorithme itératif de complexité $\mathcal{O}(n^2)$ basé sur les notions de densité maximale et de marge minimale. Cet algorithme n'effectue pas nécéssairement l'ajustement maximal à la première itération, mais s'améliore aux itérations subséquentes. Dans la pratique, cet algorithme est trois fois plus rapide que le meilleur algorithme standard d'edge-finding de la littérature.

Nous montrons aussi que l'algorithme d'extended edge-finding de Mercier et Van Hentenryck de complexité $\mathcal{O}(n^2k)$ est incorrect. Nous proposons alors un algorithme itératif d'extended edge-finding de complexité $\mathcal{O}(n^2)$ similaire à celui de l'edge-finding basé sur la

notion de marge minimale. La combinaison de ce nouvel algorithme avec celui de l'edge-finding permet d'accélérer l'exécution de certains instances et dans le cas où le branchement est dynamique, on remarque en plus une réduction du nombre de nœuds dans l'arbre de recherche sur les instances réputées fortement cumulatives. Nous démontrons que le résultat annoncé par Vilím suivant lequel la conjonction de l'edge-finding et de l'extended edge-finding est dominée par le timetable edge finding est faux.

Enfin, nous proposons deux algorithmes de not-first/not-last, l'un complet et l'autre itératif. L'algorithme complet est de complexité $\mathcal{O}(n^2|H|\log n)$ où $|H|$ désigne le nombre de dates de fin au plus tôt différentes des tâches. Nous améliorons ainsi la complexité du meilleur algorithme complet connu pour cette règle, $\mathcal{O}(n^3 \log n)$, car $|H| \leq n$. Le second algorithme de not-first/not-last est itératif et de complexité $\mathcal{O}(n^2 \log n)$. Il atteint le même point fixe que les algorithmes complets de not-first/not-last et est plus rapide en pratique que tous les autres algorithmes de not-first/not-last.

Mots Clés : Ordonnancement cumulatif, programmation par contraintes, propagation des contraintes, contrainte globale, edge-finding, extended edge-finding, not-first/not-last, timetable edge finding, RCPSP.

Abstract

A scheduling problem consists in planning the execution of a set of tasks under resource and temporal constraints. It is an NP-hard combinatorial problem even if only one resource is considered. In this thesis, we use the constraint programming approach to solve the non-preemptive cumulative scheduling problem. We focus particularly on resource constraint propagation.

We describe and compare theoretically and experimentally new propagation algorithms for resource constraint in cumulative scheduling. We focus on edge-finding, extended edge-finding and not-first/not-last algorithms. Each of these algorithms has different filtering power. It is while they are combined in the global *cumulative* constraint to increase their filtering power and for a maximum propagation. Each of this rules determines the position of the task i relatively to a set of tasks Ω all sharing the same resource. We propose for each of these rules a new filtering algorithm. The complexity of some algorithms does not strictly dominate the best existing one, but the algorithm remains useful in practice.

For the edge-finding, we propose a sound $\mathcal{O}(n^2)$ algorithm based on the notions of maximum density and minimum slack. The algorithm doesn't necessary perform the maximum adjustment at the first run, but improves at subsequent iterations. In practice, it is three time faster than the best known standard algorithm of the literature.

We prove that the extended edge-finding of Mercier and Van Hentenryck of complexity $\mathcal{O}(n^2k)$ is incorrect. We then propose a sound $\mathcal{O}(n^2)$ extended edge-finding algorithm similar to our edge-finding and based on the notion of minimum slack. The conjunction of our new extended edge-finding algorithm with the edge-finding reduces the running time of a few instances and when the dynamique branching scheme is used, we have in addition, a reduction of the number of nodes on highly cumulative instances. We prove

that the edge-finding and the extended edge-finding are not dominated by the timetable edge finding. This result contradicts the claim of Vilím that the conjunction of edge-finding and extended edge-finding is dominated by the timetable edge finding.

We end with the proposition of two filtering algorithms for the not-first/not-last rule. The first one is complete and runs in $\mathcal{O}(n^2|H|\log n)$ where $|H|$ is the number of different earliest completion times of tasks. This algorithm thus improves the complexity of the best known complete not-first/not-last algorithm, $\mathcal{O}(n^3 \log n)$, since $|H| \leq n$. The second filtering is sound and run in $\mathcal{O}(n^2 \log n)$. This algorithm reaches the same fix point as complete filtering not-first/not-last algorithms and is faster than all of them in practice.

Key Words : Cumulative scheduling, constraint programming, constraint propagation, global constraint, edge-finding, extended edge-finding, not-first/not-last, timetable edge finding, RCPSP.

Abréviations

IA	Intelligence Artificielle
RO	Recherche Opérationnelle
PPC	Programmation Par Contraintes
CuSP	Cumulative Scheduling Problem
RCPSP	Resource Constrained Projet Scheduling Problems
CSP	Constraint Satisfaction Problem
AC	Arc Consistency
HAC	Hyperarc Consistency
MLC	Maintaining Local Consistency
CSOP	Constraint Satisfaction Optimisation Problem
PSPLib	Project Scheduling Problem Library
BL	Baptiste and Le Pape RCPSP library

Liste des tableaux

Liste des figures

List of Algorithms

Liste des symboles

s_i	date de début de la tâche i
p_i	durée d'exécution de la tâche i
c_i	quantité de ressource requise par la tâche i
C	capacité de la ressource du CuSP
r_i	date de début au plus tôt de la tâche i
d_i	date de fin au plus tard de la tâche i
e_i	énergie de la tâche i
r_Ω	date de début au plus tôt de l'ensemble Ω de tâches
d_Ω	date de fin au plus tard de l'ensemble Ω de tâches
e_Ω	énergie de l'ensemble Ω de tâches
ect_i	date de fin au plus tôt de la tâche i
lst_i	date de début au plus tard de la tâche i
$i \lessdot j$	la tâche i précède la tâche j
$p_i^+(t_1)$	nombre minimum d'unités de temps de la tâche i après l'instant t_1
$p_i^-(t_2)$	nombre minimum d'unités de temps de la tâche i avant l'instant t_2
$W_{Sh}(i, t_1, t_2)$	énergie consommée par la tâche i dans l'intervalle $[t_1; t_2]$
$W_{Sh}(t_1, t_2)$	énergie consommée par toutes les tâches T sur l'intervalle $[t_1; t_2]$
p_i^{TT}	partie fixe de la tâche i
p_i^{EF}	partie libre de la tâche i
e_i^{EF}	énergie libre de la tâche i
T^{EF}	ensemble des tâches ayant une partie libre
$TT(t)$	somme de toutes les consommations qui chevauchent l'instant t

$ttAfterEst(\Omega)$	énergie consommée par toutes les tâches après r_Ω
$ttAfterLct(\Omega)$	énergie consommée par toutes les tâches après d_Ω
e_Ω^{EF}	énergie libre des tâches de Ω
$reserve(\Omega)$	énergie restant (marge) dans l'intervalle $[t_1; t_2]$
$add(r_\Omega, d_\Omega, i)$	énergie consommée par la tâche i dans l'intervalle $[r_\Omega; d_\Omega]$
$mandatoryIn(r_\Omega, d_\Omega, i)$	énergie fixe de la tâche i dans l'intervalle $[r_\Omega; d_\Omega]$
ect_Ω	date de fin au plus tôt des tâches de Ω
lst_Ω	date de début au plus tard des tâches de Ω
$\Omega \lessdot i$	les tâches de Ω finissent avant la fin de la tâche i
$\Omega \gtrdot i$	les tâches de Ω commencent après le début de la tâche i
Ect_Ω	borne inférieure de la date de fin au plus tôt de Ω
Lst_Ω	borne supérieure de la date de début au plus tard de Ω
$\alpha(\Omega, i)$	contraintes de la règle d'edge-finding
$rest(\Omega, c_i)$	énergie de Ω qui affecte l'ordonnancement de la tâche i
$\Omega_{\tau(U,i),U}$	intervalle de tâches de marge minimale par rapport à d_U
$\Theta_{\rho(U,i),U}$	intervalle de tâches de densité maximale
$Dupd_i$	valeur d'ajustement basée sur l'intervalle de tâches de densité maximale
$SLupd_i$	valeur d'ajustement basée sur l'intervalle de tâches de marge minimale pour les règles EF et EF1
$\beta(\Omega, i)$	contraintes de la règle d'extended edge-finding
$\Omega_{L,\delta(L,i)}$	intervalle de tâches de marge minimale par rapport à r_L
upd	valeur d'ajustement basée sur l'intervalle de tâches marge minimale pour la règle EEF
ECT_Ω	la plus petite date de fin au plus tôt de Ω
LST_Ω	la plus grande date de début au plus tard de Ω
$\gamma(\Omega, i)$	contraintes de la règle not-first/not-last
$LCut(T, j, Ect, i)$	coupe gauche de T par la tâche j relativement à i et à Ect
$SLCut(T, j, i)$	pseudo coupe gauche de T par la tâche j relativement à la tâche i
$Env(\Theta, i)$	enveloppe d'énergie de Θ relativement à la tâche i
H_j^k	ensemble des différentes dates de fin au plus tôt de $SLCut(T, j, i)$.

Introduction générale

Dans les organisations telles que les gouvernements et les entreprises, la grande taille des projets nécessite une planification et une optimisation de l'utilisation des ressources ou tout simplement la recherche d'une solution réalisable. C'est par exemple le cas en industrie lorsqu'on veut savoir quelles machines utiliser pour exécuter un ensemble de tâches et à quelles dates ces tâches doivent être planifiées (projets à moyens limités) pour minimiser la durée totale du projet. Ce type de problème se range dans la famille de problèmes d'ordonnancement qui sont généralement fortement combinatoires. L'ordonnancement est l'allocation des ressources aux tâches dans le temps. Plusieurs typologies complètent cette définition très générale [21, 45]. Les problèmes varient suivant la nature des tâches (préemptifs ou non), des ressources (disjonctives, cumulatives, renouvelables ou non), les contraintes portant sur les tâches (contraintes de disponibilité, contraintes de précédence) et les critères à optimiser (fin du projet, retard, coût total). La résolution d'un problème d'ordonnancement consiste à placer dans le temps les activités ou tâches, compte tenu des contraintes temporelles (délais, contraintes d'enchaînement, ...) et de contraintes portant sur l'utilisation et la disponibilité des ressources requises par les tâches.

Plusieurs communautés se sont intéressées à la résolution des problèmes d'ordonnancement : la communauté Recherche Opérationnelle (RO), la communauté Intelligence Artificielle (IA) et la communauté Programmation Par Contraintes (PPC). La flexibilité et la lisibilité des formulations des problèmes obtenus avec la PPC ont fait de cette méthode une approche émergente. Dans cette approche, le processus de résolution est basé sur la génération de contraintes et la résolution de celles-ci à travers des méthodes spécifiques. Ces techniques sont étudiées dans le domaine de l'Intelligence Artificielle. Une de ces techniques, parmi les plus importantes, est la propagation de contraintes. L'ob-

jectif essentiel de la propagation de contraintes est de définir un problème équivalent au problème de départ, mais dont la nouvelle formulation présente un espace de recherche moins volumineux. La propagation des contraintes a pour but d'éliminer les valeurs des domaines des variables ne conduisant à aucune solution. Une caractéristique importante de la PPC est qu'elle considère généralement une seule contrainte à la fois : les algorithmes de filtrage sont exécutés successivement sur chaque contrainte et les informations obtenues sont propagées localement aux autres contraintes.

Au cours de la propagation des contraintes, chaque contrainte peut être examinée indépendamment des autres. Ceci constitue une faiblesse de cette technique car la connaissance des autres contraintes peut améliorer substantiellement la réduction du domaine. C'est pour cette raison que les *contraintes globales* ont été introduites. Une contrainte globale est un sous-ensemble de contraintes, correspondant à une sous-structure du problème original, et auquel est associé un algorithme de filtrage spécialisé. Pour la résolution des problèmes d'ordonnancement cumulatif, la contrainte globale

$$cumulative([s_1, ..., s_n], [p_1, ..., p_n], [c_1, ..., c_n], C)$$

est utilisée pour la propagation de la contrainte de ressource. Elle sert à modéliser la situation suivante : On doit ordonnancer n tâches où $s_1, \ldots, s_n$ désignent les dates de début associées à chaque tâche. La durée d'exécution de chaque tâche i est p_i. Chaque tâche i demande c_i unités de ressource. C unités de ressource sont disponibles à tout moment. Les algorithmes de filtrage pour la contrainte cumulative sont très complexes et variés [40, 75, 54, 43, 42, 39, 65, 60, 8, 76, 36, 44, 67, 9, 12, 13, 33, 66].

On peut citer entre autres l'edge-finding [40, 75, 54, 60, 67], l'extended edge-finding [43, 54, 60], le not-first/not-last [42, 65, 38, 39, 37, 35], le raisonnement énergétique [10, 9, 11, 51], le timetable edge finding [77], le balayage [12], le timetabling [10, 9, 11], l'overload checking [78, 76], les coupes de Benders [33], l'énergie de précédence et le balance constraint [50].

L'efficacité et la puissance de la contrainte globale dépend de l'exactitude et de la complexité de ces algorithmes de filtrages. Les travaux présentés dans cette thèse s'inscrivent dans ce cadre. Nous décrivons et comparons tant d'un point de vue expérimental que théorique, de nouveaux algorithmes de propagation de contraintes de ressources en ordon-

nancement cumulatif. Nous nous intéressons essentiellement à l'edge-finding, l'extended edge-finding et au not-first/not-last. Les contributions de cette thèse sont les suivantes :

- Nous proposons un algorithme quadratique d'edge-finding cumulatif qui n'effectue pas nécessairement l'ajustement maximal à la première itération, mais s'améliore aux itérations subséquentes. Alors que sa complexité ne domine pas strictement celle de l'algorithme de complexité $\mathcal{O}(nk \log n)$ en temps proposé par Vilím [75], les résultats expérimentaux sur les instances de RCPSP de la librairie PSPLib [49] et l'ensemble des instances de Baptiste et Le Pape BL [8] montrent que cet algorithme est plus rapide. [1]
- Nous montrons que la seconde phase de l'algorithme d'extended edge-finding proposé par Luc Mercier et Pascal Van Hentenryck [54] de complexité $\mathcal{O}(kn^2)$ en temps et $\mathcal{O}(kn)$ en espace (où n désigne le nombre de tâches et $k \leq n$ le nombre de différentes demandes en ressource) est incorrecte.
- Puis, nous proposons un nouvel algorithme d'extended edge-finding de complexité $\mathcal{O}(n^2)$ en temps. Au point fixe de l'edge-finding, cet algorithme effectue toujours un ajustement si celui ci est justifié par la règle. Comme pour l'edge-finding, ce n'est pas nécessairement à la première itération que l'ajustement maximal est effectué, mais l'algorithme s'améliore aux itérations subséquentes. Nous montrons que la combinaison de cet algorithme et celui de l'edge-finding atteint le même point fixe que la conjonction des règles edge-finding et extended edge-finding. Les résultats expérimentaux sur les instances de RCPSP de la librairie PSPLib [49] et l'ensemble des instances de Baptiste et Le Pape BL [8] montrent que la combinaison de ce nouvel algorithme avec celui de l'edge-finding permet d'accélérer l'exécution de certains instances et dans le cas où le branchement est dynamique, on remarque en plus une réduction du nombre de nœuds dans l'arbre de recherche sur les instances réputées fortement cumulatives.
- Nous démontrons que le timetable edge finding ne domine ni l'edge-finding, ni l'extended edge-finding. Ce qui contredit le résultat annoncé par Vilím [77] suivant

1. Ce résultat a été obtenu avec la collaboration de Joseph Scott et Youcheu Ngo-Kateu thésards. L'implémentation, la conduite des expériences, la tabulation, le plottage et l'analyse des résultats de [40, 41] ont été conduites en grande partie par Joseph Scott.

lequel la conjonction de l'edge-finding et l'extended edge-finding est dominée par le timetable edge finding [2].

- Les dernières contributions portent sur le not-first/not-last. Nous proposons pour cette règle deux algorithmes : l'un complet et l'autre itératif. L'algorithme complet est de complexité $\mathcal{O}(n^2|H|\log n)$ en temps et $\mathcal{O}(n)$ en espace où $|H|$ désigne le nombre de dates de fin au plus tôt différentes des tâches [39]. Nous améliorons ainsi la complexité du meilleur algorithme complet connu pour cette règle [64].
- Nous proposons aussi un algorithme itératif de not-first/not-last qui atteint le même point fixe que les algorithmes complets de not-first/not-last [42, 38]. En effet, l'algorithme effectue toujours un ajustement lorsque celui ci est justifié par la règle. Cet ajustement n'étant pas forcement maximal, il est amélioré aux itérations subséquentes. Une comparaison des deux algorithmes de not-first/not-last sur les instances de RCPSP de la librairie PSPLib [49] et l'ensemble des instances de Baptiste et Le Pape BL [8] montre que la version itérative est plus rapide que la version complète.

Cette thèse est structurée comme suit :

Au Chapitre 1, nous présentons les notions générales utilisées dans la thèse. Il s'agit notamment de la Programmation Par Contraintes (PPC) et des problèmes d'ordonnancement cumulatif. Une attention particulière est portée sur les problèmes à une ressource connus sur le nom de Cumulative Scheduling Problems (CuSP). C'est sur ces problèmes que seront appliquées les règles de propagation étudiées dans cette thèse. Nous clôturons le chapitre par la présentation de l'état de l'art des différents algorithmes de filtrages existants et souvent utilisés dans la contrainte globale cumulative.

Le Chapitre 2 est consacré à l'edge-finding, l'extended edge-finding et au not-first/not-last. Nous proposons un algorithme de complexité $\mathcal{O}(n^2)$ en temps pour l'edge-finding. Le nouvel algorithme est basé sur les notions de marge minimale et de densité maximale et effectue toujours un ajustement tant que celui ci est justifié par la règle. Nous montrons que la seconde phase de l'algorithme d'extended edge-finding proposé par Luc Mercier et Pascal Van Hentenryck [54] est incorrecte. Nous proposons par la suite un nouvel algorithme d'extended edge-finding itératif de complexité $\mathcal{O}(n^2)$ en temps et $\mathcal{O}(n)$ en espace

2. C'est avec le concours de Joseph Scott que nous avons réfuté les arguments de Vilím sur la domination du timetable edge-finding sur l'(extended) edge-finding.

qui est très efficace au point fixe de l'edge-finding. Nous démontrons que le timetable edge finding ne domine ni l'edge-finding, ni l'extended edge-finding. La dernière partie de ce chapitre est consacré au not-first/not-last où nous proposons deux algorithmes de not-first/not-last : l'un complet de complexité $\mathcal{O}(n^2|H|\log n)$ en temps et $\mathcal{O}(n)$ en espace [39] et l'autre itératif de complexité $\mathcal{O}(n^2 \log n)$ en temps et $\mathcal{O}(n)$ en espace [38, 42]. H désigne l'ensemble des dates de fin au plus tôt des tâches.

Le Chapitre 3 est consacré aux résultats expérimentaux. Nous comparons les algorithmes ainsi proposés à ceux de la littérature sur les instances de RCPSP bien connues de la littérature et l'ensemble des instances de Baptiste et LePape BL [8]. Ces résultats montrent que nos algorithmes sont plus rapides que ceux existants.

Chapitre Premier

Revue de la littérature : Programmation par Contraintes et Ordonnancement

Dans ce chapitre, nous présentons quelques concepts de base en Programmation Par Contraintes (PPC), puis après une classification partielle des problèmes d'ordonnancement, nous décrivons les problèmes d'ordonnancement cumulatif à une ressource notés CuSP pour Cumulative Scheduling Problems. L'état de l'art sur les algorithmes de filtrage existants pour la propagation de la contrainte *cumulative* conclut le chapitre.

1.1 Programmation par contraintes

Dans ce paragraphe, nous présentons les principes de la Programmation Par Contraintes (PPC). Après une étude du problème de satisfaction de contraintes (CSP en abrégé) (Constraint Satisfaction Problem en anglais), nous décrivons les mécanismes de filtrage et de propagation des contraintes et quelques techniques de recherche de solutions. Nous présentons ensuite quelques heuristiques de choix de variables et de valeurs et décrivons comment les techniques de recherche sont utilisées dans un processus itératif pour optimiser un certain critère.

1.1.1 Problèmes de satisfaction de contraintes

La notion de contrainte apparaît naturellement dans notre vie quotidienne. C'est par exemple le cas lorsqu'il s'agit d'affecter des stages à des étudiants en respectant leurs

souhaits, de ranger des pièces de formes diverses dans une boîte rigide, de planifier le trafic aérien pour que tous les avions puissent décoller et atterrir sans se percuter, ou encore d'établir un menu à la fois équilibré et appétissant. La notion de "Problème de Satisfaction de Contraintes" (CSP) désigne l'ensemble de ces problèmes définis par des contraintes, et consistant à chercher une solution qui satisfait ces contraintes. A travers la programmation par contraintes (PPC), il est possible de modéliser plusieurs problèmes réels comme des problèmes de satisfaction de contraintes [48, 71, 55, 72, 23, 61]. Une instance du CSP est définie par la donnée d'un ensemble de variables, d'un domaine pour chaque variable spécifiant les valeurs pouvant être attribuées à cette variable et des contraintes sur les variables. Mathématiquement, le problème est défini comme suit :

Définition 1.1. *Une instance P du problème de satisfaction de contraintes (CSP) est définie par la donnée d'un triplet $P = (X, D, C)$ où $X = \{x_1, ..., x_n\}$ désigne l'ensemble des variables, $D = \{D(x_1), ..., D(x_n)\}$ l'ensemble des domaines associés aux variables ($D(x_i)$ désigne l'ensemble des valeurs possibles de la variable x_i) et $C = \{C_1, ..., C_m\}$ l'ensemble fini des contraintes. Chaque contrainte définit les combinaisons des valeurs autorisées.*

Une solution d'une instance de CSP est un n-uplet $(v_1, ..., v_n)$ de valeurs attribuées aux variables x_1,...,x_n telle que $v_i \in D(x_i)$ et toutes les contraintes de C soient satisfaites.

Dans toute la suite, nous nous intéressons au cas des domaines finis à valeurs entières comme cela sera le cas pour les problèmes d'ordonnancement qui nous intéressent.

Une variable est dite instanciée quand on lui affecte une valeur dans son domaine. Une instantiation complète est toute application qui instancie toutes les variables. Une solution partielle d'un CSP est toute instantiation partielle qui satisfait l'ensemble des contraintes de C ne portant que sur les variables instanciées.

Une instantiation (totale ou partielle) est consistante si elle ne viole aucune contrainte, et inconsistante si elle viole une ou plusieurs contraintes. Lorsqu'elle existe, une solution du CSP est donc toute instantiation complète qui satisfait toutes les contraintes.

Définition 1.2. *Pour une instance de CSP, si C_j est une contrainte alors nous notons $Var(C_j)$ l'ensemble des variables qui interviennent dans cette contrainte. L'arité d'une contrainte C_j est le nombre de variables qui interviennent dans la contrainte et est noté*

$|Var(C_j)|$.

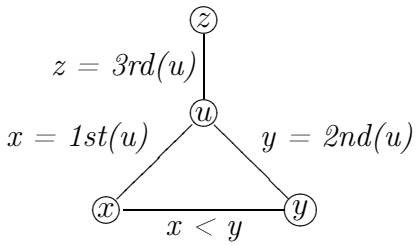
Si $|Var(C_j)| = 1$ *alors la contrainte* C_j *est dite unaire. Si* $|Var(C_j)| = 2$ *alors la contrainte est dite binaire et n-aire si* $|Var(C_j)| = n$.

Les définitions 1.1 et 1.2 sont illustrées dans l'Exemple 1.1 ci-dessous.

Exemple 1.1. *Soient* x, y *et* z *trois variables telles que* $D(x) = \{2, 3, 4\}$, $D(y) = \{1, 2, 3, 4\}$ *et* $D(z) = \{3, 4, 5\}$. *Soient* $C_1 : x < y$ *et* $C_2 : x + y = z$ *deux contraintes. Le triplet* (X, D, C) *définit un CSP où* $X = \{x, y, z\}$, $D = \{D(x), D(y), D(z)\}$ *et* $C = \{C_1, C_2\}$. $Var(C_1) = \{x, y\}$, $Var(C_2) = \{x, y, z\}$. *Comme* $|Var(C_1)| = 2$ *et* $|Var(C_2)| = 3$ *alors* C_1 *est une contrainte binaire et* C_2 *est une contrainte ternaire.*

Une instance de CSP est dite binaire si toutes ses contraintes le sont. Le problème de décision du CSP est un problème NP-complet [29] puisque le CSP binaire l'est déjà. Tout CSP d'arité supérieure à 2 peut être réduit en un CSP binaire. Cette binarisation nécessite la création de nouvelles variables, chacune encapsulant une contrainte d'arité supérieure à 2. Plusieurs techniques ont été développées pour résoudre le CSP binaire notamment les *réseaux de contraintes* [62, 6, 69]. Ici le CSP est représenté par un graphe appelé *graphe de contraintes* dans lequel les sommets sont les variables de X et les arcs entre deux sommets x_i et x_j sont les contraintes binaires liant ces deux variables. La binarisation est très peu utilisée dans la pratique malgré de nombreux travaux développés autour.

Exemple 1.2. *Considérons le CSP de l'Exemple 1.1. Sa binarisation nécessite la création d'une nouvelle variable* u *de domaine le produit cartésien des domaines des variables individuelles* $D(u) = \{(2, 1, 3), (2, 1, 4), (2, 1, 5), (2, 2, 3), (2, 2, 4), (2, 2, 5), (2, 3, 3), (2, 3, 4), (2, 3, 5), (2, 4, 3), (2, 4, 4), (2, 4, 5), (3, 1, 3), (3, 1, 4), (3, 1, 5), (3, 2, 3), (3, 2, 4), (3, 2, 5), (3, 3, 3), (3, 3, 4), (3, 3, 5), (3, 4, 3), (3, 4, 4), (3, 4, 5), (4, 1, 3), (4, 1, 4), (4, 1, 5), (4, 2, 3), (4, 2, 4), (4, 2, 5), (4, 3, 3), (4, 3, 4), (4, 3, 5), (4, 4, 3), (4, 4, 4), (4, 4, 5)\}$. *Les valeurs du domaine de* u *sont les triplets dans lequel la première composante désigne la valeur de* x *(*$x = 1st(u)$*), la sedonde composante désigne la valeur de* y *(*$y = 2nd(u)$*) et la dernière composante désigne la valeur de* z *(*$z = 3rd(u)$*). Le graphe ci-dessous représente le graphe de contraintes de ce CSP.*



Certaines techniques ont été spécifiquement développées pour explorer les contraintes n-aires ou *contraintes globales* [12]. La plus célèbre contrainte globale est la contrainte *all-different* qui s'assure qu'un ensemble de variables doivent prendre des valeurs deux à deux différentes. En ordonnancement, la contrainte globale *cumulative* [1, 12, 66] est utilisée pour modéliser la contrainte de ressources. Elle s'assure qu'à chaque instant, la capacité de la ressource n'est pas violée i.e., la quantité d'une ressource requise par un ensemble de tâches en cours d'exécution n'excède pas la capacité de la ressource.

1.1.2 Filtrage et propagation

Les contraintes sont utilisées dans un processus déductif pour détecter rapidement une inconsistance (au moins un domaine vide) ou réduire le domaine des variables. La réduction du domaine d'une variable se fait avec un algorithme de filtrage qui supprime du domaine les valeurs pour lesquelles il n'est pas possible de satisfaire la contrainte. Une des propriétés les plus intéressantes d'un tel algorithme de filtrage est la consistance.

Définition 1.3. *Lorsque la contrainte est unaire, le fait de supprimer toutes les valeurs du domaine incompatibles avec la contrainte est appelé consistance de nœuds.*

Cette consistance n'apporte que peu d'informations. Les contraintes portent généralement sur au moins deux variables.

Aussi appelée Arc Consistency (AC) pour consistance d'arc, c'est la méthode la plus employée qui s'applique aux contraintes binaires.

Définition 1.4. *Une contrainte binaire liant deux variables x_i et x_j est dite arc consistante si pour toute valeur v_i de x_i, il existe une valeur v_j dans le domaine $D(x_j)$ telle que l'instantiation (v_i, v_j) satisfait la contrainte.*

Une contrainte satisfait la consistance d'arc si chaque valeur de chaque variable appartient à une solution de la contrainte. On établit la consistance d'arc en supprimant les valeurs qui ne satisfont pas cette propriété. De nombreux algorithmes effectuent la consistance d'arc pour un CSP binaire (AC-1, AC-3, ...) [48].

Dans l'exemple 1.1, pour rendre la contrainte C_1 arc consistante, on doit supprimer la valeur 4 de $D(x)$ et les valeurs 1 et 2 de $D(y)$, ce qui permet, par exemple, de déduire que $D(x) = \{2, 3\}$ et $D(y) = \{3, 4\}$.

La consistance d'arc peut se généraliser en Hyperarc Consistency (HAC) pour *consistance d'hyperarc* ou *consistance d'arc généralisée* dans le cas d'une contrainte k-aires. Le principe est le même que pour les contraintes binaires : une contrainte est HAC si et seulement si chaque valeur de chaque variable appartient à une solution de la contrainte. On établit la consistance d'hyperarc en supprimant toutes les valeurs qui ne satisfont pas cette propriété. Une autre généralisation de la consistance d'arc est la k-consistance. Elle se définit comme suit :

Définition 1.5. *Soit C_l une contrainte sur k variables x_1, ..., x_k. $D(x_i)$ désigne le domaine de la variable x_i. C_l est "arc-consistante" si et seulement si pour toute variable x_i et pour toute valeur v_i de $D(x_i)$, il existe des valeurs v_1, ..., v_{i-1}, v_{i+1}, ..., v_k appartenant respectivement aux domaines $D(x_1)$, ..., $D(x_{i-1})$, $D(x_{i+1})$, ..., $D(x_k)$ telles que la contrainte C_l soit vérifiée lorsque $\forall j \in \{1, ..., k\}$, $x_j = v_j$.*

Le renforcement de la consistance d'arc pour une contrainte d'arité k se fait en $O(kd^k)$ dans le pire des cas où d est la taille du domaine des variables.

Définition 1.6. *Un problème est dit k-consistant si toute solution partielle de $k-1$ variables peut être étendue à une solution partielle de k variables en choisissant n'importe quelle valeur dans le domaine de la k-ième variable. Un problème est dit fortement k-consistant s'il est consistant pour tout $k' \leq k$. Lorsque $k = 2$, on parle de consistance d'arc tandis que pour $k = 3$, le problème est dit chemin-consistant.*

Pour certains problèmes de CSP, la propagation des contraintes suffit pour résoudre le problème. C'est par exemple le cas pour les problèmes d'ordonnancement simple (avec relaxation de la contrainte de ressource) modélisés en CSP. Plus k est grand, plus le filtrage est efficace. Cependant, du fait du grand nombre de combinaisons à tester, cela

reste souvent trop lourd. En général, même la 3-consistance ajoute beaucoup de contraintes au problème, augmentant ainsi le nouveau réseau de contraintes. Pour plus de détails sur ce sujet voir [72, 73].

Lorsque le domaine des variables est très grand, il est souvent plus adéquat de propager les contraintes suivant les bornes des variables, on parle alors de *consistance de bornes* ou *arc-B-consistance* [53]. On s'assure que les bornes des domaines des variables sont arc-consistantes pour la contrainte. Une variable x de domaine $D(x)$ sera alors représentée sous la forme d'un intervalle $D(x) = [\underline{x}, \overline{x}]$ où $\underline{x}$ et $\overline{x}$ sont respectivement la plus petite et la plus grande valeur de $D(x)$.

Définition 1.7. *Soit C_l une contrainte sur k variables x_1, ..., x_k, $D(x_i) = [\underline{x}_i, \overline{x}_i]$ le domaine de la variable x_i. C_l est "arc-B-consistante" si et seulement si pour toute variable x_i et pour toute valeur $v_i \in \{\underline{x}_i, \overline{x}_i\}$, il existe des valeurs v_1, ..., v_{i-1}, v_{i+1}, ..., v_k appartenant respectivement aux domaines $D(x_1)$, ..., $D(x_{i-1})$, $D(x_{i+1})$, ..., $D(x_k)$ telles que la contrainte C_l soit vérifiée lorsque $\forall j \in \{1, ..., k\}$, $x_j = v_j$.*

Dès lors qu'un algorithme de filtrage associé à une contrainte modifie le domaine d'une variable, les conséquences de cette modification sont étudiées pour les autres contraintes impliquant cette variable. Autrement dit, les algorithmes de filtrage des autres contraintes sont appelés afin de déduire éventuellement d'autres suppressions. On dit alors qu'une modification a été propagée. Ce mécanisme de propagation est répété jusqu'à ce que plus aucune modification n'apparaisse, on dit qu'on a atteint le point fixe. L'idée sous-jacente à ce mécanisme est d'essayer d'obtenir des déductions globales. En effet, on espère que la conjonction des déductions obtenues pour chaque contrainte prise indépendamment conduira à un enchaînement des déductions. C'est-à-dire que cette conjonction est plus forte que l'union des déductions obtenues indépendamment les unes des autres. Si on note Γ l'ensemble des algorithmes de filtrage et Δ l'ensemble des domaines courants des variables, l'Algorithme 1 est un algorithme de propagation.

En revenant à l'exemple 1.1, la propagation de la contrainte C_2 permet d'avoir $D(z) = \{5\}$ et une seconde propagation donne $D(x) = \{2\}$ et $D(y) = \{3\}$. On dit alors que toutes les variables sont instanciées.

Algorithm 1: Propagation des contraintes jusqu'au point fixe

Entrées: Γ ensemble des algorithmes de filtrage
Entrées: Δ ensemble des domaines courants des variables
Sorties: retourne le point fixe

1 **répéter**
2 $\quad \Delta_{old} := \Delta$;
3 $\quad$ **pour chaque** $(\gamma \in \Gamma)$ **faire**
4 $\quad\quad \Delta := \gamma(\Delta)$;
5 **jusqu'à** $\Delta = \Delta_{old}$;

1.1.3 Recherche de solution

Pour des raisons de complexité, les conséquences des contraintes sur les domaines des variables ne sont pas calculables en un temps limité. Il est donc nécessaire de développer une recherche arborescente pour déterminer s'il existe ou non une affectation valide de valeurs aux variables. Le principe commun à toutes les approches que nous allons étudier est d'explorer méthodiquement l'ensemble des affectations possibles jusqu'à, soit trouver une solution (quand le CSP est consistant), soit démontrer qu'il n'existe pas de solution (quand le CSP est inconsistant). Les heuristiques concernant l'ordre d'instantiation des variables dépendent de l'application considérée et sont difficilement généralisables. En revanche, il existe de nombreuses heuristiques d'ordre d'instantiation des variables qui permettent bien souvent d'accélérer considérablement la recherche. L'idée générale consiste à instancier en premier les variables les plus critiques, c'est-à-dire celles qui interviennent dans beaucoup de contraintes et/ou ne peuvent prendre que très peu de valeurs. L'ordre d'instantiation des variables peut être :

- **statique**, quand il est fixé avant de commencer la recherche. Par exemple, on peut ordonner les variables en fonction du nombre de contraintes portant sur elles : l'idée est d'instancier en premier les variables les plus contraintes, c'est-à-dire celles qui participent au plus grand nombre de contraintes *MostConstrained*.
- **dynamique**, quand la prochaine variable à instancier est choisie dynamiquement à chaque étape de la recherche. Par exemple, l'heuristique échec d'abord (*MinDomaine* ou *first-fail* en anglais) consiste à choisir, à chaque étape, la variable dont le domaine a le plus petit nombre de valeurs localement consistantes avec l'affectation partielle en cours. Cette heuristique est généralement couplée aux techniques de renforcement

de la consistance locale (algorithme MLC pour Maintaining Local Consistency), qui filtre les domaines des variables à chaque étape de la recherche pour ne garder que les valeurs qui satisfont un certain niveau de consistance locale.

Les principales articulations de la PPC sont la définition du problème, la propagation des contraintes, la prise de décision et le retour en arrière. Ces sous-modules coopèrent pendant le processus de recherche d'une solution.

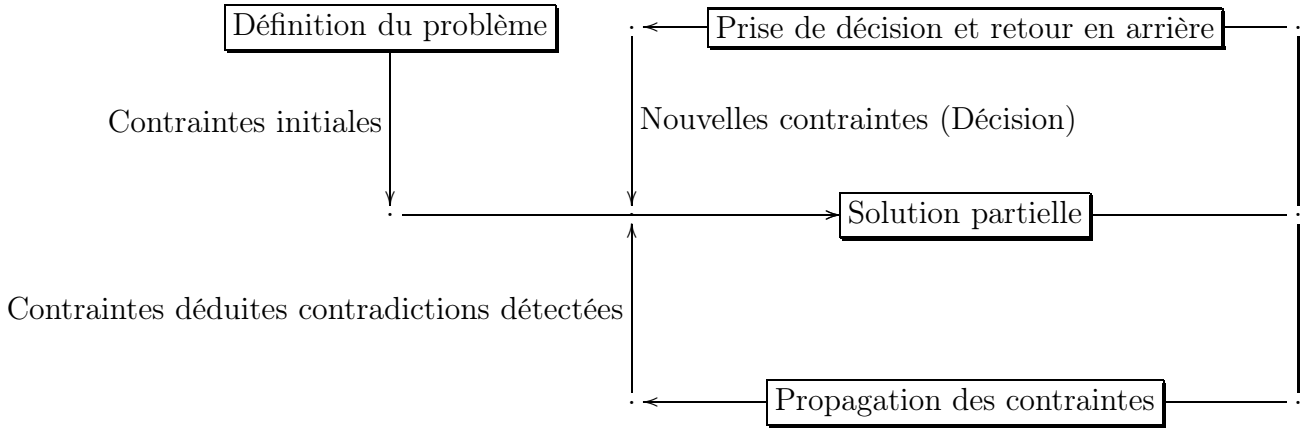


FIGURE 1.1 – Les modules d'un système de programmation par contraintes

Le comportement général d'un système de contraintes tel qu'il est proposé par Philippe Baptiste dans [9] peut se résumer à la Figure 1.1. Notons que la définition du problème, la propagation des contraintes et la phase de prise de décision sont clairement séparés.

1. En premier lieu, le problème est défini en termes de variables et de contraintes.
2. Puis, les algorithmes de propagation de ces contraintes sont spécifiés. Le problème courant est réduit après propagation des contraintes. A l'issue de cette réduction, il est décidé si le nouveau problème ainsi obtenu est résolu, réalisable ou peut être réalisable.
   - Si le problème est résolu alors on retourne la solution à l'utilisateur.
   - Si le problème est réalisable alors on fait appel au module Prise de décision et retour en arrière qui permet de prendre une décision modifiant le problème courant (On obtient ainsi une solution partielle du problème). Puis, on retourne à l'étape de propagation des contraintes sur le nouveau problème.
   - Si le problème peut être réalisable, on effectue un retour arrière (backtrack) sur

l'une des décisions prise précédemment. Cette décision est alors défaite et une autre alternative est prise. Si aucune décision n'avait été prise précédemment, alors le problème initial n'a pas de solution.

En pratique, l'utilisateur d'un système de contraintes peut soit utiliser des contraintes prédéfinies, par exemple des contraintes sur des entiers ou sur des ensembles, soit définir ses propres contraintes dont il pourra expliciter les méthodes de propagation.

3. Enfin, le mécanisme de prise de décision, c'est-à-dire la façon dont l'arbre de recherche est construit, est défini. Il précise le type de décisions qui doivent être prises au fur et à mesure de la recherche (e.g., instancier une variable à une valeur, ordonner une paire de tâches).

Les algorithmes permettant de résoudre des CSPs sont appelés des *solveurs* de contraintes. Certains de ces solveurs ont été intégrés dans des langages de programmation, définissant ainsi un nouveau *paradigme* de programmation appelé programmation par contraintes (PPC) : pour résoudre un CSP avec un langage de programmation par contraintes, il suffit de spécifier les contraintes, leur résolution étant prise en charge automatiquement (sans avoir besoin de le programmer) par les solveurs de contraintes intégrés au langage. Nous renvoyons le lecteur à [3, 23, 63, 71, 28] pour une introduction plus poussée à la PPC.

Il est possible de réutiliser des algorithmes de propagation d'une application à l'autre. C'est l'une des raisons de l'engouement des industriels pour des outils de programmation par contraintes, parfois au détriment d'autres techniques de résolution, comme la programmation linéaire qui, même si elle se montre extrêmement performante sur certains problèmes, nécessite souvent l'élaboration de modèles complexes. Parmi les systèmes de contraintes (commerciaux ou de domaine public), citons ECLiPSe [26], ILOG CP Optimizer [34], CHIP [18, 1], SICStus Prolog [68], Choco [19], Gecode [30], mistral [56] etc...

1.1.4 Optimisation

Lorsqu'il est question de déterminer une solution qui optimise un certain coût, on parle de Constraint Satisfaction Optimisation Problem (CSOP). Dans toute la suite, seul le cas de minimisation est traité sans perte de généralité puisque maximiser z revient à minimiser $-z$.

Définition 1.8. *Une instance P du CSOP est définie par la donnée d'un 4-uplet $P = (X, D, C, z)$ où $P' = (X, D, C)$ est une instance de CSP et z une fonction objectif $z : D(x_1) \times D(x_2) \times ... \times D(x_n) \to \mathbb{R}$ définie par*

$$z_{min}(P) = \begin{cases} \min\limits_{a \in Sol(P')} z(a) & si\ Sol(P') \neq \emptyset \\ \infty & sinon \end{cases}$$

$Sol(P')$ désigne l'ensemble des solutions du CSP P'.

Plusieurs méthodes ont été développées pour déterminer une solution du CSOP. Elles sont basées sur les techniques de recherche d'une solution du CSP. Soit z^* la solution minimale de $P = (X, D, C, z)$.

Une première approche consiste à résoudre une succession de problèmes $P' = (X, D, C)$. Si le problème initial P' n'a pas de solution, alors le problème d'optimisation n'a pas de solution sinon, tant que P' possède une solution X^*, on ajoute la contrainte $z(X) < z(X^*)$ à P'. La dernière solution trouvée est une solution optimale du problème. Dans une recherche arborescente, cela revient soit à repartir de la racine de l'arbre à chaque nouvelle résolution, soit à modifier le problème de décision en cours de recherche en ajoutant la contrainte dès qu'une feuille est atteinte.

Une recherche par dichotomie peut être aussi utilisée. On désigne par P'_H le problème P' auquel on a ajouté la contrainte $z(X) < H$. Connaissant LB et UB respectivement la borne inférieure et la borne supérieure de z^*, on cherche dans l'intervalle $]LB; UB]$ la plus grande (resp. petite) valeur H rendant P'_H non réalisable (resp. réalisable). Si pour une valeur de H, P'_H est non réalisable, alors l'intervalle est réduit en posant $LB = H$. Si par contre P'_H possède une solution X^*, l'intervalle est réduit en posant $UB = z(X^*)$. On remarque que tout au long du processus, $LB < z^* \leq UB$. Lorsque $LB = UB$, la solution optimale est la dernière solution trouvée.

Une recherche itérative peut aussi être adoptée suivant les valeurs croissantes ou décroissantes de H. Avec une valeur initiale de H telle que $H < z^*$, on itère suivant les valeurs croissantes de H en résolvant le problème P'_H jusqu'à obtenir une solution. Cette première solution est la solution optimale de P. Inversement, en commençant avec une valeur initiale $H > z^*$, on itère suivant les valeurs décroissantes de H en résolvant le problème P'_H jusqu'à ce que le problème soit non réalisable. La dernière solution trouvée

est la solution optimale du problème.

1.2 Problème d'ordonnancement

1.2.1 Généralités

L'ordonnancement est l'allocation des ressources aux tâches dans le temps. Plusieurs typologies complètent cette définition très générale [7, 21, 45]. Les problèmes varient suivant la nature des tâches (préemptives ou non), des ressources (disjonctives, cumulatives, renouvelables ou non), les contraintes portant sur les tâches (contraintes de disponibilité, contraintes de précédence) et les critères à optimiser (fin du projet, retard, coût total). La résolution d'un problème d'ordonnancement consiste à placer dans le temps les activités ou tâches, compte tenu des contraintes temporelles (délais, contraintes d'enchaînement, ...) et des contraintes portant sur l'utilisation et la disponibilité des ressources requises par les tâches. Nous distinguons ici les quatre dimensions de la classification considérées dans [9].

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

le plus souvent, un critère doit être optimisé. Bien que le makespan, i.e., la date de fin de l'ordonnancement, soit le critère le plus fréquemment utilisé (ce qui d'ailleurs ne correspond pas forcément à un critère d'une grande utilité concrète), d'autres critères peuvent être considérés. Citons par exemple, le nombre de tâches exécutées dans un certain délai, le retard moyen ou pondéré, ou encore le pic d'utilisation d'une ressource.

La théorie de l'ordonnancement est un champ de recherches très large qui, tant d'un point de vue pratique qu'appliqué, a donné lieu à un nombre important de publications. Nous renvoyons le lecteur à [7, 21, 45] pour une introduction plus poussée à ce domaine.

1.2.2 Notations : le problème d'ordonnancement cumulatif à une ressource

Le problème d'ordonnancement cumulatif à une ressource ou CuSP pour Constraint Scheduling Problem [10] est défini par la donnée d'une ressource de capacité supérieure à 1, un ensemble fini de tâches avec durée d'exécution et la quantité de ressource requise pour chaque tâche. Il est question de déterminer la date d'exécution de chaque tâche de sorte que la contrainte de ressource soit satisfaite, i.e., à tout instant, la capacité de la ressource reste inviolée. Formellement le problème se définit de la manière suivante :

Définition 1.9 (CuSP)**.** *Une instance du CuSP est définie par la donnée du 6-uplets* (C, T, r, d, p, c) *où*

- C *désigne la capacité de la ressource*
- T *l'ensemble des* $|T| = n$ *tâches non interruptibles ;*
- r_i $(i \in T)$ *la date de début au plus tôt de la tâche* i *;*
- d_i $(i \in T)$ *la date de fin au plus tard de la tâche* i *;*
- p_i $(i \in T)$ *la durée d'exécution de la tâche* i *;*
- c_i $(i \in T)$ *la quantité de ressource requise par la tâche* i *au cours de son exécution.*

Une solution d'un CuSP $P = (C, T, r, d, p, c)$ *est une affectation d'une date de début* $s_i \in \mathbb{Z}$ *pour chaque tâche* $i \in T$ *telle que*

$$\forall i \in T : \quad r_i \leq s_i \leq s_i + p_i \leq d_i \tag{1.1}$$

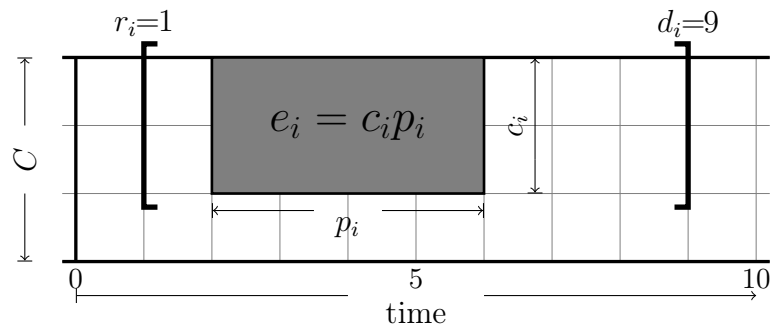


FIGURE 1.2 – Profil d'une ressource de capacité 3 et attributs d'une tâche.

$$\forall \tau : \sum_{i \in T/ \ s_i \leq \tau < s_i + p_i} c_i \leq C. \tag{1.2}$$

Les contraintes des inégalités (1.1) assurent que les dates de début et de fin de chaque tâche sont réalisables alors que l'inégalité (1.2) assure que la contrainte de ressource est satisfaite à tout instant. C'est sur les instances de CuSP que sont appliqués les algorithmes de filtrage.

Dans toute la suite, chaque fois que l'on parlera du CuSP, on adoptera les notations suivantes : L'énergie de la tâche i sera noté $e_i = c_i p_i$. Nous étendons la notion de date de début au plus tôt, date de fin au plus tard et d'énergie des tâches aux ensembles de tâches en posant pour tout ensemble non vide de tâches Ω

$$r_\Omega = \min_{j \in \Omega} r_j, \qquad d_\Omega = \max_{j \in \Omega} d_j\,, \qquad e_\Omega = \sum_{j \in \Omega} e_j.$$

Cette notation peut être étendue à l'ensemble vide en posant

$$r_\emptyset = +\infty, \qquad d_\emptyset = -\infty\,, \qquad e_\emptyset = 0.$$

A l'aide des attributs des tâches, il est possible de définir d'autres caractéristiques des tâches à savoir :

- la date de fin au plus tôt $ect_i = r_i + p_i$
- la date de début au plus tard $lst_i = d_i - p_i$

Les conditions suivantes doivent être satisfaites pour que le problème soit réalisable :

$$c_i \leq C, \quad r_i \leq lst_i \leq d_i, \quad r_i \leq ect_i \leq d_i.$$

1.2.3 La contrainte temporelle

La contrainte temporelle est une contrainte qui lie le début ou la fin de deux tâches par une relation linéaire. Par exemple, une contrainte de précédence entre i et j notée $i \lessdot j$ pour signifier que la tâche j ne peut commencer qu'après l'exécution complète de la tâche i, est représentée par l'équation linéaire $s_i + p_i \leq s_j$. Cette contrainte est propagée suivant la règle

$$i \lessdot j \Rightarrow r_j \geq \max(r_j, ect_i) \wedge lst_i \leq \min(lst_i, lst_j - p_i). \tag{1.3}$$

L'application de cette règle jusqu'à ce qu'aucun ajustement supplémentaire ne soit possible est équivalente au calcul du plus long chemin dans le graphe de précédence [20]. Quelques méthodes de propagation de la contrainte temporelle sont décrites dans [52]. [5] propose un algorithme incrémental de chemin-consistance aux bornes pour la propagation de contraintes temporelles.

1.2.4 La contrainte de ressource

Une contrainte de ressource spécifie qu'à chaque instant t, la capacité de la ressource est supérieure ou égale à la somme, sur toutes les tâches, des capacités requises à l'instant t. Cette contrainte varie suivant la nature de la ressource. Ainsi, lorsque la ressource est unaire, on utilise la contrainte globale unary ou disjunctive [74, 24]. Par contre, lorsque la ressource est de capacité discrète, on utilise la contrainte globale cumulative [1, 66].

1.2.4.1 La contrainte globale unary ou disjunctive

La ressource étant de capacité unaire, deux tâches nécessitant cette ressource ne peuvent pas s'exécuter en parallèle. Si deux tâches i et j doivent s'exécuter sur une ressource disjonctive, alors on a $s_i + p_i \leq s_j \vee s_j + p_j \leq s_i$. Le propagateur de la contrainte unary utilise plusieurs algorithmes de filtrage [74, 24, 10, 11, 16, 33].

1.2.4.2 La contrainte globale cumulative

Initialement définie dans [1], la contrainte globale cumulative s'assure que pour un ensemble T de tâches et à chaque instant τ, la capacité de la ressource n'est pas excédée,

i.e.,

$$\forall \tau : \sum_{i \in T/\ s_i \leq \tau < s_i + p_i} c_i \leq C$$

où s_i est la variable décisionnelle qui représente la date de début de la tâche i et appartient à l'intervalle d'entiers $[r_i ... lst_i]$. Une explication du propagateur de cette contrainte est donnée dans [66]. Le CuSP est un problème NP-Complet [8]. Il n'existe donc pas d'algorithme polynomial en temps implémentant la contrainte cumulative. Le propagateur de la contrainte cumulative utilise des algorithmes de filtrage implémentant une relaxation pouvant être calculée en temps polynomial [10, 77, 75, 54, 40, 64, 42, 43].

1.3 Algorithmes de filtrage existants

Il existe plusieurs algorithmes de filtrage pour la propagation de la contrainte de ressources. Ces algorithmes polynomiaux sont des relaxations de la contrainte *cumulative*. Ils sont basés sur l'intégrité des intervalles encore appelée overload en anglais et connu sous le nom de E-Réalisable [54] ou overload checking [76, 78].

1.3.1 E-réalisabilité

Elle n'est pas une règle au sens strict du terme car elle ne permet pas de réduire le domaine des variables. Il est évident que lorsque qu'une instance de CuSP admet un ordonnancement alors, pour tout ensemble Ω de tâches, son énergie est inférieure ou égale au produit de la capacité de la ressource C par la longueur de la fenêtre de temps $d_\Omega - r_\Omega$. Cette contrainte redondante est donnée dans la Définition 1.10

Définition 1.10 (E-Réalisable)**.** *Une instance de CuSP est E-Réalisable si* $\forall \Omega \subseteq T$, $\Omega \neq \emptyset$

$$C\,(d_\Omega - r_\Omega) \geq e_\Omega. \tag{1.4}$$

Il est évident que si une instance de CuSP est Réalisable alors elle est E-Réalisable mais la réciproque n'est pas toujours vraie. Si la condition (1.4) n'est pas satisfaite pour un ensemble Ω de tâches, alors l'instance de CuSP n'est pas réalisable et l'échec est détecté. Cela se traduit par la règle ci-dessous :

$$\exists \Omega \subseteq T,\ \Omega \neq \emptyset : (e_\Omega > C\,(d_\Omega - r_\Omega) \Longrightarrow Echec)$$

Il existe plusieurs algorithmes pour vérifier qu'une instance de CuSP est E-Réalisable. On montre dans [54] qu'une instance de CuSP est E-Réalisable lorsque tous ses intervalles de tâches satisfont la condition (1.4).

Définition 1.11 (Intervalle de tâches)**.** *[16] Soit $L, U \in T$ deux tâches d'une instance de CuSP (éventuellement identiques). L'intervalle de tâches $\Omega_{L,U}$ est l'ensemble de tâches*

$$\Omega_{L,U} = \{j \in T \quad | \quad r_L \leq r_j \wedge d_j \leq d_U\} \tag{1.5}$$

Il est proposé dans [60, 54] des algorithmes quadratiques pour tester qu'une instance de CuSP est E-Réalisable. Tout récemment, des algorithmes efficaces de complexité $\mathcal{O}(n \log n)$ ont été proposés pour tester qu'une instance de CuSP est E-Réalisable [76, 78]. Il est possible de modifier cette règle afin de détecter des échecs et réduire les fenêtres de temps des tâches.

Étant donné un ensemble de tâches Ω d'une instance E-Réalisable de CuSP et une nouvelle tâche $i \notin \Omega$, les nouvelles règles doivent déterminer la position de i par rapport à celle de Ω. La date de début au plus tôt (resp. de fin au plus tard) courante de la tâche i est alors ajustée à une valeur supérieure (resp. inférieure). C'est par exemple le cas dans le timetabling, l'edge-finding, l'extended edge-finding, le not-first/not-last, le timetable edge finding, le raisonnement énergétique etc... Ces règles sont des relaxations de la contrainte cumulative et peuvent être réalisées en temps polynomial. Chaque règle élimine différents types d'inconsistances. C'est pourquoi il est judicieux de combiner ces algorithmes de filtrage afin d'obtenir le maximum de filtrage.

1.3.2 La contrainte timetabling [10, 8]

Cette règle utilise la structure de données Timetable pour assurer la consistance des bornes de la formule

$$\forall \tau : \sum_{i \in T,\ s_i \leq \tau < s_i + p_i} c_i \leq C. \tag{1.6}$$

En conservant les informations à chaque instant τ sur l'utilisation des ressources et la disponibilité de celles-ci, il est possible de propager la contrainte de ressource. C'est ainsi que lorsqu'en un instant τ donné, si la quantité de ressource disponible est inférieure à la capacité d'une tâche, alors cette tâche ne peut pas être ordonnancée à cet instant-là. Cet

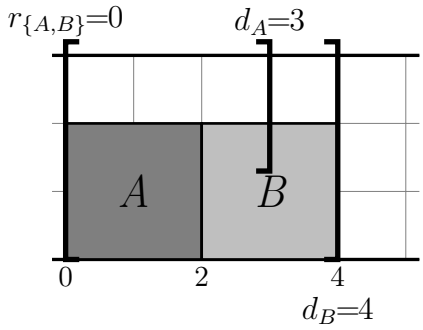


FIGURE 1.3 – Une instance de CuSP de 2 tâches s'executant sur une ressource de capacité 3.

algorithme est généralement couplé avec l'edge-finding [10, 8, 33]. Cette règle n'a pas fait l'objet de notre étude. Elle est illustrée dans l'Exemple 1.3.

Exemple 1.3. *Considérons l'instance de CuSP dans lequel deux ($n = 2$) tâches doivent être exécutées sur une ressource de capacité $C = 3$. Les dates de début au plus tôt, de fin au plus tard et la durée d'exécution de chaque tâche sont données dans la Figure 1.3.*

- *Dans cette instance, la tâche A doit nécessairement être exécutée dans l'intervalle $[1; 2]$. De ce fait, l'exécution de la tâche B ne peut commencer ni à l'instant 0, ni à 1. La date de début au plus tôt de B est alors ajustée de 0 à 2. C'est la première propagation.*
- *Après cette propagation, la date de fin au plus tard de la tâche A est alors ajustée de 3 à 2 car ne pouvant être exécutée à cette date-là.*

1.3.3 L'edge-finding

Probablement la règle de filtrage la plus utilisée en ordonnancement, la règle edge-finding détermine pour une instance E-Réalisable de CuSP, la position d'une tâche i par rapport à un ensemble de tâches Ω toutes devant être exécutées sur une ressource de capacité C. Plus précisément, elle détermine si une tâche i peut ou non être ordonnancée avant ou après l'ensemble Ω de tâches ($i \notin \Omega$) toutes nécessitant la même ressource. L'Exemple 1.4 illustre mieux cette règle.

Exemple 1.4. *Considérons l'instance de CuSP dans lequel six ($n = 6$) tâches doivent être exécutées sur une ressource de capacité $C = 3$. Les dates de début au plus tôt, de*

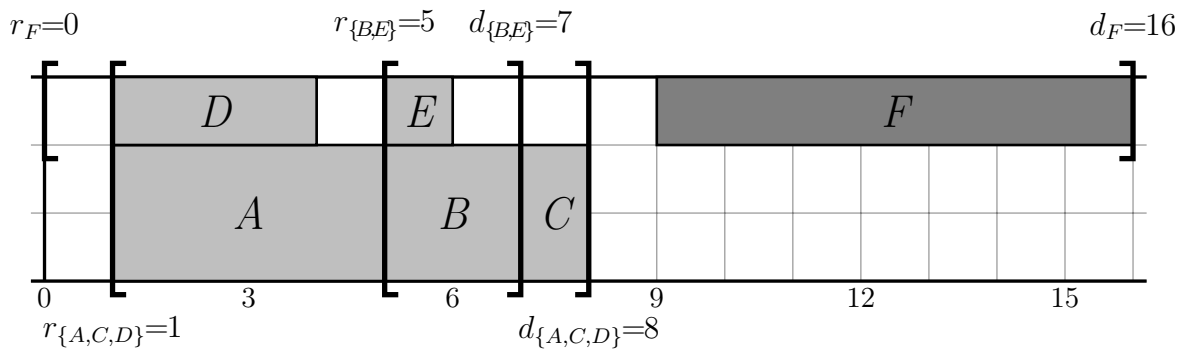


FIGURE 1.4 – Une instance de CuSP de 6 tâches devant être exécutées sur une ressource de capacité 3.

fin au plus tard, la durée d'exécution et le nombre de ressources requises par chaque tâche sont donnés dans la Figure 1.4.

Quelle que soit la manière avec laquelle on range les tâches de l'ensemble $\Omega = \{A, B, C, D, E\}$*, la tâche* F *ne précède aucune tâche de* Ω*. Cette conclusion est tirée de la logique suivante : quel que soit l'ordre donné aux tâches de l'ensemble* $\Omega \cup \{F\}$*, la tâche* F *ne peut pas s'achever avant l'instant* $t = 8$ *mais les tâches de* Ω *s'achèvent avant l'instant* $t = 8$*. Dans ce cas on dit que la tâche* F *finit après toutes les tâches de* Ω*.*

Soit $\Omega \subset T$ un ensemble de tâches et $i \in T \setminus \Omega$ une tâche toutes devant être exécutées sur une ressource de capacité C. Si on désigne par $ect_{\Omega\cup\{i\}}$ (resp. $lst_{\Omega\cup\{i\}}$) la date de fin (resp. de début) au plus tôt (resp. au plus tard) de l'ensemble $\Omega \cup \{i\}$ alors on a les règles :

$$ect_{\Omega\cup\{i\}} > d_\Omega \Longrightarrow \Omega \lessdot i \tag{1.7}$$

$$lst_{\Omega\cup\{i\}} < r_\Omega \Longrightarrow \Omega \gtrdot i \tag{1.8}$$

où $\Omega \lessdot i$ signifie que toutes les tâches de Ω finissent avant la fin de la tâche i et $\Omega \gtrdot i$ toutes les tâches de Ω commencent après le début de la tâche i. Malheureusement, le calcul de la date de fin au plus tôt (resp. début au plus tard) d'un ensemble de tâches est complexe. C'est pour quoi, la règle edge-finding utilise une formulation plus simple pour déterminer une borne inférieure (resp. supérieure) de la date de fin au plus tôt (resp. début au plus tard) d'un ensemble de tâches noté $Ect_{\Omega\cup\{i\}}$ (resp. $Lst_{\Omega\cup\{i\}}$). Cette notation est tirée de [75, 76] et on a :

$$Ect_\Omega = r_\Omega + \left\lceil \frac{e_\Omega}{C} \right\rceil \tag{1.9}$$

$$Lst_\Omega = d_\Omega - \left\lceil \frac{e_\Omega}{C} \right\rceil . \tag{1.10}$$

L'expression standard utilisée pour la règle d'edge-finding est donnée dans la définition suivante.

Définition 1.12 (Règle de détection de l'Edge-finding cumulatif classique)**.** *Soit T un ensemble de tâches d'une instance de CuSP de capacité C.*

$$e_{\Omega\cup\{i\}} > C\left(d_\Omega - r_{\Omega\cup\{i\}}\right) \Longrightarrow \Omega \lessdot i \tag{1.11}$$

$$e_{\Omega\cup\{i\}} > C\left(d_{\Omega\cup\{i\}} - r_\Omega\right) \Longrightarrow \Omega \gtrdot i \tag{1.12}$$

Dans toute la suite, nous nous intéressons à la règle (1.11) qui permet d'ajuster la date de début au plus tôt des tâches. Les résultats similaires peuvent être établis sur la version symétrique donnée par la règle (1.12)

Soit $s = (s_1, ..., s_n)$ avec $n = |T|$ un ordonnancement d'une instance de CuSP où s_j désigne la date de début de la tâche j. La condition $e_{\Omega\cup\{i\}} > C\left(d_\Omega - r_{\Omega\cup\{i\}}\right)$ étant satisfaite, on définit

$$ct(\Omega, s, c_i) = \min\{t \geq r_\Omega : \forall t' \geq t, \sum_{j\in\Omega\wedge s_j\leq t'<s_j+p_j} c_j \leq C - c_i\}. \tag{1.13}$$

comme étant le plus petit instant après lequel dans l'ordonnancement s, au moins c_i unité de la ressource est disponible. Le minimum des $ct(\Omega, s, c_i)$ à travers tous les ordonnancements possibles est la borne inférieure à laquelle la tâche i doit être ajustée. Le problème qui consiste à calculer cette borne inférieure de r_i donnée par la relation (1.13) est NP-difficile [60, 9]. Nuijten [60] propose une nouvelle borne inférieure qui peut être calculé de manière efficace. Ainsi, lorsque la condition $e_{\Omega\cup\{i\}} > C\left(d_\Omega - r_{\Omega\cup\{i\}}\right)$ est satisfaite, la tâche i doit être ordonnancée après toutes les tâches de Ω et sa date de début est alors ajustée. Pour son ajustement on divise l'énergie de Ω en deux : l'énergie qui n'influence pas l'ordonnancement de la tâche i i.e. $(C - c_i)(d_\Omega - r_\Omega)$ et celle qui affecte l'ordonnancement de la tâche i i.e. $rest\,(\Omega, c_i) = e_\Omega - (C - c_i)\,(d_\Omega - r_\Omega)$. Cette énergie est utilisée pour ajuster la date de début au plus tôt de la tâche i avec la formule

$$r_i \geq r_\Omega + \left\lceil \frac{1}{c_i} rest\,(\Omega, c_i) \right\rceil . \tag{1.14}$$

Lorsque la tâche i est précédée par toutes les tâches de Ω alors elle est précédée par toutes les tâches de tout sous ensemble Θ de Ω i.e. $(\Omega \lessdot i) \quad \Rightarrow \quad (\forall\Theta \subseteq \Omega, \Theta \lessdot i)$. On applique

alors le même raisonnement pour les sous ensembles de Ω cherchant ainsi l'ajustement maximal avec

$$r_i \geq r_\Theta + \left\lceil \frac{1}{c_i} rest\left(\Theta, c_i\right) \right\rceil \tag{1.15}$$

pour tout $\Theta : \emptyset \neq \Theta \subseteq \Omega$ vérifiant $rest\left(\Theta, c_i\right) > 0$ avec

$$rest\left(\Theta, c_i\right) = e_\Theta - \left(C - c_i\right)\left(d_\Theta - r_\Theta\right).$$

Pour plus d'informations sur cette borne inférieure voir [60, 9].

Exemple 1.5. *Considérons les données de l'instance de CuSP de la Figure 1.4. En appliquant la règle d'edge-finding pour $i = F$ et $\Omega = \{A, B, C, D, E\}$, on obtient $e_\Omega + e_F = 18 + 7 = 25$ alors que $C(d_\Omega - r_{\Omega \cup \{F\}}) = 3 \times (8 - 0) = 24$. La tâche F finit après toutes les tâches de Ω i.e., après l'instant 8. En particulier les ensembles $\{A, B, C, D, E\}$ et $\{B, E\}$ précèdent la tâche F.*

- *Pour $\Theta = \{A, B, C, D, E\}$, l'ajustement obtenu est $r_F \geq 5 = r_\Theta + \left\lceil \frac{1}{c_F} rest\left(\Theta, c_F\right) \right\rceil = 1 + \left\lceil \frac{1}{1} 4 \right\rceil$ car $rest(\Theta, c_F) = 18 - 2 \times (8 - 1) = 4 > 0$.*
- *Pour $\Theta = \{B, E\}$, l'ajustement obtenu est $r_F \geq 6 = r_\Theta + \left\lceil \frac{1}{c_F} rest\left(\Theta, c_F\right) \right\rceil = 5 + \left\lceil \frac{1}{1} \right\rceil$ car $rest(\Theta, c_F) = 5 - 2 \times (7 - 5) = 1 > 0$.*

La Proposition 1.1 donne les conditions pour lesquelles toutes les tâches de $\Omega \subset T$ finissent avant la fin de la tâche $i \in T \setminus \Omega$. Il s'agit des règles de détection d'edge-finding telles que étudiées dans [75, 40].

Proposition 1.1. *Soit $\Omega \subset T$ un ensemble non vide de tâches d'une instance E-Réalisable de CuSP et soit $i \notin \Omega$ une tâche.*

$$e_{\Omega \cup \{i\}} > C\left(d_\Omega - r_{\Omega \cup \{i\}}\right) \Rightarrow \Omega \lessdot i \tag{EF}$$

$$r_i + p_i \geq d_\Omega \quad \Rightarrow \quad \Omega \lessdot i \tag{EF1}$$

Démonstration. Nous prouverons chaque item de la proposition.

- Supposons que toutes les tâches sont ordonnancées et qu'il existe une tâche de Ω qui précède la tâche i. Posons $\Omega' = \Omega \cup \{i\}$. L'énergie consommée par les tâches de Ω' est $e_{\Omega'} = e_{\Omega \cup \{i\}}$ qui est d'après la relation $e_{\Omega \cup \{i\}} > C\left(d_\Omega - r_{\Omega \cup \{i\}}\right)$ supérieur à l'énergie diponible entre $r_{\Omega'} = r_{\Omega \cup \{i\}}$ et $d_{\Omega'} = d_\Omega$. Ce qui contredit le fait que l'instance soit E-Réalisable.

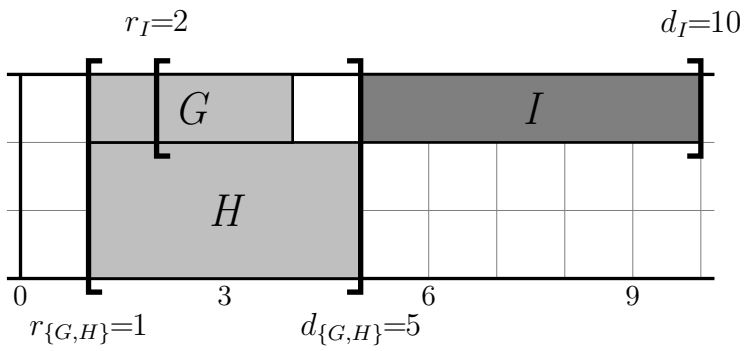


FIGURE 1.5 – Trois tâches devant être exécutées sur une ressource de capacité $C = 3$.

– Supposons une fois de plus que toutes les tâches sont ordonnancées et notons $s = (s_1, ..., s_n)$ l'ordonnacement ainsi obtenu. Comme $r_i + p_i$ est la date de fin au plus tôt de la âche i, on a : $s_i + p_i \geq r_i + p_i$. De plus, pour toute tâche $j \in \Omega$, on a : $d_\Omega \geq s_j + p_j$. De l'inégalité $r_i + p_i \geq d_\Omega$ il vient que pour toute tâche $j \in \Omega$, $\quad s_i + p_i \geq s_j + p_j$. Donc toutes les tâches de Ω finissent avant la fin de la tâche i.

□

La règle (EF) est la version standard de la règle de détection de l'edge-finding [60, 9, 54]. La règle (EF1) est l'équivalent des règles "Earliest Finishing time of First" et "Latest Starting time of Last" (EFF/LSL) de [16] a été introduite par Vilím dans [75] augmentant ainsi le pouvoir de détection de l'edge-finding et permettant à la nouvelle règle d'être légèrement plus puissante par rapport à la version standard. Dans Exemple 1.6 tiré de [40], nous illustrons le comportement de la règle (EF1) sur une instance de CuSP où trois tâches doivent être exécutées sur une ressource de capacité trois.

Exemple 1.6. *Trois travailleurs identiques (ressource de capacité $C = 3$) doivent exécuter différentes tâches $T = \{G, H, I\}$ tel que l'illustre la Figure 1.5.*

Il est claire que dans toute solution réalisable de cette instance de CuSP $\{G, H)\} \lessdot I$ car $G \lessdot I$ et $H \lessdot I$. En effet, si la tâche G ne s'achève pas avant la fin de la tâche I, la contrainte de ressource sera violée car $r_I + p_I \geq d_G$ et $c_G + c_H = C$. En appliquant la règle d'edge-finding classique (EF) *sur cette instance, la condition $\{G, H\} \lessdot I$ n'est pas détectée et l'ajustement de la date de début au plus tôt de la tâche I de 2 à 4 est manqué. Cet ajustement est détecté par la règle* (EF1).

La Définition 1.13 donne la spécification d'un algorithme complet d'edge-finding.

Définition 1.13. *Un algorithme edge-finding reçoit en entrée une instance E-Réalisable de CuSP. Il produit en sortie un vecteur*

$$\langle LB_1, ..., LB_n \rangle$$

où

$$LB_i = \max \left(r_i, \max_{\substack{\Omega \subseteq T \\ i \notin \Omega \\ \alpha(\Omega, i)}} \max_{\substack{\Theta \subseteq \Omega \\ rest(\Theta, c_i) > 0}} r_\Theta + \left\lceil \frac{1}{c_i} rest(\Theta, c_i) \right\rceil \right)$$

avec

$$\alpha(\Omega, i) \stackrel{def}{=} \left(C \left(d_\Omega - r_{\Omega \cup \{i\}} \right) < e_{\Omega \cup \{i\}} \right) \vee (r_i + p_i \geq d_\Omega)$$

et

$$rest(\Theta, c_i) = \begin{cases} e_\Theta - (C - c_i)(d_\Theta - r_\Theta) & si \;\; \Theta \neq \emptyset \\ 0 & sinon \end{cases}$$

Encore connu sur le nom de sélection immédiate [15] ou edge-finding [2], cet algorithme est bien connu dans le cas des ressources disjonctives i.e ressource de capacité unaire [15, 74, 60, 10, 33]. La version cumulative de cet algorithme de filtrage a été établie par Nuijten [60]. Il propose pour cela un algorithme de complexité $\mathcal{O}(n^2k)$ où k est le nombre de différentes consommations en ressource des tâches et n le nombre de tâches. Philippe Baptiste [10] propose un algorithme de complexité $\mathcal{O}(n^2)$. Mercier et Van Hentenryck [54] montrent que les algorithmes de Nuijten et de Baptiste sont incomplets (manquent certains ajustements). Ils proposent alors un algorithme complet à deux phases de complexité $\mathcal{O}(n^2k)$ en temps et $\mathcal{O}(nk)$ en espace. La première phase calcule les valeurs potentielles d'ajustement alors que la seconde phase utilise la première pour détecter et ajuster les fenêtres de temps des tâches. Tout récemment, cet algorithme a été amélioré de $\mathcal{O}(n^2k)$ à $\mathcal{O}(kn \log n)$ par Vilím [75]. L'algorithme de Vilìm est également à deux phases : detection et ajustement ; cet algorithme effectue l'ajustement maximal à la première itération.

Dans cette thèse, nous proposons un algorithme d'edge-finding de complexité $\mathcal{O}(n^2)$ en temps et linéaire en espace. Ce nouvel algorithme n'effectue pas nécessairement l'ajustement maximal à la première itération, mais s'améliore aux itérations subséquentes. Il atteint le même point fixe que les autres algorithmes complets d'edge-finding. Les résultats expérimentaux montrent que dans la pratique, notre algorithme est meilleur en temps sur les instances PSPLib [49] et BL [8] de la littérature (conf. Chapitre 3, paragraphe 3.3).

1.3.4 L'extended edge-finding

C'est une extension de l'edge-finding qui permet de détecter de nouvelles situations. Comme l'edge-finding, elle détermine si une tâche i peut ou non être ordonnancée avant ou après un ensemble Ω de tâches ($i \notin \Omega$) toutes devant être exécutées sur la même ressource. Cette détection est suivie de l'ajustement des fenêtres de temps de la tâche i. Plusieurs auteurs se sont intéressés et plusieurs algorithmes ont été proposés pour cette règle.

A notre connaissance, Nuijten [60] fut le tout premier auteur à proposer un algorithme de complexité $\mathcal{O}(kn^3)$ en temps où $k \leq n$ désigne le nombre de différentes demandes en consommation des tâches sur la ressource. Cet algorithme a été amélioré en 2008 par Mercier et Van Hentenryck [54] de $\mathcal{O}(kn^3)$ à $\mathcal{O}(kn^2)$ en temps. Comme pour l'edge-finding, ils proposent un algorithme de deux phases : pré-calcul et détection.

Soit $\Omega \subset T$ un ensemble de tâches et $i \in T \setminus \Omega$ une tâche vérifiant $r_i \leq r_\Omega < r_i + p_i$. Ces nouvelles conditions sont intéressantes car aucune tâche de Ω ne peut être ordonnancée dans l'intervalle $[r_i, r_\Omega]$. Sous ces conditions, Nuijten [60] montre que si

$$(C\,(d_\Omega - r_\Omega) < e_\Omega + c_i(r_i + p_i - r_\Omega))$$

alors dans tout ordonnancement réalisable, les tâches de Ω finissent avant la fin de la tâche i i.e., $\Omega \lessdot i$. De même, si $d_i - p_i < d_\Omega \leq d_i$ et $e_\Omega + c_i(d_\Omega - d_i + p_i) > C\,(d_\Omega - r_\Omega)$ alors toutes les tâches de Ω commencent après le début de la tâche i i.e., $\Omega \gtrdot i$. La Proposition 1.2 donne la condition pour laquelle toutes les tâches de Ω finissent avant la fin ou commencent après le début de la tâche i.

Proposition 1.2. *Soient Ω un ensemble de tâches d'une instance E-réalisable de CuSP et i une tâche telle que $i \notin \Omega$.*

$$(r_i \leq r_\Omega < r_i + p_i) \ \wedge \ (e_\Omega + c_i(r_i + p_i - r_\Omega) > C\,(d_\Omega - r_\Omega)) \Longrightarrow \Omega \lessdot i; \qquad \text{(EEF)}$$

$$(d_i - p_i < d_\Omega \leq d_i) \wedge (e_\Omega + c_i(d_\Omega - d_i + p_i) > C\,(d_\Omega - r_\Omega)) \Longrightarrow \Omega \gtrdot \ i. \qquad \text{(revEEF)}$$

Démonstration. La preuve de l'item 2. de la Proposition 1.2 est identique à celle du 1. Nous ne prouverons que le 1. Par l'absurde, supposons que toutes les tâches sont ordonnancées et qu'il existe une tâche de Ω qui finit après la fin de la tâche i. L'énergie consommée entre r_Ω et d_Ω est $e_\Omega + c_i(r_i + p_i - r_\Omega)$ qui est, d'après la relation $e_\Omega + c_i(r_i + p_i - r_\Omega) > C\,(d_\Omega - r_\Omega)$, supérieure à l'énergie disponible $C\,(d_\Omega - r_\Omega)$. Ce qui conduit à une contradiction. □

Dans toute la suite, nous nous intéressons à la première règle qui permet d'ajuster les dates de début au plus tôt des tâches. Pour ce qui est de l'ajustement des dates de fin au plus tard, une version symétrique de l'algorithme effectue cette autre tâche.

Comme pour l'edge-finding, lorsqu'il existe une tâche i et un ensemble de tâches $\Omega \subseteq T \setminus \{i\}$ vérifiant les conditions $(r_i \leq r_\Omega < r_i + p_i)$ et $(e_\Omega + c_i(r_i + p_i - r_\Omega) > C\,(d_\Omega - r_\Omega))$ alors la tâche i doit être ordonnancée après toutes les tâches de Ω et sa date de début est alors ajustée à

$$r_i \geq r_\Theta + \left\lceil \frac{1}{c_i} rest\,(\Theta, c_i) \right\rceil \tag{1.16}$$

pour tout $\Theta : \emptyset \neq \Theta \subseteq \Omega$ vérifiant $\quad rest\,(\Theta, c_i) > 0$ avec

$$rest\,(\Theta, c_i) = e_\Theta - (C - c_i)\,(d_\Theta - r_\Theta)\,. \tag{1.17}$$

Exemple 1.7. *Considérons l'instance de la Figure 1.6 où trois tâches doivent être exécutées sur une ressource de capacité 4.*

Cette instance de CuSP est réalisable (comme l'illustre la Figure 1.6) donc E-réalisable. De plus, en appliquant la règle d'extended edge-finding avec $i = A$ *et* $\Omega = \{B, C\}$*, on obtient* $r_A = 0 \leq r_\Omega = 1 < 5 = r_A + p_A$ *et* $e_\Omega + c_A(r_A + p_A - r_\Omega) = 17 + 1 \times (5 - 1) = 21$ *alors que* $C(d_\Omega - r_\Omega) = 4 \times (6 - 1) = 20$*. Dans tout ordonnancement possible de cette instance de CuSP, la tâche A finit après toutes les tâches de* Ω *i.e., après l'instant 6. Pour* $\Theta = \Omega = \{B, C\}$*, l'ajustement obtenu est* $r_A \geq 3$ *car* $rest(\Theta, c_A) = 17 - 3 \times (6 - 1) = 2 > 0$ *et* $r_\Theta + \left\lceil \frac{1}{c_A} rest\,(\Theta, c_A) \right\rceil = 1 + \left\lceil \frac{2}{1} \right\rceil = 3$.

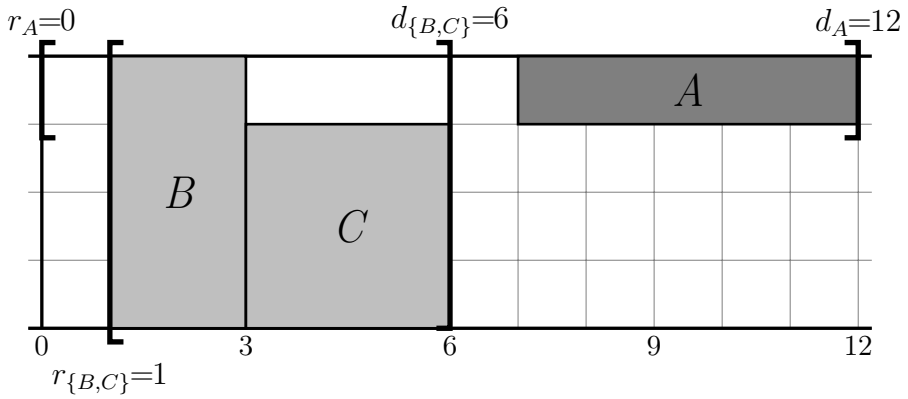


FIGURE 1.6 – Trois tâches $T = \{A, B, C\}$ s'exécutant sur une ressource de capacité $C = 4$.

Soit $\Omega \subset T$ un ensemble de tâches et $i \in T \setminus \Omega$ une tâche. L'extended edge-finding utilise la condition

$$\beta\left(\Omega, i\right) \Leftrightarrow \left(\left(C\left(d_\Omega - r_\Omega\right) < e_\Omega + c_i(r_i + p_i - r_\Omega)\right) \wedge \left(r_i \leq r_\Omega < r_i + p_i\right)\right) \tag{1.18}$$

pour détecter que la tâche i doit finir après toutes les tâches de Ω. La proposition suivante est utilisée dans [54] pour simplifier la définition de la condition $\beta(\Omega, i)$ liée à la règle.

Proposition 1.3. *Soit $\Omega \subset T$ un ensemble de tâches et $i \in T \setminus \Omega$ une tâche d'une instance E-Réalisable de CuSP.*

$$\beta(\Omega, i) \Leftrightarrow \begin{cases} r_i \leq r_\Omega \\ e_\Omega + c_i(r_i + p_i - r_\Omega) > C(d_\Omega - r_\Omega) \end{cases}.$$

Démonstration. Soit $\Omega \subset T$ un ensemble de tâches et $i \in T \setminus \Omega$ une tâche d'une instance E-Réalisable de CuSP. Etant donné que la relation membre de gauche implique membre de droite découle de la relation (1.18), il nous reste à montrer que le membre de droite implique celui de gauche.

Si $r_\Omega > r_i + p_i$ alors $e_\Omega > e_\Omega + c_i(r_i + p_i - r_\Omega)$. Par la suite, $e_\Omega > C(d_\Omega - r_\Omega)$ ce qui contredit le fait que l'instance soit E-Réalisable. □

La relaxation de la contrainte $r_\Omega < r_i + p_i$ n'accroit pas la puissance de filtrage de la règle d'extended edge-finding. Par contre, lorsque cette contrainte est ignorée, le nombre d'intervalle de tâches à considérer augmente, ce qui peut conduire à un algorithme erroné. C'est par exemple le cas des secondes phases des algorithmes d'extended edge-finding

proposés par Mercier et Van Hentenryck dans [54] comme nous allons le montrer plus loin. La Définition 1.14 donne la spécification d'un algorithme complet d'extended edge-finding.

Définition 1.14. *Un algorithme d'extended edge-finding reçoit en entrée une instance E-Réalisable de CuSP. Il produit en sortie un vecteur*

$$\langle LB_1, ..., LB_n \rangle$$

où

$$LB_i = \max \left(r_i, \max_{\substack{\Omega \subseteq T \\ i \notin \Omega \\ \beta(\Omega, i)}} \max_{\substack{\Theta \subseteq \Omega \\ rest(\Theta, c_i) > 0}} r_\Theta + \left\lceil \frac{1}{c_i} rest(\Theta, c_i) \right\rceil \right)$$

avec

$$\beta(\Omega, i) \stackrel{def}{=} ((C(d_\Omega - r_\Omega) < e_\Omega + c_i(r_i + p_i - r_\Omega)) \wedge (r_i \leq r_\Omega < r_i + p_i))$$

et

$$rest(\Theta, c_i) = \begin{cases} e_\Theta - (C - c_i)(d_\Theta - r_\Theta) & si \;\; \Theta \neq \emptyset \\ 0 & sinon \end{cases}$$

Elle a été proposée par Nuijten [60] dans un algorithme de complexité $\mathcal{O}(n^3k)$. Mercier et Van Hentenryck [54] proposent le tout premier algorithme de complexité $\mathcal{O}(n^2k)$ en temps et $\mathcal{O}(nk)$ en espace. C'est un algorithme à deux phases similaire à celui de l'edge-finding qui effectue l'ajustement maximal à la première itération.

Nous montrons au Chapitre 2 que la seconde phase de l'algorithme proposé par Mercier et Van Hentenryck [54] est incorrecte. Nous proposons alors un algorithme quadratique similaire à celui de l'edge-finding qui n'effectue pas nécessairement l'ajustement maximal à la première itération, mais s'améliore aux itérations subséquentes. Il atteint le même point fixe que les autres algorithmes complets d'extended edge-finding. Au Chapitre 3, les résultats expérimentaux montrent que dans la pratique, sur les instances PSPLib [49] et BL [8] de la littérature, le nouvel algorithme réduit le nombre de nœuds de l'arbre

de recherche pour une légère augmentation du temps d'exécution (conf. Chapitre 3, paragraphe 3.4).

1.3.5 Le not-first/not-last

La règle de not-first/not-last est un pendant de la règle d'edge-finding qui déduit qu'une tâche i ne peut être exécutée en premier (resp. en dernier) dans l'ensemble des tâches $\Omega \cup \{i\}$ toutes devant être exécutées sur une même ressource. Comme conséquence, l'ajustement de r_i (resp. d_i) correspond à la plus petite date de fin au plus tôt (resp. à la plus grande date de début au plus tard) de Ω.

Définition 1.15. *Soit $\Omega \subseteq T$ un ensemble de tâches d'une instance E-réalisable de CuSP. La date de fin au plus tôt de la tâche $i \in T$ est définie comme étant $ect_i := r_i + p_i$. La plus petite date de fin au plus tôt de l'ensemble de tâches Ω est définie comme étant $ECT_\Omega := \min_{j \in \Omega} ect_j$ si $\Omega \neq \emptyset$ et $ECT_\emptyset := +\infty$.*

Une définition similaire est donnée pour la date de début au plus tard d'une tâche et d'un ensemble de tâches.

Définition 1.16. *Soit $\Omega \subseteq T$ un ensemble de tâches d'une instance E-réalisable de CuSP. La date de début au plus tard de la tâche $i \in T$ est définie comme étant $lst_i := d_i - p_i$. La plus grande date de début au plus tard de l'ensemble de tâches Ω est définie comme étant $LST_\Omega := \max_{j \in \Omega} lst_j$ si $\Omega \neq \emptyset$ et $LST_\emptyset := -\infty$.*

La Proposition 1.4 présente la règle de not-first/not-last telle qu'elle est étudiée dans la littérature. Dans [64], par exemple, cette règle a plusieurs améliorations accroissant ainsi sa puissance.

Proposition 1.4. *[64] Soit Ω un ensemble de tâches et $i \notin \Omega$ une tâche.*

$$r_i < ECT_\Omega \wedge e_\Omega + c_i \left(\min\left(ect_i, d_\Omega\right) - r_\Omega\right) > C\left(d_\Omega - r_\Omega\right) \Rightarrow r_i \geq ECT_\Omega \tag{NF}$$

$$LST_\Omega < d_i \wedge e_\Omega + c_i \left(d_\Omega - \max\left(lst_i, r_\Omega\right)\right) > C\left(d_\Omega - r_\Omega\right) \Rightarrow d_i \leq LST_\Omega \tag{NL}$$

La règle (NF) est appelée not-first. Elle consiste à ajuster la date de début au plus tôt de toute tâche i qui ne peut pas être première dans l'exécution des tâches de l'ensemble $\Omega \cup \{i\}$. Un algorithme de not-first est une procédure qui effectue toutes ces déductions.

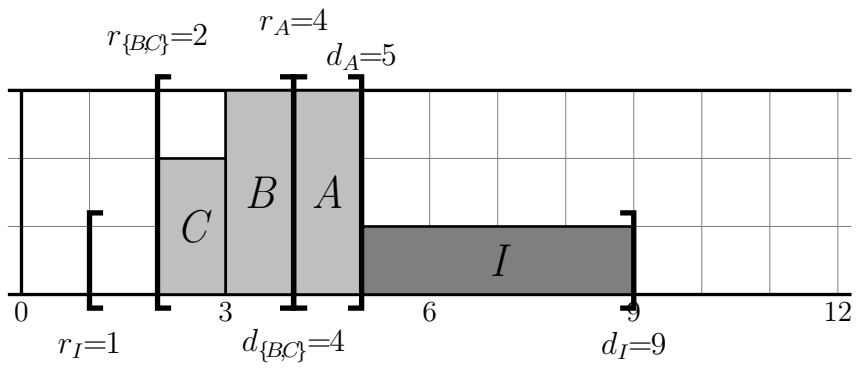


FIGURE 1.7 – Une instance de CuSP de quatre tâches devant être exécutées sur une ressource de capacité $C = 3$.

De manière similaire, la règle (NL) est appelé not-last. Elle consiste à ajuster la date de fin au plus tard de toute tâche i qui ne peut pas être dernière dans l'exécution des tâches de l'ensemble $\Omega \cup \{i\}$. Nous ne considérons dans toute la suite que la règle de not-first.

Définition 1.17 (Algorithme complet de not-first)**.** *Un algorithme complet de not-first reçoit en entrée une instance E-Réalisable de CuSP. Il renvoie en sortie un vecteur*

$$\langle LB_1, ..., LB_n \rangle$$

où

$$LB_i = \max\left(r_i, \max_{\Omega \subseteq T \setminus \{i\} | \gamma(\Omega, i)} ECT_\Omega\right)$$

et

$$\gamma(\Omega, i) \stackrel{def}{=} (r_i < ECT_\Omega) \wedge (e_\Omega + c_i(\min(ect_i, d_\Omega) - r_\Omega) > C(d_\Omega - r_\Omega))$$

Exemple 1.8. *On considère trois travailleurs identiques (ressource de capacité $C = 3$) qui doivent exécuter quatre tâches $T = \{A, B, C, I\}$.*

La condition de la règle de not-first est vérifiée pour la tâche $i = I$ et les ensembles de tâches $\Omega = \{A\}$ ou $\Omega = \{A, B, C\}$. Mais l'ajustement maximal est obtenu en utilisant la paire $(\{A\}, I)$. En effet, $r_I = 1 < ECT_\Omega = ect_A = 5$ et $e_\Omega + c_I(\min(ect_I, d_\Omega) - r_\Omega) = 3 + 1 \cdot (5 - 4) = 4 > 3 = C(d_\Omega - r_\Omega)$. D'où $LB_I = 5$.

Il est lui aussi déduit de sa version disjonctive bien étudiée [10, 74, 70]. La première version de complexité $\mathcal{O}(n^3k)$ fut proposée par Nuijten [60]. En 2005, Schutt et al [64] montrent que cet algorithme est incorrect et incomplet. Ils proposent alors un algorithme

complet de complexité $\mathcal{O}(n^3 \log(n))$ basé sur la structure de données des arbres binaires équilibrés. Dans [10], Philippe Baptiste mentionne l'existence d'un algorithme de complexité $\mathcal{O}(n^3)$ dans une communication privée de Nuijten. A notre connaissance, jusqu'a présent un tel algorithme n'est pas encore publié.

Nous proposons dans [37] un algorithme itératif de not-first/not-last de complexité $\mathcal{O}(n^3)$ en temps. Cet algorithme fut par la suite amélioré de $\mathcal{O}(n^3)$ à $\mathcal{O}(n^2 \log n)$ en utilisant la structure de données Θ-tree [38, 42]. Un algorithme de même complexité a été proposéé par Schutt et Wolf [65]. Nous proposons également dans [39] un algorithme complet de not-first/not-last améliorant ainsi la complexité du meilleur algorithme complet de la littérature [64]. Au Chapitre 3 nous comparons notre algorithme complet [39] et la version itérative [38] sur les instances PSPLib et BL [8]. Les résultats montrent que la version itérative est plus rapide que la version complète. (conf. Chapitre 3, paragraphe 3.5).

1.3.6 Le raisonnement énergétique

C'est une technique de propagation initialement amorcée par [27, 51] et approfondie dans [9, 10]. Soit une tâche i donnée et un intervalle de temps $[t_1; t_2]$, le travail minimal ou l'énergie minimale consommée de i sur l'intervalle $[t_1; t_2]$ est le nombre noté $W_{Sh}(i, t_1, t_2)$ égal à :

$$W_{Sh}(i, t_1, t_2) = c_i \min(t_2 - t_1, p_i^+(t_1), p_i^-(t_2))$$

où

- $p_i^+(t_1) = \max(0, p_i - \max(0, t_1 - r_i))$ est le nombre d'unités de temps durant lesquelles la tâche i s'exécute après l'instant t_1 lorsqu'elle est ordonnancée le plus tôt possible.
- $p_i^-(t_2) = max(0, p_i - max(0, d_i - t_2))$ est le nombre d'unités de temps durant leqsuelles la tâche i s'exécute avant l'instant t_2 lorsqu'elle est ordonnancée le plus tard possible.

L'énergie consommée par toutes les tâches T s'exécutant sur cette ressource est

$$W_{Sh}(t_1, t_2) = \sum_{i \in T} W_{Sh}(i, t_1, t_2).$$

La condition suffisante de consistance globale stipule que l'énergie fournie doit être supérieure à l'énergie consommée par l'ensemble des tâches sur tout intervalle. Si cette

condition n'est pas satisfaite, on aboutit à un échec. Mathématiquement, on écrit

$$(\exists t_1, t_2;\ t_1 < t_2,\ W_{Sh}(t_1, t_2) > C(t_2 - t_1)) \Rightarrow \quad Echec \tag{1.19}$$

Dans [9, 8], les auteurs identifient l'ensemble O des intervalles qu'il est nécessaire et suffisant de considérer pour tester la consistance globale. Un algorithme quadratique est proposé pour établir les conditions nécessaires d'existence sur tous ces intervalles. Il a été constaté expérimentalement que ces conditions nécessaires d'existence étaient particulièrement utiles dans le cas où le problème traité est fortement cumulatif [8]. Des techniques d'ajustement des dates de début au plus tôt et de fin au plus tard de complexité $\mathcal{O}(n^3)$ sont proposées à partir de ces conditions nécessaires d'existence. Plusieurs auteurs ont utilisé avec succès le raisonnement énergétique pour résoudre les problème d'ordonnancement, on peut citer entre autre [58, 47, 32]

Nous proposons dans [36] un algorithme quadratique nouveau, né de l'hybridation de l'edge-finding et du raisonnement énergétique. C'est un algorithme de filtrage à deux phases dans lequel au cours de la première phase, on calcule les valeurs potentielles d'ajustement et identifie les fenêtres des intervalles correspondants suivant l'edge-finding, la seconde phase utilise ces résultats pour détecter en s'appuyant sur le raisonnement énergétique. Cet algorithme est étendu à l'hybridation de l'edge-finding, extended edge-finding et du raisonnement énergétique dans [44]. Ce nouvel algorithme de complexité $\mathcal{O}(n^3)$ en temps domine l'edge-finding et l'extended edge-finding.

1.3.7 Le timetable edge finding [77]

Si une tâche i vérifie $d_i - p_i < r_i + p_i$, alors cette tâche doit nécessairement s'exécuter dans l'intervalle $[d_i - p_i; r_i + p_i]$. Ainsi à chaque instant $t \in [d_i - p_i; r_i + p_i]$, c_i unités de ressources sont mobilisées. On note $p_i^{TT} = \max(0, r_i + p_i - (d_i - p_i))$ la partie fixe et $p_i^{EF} = p_i - p_i^{TT}$ la partie libre de la tâche i. $T^{EF} = \{i, i \in T, p_i^{EF} > 0\}$ désigne l'ensemble des tâches ayant une partie libre. On note $TT(t)$ la somme de toutes les consommations qui chevauchent à l'instant t. Pour tout ensemble de tâches $\Omega \subset T^{EF}$, l'énergie consommée entre r_Ω et d_Ω est

$$e_\Omega^{EF} + ttAfterEst(\Omega) - ttAfterLct(\Omega) \tag{1.20}$$

où

$$ttAfterEst(\Omega) = \sum_{t \in \mathbb{N} | t \geq r_\Omega} TT(t)$$

$$ttAfterLct(\Omega) = \sum_{t \in \mathbb{N} | t \geq d_\Omega} TT(t)$$

et

$$e_\Omega^{EF} = \sum_{i \in T^{EF}} p_i^{EF} \times c_i.$$

L'énergie restant (marge) dans l'intervalle $[r_\Omega; d_\Omega]$ est donc

$$reserve(\Omega) = C(d_\Omega - r_\Omega) - (e_\Omega^{EF} + ttAfterEst(\Omega) - ttAfterLct(\Omega))$$

La condition suffisante de consistance globale stipule que l'énergie fournie doit être supérieure à l'énergie consommée par l'ensemble des tâches pour tout ensemble $\Omega \subset T^{EF}$. Si cette condition n'est pas satisfaite, on aboutit à un échec. Mathématiquement, on écrit

$$(\exists \Omega \subset T^{EF}, \quad reserve(\Omega) < 0) \Rightarrow \quad Echec \tag{1.21}$$

La règle de timetable edge finding est : $\forall \Omega \subset T^{EF}$ et $\forall i \in T^{EF} \setminus \Omega$ si

$$reserve(\Omega) < add(r_\Omega, d_\Omega, i) \tag{1.22}$$

avec

$$add(r_\Omega, d_\Omega, i) = \begin{cases} p_i^{EF} \times c_i & si \quad r_\Omega \leq r_i \wedge r_i + p_i^{EF} \leq d_\Omega \\ c_i(d_\Omega - r_i) & si \quad r_\Omega < r_i \wedge d_\Omega < r_i + p_i^{EF} \\ c_i(r_i + p_i^{EF} - r_\Omega) & si \quad r_i < r_\Omega \wedge r_i + p_i^{EF} < d_\Omega \\ c_i(d_\Omega - r_\Omega) & si \quad r_i \leq r_\Omega \wedge d_\Omega \leq r_i + p_i^{EF} \end{cases}$$

alors la date de début au plus tôt de la tâche i est ajustée à

$$r_i := d_\Omega - mandatoryIn(r_\Omega, d_\Omega, i) - \left\lfloor \frac{reserve(\Omega)}{c_i} \right\rfloor \tag{1.23}$$

avec $mandatoryIn(r_\Omega, d_\Omega, i) = \max(0, \min(d_\Omega, r_i + p_i) - \max(r_\Omega, d_i - p_i))$.

Exemple 1.9. *Considérons l'instance de CuSP de la Figure 1.8 où 5 tâches doivent s'exécuter sur une ressource de capacité 3. On a $T^{EF} = \{B, C, D, A\}$ et pour $\Omega = \{B, C, D\}$ on a $reserve(\Omega) = 1$ alors que $add(r_\Omega, d_\Omega, A) = 2$. Donc $reserve(\Omega) < add(r_\Omega, d_\Omega, A)$ par conséquent la date de début au plus tôt de la tâche A est ajustée à $r_A = 6 - 0 - 1 = 5$.*

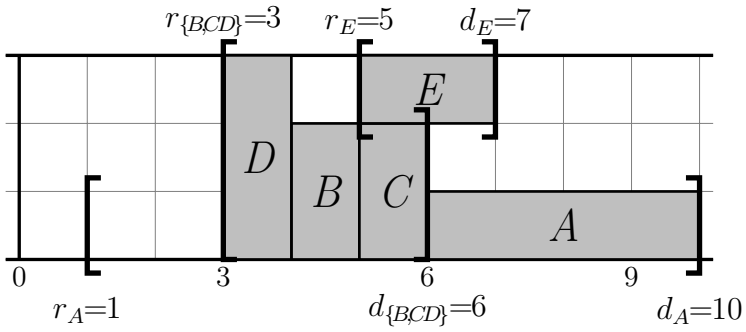


FIGURE 1.8 – Cinq tâches à ordonnancer sur une ressource de capacité $C = 3$.

Le timetable edge finding est une nouvelle règle de propagation de la contrainte de ressource basée sur la structure de données appelée timetable [77]. Il est proposé dans [77] un algorithme quadratique qui n'effectue pas nécessairement tous les ajustements à la première itération, mais s'améliore aux itérations suivantes. Il est prouvé dans cet article que cette règle est plus efficace que la conjonction de l'edge-finding et son extension et peut être améliorée en incorporant certaines idées du not-first/not-last et du raisonnement énergétique. L'Exemple 1.9 est très intéressant. En effet, sur l'instance de CuSP de cet exemple, aucune des règles edge-finding, extended edge-finding et not-first/not-last n'ajuste la fenêtre de temps de la tâche A.

Nous démontrons au Chapitre 2 de cette thèse que la règle du timetable edge finding ne domine ni l'edge-finding, ni l'extended edge-finding. Ces résultats contredisent alors les résultats annoncés par Vilím dans [77].

Il existe plusieurs autres algorithmes de propagation de la contrainte de ressources tels que les coupes de Benders [33], l'énergie de précédence et le balance constraint [50]. Les algorithmes de propagation des contraintes de ressource est un champ de recherches très large qui, tant d'un point de vue pratique qu'appliqué, a donné lieu à un nombre important de publications. Nous renvoyons le lecteur à [17, 10, 33, 50, 63, 14] pour une introduction plus poussée à ce domaine.

Chapitre Deuxième

L'edge-finding, l'extended edge-finding et le not-first/not-last cumulatif

L'edge-finding est l'une des techniques de propagation de la contrainte de ressource la plus utilisée lors de la résolution des problèmes d'ordonnancement. Elle a été utilisée pour la première fois pour la résolution des problèmes d'ordonnancement disjonctifs (cas où la ressource est unaire i.e., $C = 1$) dans [15]. En effet, il existe dans ce cas des algorithmes d'edge-finding efficicaces de complexité $\mathcal{O}(n \log n)$ [74]. Dans ce chapitre, nous nous intéressons au cas cumulatif qui est plus complexe car une tâche peut nécessiter plus d'une unité de ressource pour son exécution. Nous proposons un nouvel algorithme d'edge-finding de complexité $\mathcal{O}(n^2)$. Notre algorithme n'effectue pas nécessairement l'ajustement maximal à la première itération, mais elle s'améliore aux itérations subséquentes. Nous montrons que la seconde phase de l'algorithme d'extended edge-finding proposée par Mercier et Van Hentenryck [54] est incorrecte et nous proposons un nouvel algorithme de complexité $\mathcal{O}(n^2)$ similaire à celui de l'edge-finding de ce chapitre. Cet algorithme est itératif car n'effectue pas nécessairement l'ajustement maximal à la première itération, mais s'améliore aux itérations subséquentes. Au point fixe de l'edge-finding, le nouvel algorithme d'extended edge-finding effectue toujours un ajustement lorsque celui-ci est justifié par la règle. La combinaison notre précédent algorithme d'edge-finding à ce nouvel algorithme d'extended edge-finding effectue les même ajustements que la conjonction des règles edge-finding et extended edge-finding. Nous proposons également dans ce chapitre deux algorithmes pour la règle not-first/not-last. Le premier complet est de complexité

$\mathcal{O}(n^2|H|\log n)$ en temps où $|H|$ désigne le nombre de dates de fin au plus tôt différentes des tâches. Cet algorithme améliore ainsi la complexité de l'algorithme de complexité $\mathcal{O}(n^3 \log n)$ proposé par Schutt et al [64]. Le second algorithme, de complexité $\mathcal{O}(n^2 \log n)$, est incomplet à la première itération, mais s'améliore aux itérations subséquentes pour atteindre le même point fixe que les algorithmes de la littérature [64, 37, 65, 39].

2.1 L'edge-finding

2.1.1 Propriétés de dominance de la règle d'edge-finding

Il est clair que pour l'ajustement d'une tâche i, un algorithme d'edge-finding ne peut pas considérer tous les ensembles de tâches $\Theta \subseteq \Omega \subseteq T$. Les propriétés de dominance sont donc utilisées pour caractériser et réduire les paires (Ω, Θ) qui doivent être considérées à cet effet. On démontre dans [54] qu'un algorithme d'edge-finding pour lequel les ensembles de tâches $\Omega \subseteq T$ et $\Theta \subseteq \Omega$ sont des intervalles de tâches peut être complet. Nous dirons que l'application de la règle d'edge-finding sur la tâche i et la paire (Ω, Θ) permet de réduire la date de début au plus tôt d'une tâche i si $\Omega \lessdot i$, $rest\,(\Theta, c_i) > 0$ et $r_\Theta + \frac{1}{c_i} rest\,(\Theta, c_i) > r_i$. Les Propositions 2.1 et 2.2 résument les principales propriétés de dominance des règles (EF) et (EF1) respectivement. Certaines de ces propriétés sont disponibles dans [54, 60].

Proposition 2.1. *[54] Soit $i \in T$ une tâche d'une instance E-Réalisable de CuSP, $\Omega \subseteq T \setminus \{i\}$ et $\Theta \subseteq \Omega$ deux ensembles de tâches. Si l'application de la règle d'edge-finding* (EF) *sur la tâche i et la paire (Ω, Θ) permet d'augmenter la date de début au plus tôt de la tâche i alors*

(i) il existe quatre tâches L, U, l, u telles que $r_L \leq r_l < d_u \leq d_U < d_i \wedge r_L \leq r_i$ et

(ii) la règle d'edge-finding (EF) *appliquée à la tâche i et la paire $(\Omega_{L,U}, \Omega_{l,u})$ permet d'effectuer un meilleur ajustement de la date de début au plus tôt de la tâche i.*

Les propriétés de dominance suivantes permettent de choisir les ensembles Ω et Θ pour la règle (EF1).

Proposition 2.2. *Soit $i \in T$ une tâche d'une instance E-Réalisable de CuSP, $\Omega \subseteq T \setminus \{i\}$ et $\Theta \subseteq \Omega$ deux ensembles de tâches. Si l'application de la règle d'edge-finding* (EF1) *sur la tâche i et la paire (Ω, Θ) permet d'augmenter la date de début au plus tôt de la tâche i alors*

(i) il existe quatre tâches L, U, l, u telles que $r_L \leq r_l < d_u \leq d_U < d_i \wedge r_L \leq r_i$ et

(ii) la règle d'edge-finding (EF1) *appliquée à la tâche i et la paire $(\Omega_{L,U}, \Theta_{l,u})$ permet d'effectuer un meilleur ajustement de la date de début au plus tôt de la tâche i.*

Démonstration. Les ensembles Ω et Θ sont non vides car $\Theta \subseteq \Omega$ et $rest(\Theta, c_i) > 0$. Ainsi, il existe quatre tâches $L \in \Omega$, $U \in \Omega$, $l \in \Theta$, $u \in \Theta$ vérifiant $r_L = r_\Omega$, $d_U = d_\Omega$, $r_l = r_\Theta$ et $d_u = d_\Theta$ (s'il existe plusieurs tâches ayant cette propriété, choisir arbitrairement). Nous avons $r_L \leq r_l < d_u \leq d_U$ car $\Theta \subseteq \Omega$ et $\Theta \neq \emptyset$. Par contradiction, si $d_i \leq d_\Omega$ alors $r_i + p_i = d_i$ car $r_i + p_i \leq d_i$ et $r_i + p_i \geq d_\Omega$. Ainsi, la tâche i est déjà ordonnancée et aucune règle de propagation ne peut permettre de réduire la fenêtre de temps de la tâche i. Ceci contredit le fait que la règle (EF1) appliquée à la tâche i et la paire (Ω, Θ) permet d'ajuster la date de début au plus tôt d'une tâche i. L'inclusion $\Theta \subseteq \Omega_{l,u}$ implique $rest(\Theta_{l,u}, c_i) > 0$ (car $rest(\Theta, c_i) > 0$) et $r_l + \frac{1}{c_i} rest\,(\Theta_{l,u}, c_i) \geq r_\Theta + \frac{1}{c_i} rest\,(\Theta, c_i)$. Donc, la règle d'edge-finding (EF1) appliquée à la tâche i et la paire $(\Omega_{L,U}, \Theta_{l,u})$ permet d'effectuer un meilleur ajustement de la date de début au plus tôt de la tâche i. □

2.1.2 Nouvel algorithme d'edge-finding

Dans ce paragraphe, nous présentons un nouvel algorithme quadratique d'edge-finding qui n'effectue pas nécessairement l'ajustement maximal à la première itération, mais s'améliore aux itérations subséquentes. Il atteint le même point fixe que les autres algorithmes d'edge-finding [75, 54].

Nous commençons par considérer l'algorithme d'edge-finding de complexité $\mathcal{O}(n^2)$ proposé dans [9]. Dans cet algorithme, les ensembles de tâches Ω et Θ utilisés dans la Définition 1.13 pour détecter et ajuster la fenêtre de temps d'une tâche sont localisés à travers l'intervalle de tâches de marge minimale. La marge d'un ensemble de tâches est donnée par la définition suivante.

Définition 2.1. *Soit Ω un ensemble de tâches d'une instance E-réalisable de CuSP. La marge (en anglais* slack*) d'un ensemble de tâches Ω, notée SL_Ω, est définie par :*

$$SL_\Omega = C(d_\Omega - r_\Omega) - e_\Omega. \tag{2.1}$$

Définition 2.2. *Soient i et U deux tâches d'une instance E-Réalisable de CuSP. On dit que la tâche $\tau(U,i)$ avec $r_{\tau(U,i)} \leq r_i$ définit la borne inférieure de l'intervalle de tâches de marge minimale (en anglais* minimum slack*) si pour toute tâche $L \in T$ telle que $r_L \leq r_i$ on a :*

$$C(d_U - r_{\tau(U,i)}) - e_{\Omega_{\tau(U,i),U}} \leq C(d_U - r_L) - e_{\Omega_{L,U}} \tag{2.2}$$

Comme l'algorithme d'edge-finding proposé par Baptiste [9], pour une tâche i donnée, l'algorithme détecte la relation $\Omega \lessdot i$ en vérifiant la condition $SL_\Omega < e_i$ pour tout $\Omega = \Omega_{\tau(U,i),U}$ tel que $d_U < d_i$ et $r_{\tau(U,i)} \leq r_i$. Mieux encore, si l'intervalle de tâches $\Theta_{l,u}$ responsable de l'ajustement maximal de r_i vérifie $r_l \leq r_i$, alors $\Theta_{l,u}$ est l'intervalle de tâches de marge minimale. C'est par exemple le cas pour l'instance de la Figure 1.5. Au lieu de déterminer r_l, notre nouvel algorithme calcule la valeur potentielle d'ajustement de r_i en utilisant $rest(\Omega, c_i)$. Par contre, si l'intervalle de tâches $\Theta_{l,u}$ vérifie $r_i < r_l$, alors l'intervalle $\Theta_{l,u}$ n'est pas nécessairement l'intervalle de marge minimale. C'est par exemple le cas pour l'instance de la Figure 1.4. Pour ce cas, nous avons introduit la notion de densité d'intervalle.

Définition 2.3. *Soit Θ un ensemble de tâches d'une instance E-Réalisable de CuSP. La* densité *de l'ensemble de tâches Θ, noté $Dens_\Theta$, est définie par :*

$$Dens_\Theta = \frac{e_\Theta}{d_\Theta - r_\Theta}. \tag{2.3}$$

Définition 2.4. *Soient i et u deux tâches d'une instance E-Réalisable de CuSP. On dit que la tâche $\rho(u,i)$ avec $r_i < r_{\rho(u,i)}$ définit la borne inférieure de l'intervalle de tâches de densité maximale si pour toute tâche $l \in T$ telle que $r_i < r_l$ on a :*

$$\frac{e_{\Theta_{l,u}}}{d_u - r_l} \leq \frac{e_{\Theta_{\rho(u,i),u}}}{d_u - r_{\rho(u,i)}}. \tag{2.4}$$

Si l'intervalle de tâches $\Theta_{l,u}$ responsable de l'ajustement maximal de r_i vérifie $r_i < r_l$, alors $\Theta_{l,u}$ est l'intervalle de tâches de densité maximale. Une nouvelle valeur potentielle d'ajustement de r_i est alors calculée en utilisant $rest(\Theta, c_i)$ avec $\Theta = \Theta_{\rho(u,i),u}$ pour toute

tâche $u \in T$ vérifiant $d_u \leq d_U < d_i$ et $r_i < r_{\rho(u,i)}$. L'intervalle de tâches de densité maximale est calculé suivant la Définition 2.4 et est utilisé pour le calcul de la seconde valeur potentielle d'ajustement. Au départ de l'algorithme, nous n'avons pas l'information selon laquelle $r_l \leq r_i$ ou non C'est pourquoi les intervalles de tâches de marge minimale et de densité maximale sont des bons candidats pour localiser et effectuer le bon ajustement. Pour chaque tâche i, les deux valeurs potentielles d'ajustement sont calculées et le meilleur ajustement est alors appliqué.

Considérons l'instance de CuSP de la Figure 1.4. Avec $\Omega = \{A, B, C, D, E\}$ et $i = F$, la règle (EF) détecte la condition $\Omega \lessdot i$. Cependant, les algorithmes de [60, 9] manquent l'ajustement de r_F car pour tout $\Theta \subseteq \Omega$ nous avons $r_\Theta > r_F$. Le nouvel algorithme ajuste r_F en utilisant l'intervalle de tâches de densité maximale $\Theta_{\rho(u,i),u}$. Pour $u \in \{A, C, D\}$, l'intervalle de tâches de densité maximale est $\Theta_{A,D} = \{A, B, C, D, E\}$ qui a une densité de $18/7 \approx 2.6$. En utilisant (1.15) avec $\Theta = \Theta_{A,D}$, la date de début au plus tôt de la tâche F est ajustée à $r_F \geq 5$. Cependant, pour $u \in \{B, E\}$, l'intervalle de tâches de densité maximale est $\Theta_{B,E} = \{B, E\}$ qui a pour densité $5/2 = 2.5$. En utilisant (1.15) avec $\Theta = \Theta_{B,E}$ la date de début au plus tôt de la tâche F est ajustée à $r_F \geq 6$ qui est le meilleur ajustement possible.

Pour toute tâche $i \in T$, l'Algorithme 2 réalise ces ajustements avec une complexité de $\mathcal{O}(n^2)$ en temps. En effet :

- La boucle externe (ligne 3) parcourt l'ensemble T des tâches triées dans l'ordre croissant des dates de fin au plus tard et pour chaque tâche $U \in T$, sélectionne la borne supérieure des intervalles de tâches.
- La boucle interne (ligne 5) par contre itère suivant l'ensemble T triées dans l'ordre décroissant des dates de début au plus tôt et sélectionne la tâche $i \in T$, borne inférieure des intervalles de tâches. Si $d_i \leq d_U$, alors l'énergie et la densité de l'intervalle $\Omega_{i,U}$ sont calculées. L'énergie est sauvegardée et la densité est comparée à celle de $\Omega_{\rho(U,i),U}$. Si la nouvelle densité est la plus grande, $\rho(U, i)$ devient L. Si $d_i > d_U$, alors la première valeur potentielle d'ajustement $Dupd_i$ de la date de début au plus tôt de la tâche i est calculée avec la tâche courante $\rho(U, i)$. Cette valeur potentielle d'ajustement n'est sauvegardée que si elle est supérieure aux précédentes valeurs d'ajustement de i obtenues avec les précédentes densités maximales.

– La seconde boucle interne (ligne 16) itère suivant l'ensemble T des tâches triées dans l'ordre croissant des dates de début au plus tôt et sélectionne une tâche $i \in T$. L'énergie de l'intervalle $\Omega_{i,U}$ déjà calculée dans la première boucle interne est utilisée pour calculer sa marge. Si cette marge est inférieure à celle de $\Omega_{\tau(U,i),U}$, alors la tâche $\tau(U,i)$ devient i. Si $d_i > d_U$, alors la seconde valeur potentielle d'ajustement de la date de début au plus tôt de la tâche i est calculée avec la tâche courante $\tau U, i$. Cette valeur n'est sauvegardée que lorsqu'elle est supérieure aux valeurs potentielles d'ajustement déjà calculée avec les intervalles de tâches de marge minimale. La deuxième partie nous permet de connaître la date à laquelle la tâche i doit être ajustée. On vérifie aussi si les conditions des règles (EF) ou (EF1) sont satisfaites. Si tel est le cas, alors la date de début au plus tôt de la tâche i est ajustée suivant les valeurs pré-calculées à cet effet.

– A l'itération suivante des boucles internes, les tâches $\tau(U,i)$ et $\rho(U,i)$ sont réinitialisées.

Afin de montrer que l'Algorithme 2 est correct, prouvons tout d'abord les propriétés de ses boucles internes.

Proposition 2.3. *Pour toute tâche i donnée, l'Algorithme 2 calcule la valeur potentielle d'ajustement $Dupd_i$ de la tâche i basée sur l'intervalle de tâches de densité maximale telle que :*

$$Dupd_i = \max_{U\,:\,d_U < d_i \wedge \mathrm{rest}(\Theta_{\rho(U,i),U}, c_i) > 0} \left(r_{\rho(U,i)} + \left\lceil \mathrm{rest}(\Theta_{\rho(U,i),U}, c_i) \cdot \frac{1}{c_i} \right\rceil \right) \tag{2.5}$$

Démonstration. Soit $i \in T$ une tâche. Chaque choix d'une tâche $U \in T$ dans la boucle externe (ligne 3) commence avec les valeurs $r_\rho = -\infty$ et $maxEnergy = 0$. La boucle interne de la ligne 5 itère à travers l'ensemble des tâches $i' \in T$ (T trié dans l'ordre décroissant des dates de début au plus tôt). Pour chaque tâche $i' \in T$ telle que $r_{i'} > r_i$, si $d_{i'} \leq d_U$, alors $i' \in \Theta_{i',U}$, et $e_{i'}$ est ajouté à $Energy$ (ligne 7). D'où $Energy = e_{\Theta_{i',U}}$ à chaque itération. La condition de la ligne 8 assure que r_ρ et $maxEnergy = e_{\Theta_{\rho,U}}$ sont à jour et reflètent la tâche $\rho(U,i)$ pour l'intervalle de tâches courant $\Theta_{i',U}$. Par conséquent, à la i^{ieme} itération de la boucle interne, si $d_i > d_U$ alors la ligne 11 calcule $rest(i,U) = \mathrm{rest}(\Theta_{\rho(U,i),U}, c_i)$ et sa valeur potentielle de mise à jour $r_\rho + \lceil \mathrm{rest}(i,U) \cdot \frac{1}{c_i} \rceil$ si $rest(i,U) > 0$, et $-\infty$ sinon. A la ligne 13, $Dupd_i$ est mise à jour seulement si $r_\rho + \lceil \mathrm{rest}(i,U) \cdot \frac{1}{c_i} \rceil$ est

Algorithm 2: Algorithme d'edge-finding de complexité $\mathcal{O}(n^2)$ en temps

Entrées: T, T' et T'' trois vecteurs de tâches
Sorties: La borne inférieure LB'_i de la date de début au plus tôt de chaque tâche i

1 **pour** $i \in T$ **faire**
2 $\quad LB'_i := r_i, \quad Dupd_i := -\infty, \quad SLupd_i := -\infty;$
3 **pour** $U \in T$ *trié dans l'ordre croissant des dates de fin au plus tôt* **faire**
4 $\quad Energy := 0, \quad maxEnergy := 0, \quad r_\rho := -\infty;$
5 $\quad$ **pour** $i \in T'$ *trié dans l'ordre décroissant des dates de début au plus tôt* **faire**
6 $\quad\quad$ **si** $d_i \leq d_U$ **alors**
7 $\quad\quad\quad Energy := Energy + e_i;$
8 $\quad\quad\quad$ **si** $\left(\frac{Energy}{d_U - r_i} > \frac{maxEnergy}{d_U - r_\rho}\right)$ **alors**
9 $\quad\quad\quad\quad maxEnergy := Energy, \quad r_\rho := r_i$;
10 $\quad\quad$ **sinon**
11 $\quad\quad\quad rest := maxEnergy - (C - c_i)(d_U - r_\rho);$
12 $\quad\quad\quad$ **si** $(rest > 0)$ **alors**
13 $\quad\quad\quad\quad Dupd_i := \max(Dupd_i, \quad r_\rho + \lceil \frac{rest}{c_i} \rceil);$
14 $\quad\quad E_i := Energy$;
15 $\quad minSL := +\infty, r_\tau := d_U$;
16 $\quad$ **pour** $i \in T''$ *trié dans l'ordre croissant des dates de début au plus tôt* **faire**
17 $\quad\quad$ **si** $(C(d_U - r_i) - E_i < minSL)$ **alors**
18 $\quad\quad\quad r_\tau := r_i, \quad minSL := C(d_U - r_\tau) - E_i$;
19 $\quad\quad$ **si** $(d_i > d_U)$ **alors**
20 $\quad\quad\quad rest' := c_i(d_U - r_\tau) - minSL$;
21 $\quad\quad\quad$ **si** $(r_\tau \leq d_U \wedge rest' > 0)$ **alors**
22 $\quad\quad\quad\quad SLupd_i := max(SLupd_i, r_\tau + \lceil \frac{rest'}{c_i} \rceil);$
23 $\quad\quad\quad$ **si** $(r_i + p_i \geq d_U \vee minSL - e_i < 0)$ **alors**
24 $\quad\quad\quad\quad LB'_i := \max(LB'_i, Dupd_i, SLupd_i);$
25 **pour** $i \in T$ **faire**
26 $\quad r_i := LB'_i;$

supérieur à la valeur courante de $Dupd_i$. Et comme la boucle externe sélectionne les tâches U dans l'ordre croissant des d_U, nous avons :

$$Dupd_i = \max_{U\,:\,d_U < d_i \wedge \mathrm{rest}(i,U) > 0} r_\rho + \lceil \mathrm{rest}(i,U) \cdot \frac{1}{c_i} \rceil \ . \tag{2.6}$$

Par conséquent (2.5) est vérifiée et la proposition est correcte. □

Proposition 2.4. *Pour chaque tâche i, l'Algorithme 2 calcule la valeur potentielle d'ajustement $SLupd_i$ de la tâche i basée sur l'intervalle de tâche de marge minimale telle que :*

$$SLupd_i = \max_{U\,:\,d_U < d_i \wedge \mathrm{rest}(\Omega_{\tau(U,i),U}, c_i) > 0} \left(r_{\tau(U,i)} + \left\lceil \mathrm{rest}(\Omega_{\tau(U,i),U}, c_i) \cdot \frac{1}{c_i} \right\rceil \right) \ . \tag{2.7}$$

Démonstration. Soit $i \in T$ une tâche. Chaque choix de la tâche U dans la boucle externe (line 3) commence avec les valeurs $r_\tau = d_U$ et $minSL = +\infty$ (line 15). La boucle interne à la ligne 16 itère à travers l'ensemble de tâches $i' \in T$ (T trié dans l'ordre croissant des dates de début au plus tôt). Pour chaque tâche $i' \in T$, $e_{\Omega_{i',U}}$ a déjà été calculée dans la première boucle interne et sauvegardée dans la variable $E_{i'}$ (ligne 14). Cet énergie est utilisée pour calculer la marge de l'intervalle de tâches $\Omega_{i',U}$. Si $SL_{\Omega_{i',U}} < minSL$, alors les mises à jours $r_\tau = r_{i'}$ et $minSL = C(d_U - r_{i'}) - e_{\Omega_{i',U}}$ sont effectuées pour refléter $\tau(U,i)$ de l'intervalle de tâches courant $\Omega_{\tau(U,i),U}$. A la i^{ieme} itération, si $d_i > d_U$, alors la ligne 20 calcule $rest'(i,U) = \text{rest}(\Theta_{\tau(U,i),U}, c_i)$ et sa valeur potentielle de mise à jour $r_\tau + \lceil \text{rest}'(i,U) \cdot \frac{1}{c_i} \rceil$ si $rest'(i,U) > 0$, et $-\infty$ sinon. A la ligne 22, $SLupd_i$ est mise à jour seulement si $r_\tau + \lceil \text{rest}'(i,U) \cdot \frac{1}{c_i} \rceil$ est supérieur à la valeur courante de $SLupd_i$. Comme la boucle externe sélectionne les tâches U dans l'ordre croissant des d_U nous avons :

$$SLupd_i = \max_{U\,:\, d_U \leq d_i \wedge r_\tau \leq d_U \wedge \text{rest}'(i,U) > 0} r_\tau + \lceil \text{rest}'(i,U) \cdot \frac{1}{c_i} \rceil \ . \tag{2.8}$$

Par conséquent (2.7) est vérifiée et la proposition est correcte. □

Nous pouvons maintenant montrer que la condition de la règle (EF) peut être détectée en utilisant l'intervalle de tâches de marge minimale.

Théorème 2.1. *Pour une tâche $i \in T$ donnée et pour un ensemble de tâches $\Omega \subseteq T \setminus \{i\}$,*

$$e_{\Omega \cup \{i\}} > C(d_\Omega - r_{\Omega \cup \{i\}}) \vee r_i + p_i \geq d_\Omega \tag{2.9}$$

si et seulement si

$$e_i > C\left(d_U - r_{\tau(U,i)}\right) - e_{\Omega_{\tau(U,i),U}} \vee r_i + p_i \geq d_U \tag{2.10}$$

pour une tâche $U \in T$ vérifiant $d_U < d_i$, et $\tau(U,i)$ tel qu'il est spécifié dans la Définition 2.2.

Démonstration. Soit $i \in T$ une tâche donnée.

Il est évident que la condition de la règle (EF1) peut être détectée avec la condition $r_i + p_i \geq d_U$ pour toute tâche $U \in T$ vérifiant $d_i > d_\Omega$. Dans la suite de la preuve, nous nous intéressons uniquement à la condition de la règle (EF). Nous commençons par démontrer que (2.9) implique (2.10). Supposons qu'il existe un ensemble $\Omega \subseteq T \setminus \{i\}$ de tâches vérifiant $C(d_\Omega - r_{\Omega \cup \{i\}}) < e_\Omega + e_i$. Avec la règle (EF) on peut déduire la relation

$\Omega \lessdot i$. D'après la Proposition 2.1, il existe un intervalle de tâches $\Omega_{L,U} \lessdot i$, tel que $d_i > d_U$ et $r_L \leq r_i$. D'après la Définition 2.2 nous avons

$$C(d_U - r_{\tau(U,i)}) - e_{\Omega_{\tau(U,i),U}} \leq C(d_U - r_L) - e_{\Omega_{L,U}}. \tag{2.11}$$

En ajoutant e_i à chaque membre des inequations (2.11) et en utilisant le fait que

$$C(d_U - r_L) < e_{\Omega_{L,U}} + e_i, \tag{2.12}$$

il s'en suit que

$$C\left(d_U - r_{\tau(U,i)}\right) < e_{\Omega_{\tau(U,i),U}} + e_i. \tag{2.13}$$

Maintenant, il reste à montrer que (2.10) implique (2.9). Soit $U \in T$ une tâche vérifiant $d_U < d_i$ et $\tau(U,i) \in T$ une tâche vérifiant (2.10). Par définition d'intervalle de tâches, la condition $d_i > d_U$ implique $i \notin \Omega_{\tau(U,i),U}$. Comme $r_{\tau(U,i)} \leq r_i$, nous avons

$$e_{\Omega_{\tau(U,i),U}} + e_i > C(d_U - r_{\tau(U,i)}) \geq C(d_{\Omega_{\tau(U,i),U}} - r_{\Omega_{\tau(U,i),U} \cup \{i\}}). \tag{2.14}$$

D'où, (2.9) est satisfaite pour $\Omega = \Omega_{\tau(U,i),U}$. □

En vertu de la Proposition 2.4, la tâche $\tau(U,i)$ et l'intervalle de tâches de marge minimale sont bien déterminés dans la boucle interne de la ligne 16. Combinée avec le Théorème 2.1, ceci justifie l'utilisation de l'inégalité $minSL - e_i < 0$ à la ligne 23 pour vérifier (EF). C'est aussi à cette ligne que la condition (EF1) est testée. Donc, pour toute tâche i, l'Algorithme 2 détecte correctement l'ensemble $\Omega \subseteq T \setminus i$ pour lequel les règles (EF) et (EF1) détecte la relation $\Omega \lessdot i$.

Un algorithme complet d'edge-finding doit à chaque instant rechercher pour chaque tâche i, l'ensemble de tâches Θ pour lequel l'ajustement de r_i est maximal. Dans le théorème qui suit, nous montrons que notre algorithme effectue correctement les mises à jour des dates de début au plus tôt des tâches, mais l'ajustement n'est pas nécessairement maximal à la première itération.

Théorème 2.2. *Pour une tâche $i \in T$ donnée et pour l'ajustement maximal de la date de début au plus tôt de cette tâche LB_i tel qu'il a été spécifié dans la Définition 1.13, l'Algorithme 2 est un bon algorithme pour calculer une borne inférieure LB'_i, vérifiant $r_i < LB'_i \leq LB_i$ si $r_i < LB_i$, et $LB'_i = r_i$ si $r_i = LB_i$.*

Démonstration. Soit $i \in T$ une tâche.

LB'_i est initialisée à r_i. Après chaque détection, LB'_i est mise à jour avec $\max(Dupd_i, SLupd_i, LB'_i)$ (ligne 24). Il s'en suit que $LB'_i \geq r_i$. Si l'égalité $LB_i = r_i$ est vérifiée alors, aucune des conditions (EF) ou (EF1) n'est détectée par l'Algorithme 2 et par la suite $LB'_i = r_i$ avec la boucle de la ligne 25. Dans la suite de la preuve, nous supposons que $r_i < LB_i$. D'après les Propositions 2.1 et 2.2, il existe deux intervalles de tâches $\Theta_{l,u} \subseteq \Omega_{L,U} \subseteq T \setminus \{i\}$ vérifiant $r_L \leq r_l < d_u \leq d_U < d_i$ et $r_L \leq r_i$ pour lesquels on a :

$$\alpha\left(\Omega_{L,U}, i\right) \quad \wedge \quad LB_i = r_{\Theta_{l,u}} + \left\lceil \frac{1}{c_i} \operatorname{rest}(\Theta_{l,u}, c_i) \right\rceil \tag{2.15}$$

Nous avons déjà démontré qu'avec la Proposition 2.4 et le Théorème 2.1, l'Algorithme 2 détecte correctement les conditions de l'edge-finding. Il reste à montrer que les valeurs potentielles d'ajustement calculées par cet algorithme sont supérieures à la valeur courante de la date de début au plus tôt des tâches lorsqu'il y a détection. Et comme l'expression d'ajustement est identique pour les deux règles, la preuve qui suit est valable pour les deux règles. Nous distinguerons deux cas :

1. $\boldsymbol{r_i < r_l}$: Ici, nous prouvons que l'ajustement peut être effectué avec l'intervalle de tâches de densité maximale. D'après la Définition 2.4, nous avons

$$\frac{e_{\Theta_{l,u}}}{d_u - r_l} \leq \frac{e_{\Theta_{\rho(u,i),u}}}{d_u - r_{\rho(u,i)}}. \tag{2.16}$$

La condition $\operatorname{rest}(\Theta_{l,u}, c_i) > 0$ est vérifiée car la paire $(\Omega_{L,U}, \Theta_{l,u})$ permet d'ajuster la date de début au plus tôt de la tâche i. Par conséquent

$$\frac{e_{\Theta_{l,u}}}{d_{\Theta_{l,u}} - r_{\Theta_{l,u}}} > C - c_i. \tag{2.17}$$

Les relations (2.16) et (2.17) permettent de déduire que $\operatorname{rest}(\Theta_{\rho(u,i),u}, c_i) > 0$. Nous avons $r_{\rho(u,i)} + \left\lceil \frac{1}{c_i} \operatorname{rest}\left(\Theta_{\rho(u,i,u}, c_i\right) \right\rceil > r_i$ car $r_i < r_{\rho(u,i)}$. D'après la Proposition 2.8, la valeur $Dupd_i = r_{\rho(u,i)} + \left\lceil \frac{1}{c_i} \operatorname{rest}\left(\Theta_{\rho(u,i),u}, c_i\right) \right\rceil > r_i$ est calculée par l'Algorithme 2 à la ligne 13. Par la suite, après la détection de la condition d'edge-finding à la ligne 23, la date de début au plus tôt de tâche i est ajustée à $LB'_i \geq Dupd_i > r_i$.

2. $\boldsymbol{r_l \leq r_i}$: Ici nous prouvons que l'ajustement peut être effectué avec l'intervalle de tâches de marge maximale. D'après la Définition 2.2, nous avons :

$$C(d_u - r_{\tau(u,i)}) - e_{\Theta_{\tau(u,i),u}} \leq C(d_u - r_l) - e_{\Theta_{l,u}} \tag{2.18}$$

En ajoutant $-c_i(d_u - r_{\tau(u,i)}) - c_i \cdot r_{\tau(u,i)}$ au membre de gauche et $-c_i(d_u - r_l) - c_i \cdot r_l$ au membre de droite de l'inégalité (2.18) nous avons

$$-c_i \cdot r_{\tau(u,i)} - \text{rest}(\Theta_{\tau(u,i),u}, c_i) \leq -c_i \cdot r_l - \text{rest}(\Theta_{l,u}, c_i) \tag{2.19}$$

Par conséquent

$$r_{\tau(u,i)} + \frac{1}{c_i}\,\text{rest}(\Theta_{\tau(u,i),u}, c_i) \geq r_l + \frac{1}{c_i}\,\text{rest}(\Theta_{l,u}, c_i) \tag{2.20}$$

et

$$r_{\tau(u,i)} + \frac{1}{c_i}\,\text{rest}(\Theta_{\tau(u,i),u}, c_i) > r_i \tag{2.21}$$

car $r_{\Theta_{l,u}} + \frac{1}{c_i}\,\text{rest}(\Theta_{l,u}, c_i) > r_i$. De l'inégalité (2.21), il vient que

$$\text{rest}(\Theta_{\tau(u,i),u}, c_i) > 0 \tag{2.22}$$

car $r_{\tau(u,i)} \leq r_i$. D'après la Proposition 2.4, la valeur

$$SLupd_i = r_{\tau(u,i)} + \frac{1}{c_i}\,\text{rest}\big(\Theta_{\tau(u,i),u}, c_i\big) > r_i \tag{2.23}$$

est calculée par l'Algorithme 2 à la ligne 22. Par la suite, après la détection de la condition d'edge-finding à la ligne 23, la date de début au plus tôt de tâche i est ajustée à $LB'_i \geq SLupd_i > r_i$.

Par conséquent l'Algorithme 2 est correct. □

2.1.3 Complexité pour l'ajustement maximal

D'après le Théorème 2.2, l'Algorithme 2 doit chaque fois effectuer une mise à jour de r_i si celle-ci est justifiée par la règle d'edge-finding. Toutefois cette mise à jour ne sera pas toujours maximale. Étant donné qu'il y a un nombre fini d'ajustements, l'Algorithme 2 finit par atteindre le même point fixe que les autres algorithmes complets d'edge-finding. Cette approche naïve a été tout récemment utilisée pour réduire la complexité de l'algorithme de not-first/not-last cumulatif [38, 65, 42]. La situation est différente du précédent algorithme quadratique d'edge-finding [9] qui est incomplet car certains ajustements sont ignorées [54]. Nous montrons dans le théorème suivant que l'Algorithme 2 peut effectuer l'ajustement maximal immédiatement et si ce n'est pas le cas, il nécessite au plus $n-1$ itérations pour le faire.

Théorème 2.3. *Soit $i \in T$ une tâche d'une instance E-Réalisable de CuSP. Soit $\Theta \subseteq T \setminus \{i\}$ l'ensemble de tâches utilisé pour effectuer l'ajustement maximal de r_i avec la règle*

d'edge-finding. Soit $\rho(u,i) \in T$ la tâche donnée dans la Définition 2.4 appliquée à i et $u \in T$ vérifiant $d_u = d_\Theta$. L'Algorithme 2 effectue l'ajustement maximal de r_i

1. *A la première itération si $r_i \geq r_\Theta$ ou $r_i < r_\Theta \leq r_{\rho(u,i)}$, sinon*
2. *Après au plus $n-1$ itérations si $r_i < r_{\rho(u,i)} \leq r_\Theta$.*

Démonstration. Étant donné une tâche $i \in T$, soit $\Theta \subseteq T \setminus \{i\}$ l'ensemble de tâches utilisé pour effectuer l'ajustement maximal de r_i par la règle edge-finding. Soit $\rho(u,u) \in T$ la tâche donnée dans la Définition 2.4 appliquée à i et $u \in T$ avec u vérifiant $d_u = d_\Theta$.

1. Supposons $r_\Theta \leq r_i$. D'après les Propositions 2.1 et 2.2, et la seconde partie de la preuve du Théorème 2.2, l'inégalité (2.20) est vérifiée. Ainsi avec la Proposition 2.4, lorsque l'Algorithme 2 considère u dans la boucle externe et i dans la seconde boucle interne, nous avons

$$SLupd_i = r_{\tau(u,i)} + \lceil \frac{1}{c_i} \operatorname{rest}(\Theta_{\tau(u,i),u}, c_i) \rceil \geq r_\Theta + \lceil \frac{1}{c_i} \operatorname{rest}(\Theta, c_i) \rceil. \tag{2.24}$$

   Et comme l'ajustement effectué avec l'ensemble Θ est maximal, r_i est mise à jour à $r_\Theta + \lceil \frac{1}{c_i} \operatorname{rest}(\Theta, c_i) \rceil$.

2. Supposons $r_i < r_\Theta$. Nous analysons deux sous cas :

   (a) $\boldsymbol{r_i < r_\Theta \leq r_{\rho(u,i)}}$: D'après la définition de la tâche $\rho(u,i)$, nous avons

$$\frac{e_\Theta}{d_\Theta - r_\Theta} \leq \frac{e_{\Theta_{\rho(u,i),u}}}{d_u - r_{\rho(u,i)}}. \tag{2.25}$$

   En soustrayant $\frac{e_{\Theta_{\rho(u,i),u}}}{d_\Theta - r_\Theta}$ à chaque membre de (2.25), nous avons

$$e_\Theta - e_{\Theta_{\rho(u,i),u}} \leq \frac{e_{\Theta_{\rho(u,i),u}}}{d_u - r_{\rho(u,i)}}(r_{\rho(u,i)} - r_\Theta). \tag{2.26}$$

   Et comme le problème est E-réalisable, on a

$$\frac{e_\Theta}{d_\Theta - r_\Theta} \leq C. \tag{2.27}$$

   En combinant les inégalités (2.26) et (2.27) on obtient

$$Cr_\Theta + e_\Theta \leq Cr_{\rho(u,i)} + e_{\Theta_{\rho(u,i),u}}. \tag{2.28}$$

   De manière triviale, l'inégalité (2.28) est équivalente à

$$r_\Theta + \lceil \frac{1}{c_i} \operatorname{rest}(\Theta, c_i) \rceil \leq r_{\rho(u,i)} + \lceil \frac{1}{c_i} \operatorname{rest}(\Theta_{\rho(u,i),u}, c_i) \rceil. \tag{2.29}$$

La Proposition 2.8 montre que lorsque l'Algorithme 2 considère u dans la boucle externe et i dans la première boucle interne (ligne 5), nous avons

$$SLupd_i = r_{\rho(u,i)} + \left\lceil \frac{1}{c_i} \text{rest}(\Theta_{\rho(u,i),u}, c_i) \right\rceil \geq r_\Theta + \left\lceil \frac{1}{c_i} \text{rest}(\Theta, c_i) \right\rceil \tag{2.30}$$

Et comme l'ajustement effectué avec l'ensemble Θ est maximal, r_i est mise à jour à $r_\Theta + \left\lceil \frac{1}{c_i} \text{rest}(\Theta, c_i) \right\rceil$.

(b) $\boldsymbol{r_i < r_{\rho(u,i)} \leq r_\Theta}$: A la k^{ieme} itération de l'Algorithme 2, considérons les ensembles de tâches $\Theta^k_{\leq i} := \{j, j \in T \wedge r_j \leq r_i \wedge d_j \leq d_\Theta\}$ et $\Theta^k_{>i} := \{j, j \in T \wedge r_j > r_i \wedge d_j \leq d_\Theta\}$. Si l'ajustement maximal n'est pas atteint après cette itération, alors au moins une tâche est déplacée de $\Theta^k_{>i}$ pour $\Theta^k_{\leq i}$. En effet, si à la k^{ieme} et la $k+1^{ieme}$ itération, $\Theta^k_{\leq i} = \Theta^{k+1}_{\leq i}$, $\Theta^k_{>i} = \Theta^{k+1}_{>i}$ et l'ajustement n'est toujours pas maximal alors la tâche $\tau(u,i) \in \Theta^k_{\leq i} = \Theta^{k+1}_{\leq i}$ (Définition 2.2) et $\rho(u,i) \in \Theta^k_{>i} = \Theta^{k+1}_{>i}$ (Définition 2.4) sont identiques dans les deux itérations. Par conséquent à la $k+1^{ieme}$ itération, aucun ajustement n'est effectué et l'ajustement maximal de la date de début au plus tôt de la tâche i n'est jamais atteint. Ceci contredit le fait que l'Algorithme 2 atteint le même point fixe que les autres algorithmes d'edge-finding. D'où, l'ajustement maximal de la date de début au plus tôt de la tâche i est atteint après au plus $|\Theta^1_{>i}| \leq n-1$ itérations.

□

Les illustrations expérimentales réalisées montrent qu'en pratique, notre algorithme effectue l'ajustement maximal après un nombre réduit d'itérations. La règle d'edge-finding n'étant pas idempotente, l'ajustement des dates de début au plus tôt et de fin au plus tard n'est pas suffisant à la première itération pour atteindre le point fixe. C'est en combinant plusieurs de ces ajustements et ceux réalisés par d'autres propagateurs que l'on atteint le point fixe. Le nombre d'ajustements de la date de début au plus tôt d'une tâche n'approche pas généralement le pire des cas, car dans les problèmes d'ordonnancement cumulatif, il y a un nombre réduit de tâches qui admettent des ajustements. Cette affirmation est justifiée par les observations expérimentales effectuées au Chapitre 3.

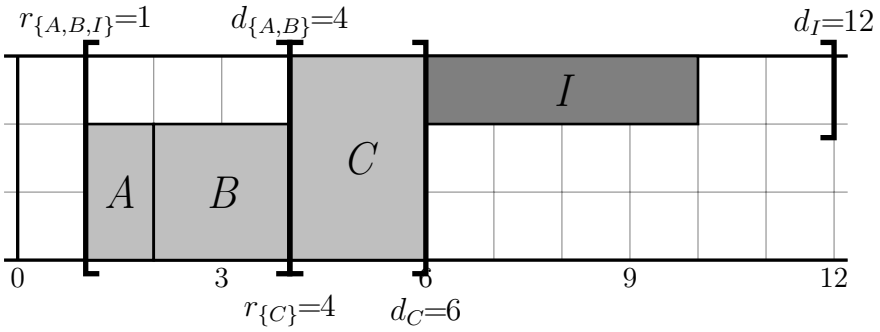


FIGURE 2.1 – Instance de CuSP où quatre tâches doivent s'exécuter sur une ressource de capacité $C = 3$.

2.1.4 Comparaison avec le timetable edge finding [77]

Il est démontré dans [77] qu'au point fixe de la règle timetable edge finding, l'edge finding est incapable de réduire la fenêtre de temps d'une tâche. Ceci traduit la dominance du timetable edge finding sur l'edge-finding. Nous montrons dans ce paragraphe que cette affirmation est fausse. Pour cela, considérons l'instance de CuSP de la Figure 2.1 tiré de [41] où quatre tâches doivent s'exécuter sur une ressource de capacité 3.

En observant les données de la Figure 2.1 on constate que seule les tâches A, B et I peuvent être ajustée par un algorithme de filtrage car la tâche C est déjà ordonnancée. Une application de la règle d'edge-finding à cette instance permet d'ajuster la date de début au plus tôt de la tâche I de 1 à 6. En effet, pour $\Omega = \{A, B, C\}$, nous avons $e_{\Omega\cup\{I\}} = 16 > C(d_\Omega - r_{\Omega\cup\{I\}}) = 15$. Par conséquent, la condition de la détection de la règle d'edge-finding est satisfaite. Pour l'ajustement, en considérant $\Theta = \{C\}$, nous avons $rest(\Theta, c_I) = 2$ et $r_\Theta + \lceil \frac{rest(\Theta,c_I)}{c_I} \rceil = 6$. Par conséquent, la date de début au plus tôt de la tâche I est ajustée de 1 à 6.

Par contre, une application de la règle du timetable edge finding à cette instance ne produit pas le même résultat. Nous pouvons maintenant prouver formellement que le timetable edge finding ne domine pas l'edge-finding.

Théorème 2.4. *La règle d'edge finding n'est pas dominée par la règle du timetable edge finding.*

Démonstration. Considérons l'instance de CuSP de la Figure 2.1 où quatre tâches s'exé-

cutent sur une ressource de capacité 3. Nous avons montré plus haut qu'en utilisant la règle d'edge-finding, la date de début au plus tôt de la tâche I est ajustée de 1 à 6. Nous allons prouver qu'en appliquant la règle du timetable, aucun ajustement n'est produit (le timetable ne produit pas le même ajustement). Si nous voulons appliquer le timetable edge finding pour ajuster la date de début au plus tôt de la tâche I, alors nous devons considérer un ensemble de tâches $\Omega \subseteq T^{EF} \setminus \{I\}$. D'après les données de l'instance de CuSP de la Figure 2.1, nous avons $T^{EF} = \{A, B\}$. Par conséquent, aucun ensemble de tâches $\Omega \subseteq T^{EF} \setminus \{I\}$ ne permet de détecter la règle. Par conséquent, le timetable edge finding n'effectue aucun ajustement sur cette instance. Ce qui contredit le résultat de Vilím suivant lequel la règle d'edge finding est dominée par la règle du timetable edge finding. Le même raisonnement est appliqué pour les versions améliorées de cette règle. □

Dans l'instance de CuSP de la Figure 2.2, le point fixe du timetable edge finding est atteint, mais les règles d'edge-finding et d'extended edge-finding sont satisfaites lorsqu'on considère la paire (Ω, I) où $\Omega = \{A, B, C\}$ avec la condition $r_I + p_I < d_\Omega$. Par conséquent, le Lemme 1 de [77] est incorrect. Il en est de même pour la Proposition 1 de [77] qui est déduite de ce lemme. Le problème avec le timetable edge finding vient du fait que la règle ne considère que les intervalles de tâches ayant pour bornes les tâches ayant une partie libre. Cette restriction réduit alors la puissance de filtrage de la règle. Cette spécification n'a pas été prise en compte dans [77] dans la preuve de la domination de l'edge-finding par le timetable edge finding.

2.2 L'extended edge-finding

2.2.1 Propriétés de dominance de la règle d'extended edge-finding

Les propriétés de dominance de l'extended edge-finding sont similaires en nature à celles de l'edge-finding. Certaines sont développées dans [54, 60].

La proposition suivante montre le lien entre la condition de l'edge-finding et celle de l'extended edge-finding et permet de justifier l'origine du nom d'extended edge-finding.

Proposition 2.5. *Soit $\Omega \subset T$ un ensemble de tâches et $i \in T \setminus \Omega$ une tâche.*

$$r_i \leq r_\Omega < r_i + p_i \wedge \alpha(\Omega, i) \Rightarrow \beta(\Omega, i)$$

Démonstration. Supposons que les conditions $r_i \leq r_\Omega < r_i + p_i$ et $\alpha(\Omega, i)$ sont satisfaites et montrons que la condition $\beta(\Omega, i)$ est vérifiée.

De l'inégalité $r_i \leq r_\Omega$, il vient que :

$$C\left(d_\Omega - r_{\Omega\cup\{i\}}\right) = C(d_\Omega - r_\Omega) + C(r_\Omega - r_i).$$

Comme $i \notin \Omega$, on a $e_{\Omega\cup\{i\}} = e_\Omega + c_i \times p_i$ et la condition $\alpha(\Omega, i)$ devient alors

$$C(d_\Omega - r_\Omega) + C(r_\Omega - r_i) < e_\Omega + c_i \times p_i. \tag{2.31}$$

De plus

$$C(d_\Omega - r_\Omega) + c_i(r_\Omega - r_i) < C(d_\Omega - r_\Omega) + C(r_\Omega - r_i) \tag{2.32}$$

car $c_i \leq C$. Des inégalités (2.31) et (2.32), on déduit

$$C(d_\Omega - r_\Omega) + c_i(r_\Omega - r_i) < e_\Omega + c_i \times p_i$$

et par conséquent la condition $\beta(\Omega, i)$ est satisfaite. □

Les propriétés de dominance sont utilisées pour caractériser afin de réduire les paires (Ω, Θ) qui doivent être considérées pour réduire la date de début au plus tôt d'une tâche i. Nous dirons que l'application de la règle d'extended edge-finding sur la tâche i et la paire (Ω, Θ) permet de réduire la date de début au plus tôt d'une tâche i si $\Omega \lessdot i$, $rest(\Theta, c_i) > 0$ et $r_\Theta + \frac{1}{c_i} rest(\Theta, c_i) > r_i$. La Proposition 2.6 résume les principales propriétés de dominance de la règle d'extended edge-finding.

Proposition 2.6. *[54] Soit $i \in T$ une tâche d'une instance E-Réalisable de CuSP, $\Omega \subseteq T \setminus \{i\}$ et $\Theta \subseteq T \setminus \{i\}$ deux ensembles de tâches avec $\Theta \subseteq \Omega$. Si l'application de la règle d'extended edge-finding sur la tâche i et la paire (Ω, Θ) permet de réduire la date de début au plus tôt d'une tâche i alors*

(i) il existe quatre tâches L, U, l, u telles que $r_L \leq r_l < d_u \leq d_U < d_i \wedge r_i \leq r_L < r_i + p_i$;

(ii) la règle d'extended edge-finding appliquée à la tâche i et la paire $(\Omega_{L,U}, \Omega_{l,u})$ permet d'effectuer un meilleur ajustement de la date de début au plus tôt d'une tâche i.

2.2.2 L'algorithme d'extended edge-finding de Mercier et Van Hentenryck est incorrect [54]

Nous montrons dans ce paragraphe que l'algorithme CalcEEF ([54], Algorithme 7) qui est reproduit ici à l'Algorithme 3 est incorrect. L'algorithme CalcEEF prend en entrée deux vecteurs de tâches : le vecteur X où les tâches sont triées dans l'ordre croissant des dates de début au plus tôt et le vecteur Y où les tâches sont triées dans l'ordre croissant des dates de fin au plus tard. Cet algorithme est supposé fonctionner comme suit : La procédure CalcEF de la ligne 1 est l'algorithme d'edge-finding ([54], Algorithme 5) pour l'ajustement des dates de début au plus tôt. C'est dans cette procédure que le vecteur LB est initialisé. CalcEF est l'algorithme d'edge-finding de deux phases où les valeurs potentielles d'ajustement sont calculées dans la procédure CalcR ([54], Algorithme 3) tel que l'indique la spécification de la règle. La boucle externe (ligne 2) énumère toutes les possibles dates de fin au plus tard d_Ω. Chaque exécution de cette boucle externe essaie de trouver l'ensemble des tâches Ω correspondant à la date de fin au plus tard d_Ω et considère la mise à jour des tâches i vérifiant $d_i > d_\Omega$. La première boucle interne (ligne 4) calcule et sauvegardée dans E l'énergie des pseudo intervalles de tâches. La seconde boucle interne (ligne 8) utilise l'énergie des pseudo intervalles de tâches sauvegardée à la précédente boucle pour calculer l'énergie des intervalles de tâches. La troisième boucle externe (ligne 11) initialise le vecteur $CEEF$. La quatrième boucle interne (ligne 13) sauvegarde pour chaque consommation en ressource $c \in Sc$ avec $Sc = \{c_i, i \in T\}$ l'expression $CEEF[c, i] = \max_{x \geq i}((C - c)r_{X[x]} + E[x] - Cd_{Y[y]})$ qui est la partie de la condition d'extended edge-finding indépendante de la tâche devant être ajustée. La dernière boucle interne (ligne 16) identifie la tâche pour laquelle les conditions d'extended edge-finding sont vérifiées , i.e. $CEEF[c_{X[i]}, i] + c_{X[i]}(r_{X[i]} + p_{X[i]}) > 0$ et $d_{X[i]} > d_{Y[y]}$.

L'affirmation de Mercier et Van Hentenryck suivant laquelle l'algorithme CalcEEF calcule LB_i pour tout $i \in T$ est incorrecte. En effet, considérons l'instance de CuSP de la Figure 2.2 où trois tâches doivent être exécutées sur une ressource de capacité 4. Cette instance a exactement une solution telle que l'illustre la Figure 2.2.

Une observation des données de cet exemple montre que les tâches A et B sont déjà ordonnancées i.e. $r_A + p_A = d_A$ et $r_B + p_B = d_B$. En appliquant un algorithme de fil-

Algorithm 3: CALCEEF : Algorithme d'extended edge-finding [54]

Entrées: X vecteur des tâches trié dans l'ordre croissant des dates de début au plus tôt
Entrées: Y vecteur des tâches trié dans l'ordre croissant des dates de fin au plus tard
Sorties: $LB[i] = LB(X[i]) \quad (1 \leq i \leq n)$.

1 $CalcEF()$;
2 **pour** $y \leftarrow 1$ *to* $n-1$ **faire**
3 $\quad E \leftarrow 0$;
4 $\quad$ **pour** $x \leftarrow n$ *down to* 1 **faire**
5 $\quad\quad$ **si** $d_{X[x]} \leq d_{Y[y]}$ **alors**
6 $\quad\quad\quad E \leftarrow E + e_{X[x]}$;
7 $\quad\quad E[x] \leftarrow E$;
8 $\quad$ **pour** $x \leftarrow 1$ *to* $n-1$ **faire**
9 $\quad\quad$ **si** $r_{X[x]} = r_{X[x+1]}$ **alors**
10 $\quad\quad\quad E[x+1] \leftarrow E[x]$;
11 $\quad$ **pour chaque** $c \in Sc$ **faire**
12 $\quad\quad CEEF[c, n+1] \leftarrow -\infty$;
13 $\quad$ **pour** $x \leftarrow n$ *down to* 1 **faire**
14 $\quad\quad$ **pour chaque** $c \in Sc$ **faire**
15 $\quad\quad\quad CEEF[c, x] \leftarrow max(CEEF[c, x+1], (C-c)r_{X[x]} + E[x] - Cd_{Y[y]})$;
16 $\quad$ **pour** $i \leftarrow 1$ *to* n **faire**
17 $\quad\quad$ **si** $CEEF[c_{X[i]}, i] + c_{X[i]}(r_{X[i]} + p_{X[i]}) > 0$ **alors**
18 $\quad\quad\quad$ **if** $d_{X[i]} > d_{Y[y]}$ **then**
19 $\quad\quad\quad\quad LB[i] \leftarrow max(LB[i], R[c_{X[i]}, y])$;

trage à cette instance, seule les fenêtres de la tâche I peuvent être modifiées. Comme l'intervalle de temps de la tâche B n'interfère pas avec celui des tâches A et I, seule la tâche A peut intervenir dans l'ajustement de la date de début au plus tôt de I. En appliquant l'edge-finding et l'extended edge-finding à cette instance, aucun ajustement n'est réalisé. En effet, pour $\Omega = \{A\}$, nous avons $e_{\Omega \cup \{I\}} = 12 < C(d_\Omega - r_{\Omega \cup \{I\}}) = 16$. Par conséquent aucune détection n'est effectuée par la règle edge-finding. On a également $e_\Omega + c_I(r_I + p_I - r_\Omega) = 8 \leq C(d_\Omega - r_\Omega) = 8$. D'où, aucune détection n'est réalisée par la règle d'extended edge-finding et par conséquent la date de début au plus tôt de la tâche I reste $LB_I = 0$ après application de ces deux algorithmes de filtrage. Mais, en appliquant l'algorithme d'extended edge-finding de Mercier et Van Hentenryck, la conclusion est différente. Le problème avec l'algorithme CALCEEF provient de la quatrième boucle interne (ligne 13) où les $CEEF[c, i]$ sont calculées pour chaque consommation en ressource c. La valeur $CEEF[c, i]$ est calculée pour chaque intervalle de tâches $\Omega_{X[x],Y[y]}$ même pour

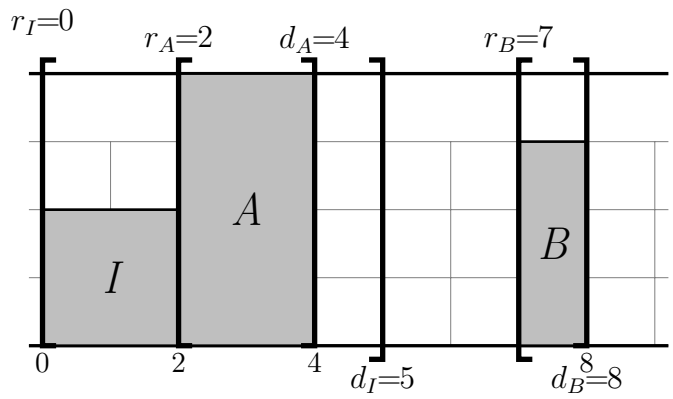


FIGURE 2.2 – Instance de CuSP de trois tâches devant être exécutées sur une ressource de capacité $C = 4$.

les intervalles vides. Ce manque de contrôle peut aboutir à une détection erronée comme c'est le cas dans l'exemple de la Figure 2.2. Nous pouvons maintenant montrer de manière formel que CALCEEF est incorrect.

Théorème 2.5. *L'algorithme* CALCEEF *([54], Algorithm 7) ne calcule pas* LB_i *pour tout* $i \in T$.

Démonstration. Considérons l'instance de CuSP de la Figure 2.2 où trois tâches doivent être exécutées sur une ressource de capacité $C = 4$. Nous avons déjà montré que $LB_I = 0$ et que le problème possède exactement une solution. Appliquons maintenant l'algorithme CALCEEF à cette instance. A partir des données, nous avons les vecteurs $X = (I, A, B)$ et $Y = (A, I, B)$ exigés à l'entrée par l'algorithme. Aucun ajustement n'est réalisé par la procédure CALCEF à la ligne 1. Considérons A à la boucle externe (ligne 2) i.e. $d_{Y[y]} = d_A = 4$. Dans la première boucle interne, nous avons $E[B] = 0$, $E[A] = 8$ et $E[I] = 8$. Les tâches ayant des dates de début au plus tôt différentes, les pseudo intervalles de tâches sont les intervalles de tâches. Ainsi aucune modification n'est apportée sur les énergies dans la seconde boucle interne. A la quatrième boucle interne (ligne 13), pour $c = c_I$, nous avons $CEEF[c_I, B] = -2$, $CEEF[c_I, A] = -2$ car $-2 > (C - c_I)r_A + E[A] - Cd_A = -4$. Nous avons aussi $CEEF[c_I, I] = -2$ car $-2 > (C - c_I)r_I + E[I] - Cd_A = -8$. Dans la dernière boucle interne (ligne 16), pour $X[i] = I$ nous avons $CEEF[c_I, I] + c_I(r_I + p_I) = -2 + 4 = 2 > 0$ et $d_I > d_A$. Par la suite, $LB_I = r_A + \lceil \frac{rest(\{A\}, c_I)}{c_I} \rceil = 4$. car $rest(\{A\}, c_I) = 4$. Après

cet ajustement, nous avons $r_I + p_I = 6 > d_I = 5$ et le problème n'admet plus de solutions. Ceci prouve que l'algorithme CALCEEF ajuste incorrectement les fenêtres de temps des tâches. □

L'algorithme CALCEEF est une amélioration de l'algorithme CALCEEFI de complexité $\mathcal{O}(n^3)$ en temps. C'est aussi une seconde phase de l'algorithme d'extended edge-finding proposée par Mercier et Van Hentenryck dans ([54], Algorithm 6). Cet algorithme est aussi incorrect car souffre du même problème. En effet, en utilisant le contre exemple de la Figure 2.2, il est possible de montrer que l'algorithme CALCEEFI est incorrect. Considérons A dans la boucle externe i.e. $d_{Y[y]} = d_A = 4$ ([54], Algorithm 6, lignes 2-19). Dans la première boucle interne ([54], Algorithm 6, lignes 4-9), nous avons $E[B] = 0$, $E[A] = 8$ et $E[I] = 8$. La seconde boucle interne contient une autre boucle et lorsque $X[x] = B$ pour la boucle interne ([54], Algorithm 6, lignes 10-18) et $X[i] = I$ pour la boucle plus interne ([54], Algorithm 6, lignes 11-17), nous avons : $E[B] + c_I(r_I + p_I - r_{X[x]}) = -10$ et $C(d_{Y[y]} - r_{X[x]}) = -12$. Par conséquent, l'algorithme CALCEEFI détecte $I \lessdot \Omega_{B,A}$ et la date de début au plus tôt de la tâche I est mise à jour de 0 à 4. Ceci prouve que l'algorithme CALCEEFI ajuste incorrectement les fenêtres de temps des tâches, ce qui contredit les allégations de Mercier et Van Hentenryck suivant lesquelles l'algorithme CALCEEFI est un algorithme d'extended edge-finding. De cette dernière analyse, il ressort qu'il n'existe pas d'algorithme correct d'extended edge-finding dans la littérature.

L'idée théorique soutenant les algorithmes CALCEEFI et CALCEEF est bonne, mais au cours de son application, il est important de tenir compte de quelques contôles. Ces contrôles apparaissent naturellement et ne réduisent pas la puissance de filtrage des algorithmes. Au contraire, ils réduisent les potentiels intervalles de tâches utilisés pour détecter les conditions de l'extended edge-finding sans perdre sa complétude. Ces contrôles peuvent être considérés comme des propriétés de dominance car ils permettent de réduire les potentiels paires (Ω, i) utilisées pour détecter la règle d'extended edge-finding.

- Pour déterminer si une tâche i peut ou non être ordonnancée avant ou après une ensemble de tâches Ω ($i \notin \Omega$), il est nécessaire de s'assurer que l'ensemble Ω est non vide. L'ensemble vide n'apporte aucune information dans la résolution, mais peut conduire à un ajustement érroné des fenêtres de temps des tâches. C'est par exemple le cas dans notre contre exemple car l'intervalle de tâches $\Omega_{B,A}$ est vide

puisque $r_B > d_A$ et la relation $I \lessdot \Omega_{B,A}$ est détectée par les algorithmes CalcEEF et CalcEEFI. Cette détection a permis d'ajuster incorrectement la date de début au plus tôt de la tâche I de 0 à 4.

– Pour une instance E-Réalisable de CuSP, si la relation $i \lessdot \Omega$ est détecté par la règle d'extended edge-finding, alors au moins une unité d'énergie de la tâche i est utilisée après r_Ω. Cette condition est garantie si $r_i + p_i > r_\Omega$ et il est utilisé pour restreindre les différentes tâches i à considérer pour détecter la relation $i \lessdot \Omega$. Dans notre contre exemple, la relation $I \lessdot \Omega_{B,A}$ ne devait pas être détectée si ce contrôle avait été ajouté dans les algorithmes CalcEEF et CalcEEFI.

L'introduction des contrôles $r_\Omega < d_\Omega$ et $r_i + p_i > r_\Omega$ dans les algorithmes CalcEEF et CalcEEFI avant l'ajustement permet de corriger les imperfections de ces algorithmes. La complexité des algorithmes CalcEEF et CalcEEFI reste inchangé et elle effectue le même filtrage que la règle d'extended edge-finding. Au chapitre 3, nous introduirons une version de l'algorithme CalcEEFI pour étendre la phase de détection de l'algorithme d'edge-finding de Vilím [75]. La phase de détection passe alors de $\mathcal{O}(n \log n)$ à $\mathcal{O}(n^3)$ alors que la phase d'ajustement reste inchangé $\mathcal{O}(kn \log n)$ en temps. On obtient ainsi un algorithme complet d'extended edge-finding.

2.2.3 Nouvel algorithme d'extended edge-finding

Dans ce paragraphe, nous présentons un algorithme d'extended edge-finding de complexité $\mathcal{O}(n^2)$ qui effectue les ajustements complémentaires non effectués par l'edge-finding. L'ajustement effectué par cet algorithme n'est pas nécessairement maximal à la première itération, mais s'améliore aux itérations subséquentes. La combinaison de ce nouvel algorithme avec notre algorithme quadratique d'edge-finding [40] atteint le même point fixe que la conjonction des algorithmes d'edge-finding et d'extended edge-finding existant [75, 40, 54]. Notre algorithme d'extended edge-finding utilise la notion de marge d'un ensemble de tâches déjà utilisée pour l'edge-finding dans [40]. Pour une tâche i, l'algorithme identifie l'ensemble de tâches Ω de marge minimale qui satisfait $r_i < r_\Omega$.

L'une des importantes propriétés utilisées dans notre algorithme est le fait qu'au point fixe de la règle d'edge-finding, si la règle d'extended edge-finding détecte $\Omega \lessdot i$ pour un ensemble Ω de tâches et une tâche i avec $i \notin \Omega$ alors l'ensemble Ω peut être utilisée pour

calculer la valeur potentielle d'ajustement de r_i i.e., $rest(\Omega, c_i) > 0$ et $r_\Omega + \lceil \frac{1}{c_i} rest(\Omega, c_i) \rceil > r_i$. Cette propriété est formellement démontrée dans la proposition suivante.

Proposition 2.7. *Soient $\Omega \subset T$ un ensemble de tâches et $i \in T \backslash \Omega$ une tâche, toutes d'une instance E-Réalisable de CuSP. Au point fixe de l'edge-finding, si l'extended edge-finding détecte $\Omega \lessdot i$ alors $rest(\Omega, c_i) > 0$ et $r_\Omega + \lceil \frac{1}{c_i} rest(\Omega, c_i) \rceil > r_i$.*

Démonstration. Soit $\Omega \subset T$ un ensemble de tâches d'une instance de CuSP au point fixe de l'edge-finding i.e., pour tout ensemble de tâche $\Omega' \subset T$, pour toute tâche $i \in T \setminus \Omega'$, si $C\left(d_{\Omega'} - r_{\Omega' \cup \{i\}}\right) < e_{\Omega'} + e_i$ ou $r_i + p_i \geq d_{\Omega'}$ alors $\forall \Theta' \subseteq \Omega'$ avec $rest(\Theta', c_i) > 0$, nous avons $r_{\Theta'} + \lceil \frac{1}{c_i} rest(\Theta', c_i) \rceil \leq r_i$.
Soient $\Theta \subseteq \Omega$ un ensemble de tâches et $i \in T \setminus \Omega$ une tâche telle que l'application de la règle d'extended edge-finding à la paire (Ω, i) permet de détecter la relation $\Omega \lessdot i$ et d'utiliser l'ensemble Θ pour ajuster la date de début au plus tôt de la tâche i i.e, $C\left(d_\Omega - r_\Omega\right) < e_\Omega + c_i(r_i + p_i - r_\Omega)$, $r_i \leq r_\Omega$, $r_\Theta + \lceil \frac{1}{c_i} rest(\Theta, c_i) \rceil > r_i$ avec $rest(\Theta, c_i) > 0$. Nous allons montrer que $rest(\Omega, c_i) > 0$ et $r_\Omega + \lceil \frac{1}{c_i} rest(\Omega, c_i) \rceil > r_i$. Nous avons $r_i < r_\Omega$ car le point fixe de l'edge-finding est atteint. L'inégalité

$$C\left(d_\Omega - r_\Omega\right) < e_\Omega + c_i(r_i + p_i - r_\Omega). \tag{2.33}$$

est algébriquement équivalent à

$$rest(\Omega, c_i) > c_i(d_\Omega - r_i - p_i). \tag{2.34}$$

Le point fixe de l'edge-finding étant atteint, il est clair que $d_\Omega > r_i + p_i$ et par conséquent $rest(\Omega, c_i) > 0$. De l'inégalité $r_i < r_\Omega$ découle $r_\Omega + \lceil \frac{1}{c_i} rest(\Omega, c_i) \rceil > r_i$. □

D'après la Proposition 2.7, au point fixe de l'edge-finding, l'ensemble de tâches utilisé pour détecter la relation $\Omega \lessdot i$ peut être utilisé pour calculer une valeur potentielle d'ajustement de date de début au plus tôt de la tâche i. Ce n'était pas le cas pour l'edge-finding et c'était pour cela que l'algorithme quadratique d'edge-finding proposé par Baptiste [9] était incomplète. Nous allons adopter cette approche pour obtenir un algorithme quadratique d'extended edge-finding. Ce nouvel algorithme, Algorithme 4 effectue l'ajustement maximal après un nombre fini d'itérations.

Cet algorithme fonctionne comme suit : Pour chaque tâche i, l'algorithme identifie l'intervalle de tâches Ω de marge minimale vérifiant $r_i < r_\Omega$ et $d_\Omega \leq d_i$. Si les conditions

de l'extended edge-finding sont satisfaites avec la paire (Ω, i), alors l'ensemble Ω est utilisé par l'algorithme pour ajuster r_i.

La localisation de l'intervalle de tâches de marge minimale est différente de celle utilisée pour l'edge-finding dans [40].

Définition 2.5. *[40] Soit Ω un ensemble de tâches d'une instance E-Réalisable de CuSP. La* marge *de l'ensemble de tâches Ω, notée SL_Ω, est définie par :*

$$SL_\Omega = C(d_\Omega - r_\Omega) - e_\Omega. \tag{2.35}$$

Définition 2.6. *Soient i et L deux tâches d'une instance E-Réalisable de CuSP avec $r_i < r_L$. On dit que la tâche $\delta(L, i)$ avec $d_{\delta(L,i)} \leq d_i$ définit la borne supérieure l'intervalle de tâches de marge minimale si pour toute tâche $U \in T$ telle que $d_U \leq d_i$,*

$$C(d_{\delta(L,i)} - r_L) - e_{\Omega_{L,\delta(L,i)}} \leq C(d_U - r_L) - e_{\Omega_{L,U}} \ . \tag{2.36}$$

L'Algorithme 4 est de complexité $\mathcal{O}(n^2)$ en temps. En effet :

- La boucle externe (ligne 3) parcourt l'ensemble des tâches $L \in T$ ($|T| = n$) et sélectionne les différentes bornes inférieures des intervalles de tâches.
- La boucle interne (ligne 5) parcourt l'ensemble des tâches ($|T| = n$) et sélectionne la tâche $i \in T$ des possibles bornes supérieures des intervalles de tâches dans l'ordre croissant des dates de fin au plus tard. Si $r_L \leq r_i$, alors l'énergie et la marge de l'intervalle de tâches $\Omega_{L,i}$ sont calculées. L'énergie est sauvegardée, la marge est comparée à celle de l'intervalle $\Omega_{L,\delta(L,i)}$ et si $SL_{\Omega_{L,i}} \leq SL_{\Omega_{L,\delta(L,i)}}$ alors $\delta(L, i)$ devient i. Si par contre $r_L > r_i$ et si les conditions des lignes 12 et 13 sont vérifiées, alors la valeur potentielle d'ajustement de la date de début au plus tôt de la tâche i est calculée avec la valeur courante de $\delta(L, i)$. Les conditions des lignes 12 et 13 garantissent que $rest = rest(\Omega_{L,\delta(L,i)}) > 0$ (ligne 14). Si ces conditions sont satisfaites, alors la date de début au plus tôt de la tâche i est ajustée à la ligne 16.
- A l'itération suivante de la seconde boucle externe, $\delta(L, i)$ est réinitialisée.

Avant de montrer que l'Algorithme 4 est correct au point fixe de l'edge-finding, montrons quelques propriétés de sa boucle interne (ligne 5).

Proposition 2.8. *Pour chaque tâche i, l'Algorithme 4 calcule la valeur d'ajustement upd de r_i basée sur l'intervalle de tâches de marge minimale telle que pour une tâche L donnée*

Algorithm 4: Algorithme d'extended edge-finding de complexité $\mathcal{O}(n^2)$ en temps

Entrées: T vecteur de tâches
Sorties: LB'_i pour toute tâche $i \in T$

1 **pour chaque** $i \in T$ **faire**
2 $\quad LB'_i := r_i$
3 **pour chaque** $L \in T$ **faire**
4 $\quad Energy := 0, maxEnergy := 0, d_\delta := r_L;$
5 $\quad$ **pour** $i \in T$ *trié dans l'ordre croissant des dates de fin au plus tard* **faire**
6 $\quad\quad$ **si** $r_L \leq r_i$ **alors**
7 $\quad\quad\quad Energy := Energy + e_i;$
8 $\quad\quad\quad MinSalck :=$ **si** $(maxEnergy \neq 0 \wedge d_\delta > r_L)$ **alors** $C(d_\delta - r_L) - maxEnergy$ **sinon** $+\infty;$
9 $\quad\quad\quad$ **si** $(C(d_i - r_L) - Energy \leq MinSlack)$ **alors**
10 $\quad\quad\quad\quad maxEnergy := Energy, d_\delta := d_i$;
11 $\quad\quad$ **sinon**
12 $\quad\quad\quad$ **si** $maxEnergy + c_i(r_i + p_i - r_L) > C(d_\delta - r_L)$ **alors**
13 $\quad\quad\quad\quad$ **si** $r_i + p_i < d_\delta \wedge r_L < r_i + p_i \wedge r_L < d_\delta$ **alors**
14 $\quad\quad\quad\quad\quad rest := maxEnergy - (C - c_i)(d_\delta - r_L);$
15 $\quad\quad\quad\quad\quad upd := r_L + \lceil rest/c_i \rceil;$
16 $\quad\quad\quad\quad\quad LB'_i := \max(LB'_i, upd);$
17 **pour chaque** $i \in T$ **faire**
18 $\quad r_i := LB'_i;$

avec $r_i < r_L$

$$upd = r_L + \left\lceil \text{rest}(\Omega_{L,\delta(L,i)}, c_i) \cdot \frac{1}{c_i} \right\rceil \tag{2.37}$$

si $e_{\Omega_{L,\delta(L,i)}} + c_i(r_i + p_i - r_L) > C(d_{\delta(L,i)} - r_L)$, $r_i + p_i < d_{\delta(L,i)}$, $r_L < d_{\delta(L,i)}$ *et* $r_L < r_i + p_i$.

Démonstration. Soit $i \in T$ une tâche. Le choix d'une tâche $L \in T$ à la boucle externe de la ligne 3 commence avec les valeurs $d_{\delta(L,i)} = r_L$ et $maxEnergy = 0$. La boucle interne de la ligne 5 parcourt l'ensemble des tâches $i' \in T$ (T trié dans l'ordre croissant des dates de fin au plus tard). Pour une tâche $i' \in T$ vérifiant $d_{i'} \leq d_i$, si $r_L \leq r_{i'}$, alors $i' \in \Omega_{L,i'}$, et $e_{i'}$ est ajoutée à $Energy$ (ligne 7), valeur courante de l'énergie de $\Omega_{L,i'}$. Donc, à chaque itération, nous avons $Energy = e_{\Omega_{L,i'}}$. Le test de la ligne 9 garantie que $d_{\delta(L,i)}$ et $maxEnergy = e_{\Omega_{L,\delta(L,i)}}$ reflètent les propriétés de la tâche $\delta(L,i)$ pour la valeur courante de l'intervalle de tâches $\Omega_{i',U}$. Ainsi, à la i^{ieme} itération de la boucle interne (ligne 5) si $r_i < r_L$ alors si les conditions des lignes 12 et 13 sont vérifiées, alors la ligne 14 calcule $rest_{L,i} = \text{rest}(\Omega_{L,\delta(L,i)}, c_i)$, et la valeur potentielle d'ajustement $r_L + \lceil \text{rest}_{L,i} \cdot \frac{1}{c_i} \rceil$ (ligne 15). Par conséquent, la proposition est correcte. □

Théorème 2.6 montre qu'au point fixe de l'edge-finding, les conditions d'extended edge-finding peuvent être vérifiées en utilisant l'intervalle de tâches de marge minimale tel qu'il est donné dans la Définition 2.6.

Théorème 2.6. *Au point fixe de l'edge-finding, pour chaque tâche $i \in T$ et pour tout ensemble de tâches $\Omega \subseteq T \setminus \{i\}$,*

$$r_i < r_\Omega < r_i + p_i \ \wedge \ e_\Omega + c_i(r_i + p_i - r_\Omega) > C(d_\Omega - r_\Omega) \tag{2.38}$$

si et seulement si

$$r_i < r_L < r_i + p_i \ \wedge \ e_{\Omega_{L,\delta(L,i)}} + c_i(r_i + p_i - r_L) > C(d_{\delta(L,i)} - r_L) \tag{2.39}$$

pour une tâche $L \in T$ vérifiant $r_i < r_L$, et $\delta(L,i)$ tel que spécifié dans la Définition 2.6.

Démonstration. Soit $i \in T$ une tâche. Nous montrons dans un premier temps que (2.38) implique (2.39). Supposons qu'il existe un ensemble de tâches $\Omega \subseteq T \setminus \{i\}$ vérifiant $r_i < r_\Omega < r_i + p_i$ et $e_\Omega + c_i(r_i + p_i - r_\Omega) > C(d_\Omega - r_\Omega)$. Alors $\Omega \lessdot i$ et d'après la Proposition 2.6, il existe un intervalle de tâches $\Omega_{L,U}$ vérifiant $\Omega_{L,U} \lessdot i$ et tel que $r_i \leq r_L < r_i + p_i$. Étant au point fixe de l'edge-finding, il est évident que $r_i < r_L < r_i + p_i$. D'après la Définition 2.6 nous avons

$$C(d_{\delta(L,i)} - r_L) - e_{\Omega_{L,\delta(L,i)}} \leq C(d_U - r_L) - e_{\Omega_{L,U}}. \tag{2.40}$$

En ajoutant $-c_i(r_i + p_i - r_L)$ à chaque membre de (2.40) et en utilisant le fait que

$$C(d_U - r_L) - e_{\Omega_{L,U}} - c_i(r_i + p_i - r_L) < 0 \tag{2.41}$$

il s'en suit que

$$C(d_{\delta(L,i)} - r_L) < e_{\Omega_{L,\delta(L,i)}} + c_i(r_i + p_i - r_L). \tag{2.42}$$

Nous pouvons alors montrer que (2.39) implique (2.38). Soient $L \in T$ une tâche telle que $r_i < r_L < r_i + p_i$, et $\delta(L,i) \in T$, la tâche spécifiée dans la Définition 2.6 et qui vérifie (2.39). Il est évident que (2.38) est vérifiée pour $\Omega = \Omega_{L,\delta(L,i)}$. □

A la Proposition 2.8, nous avons montré que $\delta(L,i)$ et la marge minimale sont correctement déterminés par l'Algorithme 4 à la boucle interne de la ligne 5. La combinaison ce résultat avec celui de la Proposition 2.7 et du Théorème 2.6, explique pourquoi l'Algorithme 4 détecte correctement les conditions de l'extended edge-finding aux lignes 12

et 13 avec l'intervalle de tâches de marge minimale et effectue toujours un ajustement lorsque celui ci est justifié par la règle. Ainsi, au point fixe de l'edge-finding, pour toute tâche i, l'Algorithme 4 identifie correctement l'ensemble de tâches $\Omega \subseteq T \setminus \{i\}$ pour lequel la relation $\Omega \lessdot i$ est vérifiée.

Comme pour l'algorithme d'edge-finding présenté à la section 2.1.2, ce nouvel algorithme n'effectue pas l'ajustement maximal à la première itération, mais s'améliore aux autres itérations. Cette approche naïve nous a permis d'améliorer la vitesse et la complexité de l'algorithme d'edge-finding. Dans le théorème qui suit, nous montrons qu'au point fixe de l'edge-finding, l'Algorithme 4 possède cette propriété i.e., détecte correctement les conditions de l'extended edge-finding et effectue toujours un ajustement lorsque celui est justifié par la règle.

Théorème 2.7. *Soit $i \in T$ une tâche et LB_i l'ajustement maximal de r_i tel qu'il est spécifié dans la Définition 1.14. Au point fixe de l'edge-finding, l'Algorithme 4 calcule une borne inférieure LB'_i de r_i qui est telle que $r_i < LB'_i \leq LB_i$ si $r_i < LB_i$, et $LB'_i = r_i$ si $r_i = LB_i$. La complexité de cet algorithme est de $\mathcal{O}(n^2)$ en temps.*

Démonstration. Soit $i \in T$ une tâche.
Les LB'_i sont initialisées à r_i dans la boucle de la ligne 1. Après chaque détection des conditions d'extended edge-finding (lignes 12 et 13), r_i est mise à jour avec $\max(upd_i, LB'_i)$ (ligne 16). Donc, on a toujours $LB'_i \geq r_i$. Si l'égalité $LB_i = r_i$ est satisfaite, alors aucune détection n'est effectuée par la règle et donc par l'Algorithme 4. Par conséquent, on a $LB'_i = r_i$ avec la boucle de la ligne 17. Dans la suite de preuve nous supposons que $r_i < LB_i$. Alors les conditions de la règle d'extended edge-finding sont détectées avec la paire (Ω, i) où Ω est un ensemble de tâches et la date de début au plus tôt de la tâche i est ajustée. D'après le Théorème 2.6, il existe deux tâches L et $\delta(L, i)$ telles que la relation $\Omega_{L,\delta(L,i)} \lessdot i$ est détectée par l'extended edge-finding. Par la Proposition 2.7, l'ensemble de tâches $\Omega_{L,\delta(L,i)}$ peut être utilisé pour ajuster r_i. Comme démontré dans la Proposition 2.8, la tâche $\delta(L, i)$ et l'intervalle de tâches de marge minimale sont correctement déterminés par l'Algorithme 4. Ainsi, après la détection des conditions de l'extended edge-finding aux lignes 12 et 13, la date de début au plus tôt de la tâche i est ajustée à $LB'_i = upd_i > r_i$.

La complexité de l'Algorithme 4 est de $\mathcal{O}(n^2)$ en temps et $\mathcal{O}(n)$ en espace. Le tri du

vecteur T se fait en $\mathcal{O}(n \log n)$ en temps. La boucle externe de la ligne 3 qui est parcourue $|T| = n$ fois contient une boucle interne (ligne 5) de complexité $\mathcal{O}(n)$ en temps, de sorte que la complexité en temps de l'algorithme est égale à $\mathcal{O}(n \log n) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$. □

2.2.4 Comparaison avec la conjonction des règles edge-finding et extended edge-finding

Au point fixe de l'edge-finding, le théorème 2.2 montre que l'Algorithme 4 effectue toujours un ajustement de r_i si cet ajustement est justifié par la règle d'extended edge-finding. Même si l'ajustement n'est pas maximal à la première itération, elle s'améliore aux itérations subséquentes. Le nombre d'ajustements étant fini, la combinaison de l'Algorithme 4 avec l'Algorithme 2 doit atteindre le même point fixe que la conjonction des règles edge-finding et extended edge-finding. C'est ce que nous montrons dans le théorème suivant tiré de [43].

Théorème 2.8. *Lorsque le point fixe de la combinaison de l'Algorithme 4 avec l'Algorithme 2 est atteint, alors aucune propagation n'est effectuée par les règles edge-finding et extended edge-finding (le point fixe des ces règles est atteint).*

Démonstration. Par l'absurde, supposons que le point fixe de la combinaison de l'Algorithme 2 d'edge-finding avec l'Algorithme 4 est atteint et que la règle d'edge-finding où l'extended edge-finding peut réduire la fenêtre de temps d'une tâche i.e. il existe une tâche i et deux ensembles de tâches Ω et Θ avec $\Theta \subseteq \Omega \subseteq T \setminus \{i\}$ tels que l'une des règles (EF) ou (EF1) ou (EEF) soit satisfaite et la formule (1.15) permet d'ajuster r_i en utilisant Θ. Nous allons montrer que soit l'Algorithme 2, soit l'Algorithme 4 peut réduire la fenêtre de temps de la tâche i, ce qui contredit que le point fixe de ces algorithmes est atteint. Nous distinguons deux cas :

1. **L'une des règles** (EF) **ou** (EF1) **est détectée et la fenêtre de temps de la tâche** i **est ajustée.** Comme l'algorithme 2 effectue toujours un ajustement lorsque celui ci est justifié par la règle, il vient que la fenêtre de temps de la tâche i sera réduite par cet algorithme. Ce qui contredit l'hypothèse suivant laquelle le point fixe de la combinaison de l'Algorithme 2 et l'Algorithme 4 est atteint.

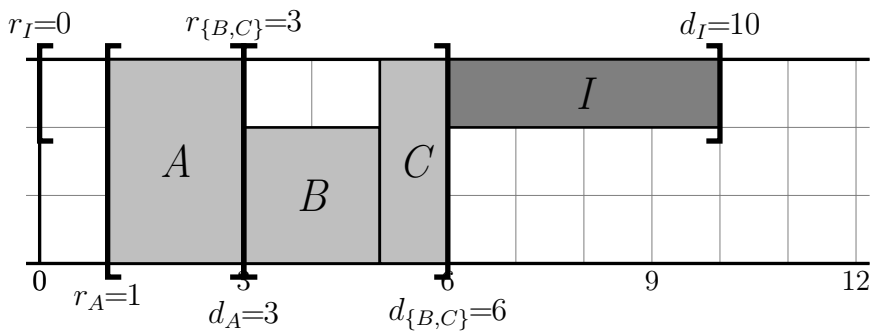


FIGURE 2.3 – Instance de CuSP de quatre tâches devant être exécutées sur une ressource de capacité $C = 3$.

2. **La** (EEF) **est détectée et la fenêtre de temps de la tâche** i **est ajustée.** Nous distinguons ici deux sous cas.

   (a) **Le point fixe des règles** (EF) **et** (EF1) **est atteint.** Comme au point fixe des règles (EF) et (EF1), l'algorithme 4 effectue toujours un ajustement lorsque celui ci est justifié par la règle, il vient que la fenêtre de temps de la tâche i sera réduite par cet algorithme. Ce qui contredit l'hypothèse que le point fixe de la combinaison de l'Algorithme 2 et l'Algorithme 4 est atteint.

   (b) **Le point fixe de** (EF) **ou** (EF1) **n'est pas atteint.** On revient au cas du premier item et la contradiction est détectée.

□

2.2.5 Comparaison avec le timetable edge finding [77]

Il est démontré dans [77], qu'au point fixe du timetable edge finding, la conjonction de l'edge-finding et de l'extended edge-finding ne filtre aucun domaine de tâches. Nous avons déjà démontré que ce résultat était incorrect avec l'edge-finding [41]. On se propose de montrer le résultat similaire avec l'extended edge-finding [43]. Pour cela, considérons l'instance de CuSP de la Figure 2.3 où quatre tâches doivent s'exécuter sur une ressource de capacité 3.

Une application de l'extended edge-finding à cette instance montre que pour $\Omega = \{A, B, C\}$, la relation $\Omega \lessdot I$ est détectée et la date de début au plus tôt de la tâche I est

ajustée de 0 à 4. Par contre, le timetable edge finding n'effectue aucun ajustement sur cette instance, preuve qu'il ne domine pas l'extended edge-finding.

Théorème 2.9. *La règle d'extended edge finding n'est pas dominée par la règle du timetable edge finding.*

Démonstration. Considérons l'instance de CuSP de la Figure 2.3 où quatre tâches s'exécutent sur une ressource de capacité 3. Nous avons montré précédemment qu'en utilisant la règle d'extended edge-finding, la date de début au plus tôt de la tâche I est ajustée de 0 à 4. Nous allons prouver qu'en appliquant la règle du timetable, aucun ajustement n'est produit (le timetable ne produit pas le même ajustement). Si nous voulons appliquer le timetable edge finding pour ajuster la date de début au plus tôt de la tâche I, alors nous devons considérer un ensemble de tâches $\Omega \subseteq T^{EF} \setminus \{I\}$. D'après les données de l'instance de CuSP de la Figure 2.3, nous avons $T^{EF} = \{I, B, C\}$. Par conséquent, aucun ensemble de tâches $\Omega \subseteq T^{EF} \setminus \{I\}$ ne permet de détecter le règle. Par conséquent, le timetable edge finding n'effectue aucun ajustement sur cette instance. Ce qui contredit les résultats de Vilím suivant lequel la règle d'extended edge-finding est dominée par la règle du timetable edge finding. Le même raisonnement est appliqué pour les versions améliorées de cette règle. □

2.3 Le not-first/not-last

2.3.1 Introduction

Le filtrage effectué par l'edge-finding et l'extended edge-finding est généralement complété par celui du not-first/not-last. Il s'agit d'une autre technique de propagation de la contrainte de ressource utilisée tant en ordonnancement disjonctif que cumulatif. Cette règle permet de déterminer la position relative d'une tâche par rapport à un ensemble de tâches toutes devant être exécutées sur une même ressource. Plus précisément, elle vérifie si une tâche i ne peut être exécutée en premier (resp. en dernier) relativement à l'ensemble de tâches $\Omega \cup \{i\}$ et comme conséquence, effectue une mise à jour de la date de début au plus tôt (resp. date de fin au plus tard) de la tâche i. Il existe des algorithmes efficaces pour cette règle dans le cas disjonctif $\mathcal{O}(n \log n)$ où n désigne le nombre de tâches utilisant la

ressource [74]. La règle est plus complexe dans le cas cumulatif. Schutt et al [64] montrent que l'implémentation de Nuijten [60] de complexité $\mathcal{O}(|Sc|n^3)$ (où Sc désigne l'ensemble des différentes consommations des tâches) est incorrecte et incomplète. Ils proposent alors un algorithme complet de complexité $\mathcal{O}\left(n^3 \log n\right)$ en temps. Dans [37], nous proposons un algorithme itératif de not-fist/not-last de complexité $\mathcal{O}\left(n^3\right)$ qui atteint le même point fixe que l'algorithme complet après au plus n itérations. Tout récemment, Schutt et Wolf [65] et nous autres indépendamment [38, 42] avons proposé un algorithme de not-first/not-last incomplet de complexité $\mathcal{O}(n^2 \log n)$ en temps. Ces algorithmes n'effectuent pas nécessairement l'ajustement maximal à la première itération, mais s'améliorent aux itérations subséquentes.

Dans cette section, nous proposons deux algorithmes de not-first/not-last cumulatif de complexité $\mathcal{O}(n^2|H| \log n)$ et $\mathcal{O}(n^2 \log n)$ respectivement. Le premier est complet, le second est itératif et effectue l'ajustement maximal après au plus $|H|$ itérations. En somme, les deux algorithmes ont la même complexité pour l'ajustement maximal mais dans la pratique, la version itérative est plus rapide que l'algorithme complet. Les deux algorithmes utilisent la structure de données cumulative Θ-tree [76].

2.3.2 Algorithme complet de not-first en $\mathcal{O}(n^2|H| \log n)$

Un algorithme complet de not-first recherche le meilleur ajustement possible LB_i pour chaque tâche i tel qu'il est spécifié dans la Définition 1.17. Par contre, un algorithme incomplet tel qu'il est présenté dans [37, 65, 40] recherche un ajustement LB'_i qui vérifie $r_i < LB'_i \leq LB_i$ si $r_i < LB_i$ et $LB'_i = r_i$ sinon. Dans ce paragraphe, nous décrivons un algorithme complet de not-first de complexité $\mathcal{O}(n^2|H| \log n)$ en temps et $\mathcal{O}(n)$ en espace. Il est basé sur une adaptation du concept de coupe gauche de l'ensemble de tâches T introduit par Vilím dans [74] et de la notion d'enveloppe d'énergie introduit par le même auteur dans [76].

Nous décrivons dans un premier temps comment les notions de coupe gauche et d'enveloppe d'énergie peuvent être utilisées pour vérifier la condition de la règle not-first. Nous utilisons la structure de données Θ-arbre cumulatif (cumulative Θ-tree en anglais) pour un calcul efficace de l'enveloppe d'énergie des ensembles de tâches. Enfin, nous présentons notre nouvel algorithme de not-first, montrons qu'il est complet et analysons sa

complexité.

Définition 2.7 donne une nouvelle définition de la coupe de T par une tâche, liée à la règle de not-first.

Définition 2.7. *[39] Soient j et i deux tâches d'une instance E-Réalisable de CuSP de T tâches telles que $i \neq j$. Soit $Ect \in H$ une date de fin au plus tôt d'une tâche. La coupe gauche de T par la tâche j relativement à la tâche i et à la date de fin au plus tôt Ect est l'ensemble noté $LCut(T, j, Ect, i)$ et défini comme suit :*

$$LCut(T, j, Ect, i) := \{k,\ k \in T\ \wedge\ k \neq i\ \wedge\ Ect \leq r_k \wedge r_i < Ect\ \wedge\ d_k \leq d_j\}. \quad (2.43)$$

Définition 2.8. *Soient Θ un ensemble de tâches et i une tâche d'une instance E-Réalisable de CuSP. L'enveloppe d'énergie de Θ relativement à la tâche i noté $Env(\Theta, i)$ est le réel défini comme suit :*

$$Env(\Theta, i) := \max_{\Omega \subseteq \Theta}\{(C - c_i)r_\Omega + e_\Omega\}. \quad (2.44)$$

Une tâche $i \in T$ et un ensemble de tâches $\Omega \subseteq T \setminus \{i\}$ satisfont la condition de la règle de not-first si $r_i < ECT_\Omega$ et $e_\Omega + c_i(\min(ect_i, d_\Omega) - r_\Omega) > C(d_\Omega - r_\Omega)$. Nous montrons dans le théorème suivant que les conditions de la règle not-first peuvent être détectées en utilisant l'inégalité suivante pour toute paire (i, j) de tâches avec $i \neq j$ et la donnée d'une date de fin au plus tôt $Ect \in H$:

$$Env(LCut(T, j, Ect, i), i) > Cd_j - c_i \min(ect_i, d_j). \quad (2.45)$$

Théorème 2.10. *Soit $i \in T$ une tâche d'une instance E-Réalisable de CuSP. Il existe un ensemble de tâches $\Omega \subseteq T \setminus \{i\}$ satisfaisant les conditions de la règle not-first si et seulement s'ils existent une tâche $j \in T \setminus \{i\}$ et une date de fin au plus tôt $Ect \in H$ telle que l'inégalité (2.45) soit satisfaite pour les tâches i et j et la date de fin au plus tôt Ect.*

Démonstration. Soit $i \in T$ une tâche d'une instance E-Réalisable de CuSP. Nous montrons chaque implication de l'équivalence.

$\Rightarrow$: Supposons qu'il existe un ensemble de tâches $\Omega \subseteq T \setminus \{i\}$ tel que les conditions de la règle not-first soient satisfaites pour Ω et i, i.e. $r_i < ECT_\Omega$ et $e_\Omega + c_i(\min(ect_i, d_\Omega) - r_\Omega) > C(d_\Omega - r_\Omega)$. Il faut montrer qu'il existe une tâche $j \in T \setminus \{i\}$ et une date de fin au plus tôt $Ect \in H$ qui vérifie la condition $Env(LCut(T, j, Ect, i), i) > Cd_j - c_i \min(ect_i, d_j)$.

Soit $j \in \Omega$ une tâche satisfaisant la condition $d_j = d_\Omega$ (une telle tâche existe toujours). Parce que $r_i < ECT_\Omega$ et $i \notin \Omega$, il est évident que $\Omega \subseteq LCut(T, j, ECT_\Omega, i)$. L'inégalité $e_\Omega + c_i(\min(ect_i, d_\Omega) - r_\Omega) > C(d_\Omega - r_\Omega)$ est vérifiée car la condition $\gamma(\Omega, i)$ est satisfaite. Cette inégalité est équivalente à

$$(C - c_i)r_\Omega + e_\Omega > Cd_j - c_i \min(ect_i, d_j). \tag{2.46}$$

D'après la Définition 2.8 et l'inclusion $\Omega \subseteq LCut(T, j, ECT_\Omega, i)$ il vient que

$$Env(LCut(T, j, ECT_\Omega, i), i) \geq (C - c_i)r_\Omega + e_\Omega > Cd_j - c_i \min(ect_i, d_j). \tag{2.47}$$

D'où l'inégalité (2.45) est vérifiée pour la tâche j et la date de fin au plus tôt ECT_Ω.

$\Leftarrow$: Soient $j \in T \setminus \{i\}$ une tâche et $Ect \in H$ une date de fin au plus tôt vérifiant l'inégalité (2.45). Nous allons montrer l'existence d'un ensemble de tâches $\Omega \subseteq T \setminus \{i\}$ vérifiant les conditions de la règle not-first. Soit $\Omega' \subseteq LCut(T, j, Ect, i)$ l'ensemble de tâches qui détermine $Env(LCut(T, j, Ect, i), i)$, i.e. $Env(LCut(T, j, Ect, i), i) = (C - c_i)r_{\Omega'} + e_{\Omega'}$. De l'inégalité $d_{\Omega'} \leq d_j$ il vient que

$$Cd_j - c_i \min(ect_i, d_j) \geq Cd_{\Omega'} - c_i \min(ect_i, d_{\Omega'}). \tag{2.48}$$

En effet,

- Si $ect_i \leq d_{\Omega'}$ alors

$$\begin{aligned} Cd_j - c_i \min(ect_i, d_j) &= Cd_j - c_i \cdot ect_i \\ &\geq Cd_{\Omega'} - c_i \cdot ect_i \\ &= Cd_{\Omega'} - c_i \min(ect_i, d_{\Omega'}) \end{aligned}$$

- Si $ect_i > d_{\Omega'}$ alors

$$\begin{aligned} Cd_j - c_i \min(ect_i, d_j) &\geq (C - c_i)d_j \\ &\geq (C - c_i)d_{\Omega'} \\ &= Cd_{\Omega'} - c_i \min(ect_i, d_{\Omega'}) \end{aligned}$$

De l'inégalité (2.48), il s'en suit que

$$\begin{aligned} Env(LCut(T, j, Ect, i), i) &= (C - c_i)r_{\Omega'} + e_{\Omega'} \\ &> Cd_j - c_i \min(ect_i, d_j) \\ &\geq Cd_{\Omega'} - c_i \min(ect_i, d_{\Omega'}) \end{aligned}$$

et les conditions du not-first sont vérifiées pour Ω' et i. $\square$

Soient $i \in T$ une tâche et $Ect \in H$ une date de fin au plus tôt. Dans toute la suite, nous supposerons que l'ensemble $T \setminus \{i\} = \{j_1, j_2, ..., j_{n-1}\}$ des tâches est trié dans l'ordre croissant des dates de fin au plus tard i.e. $d_{j_1} \leq d_{j_2} \leq \cdots \leq d_{j_{n-1}}$. Ainsi, on a les inclusions

$$LCut(T, j_1, Ect, i) \subseteq LCut(T, j_2, Ect, i) \subseteq \cdots \subseteq LCut(T, j_{n-1}, Ect, i).$$

Avec cet ordre sur les tâches, l'ensemble $LCut(T, j_{l+1}, Ect, i)$ peut être rapidement déterminé à partir de $LCut(T, j_l, Ect, i)$. Après avoir trié l'ensemble H dans l'ordre croissant des Ect, nous parcourons tous les ensembles $LCut(T, j, Ect, i)$ et le premier ensemble pour lequel les conditions du not-first sont satisfaites nous permet d'avoir l'ajustement maximal. Pour détecter les conditions du not-first comme il est spécifié dans la formule (2.45), nous avons besoin de calculer l'enveloppe d'énergie de chaque ensemble $LCut(T, j, Ect, i)$. Nous utilisons pour cela la structure de donnée "cumulative Θ-tree" introduite par Vilím [76]. L'algorithme 5 que nous proposons organise les ensembles $LCut(T, j, Ect, i)$ en arbre binaire équilibré pour calculer rapidement ces valeurs. Ainsi, les opérations classiques (ajout et suppression de nœuds dans l'arbre) sont faites en $\mathcal{O}(\log n)$ et on peut intégrer le calcul de l'enveloppe d'énergie sans changer cette complexité. Une présentation plus détaillée de cette structure de donnée est faite plus bas.

Dans l'Algorithme 5, les ensembles $LCut(T, j, Ect, i)$ sont déterminés dans les boucles des lignes 3 et 4. L'arbre binaire devant organiser les ensembles $LCut(T, j, Ect, i)$ est initialisé à la ligne 6. La boucle interne de la ligne 7 parcourt l'ensemble des tâches dans l'ordre croissant des dates de fin au plus tard. Ainsi, les différents ensembles $LCut(T, j, Ect, i)$ sont construits de proche en proche en commençant par la plus grande date de fin au plus tôt. Les breaks des lignes 12 et 14 permettent de vite recommencer la détection et l'ajustement d'une nouvelle tâche. En effet, étant donné deux tâches j et j' avec $d_j \leq d_{j'}$, on a l'inclusion $LCut(T, j, Ect, i) \subseteq LCut(T, j', Ect, i)$. Si les conditions de not-first sont satisfaites pour ces deux ensembles, alors le même ajustement sera effectué. Lorsqu'une tâche $i \in T$ est ajustée à $Ect \in H$, alors la prochaine date de fin au plus tôt $Ect' \in H$ nous conduira à un ajustement plus faible car l'ensemble H est parcouru dans l'ordre décroissant de ses valeurs (ligne 4).

Algorithm 5: Algorithme de Not-first de Complexité $\mathcal{O}(n^2|H|\log n)$ en temps

Entrées: T ensemble de tâches
Données: Θ : representant les ensembles $LCut(T, j, Ect, i)$ équilibré suivant les r_l
Sorties: LB'_i nouvelle borne inférieure de la date de début au plus tôt des tâches

1 **pour chaque** $i \in T$ **faire**
2 $\quad LB'_i := r_i$;
3 **pour chaque** $i \in T$ **faire**
4 $\quad$ **pour chaque** $Ect \in H$ // *H trié dans l'ordre décroissant des Ect* **faire**
5 $\quad\quad$ **si** $r_i < Ect$ **alors**
6 $\quad\quad\quad \Theta := \emptyset$;
7 $\quad\quad\quad$ **pour chaque** $j \in T$ // *T trié dans l'ordre croissant des dates de fin au plus tard* **faire**
8 $\quad\quad\quad\quad$ **si** $Ect \leq r_j \wedge j \neq i$ **alors**
9 $\quad\quad\quad\quad\quad \Theta := \Theta \cup \{j\}$;
10 $\quad\quad\quad\quad\quad$ **si** $Env(\Theta, i) > Cd_j - c_i \min(ect_i, d_j)$ **alors**
11 $\quad\quad\quad\quad\quad\quad LB'_i := \max(LB'_i, Ect)$;
12 $\quad\quad\quad\quad\quad\quad$ **break**;
13 $\quad\quad$ **si** $LB'[i] == Ect$ **alors**
14 $\quad\quad\quad$ **break** ;
15 **pour** $i \in T$ **faire**
16 $\quad r_i := LB'_i$;

Afin de réduire la complexité de l'algorithme et calculer rapidement l'enveloppe d'énergie de $\Theta = LCut(T, j, Ect, i)$, nous organisons l'ensemble $LCut(T, j, Ect, i)$ en arbre binaire équilibré appelé Θ-tree [76]. Les tâches sont représentées par les feuilles et rangées de la gauche vers la droite suivant les valeurs croissantes de r_l. Chaque nœud v de l'arbre possède les valeurs suivantes :

$$e_v = e_{Leaves(v)} \tag{2.49}$$

$$Env_v(i) = Env(Leaves(v), i) \tag{2.50}$$

où $Leaves(v)$ représente l'ensemble des tâches du sous arbre de racine v. Pour une feuille représentant une tâche $k \in T$, les valeurs dans l'arbre sont :

$$e_v = e_k$$

$$Env_v(i) = Env(\{k\}, i) = (C - c_i)r_k + e_k$$

Pour un nœud interne v, ces valeurs peuvent être calculées de manière recursive à partir de ses nœuds fils $left(v)$ et $right(v)$ tel qu'il est spécifié dans la Proposition 2.9.

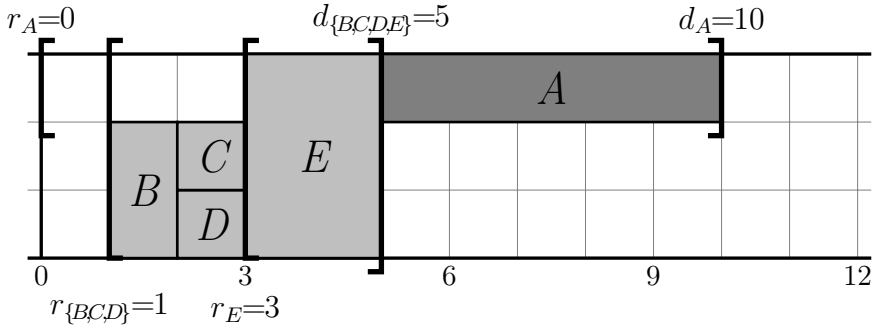


FIGURE 2.4 – Instance de CuSP de 5 tâches devant être exécutées sur une ressource de capacité $C = 3$.

Proposition 2.9. *Pour une tâche i donnée, pour un nœud interne v, les valeurs e_v et $Env_v(i)$ peuvent être calculées par les formules de récurrence suivantes :*

$$e_v = e_{left(v)} + e_{right(v)} \tag{2.51}$$

$$Env_v(i) = \max\{Env_{left(v)}(i) + e_{right(v)}, Env_{right(v)}(i)\} \tag{2.52}$$

Démonstration. Similaire à la preuve de la Proposition 2 dans [76]. Il suffit de remplacer C dans la preuve du second item par $C - c_i$. □

Exemple 2.1. *La Figure 2.4 contient les données d'une instance de CuSP de 5 tâches devant être exécutées sur une ressource de capacité $C = 3$.*

Pour une tâche $i \in T$ donnée et la donnée d'une date de fin au plus tôt $Ect \in H$, les formules (2.51) et (2.52) sont intégrées dans les opérations élémentaires (ajout et suppression d'éléments dans l'arbre) et utilisées pour calculer e_v et $Env_v(i)$ sans changer sa complexité connue être en $\log n$ [76].

Avant de montrer que l'Algorithme 5 est correct, complet et est de complexité $\mathcal{O}(n^2|H|\log n)$, montrons quelques propriétés de la boucle interne de la ligne 7. Soient $i \in T$ une tâche et $Ect \in H$ une date de fin au plus tôt (H trié dans l'ordre décroissant de ses valeurs). Posons $\Theta_j := LCut(T, j, Ect, i)$ avec $j \in T = \{1, 2, ..., n\}$ (T trié dans l'ordre croissant des dates de fin au plus tard) et $\Theta_0 := \emptyset$.

Proposition 2.10. *Soient $i \in T$ une tâche et $Ect \in H$ une date de fin au plus tôt. Dans l'Algorithme 5, nous avons avant la j^{ieme} itération de la boucle interne de la ligne 7 :*

$$\Theta_{j-1} = \Theta \quad et \tag{2.53}$$

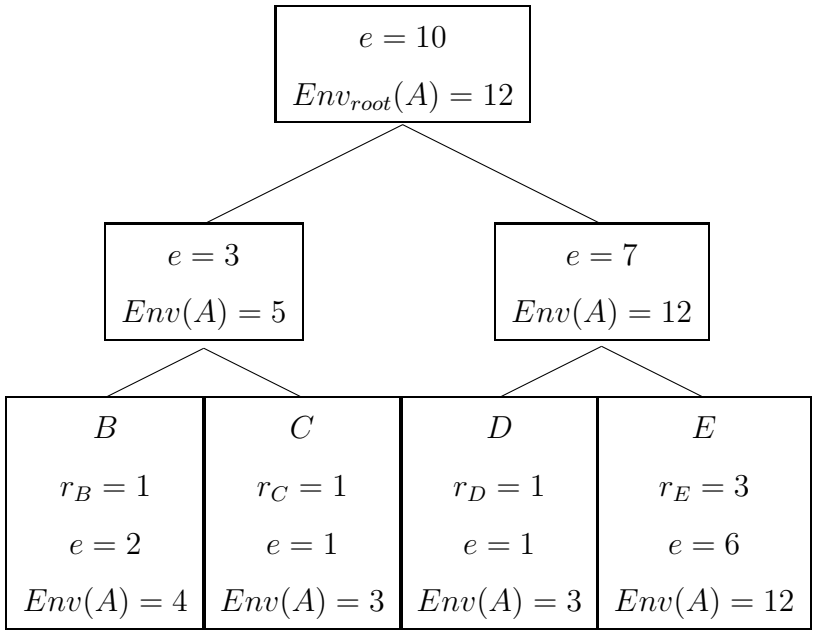


FIGURE 2.5 – Calcule de $Env(\Theta, A)$ avec $\Theta = LCut(T, B, 2, A) = \{A, B, C, D\}$. Les valeurs à la racine sont $e_\Theta = 10$ et $Env_{root} = 12$. Il est trivial que les condition du not-first sont vérifiées car $Env_{root} = 12 > Cd_B - c_A \min(ect_A, d_B) = 10$.

$$Env(\Theta_{j-1}, i) = Env(\Theta, i) \tag{2.54}$$

Démonstration. Par récurrence sur le vecteur $T = \{1, 2, ..., n\}$ où les tâches sont triées dans l'ordre croissant des dates de fin au plus tard.
Cas de base $j = 1$: Étant donné une tâche $i \in T$ et une date de fin au plus tôt $Ect \in H$, aucune itération de la boucle interne de la ligne 7 n'est effectuée. Alors, $\Theta = \emptyset$ et aucune enveloppe d'énergie n'est calculée. Parce que $\Theta_{1-1} = \Theta_0 = \emptyset$, le cas de base est vérifié.
Étape de récurrence $j \to j + 1$: L'hypothèse de récurrence est : pour tout $j' < j$ avant la j'^{ieme} itération de la boucle de la ligne 7, les relations (2.53) et (2.54) sont vérifiées. Le vecteur T étant trié dans l'ordre croissant des dates de fin au plus tard, nous avons $\Theta_{j-1} = \Theta_j \setminus \{j\}$. Pour montrer l'étape de récurrence, nous distinguerons deux cas, à savoir $j \in \Theta_j$ et $j \notin \Theta_j$.

Dans le premier cas, considérons $j \in \Theta_j$. Ici les conditions de la ligne 8 sont satisfaites à la j^{ieme} itération de la boucle. D'après l'hypothèse de récurrence on a $\Theta = \Theta_{j-1}$. L'ensemble Θ est alors mis à jour à $\Theta \cup \{j\}$ (ligne 9). Par la suite, les conditions (2.53) et (2.54) sont satisfaites.

Le second cas est $j \notin \Theta_j$. Ici les conditions de la ligne 8 ne sont pas vérifiées. Parce que

la tâche j n'est pas dans l'arbre, il vient d'après l'hypothèse de récurrence que $\Theta_j = \Theta_{j-1}$ et les conditions (2.53) et (2.54) sont satisfaites. □

Théorème 2.11. *L'Algorithme 5 est correct, complet et s'exécute en $\mathcal{O}(n^2|H|\log n)$ en temps. Après son exécution, nous avons : $LB'_i = LB_i = \max\left(r_i, \max_{\Omega \subseteq T\setminus\{i\} | \gamma(\Omega,i)} ECT_\Omega\right)$ avec $\gamma(\Omega, i) \stackrel{def}{=} (r_i < ECT_\Omega) \wedge (e_\Omega + c_i(\min(ect_i, d_\Omega) - r_\Omega) > C(d_\Omega - r_\Omega))$.*

Démonstration. Nous montrons dans un premier temps que l'Algorithme 5 est correct et complet, puis nous étudions sa complexité.

Complétude : L'Algorithme 5 est correct et complet si nous avons pour chaque tâche $i \in T$:

$$LB'_i = LB_i = \max\left(r_i, \max_{\Omega \subseteq T\setminus\{i\} | \gamma(\Omega,i)} ECT_\Omega\right)$$

avec $\gamma(\Omega, i) \stackrel{def}{=} (r_i < ECT_\Omega) \wedge (e_\Omega + c_i(\min(ect_i, d_\Omega) - r_\Omega) > C(d_\Omega - r_\Omega))$.

Soit $i \in T$ une tâche. Il faut montrer que $LB'_i = LB_i$. Si $LB_i = r_i$, alors nous avons $LB'_i \geq r_i$ à cause de la ligne 11. Sans perdre la généralité, supposons $r_i < LB_i$. Alors il existe un ensemble de tâches $\Omega \subseteq T \setminus \{i\}$ vérifiant :

$$LB_i = ECT_\Omega \wedge e_\Omega + c_i(\min(ect_i, d_\Omega) - r_\Omega) > C(d_\Omega - r_\Omega). \tag{2.55}$$

Soit $j \in \Omega$ une tâche vérifiant $r_j = r_\Omega$ et $Ect := ECT_\Omega \in H$ la date de fin au plus tôt de l'ensemble Ω. Nous avons $\Omega \subseteq \Theta_j$, et par conséquent la condition (2.55) est satisfaite pour l'ensemble Θ_j car $Env(\Theta_j, i) \geq (C - c_i)r_\Omega + e_\Omega > Cd_j - c_i \min(ect_i, d_j)$. D'après la Proposition 2.10, l'Algorithme 5 détecte les conditions du not-first (ligne 10 11) (confère Théorème 2.11) et ajuste la date de début au plus tôt de la tâche r_i à $LB'_i = Ect = ECT_\Omega = LB_i$. En d'autres termes, l'Algorithme 5 est correct et complet.

Complexité : Les boucles des lignes 1 et 15 sont de complexité $\mathcal{O}(n)$ en temps. La seconde boucle externe (ligne 3 de complexité $\mathcal{O}(n)$) contient une boucle interne (ligne 4 de complexité $\mathcal{O}(|H|)$) qui contient à son tour une boucle (ligne 7) de complexité $\mathcal{O}(n)$. Dans cette dernière boucle, la complexité de la ligne 9 est de $\mathcal{O}(\log n)$ en temps. La complexité de la boucle externe (ligne 3) est alors $\mathcal{O}(n^2|H|\log n)$ en temps. Et la complexité totale de l'algorithme est alors $\mathcal{O}(n) + \mathcal{O}(n^2|H|\log n) + \mathcal{O}(n) = \mathcal{O}(n^2|H|\log n)$. □

2.3.3 Algorithme itératif de not-first en $\mathcal{O}(n^2 \log n)$

Dans cette section, nous décrivons un algorithme itératif de not-first cumulatif de complexité $\mathcal{O}(n^2 \log(n))$. C'est un algorithme similaire à celui proposé pour l'edge-finding. Il n'effectue pas nécessairement l'ajustement maximal à la première itération, mais s'améliore aux itérations subséquentes. Il atteint le même point fixe que les autres algorithmes de not-first [64, 39, 38, 37, 65]. Cet algorithme a été décrit dans [38] et étendu dans [42].

Comme pour l'Algorithme 5, nous utilisons une relaxation de la notion de coupe gauche que nous appelons ici pseudo coupe gauche. Dans [39] et [38, 42] la même notation est utilisée pour les différentes versions de la coupe gauche. C'est le nombre de paramètres qui varie. Pour des raisons de clarté nous noterons différemment la version relaxée.

Définition 2.9. *Soient j et i deux tâches avec $i \neq j$. On appelle pseudo coupe gauche de T par la tâche j relativement à la tâche i l'ensemble de tâches noté $SLCut(T, j, i)$ et défini comme suit :*

$$SLCut(T, j, i) := \{k,\ k \in T\ \wedge\ k \neq i\ \wedge\ r_i < ect_k\ \wedge\ d_k \leq d_j\}. \tag{2.56}$$

Comme dans le cas de l'Algorithme 5, nous utilisons la notion d'enveloppe d'énergie pour détecter les conditions du not-first. Soient $i \in T$ une tâche et $\Omega \subseteq T \setminus \{i\}$ un ensemble de tâches vérifiant les conditions du not-first i.e. $r_i < ECT_\Omega$ et $e_\Omega + c_i(\min(ect_i, d_\Omega) - r_\Omega) > C(d_\Omega - r_\Omega)$. Le Théorème 2.12 montre que les conditions du not-first sont satisfaites si et seulement si il existe une tâche $j \in T$ avec $i \neq j$ vérifiant

$$Env(SLCut(T, j, i), i) > Cd_j - c_i \min(ect_i, d_j). \tag{2.57}$$

Théorème 2.12. *Soit $i \in T$ une tâche d'une instance de CuSP. Il existe un ensemble de tâches $\Omega \subseteq T \setminus \{i\}$ vérifiant les conditions du not-first si et seulement si il existe une tâche $j \in T \setminus \{i\}$ pour laquelle l'inégalité* (2.57) *est satisfaite pour les tâches i et j.*

Démonstration. Similaire à la preuve du Théorème 2.10. □

Comme dans [38, 39], nous supposons que l'ensemble des tâches T est trié dans l'ordre croissant des dates de fin au plus tard. Avec ce tri des tâches, les ensembles $SLCut(T, j, i)$ sont rapidement déterminés à partir des ensembles précédents i.e., si $T = \{j_1, j_2, ..., j_n\}$

alors $SLCut(T, j_1, i) \subseteq SLCut(T, j_2, i) \subseteq ... \subseteq SLCut(T, j_n, i)$. Le principe de l'Algorithme 6 est similaire à celui de l'Algorithme 5 à la différence que l'on supprime la boucle sur l'ensemble H des dates de fin au plus tôt et on détermine la date de fin au plus tôt du plus petit ensemble des tâches de $SLCut(T, j_{min}, i)$ pour lequel la condition $Env(SLCut(T, j_{min}, i), i) > Cd_{j_{min}} - c_i \min(ect_i, d_{j_{min}})$ est satisfaite. Si cette condition est vérifiée, alors la date de début au plus tôt r_i est ajustée à $r_i = ECT_{SLCut(T,j_{min},i)}$.

Exemple 2.2. *Considérons l'instance de CuSP de la Figure 1.7 avec $T = \{B, C, A, I\}$. Les conditions de not-first sont satisfaites avec l'ensemble $SLCut(T, A, I) = \{B, C, A\}$. La date de début au plus tôt de la tâche I est alors ajustée à $ECT_{SLCut(T,A,I)} = 3$ qui n'est pas le meilleur ajustement possible comme il a été montré dans l'Exemple 1.8.*

Comme dans l'Algorithme 5, pour réduire la complexité de l'algorithme, les ensembles $SLCut(T, j, i)$ sont organisés en arbres binaires équilibrés appelés Θ-tree. Les tâches représentent les feuilles et sont rangées de la gauche vers la droite dans l'ordre croissant des r_l. Chaque nœud v de l'arbre possède les valeurs suivantes :

$$e_v = e_{Leaves(v)} \tag{2.58}$$

$$Env_v(i) = Env(Leaves(v), i) \tag{2.59}$$

où $Leaves(v)$ représente l'ensemble des tâche du sous arbre de racine v. Pour une feuille représentant une tâche $k \in T$, les valeurs dans l'arbre sont :

$$e_v = e_k$$

$$Env_v(i) = Env(\{k\}, i) = (C - c_i)r_k + e_k$$

Pour un nœud interne v, Ces valeurs peuvent être calculées de manière recursive à partir de ses nœuds fils $left(v)$ et $right(v)$ tel qu'il est spécifié dans la Proposition 2.11.

Proposition 2.11. *Pour une tâche i donnée, pour un nœud interne v, les valeurs e_v et $Env_v(i)$ peuvent être calculées par les formules de récurrence suivantes :*

$$e_v = e_{left(v)} + e_{right(v)} \tag{2.60}$$

$$Env_v(i) = \max\{Env_{left(v)}(i) + e_{right(v)}, Env_{right(v)}(i)\} \tag{2.61}$$

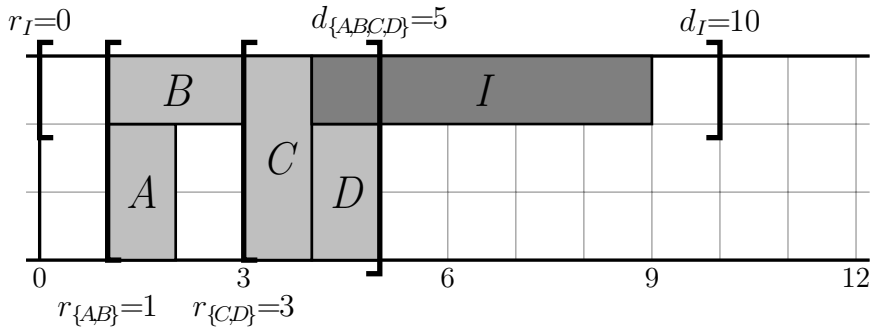


FIGURE 2.6 – Instance de CuSP de 5 tâches devant être exécutées sur une ressource de capacité $C = 3$.

Algorithm 6: Algorithme de Not-first en $\mathcal{O}(n^2 \log n)$ en temps et $\mathcal{O}(n)$ en espace

Entrées: T ensemble de tâches
Entrées: Θ : arbre binaire équilibré suivant les r_l représentant les $SLCut(T, j, i)$
Sorties: LB'_i borne inférieure des dates de début au plus tôt pour chaque tâche i

1 **for** $i \in T$ **do**
2 $\quad LB'_i := r_i$;
3 **pour** $i \in T$ **faire**
4 $\quad \Theta := \emptyset, \quad minEct := +\infty$;
5 $\quad$ **pour** $j \in T$ // *T trié dans l'ordre croissant des dates de fin au plus tard* **faire**
6 $\quad\quad$ **si** $r_i < ect_j \wedge j \neq i$ **alors**
7 $\quad\quad\quad \Theta := \Theta \cup \{j\}$;
8 $\quad\quad\quad minEct := \min(minEct, ect_j)$;
9 $\quad\quad\quad$ **si** $Env(\Theta, c_i) > Cd_j - c_i \min(ect_i, d_j)$ **alors**
10 $\quad\quad\quad\quad LB'_i := \max(LB'_i, minEct)$;
11 $\quad\quad\quad\quad$ **break**;
12 **pour** $i \in T$ **faire**
13 $\quad r_i := LB'_i$;

Démonstration. Similaire à la preuve de la Proposition 2.9. □

Dans l'Algorithme 6, la boucle externe (ligne 3) parcourt l'ensemble des tâches $i \in T$ formé des possibles tâches non premières. La boucle interne (ligne 5) sélectionne la tâche $j \in T$ potentielle borne supérieure de la pseudo coupe gauche $SLCut(T, j, i)$ dans l'ordre croissant des dates de fin au plus tard. Si $r_i < ect_j$ et $j \neq i$ alors la tâche j est ajoutée à l'arbre $\Theta = SLCut(T, j, i)$ (ligne 7) et la date de fin au plus tôt de l'ensemble Θ est alors mise à jour pour refléter ECT_Θ (ligne 8). Si la condition du not-first (2.45) est satisfaite à la ligne 9, alors la date de début au plus tôt de la tâche i est ajustée à ECT_Θ (ligne 10). Le "break" de la ligne 11 nous permet de recommencer rapidement la détection

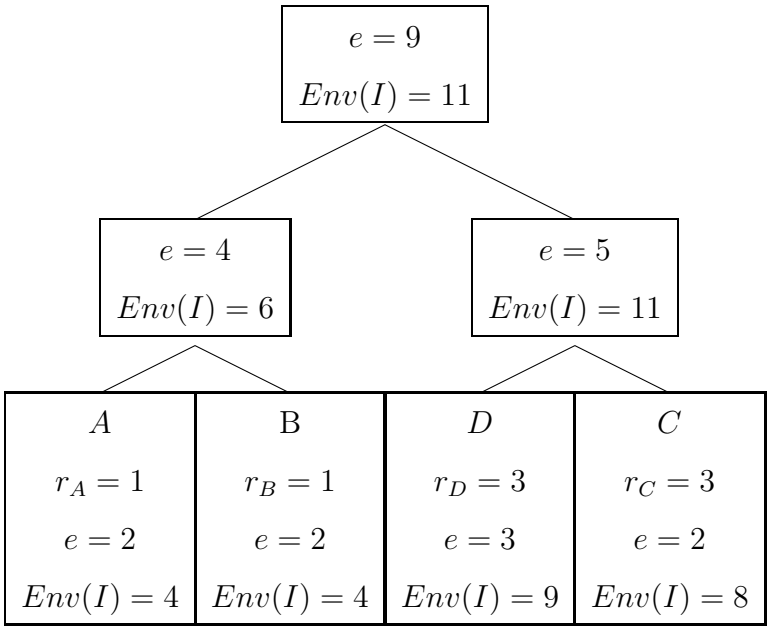


FIGURE 2.7 – Les valeurs de e_Θ et $Env(\Theta, I)$ pour $\Theta = SLCut(T, A, I) = \{A, B, C, D\}$ sont obtenues à la racine de l'arbre $e_\Theta = 9$ et $Env_{root} = 11$. Il est évident que les conditions de not-first sont satisfaites car $Env_{root} = 11 > Cd_A - c_I \min(ect_I, d_A) = 10$.

et l'ajustement d'une nouvelle tâche. A l'itération suivante de la boucle externe, Θ est réinitialisé.

Avant de montrer que l'Algorithme 6 effectue toujours un ajustement lorsque celui est justifié par la règle not-first, montrons quelques propriétés de la boucle interne de la ligne 5. Soit $i \in T$ une tâche. Posons $\Psi_j := SLCut(T, j, i)$ avec $j \in T$ (T trié dans l'ordre croissant des dates de fin au plus tard) et $\Psi_0 := \emptyset$.

Proposition 2.12. *Soit $i \in T$ une tâche. Dans l'Algorithme 6, avant la j^{ieme} itération de la boucle interne de la ligne 5, nous avons :*

$$\Theta_{j-1} = \Theta \tag{2.62}$$

$$ECT_{\Theta_{j-1}} = minEct \quad et \tag{2.63}$$

$$Env(\Theta_{j-1}, i) = Env(\Theta, i) \tag{2.64}$$

Démonstration. Similaire à la preuve de la Proposition 2.10. □

La Proposition 2.12 montre que $Env(SLCut(T, j, i), i)$ et $ECT_{SLCut(T,j,i)}$ sont correctement calculés par l'Algorithme 6. Combiné au Théorème 2.12, ceci justifie le fait que

pour toute tâche i, l'Algorithme 6 détecte correctement l'ensemble $\Omega \subseteq T \setminus \{i\}$ pour lequel la règle not-first est vérifiée.

Dans le théorème suivant, nous montrons que l'Algorithme 6 possède une propriété légèrement faible par rapport à celle de l'Algorithme 5. En effet elle effectue l'ajustement d'une tâche lorsque celle-ci est justifiée par la règle et cet ajustement n'est pas nécessairement maximal à la première itération. C'est l'approche similaire utilisée pour les algorithmes d'edge-finding et d'extended-edge-finding.

Théorème 2.13. *Pour chaque tâche $i \in T$, et l'ajustement maximal de la date de début au plus tôt LB_i tels que spécifiés dans la Définition 1.17, l'Algorithme 6 calcule une borne inférieure LB'_i, telle que $r_i < LB'_i \leq LB_i$ si $r_i < LB_i$, et $LB'_i = r_i$ si $r_i = LB_i$.*

Démonstration. Soit $i \in T$ une tâche. LB'_i est initialisée à r_i (ligne 1). Après chaque détection, les LB'_i sont mises à jour à $\max(LB'_i, minEct)$ (ligne 10). Il s'en suit que $LB'_i \geq r_i$. Si on a $LB_i = r_i$ alors aucune détection n'est faite par la règle, par conséquent, par l'Algorithme 6, et d'après la boucle de la ligne 12, nous avons $LB'_i = r_i$. Dans la suite de la preuve, nous supposerons que $r_i < LB_i$.

Il existe alors un ensemble de tâches $\Omega \subseteq T \setminus \{i\}$ vérifiant :

$$LB_i = ECT_\Omega \wedge e_\Omega + c_i \left(\min\left(ect_i, d_\Omega\right) - r_\Omega\right) > C\left(d_\Omega - r_\Omega\right). \tag{2.65}$$

Soit $j \in \Omega$ une tâche vérifiant $d_j = d_\Omega$. D'après la Définition 2.9 de pseudo coupe gauche de T, $\Omega \subseteq \Psi_j$. Comme démontré dans la Proposition 2.12 et le Théorème 2.12, l'Algorithme 6 détecte correctement les conditions du not-first lorsque l'on considère i à la boucle externe et j à la boucle interne. De la définition de $SLCut(T, j, i)$, nous avons $r_i < ECT_{SLCut(T,j,i)} = ECT_{\Psi_j} = minEct$. Par la suite, après que les conditions de détection soit satisfaites à la ligne 9, la date de début au plus tôt de la tâche i est ajustée à $LB'_i = \max(LB'_i, minEct) = minEct > r_i$. Par conséquent, l'Algorithme 6 est correct et effectue chaque fois un ajustement lorsqu'il est justifié par la règle. □

2.3.4 Complexité pour l'ajustement maximal

D'après le Théorème 2.13, l'Algorithme 6 effectue toujours un ajustement de r_i lorsqu'il est justifié par la règle de not-first. Cependant, l'ajustement effectué n'est pas nécessairement maximal à la première itération. Comme il existe un nombre fini d'ajustements,

l'Algorithme 6 atteint le même point fixe que les autres algorithmes complets de not-first. Cette approche naïve a été récemment utilisée pour réduire la complexité de l'edge-finding en ordonnancement cumulatif [40]. Nous allons maintenant montrer que l'Algorithme 6 effectue l'ajustement maximal après au plus $|H|$ itérations où H désigne l'ensemble de différentes dates de fin au plus tôt des tâches.

Théorème 2.14. *Soit $i \in T$ une tâche d'une instance E-Réalisable de CuSP. Soit H l'ensemble de différentes dates de fin au plus tôt des tâches. Soit $\Omega \subseteq T \setminus \{i\}$ un ensemble de tâches utilisé pour réaliser l'ajustement maximal de r_i par la règle not-first. Alors l'Algorithme 6 effectue l'ajustement maximal de r_i après au plus $|H|$ itérations.*

Démonstration. Soit $j \in T$ une tâche vérifiant $d_j = d_\Omega$. Soit $H_j^k := \{ect_l, l \in SLCut(T, j, i)\}$. Les éléments de H_j^k peuvent être considérés comme des classes de tâches. Un élément ect_l représente l'ensemble des tâches ayant ect_l pour date de fin au plus tôt. l'ensemble des différentes dates de fin au plus tôt des tâches de $SLCut(T, j, i)$ à la k^{th} itération de l'Algorithme 6. Si l'ajustement maximal n'est pas atteint après cette itération, alors au moins un élément est extrait (dans l'ordre croissant) de H_j^k à la $k + 1^{th}$ itération. En effet, si à la k^{th} et à la $k + 1^{th}$ itération, $H_j^k = H_j^{k+1}$ et l'ajustement maximal n'est pas atteint, à aucune de ces itérations, l'Algorithme 6 détecte les conditions du not-first en utilisant l'ensemble $SLCut(T, j, i)$. Ainsi, à la $k + 1^{th}$ itération, aucune nouvelle détection n'est observée malgré le fait que l'ajustement maximal ne soit pas atteint. Ceci contredit le fait que l'Algorithme 6 effectue toujours un ajustement lorsque celui-ci est justifié par la règle not-first. Par conséquent, l'ajustement maximal de la date de fin au plus tôt de la tâche i est atteint après au plus $|H| \leq n$ itérations. □

La règle du not-first/not-last n'étant pas idempotente, l'ajustement des dates de début au plus tôt et de fin au plus tard n'est pas suffisant à la première itération pour atteindre le point fixe. C'est en combinant plusieurs de ces ajustements et ceux réalisés par d'autres propagateurs que l'on atteint le point fixe. Comme on le verra dans le chapitre suivant, les algorithmes itératifs épousent très bien la philosophie de la propagation des contraintes. L'Algorithme 6 approchera rarement sa complexité pour l'ajustement maximal. Ces affirmations sont justifiées par les observations expérimentales ci-dessous.

Chapitre Troisième

Résultats Expérimentaux

Dans ce chapitre, après une description des problèmes d'ordonnancement à moyens limités encore noté RCPSP pour Resource Constrained Project Scheduling Problems, nous présentons le cadre général dans lequel nous effectuons les évaluations des différents algorithmes de filtrage proposés au Chapitre 2.

3.1 Le RCPSP

3.1.1 Description formelle et exemple de RCPSP

Un problème de RCPSP est défini par la donnée d'un ensemble fini de ressources, toutes de capacité finie, un ensemble fini de tâches toutes liées par des contraintes de précédence ou contraintes d'antériorité. Chaque tâche a une durée d'exécution et la quantité de ressources requises pour son exécution. Il est question de déterminer la date de début d'exécution pour chaque tâche de sorte que les contraintes de précédence et de ressource soient satisfaites i.e., pour chaque ressource, à tout instant, la capacité de la ressource reste inviolée. Formellement le problème se définit de la manière suivante :

Définition 3.1. *Une instance de RCPSP est définie par la donnée d'un 6-uplets* (T, R, G, c, p, H) *où*

- T *désigne l'ensemble de* $|T| = n$ *tâches non interruptibles ou non-préemptives ;*
- R *l'ensemble des ressources. Chaque ressource* $k \in R$ *est de capacité* C_k *;*
- G *le graphe acyclique représentant les contraintes de précédence simple de la forme* $i \lessdot j$ *pour signifier que la tâche* j *ne peut commencer qu'après l'exécution complète de la tâche* i*.*

- *c_{ik} ($i \in T$ and $k \in R$) la quantité de la ressource k utilisée par la tâche i tout au long de son exécution ;*
- *p_i ($i \in T$) la durée d'exécution de la tâche $i \in T$;*
- *H l'horizon du projet.*

Une solution d'un RCPSP $P = (T, R, G, c, p, H)$ est une affectation d'une date de début $s_i \in \mathbb{Z}$ pour chaque tâche i telle que

1. *La durée totale du projet soit minimale*
2. *Les contraintes de précédence soient respectées :*

$$\forall i, j \in T \quad si \quad i \lessdot j \quad alors \quad s_j \geq s_i + p_i \tag{3.1}$$

3. *Les contraintes d'utilisation des resources soient respectées :*

$$\forall t \quad \forall k \in R : \sum_{\substack{i \in T \\ s_i \leq t < s_i + p_i}} c_{ik} \leq C_k. \tag{3.2}$$

On représente couramment le projet par un graphe valué, orienté et sans circuit dans lequel les sommets sont les tâches, les arcs les relations de précédence et les poids la durée des tâches. On ajoute à ce graphe deux tâches fictives de durées nulles représentant respectivement le début et la fin du projet. La tâche de début 0 précède toutes les tâches du projet alors que la tâche de fin $n+1$ succède toutes les tâches du projet. On note aussi $i \lessdot j$ pour signifier que la tâche j ne peut commencer qu'après l'exécution complète de la tâche i. Si $G = (V, E, p)$ désigne ce graphe potentiel-tâches, alors $V = T \cup \{0, n+1\}$ et $E = \{(i, j) \ / \ i \lessdot j; i, j \in V\}$.

Le RCPSP appartient à la classe des problèmes combinatoires les plus difficiles. Garey et Johnson [29] ont montré par une réduction au problème de 3-partition que le problème d'ordonnancement à contraintes de ressources, et sans contraintes de précédence, avec une unique ressource, est NP-difficile au sens fort.

Exemple 3.1 (Exemple d'instance du RCPSP)**.** *Nous reprenons ici l'exemple d'une instance de 7 tâches s'exécutant sur 3 ressources proposé par Néron dans [57].*

Dans le réseau à droite, chaque nœud représente une tâche (les tâches 0 et 8 sont les tâches fictives de début et de fin du projet) avec sa consommation sur chacune des

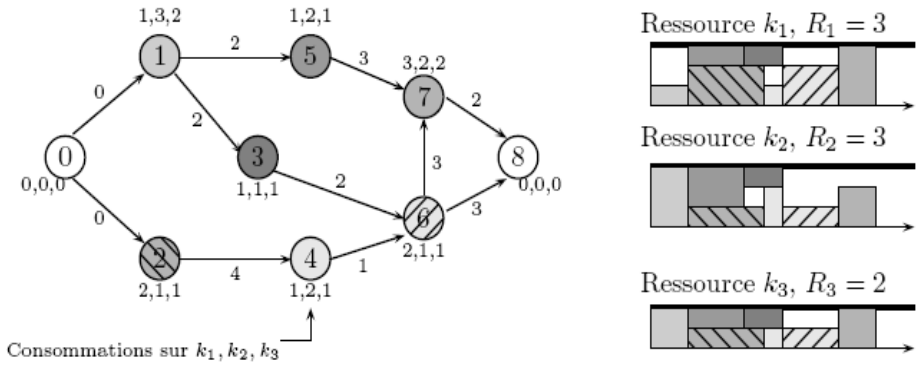


FIGURE 3.1 – Exemple d'instance du RCPSP tiré de [22]

ressources, le coût des arcs représente la durée des tâches. Le diagramme de Gantt à droite est une solution optimale à ce problème.

3.1.2 Méthodes de résolution du RCPSP

Compte-tenu de l'intérêt que suscite le RCPSP, il existe une littérature importante concernant sa résolution, qu'on peut regrouper comme dans [46, 14] en trois catégories :

- les méthodes de calcul des bornes inférieures ;
- les méthodes de calcul des bornes supérieures (résolution approchée) ;
- les méthodes de résolution exactes.

3.1.2.1 Les méthodes de calcul des bornes inférieures

Ces méthodes consistent à autoriser la violation de certaines contraintes du problème ou à la réduction du problème initial à un problème plus simple dont la solution fournit une borne inférieure pour la valeur optimale du problème initial. La relaxation des contraintes de précédence conduit à l'obtention des bornes inférieures basées sur les ressources, comme la borne énergétique, tandis que la relaxation des contraintes de ressources donnent des bornes de chemin critique. Explicitement, il s'agit donc d'élargir l'espace des solutions en autorisant certaines solutions irréalisables pour le problème initial. Nous renvoyons le lecteur au chapitre 3 de [14] et à [59, 31].

3.1.2.2 Les méthodes de calcul des bornes supérieures (résolution approchée)

Ces méthodes sont généralement regroupées en heuristiques et métaheuristiques. Elles sacrifient le caractère optimal de la solution pour obtenir, en un temps de calcul raisonnable, des solutions sous-optimales de bonne qualité. Ces méthodes reposent généralement sur un mécanisme de déplacement (aléatoire ou non) dans l'espace des solutions. Elles ne sont pas exactes, mais permettent en général d'obtenir des solutions proches de l'optimum. Nous renvoyons le lecteur au chapitre 3 de [14, 31].

3.1.2.3 Les méthodes de résolution exactes

Ces méthodes garantissent la complétude de la résolution et le caractère optimal des solutions trouvées, mais sont coûteuses en temps de calcul. Elles se caractérisent par une exploration déterministe de l'espace de solutions. On peut regrouper les travaux sur la résolution exacte du RCPSP en trois grandes familles : les travaux basés sur un branch-and-bound [14, 31, 25], ceux utilisant la programmation par contraintes (PPC) [14, 31, 17, 10, 60, 40, 43] et enfin ceux faisant appel à la programmation linéaire en nombres entiers (PLNE) [46, 31, 4]. Cependant, dans la pratique, on retrouve assez souvent des propositions de modèles combinant la PPC avec la PLNE [22, 31] ou utilisant la PPC en prétraitement dans les branch-and-bound [31].

3.1.3 Quelques extensions du RCPSP [14, 22]

Il existe plusieurs variantes et extensions du RCPSP suivant que l'on accepte la préemption des tâches, l'ordonnancement simultané de plusieurs projets, que l'on généralise les contraintes de précédence ou change le critère à optimiser.

Lorsque toutes les tâches sont interruptibles ou préemptives, le problème est dit préemptif. Une tâche préemptive est une tâche dont on peut suspendre son exécution pour un instant et reprendre bien plus tard. On peut également accorder plus de flexibilité aux tâches en faisant varier leur consommation de ressources pendant leur exécution. Les tâches sont alors décomposées en une partie ininterruptible à consommation fixe et un ensemble de petites tâches interruptibles. Ce nouveau problème est considéré comme une relaxation semi-préemptive du RCPSP.

Dans le cas multi-mode, différentes alternatives sont envisagées, à chacune correspond des durées et consommations fixes. Il peut s'agir d'alternatives temps/ressources (1 machine pendant 8 heures ou 4 machines pendant 2 heures) ou ressources/ressources (2 unités sur une ressource ou 1 unité sur 2 ressources). Capacités, consommations et durées peuvent aussi varier en fonction du temps. Si les capacités varient, on se ramène à considérer le cas des ressources partiellement renouvelables, c'est-à-dire, non-renouvelables sur des intervalles de temps donnés, ce qui généralise les ressources renouvelables et/ou non-renouvelables.

Dans le RCPSP, on peut généraliser les contraintes de précédence. La contrainte de précédence $i \lessdot j$ sera généralisée par la relation :

$$s_i + d_{ij} \leq s_j \tag{3.3}$$

où d_{ij} est un entier arbitraire. L'interprétation de la relation (3.3) dépend du signe de d_{ij}.

- Si $d_{ij} \geq 0$, alors la tâche j ne peut commencer qu'à d_{ij} unités de temps après le début de la tâche i.
- Si $d_{ij} \leq 0$, alors la date de début au plus tôt de la tâche j est $-d_{ij}$ unités de temps avant le début de la tâche i.

Pour $d_{ij} = p_i$, on retrouve la relation de précédence $i \lessdot j$. Ce type de contraintes de précédence complexes correspondent à des situations réelles de processus de fabrication. On peut par exemple imposer qu'une tâche ne puise commencer qu'après qu'une proportion d'une autre (3/4, par exemple) soit terminée.

3.2 Cadre général

Dans ce paragraphe, nous présentons notre approche générale ainsi que les différents ingrédients que nous avons incorporés dans notre approche par contraintes pour la résolution du RCPSP. Les ingrédients introduits ici sont les plus simples car nous limiterons notre discussion au RCPSP standard.

3.2.1 Optimisation

Le RCPSP est un problème d'optimisation. L'objectif est de déterminer une solution avec un makespan (date de fin de projet) minimal. Comme décrit au Chapitre 1 (Paragraphe 1.1.4), plusieurs stratégies peuvent être envisagées pour réduire la valeur de la variable entière *makespan* représentant le makespan. Dans nos expériences, nous utilisons l'horizon du projet tiré des données noté *horizon* pour initialiser le domaine des variables y compris celui du makespan à l'intervalle d'entiers $[0...horizon]$. Nous résolvons alors une succession de CSPs. Si le problème initial n'a pas de solution, alors le problème d'optimisation n'a pas de solution sinon, tant que une solution existe, nous ajoutons à la nouvelle instance de CSP la contrainte suivant laquelle le makespan est inférieur à sa valeur courante. La dernière solution trouvée est une solution optimale du problème.

3.2.2 Stratégie de sélection et de branchement

Une procédure de branchement avec la propagation de contraintes est utilisée à chaque nœud de l'arbre de recherche pour déterminer une solution du CSP. Dans la littérature, il existe plusieurs stratégies de branchement allant des plus complexes aux plus simples [9] pour la résolution du RCPSP. On distinguera chaque fois le branchement dynamique du branchement statique.

- Au cours du branchement dynamique, à chaque nœud de l'arbre de recherche, les variables dont le rapport entre la longueur du domaine et le degré de contraintes sera minimale sont sélectionnées. Pour cette variable pris dans l'ordre chronologique, on choisit sa valeur minimale.
- Pour le branchement statique, on choisira à chaque nœud de l'arbre de recherche la première tâche non ordonnancée (pris dans l'ordre chronologique) et pour celle-ci, on lui attribuera sa plus petite valeur.

La recherche de la date minimale de fin du projet se fera avec du "branch and bound".

3.2.3 Propagation des contraintes

Pour le RCPSP, il existe deux contraintes, à savoir la contrainte de précédence et la contrainte de ressource. La propagation de la contrainte de précédence se fera par une série

de relations linéaires entres les dates de début et de fin des tâches. Les tâches s'exécutent sur des ressources de capacité finie. L'exécution d'un ensemble de tâches sur une ressource sera modélisé avec la contrainte cumulative. Cette contrainte globale est constituée très souvent de plusieurs algorithmes de filtrage tous effectuant différents ajustements. La base de nos propagateurs cumulatifs sera faite de l'edge-finding, du overload checking et du timetabling. On pourra par la suite combiner ces algorithmes de filtrage avec d'autres tels que l'extended edge-finding, le not-first/not-last, le timetable edge finding.

3.3 Evaluation de l'edge-finding

Nous avons mené nos expériences sur les instances de RCPSP obtenues de benchmarks bien connus. Les instances de la librairie PSPLib [49] contenant les ensembles J30, J60, J90 et chacun comprenant 480 instances de 30, 60 et 90 tâches respectivement. Les instances de la librairie BL de Baptiste et Le Pape [8] qui sont réputées fortement cumulatives comprenant 40 instances de 20 et 25 tâches. Pour comparer la vitesse des quatre propagateurs, nous avons analysé le temps nécessaire pour trouver un makespan optimal. Chaque instance a été exécutée trois fois, et la meilleure solution a alors été considérée.

Les tests sont effectués sur un processeur Core i7 intel, CPU 3.07 ghz avec un temps limite de 300 secondes. Les algorithmes de filtrage ont été implémentés en C++ sur le solveur de contraintes Gecode 3.7.3 [30]. Les propagateurs suivants ont été implémentés :

- THETA : C'est le propagateur de la contrainte cumulative de Gecode pour des tâches de durées fixées qui utilise une implémentation de l'algorithme d'edge-finding de complexité $\mathcal{O}(kn \log n)$ de [75], combiné à l'algorithme de vérification de l'intégrité des intervalles (overload checking) et le timetabling.
- QUAD : C'est une version modifiée de THETA dans laquelle l'algorithme de filtrage d'edge-finding utilisant le Θ-tree est substitué par notre nouvel algorithme quadratique d'edge-finding [40].
- MVH : C'est une version modifiée de THETA dans laquelle l'algorithme de filtrage d'edge-finding utilisant le Θ-tree est substitué par l'algorithme d'edge-finding de Mercier et Van Hentenryck [54].
- TTEF : C'est une autre version modifiée de THETA dans laquelle l'algorithme de

Table I Comparaison de THETA, QUAD, MVH, TTEF pour le branchement dynamique.

	J30			J60			J90			BL		
	solve	time	nodes	solve	time	nodes	solve	time	nodes	solve	time	nodes
MVH	377	1	332	334	1	**325**	322	1	**314**	38	0	10
THETA	381	0	333	**336**	0	**325**	322	0	**314**	**40**	0	12
QUAD	**386**	**250**	333	**336**	117	**325**	**324**	61	**314**	**40**	15	12
TTEF	378	136	**364**	327	**219**	316	317	**263**	310	39	**25**	**29**

filtrage d'edge-finding utilisant le Θ-tree est substitué par l'algorithme du timetable edge finding de Vilím [77].

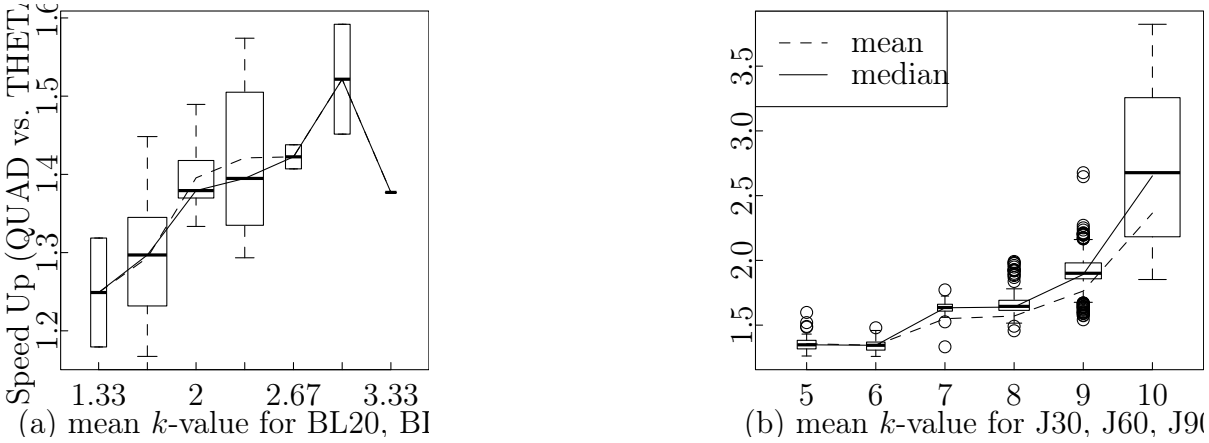
3.3.1 Branchement dynamique

3.3.1.1 Le temps CPU

Le Tableau I compare les quatre propagateurs lorsque le branchement est dynamique. Il y est indiqué le nombre d'instances résolues (solve), rapidement (time), avec très peu de nœuds (nodes). Le propagateur QUAD est capable de déterminer la solution optimale (avant le temps limite) dans la majorité des instances étudiées. De plus sur J30, QUAD a le meilleur temps dans plus d'instances que tous les autres propagateurs. Par contre, sur les ensembles J60, J90 et BL, TTEF est meilleur en général. L'écart de performance entre QUAD et TTEF s'agrandit d'avantage en faveur de TTEF sur les instances réputées difficiles (J60 et J90).

La comparaison des propagateurs QUAD et THETA montre que QUAD est meilleur dans tous les cas. Dans le propagateur THETA, la complexité de l'algorithme d'edge-finding dépend de k (le nombre de différentes consommations en ressource des tâches). On s'attendrait qu'au cours de nos évaluations, la performance de QUAD croisse avec la valeur de k. La Figure 3.2 compare le rapport des vitesses de QUAD sur THETA en fonction de la valeur moyenne de k sur les instances de la librairie (a) BL et (b) PSPLib. L'aire des rectangles est proportionnelle à la racine carrée du nombre d'instances

pour chaque valeur de k. Aux extrémités des rectangles, sont représentés les données les plus hautes/basses à moins de 1,5 fois l'écart interquartile, les valeurs marginales sont individuellement représentées. Nous avons représenté séparément BL et PSPLib car ces instances sont générées avec des paramètres différents. Il apparaît que chaque instance a soit 3 (BL) ou 4 (J30, J60, J90) ressources avec différentes valeurs de k.



(a) mean k-value for BL20, BI (b) mean k-value for J30, J60, J9(

FIGURE 3.2 – QUAD/THETA en fonction de k sur (a) BL et (b) PSPLib.

Dans la plupart des cas, TTEF est plus rapide que QUAD. Mais, avec un peu de recul, lorsqu'on examine la distribution du rapport de la vitesse de QUAD sur TTEF (Figure 3.3 (a)), on s'aperçoit que les temps d'exécution sont très proches dans la plupart des cas. La partie bleue de chaque bare représente les instances pour lesquelles au moins une fois, l'un des propagateurs a modifié le domaine d'une variable et la partie rouge la totalité des instances où la solution optimale a été trouvée. Il y existe aussi plusieurs cas où QUAD est plus rapide que TTEF. Ceci peut être dû à l'utilisation d'un branchement dynamique. C'est pour cela que nous avons effectué plus loin une comparaison similaire des deux propagateurs pour un branchement statique.

3.3.1.2 Les nœuds

Le Tableau II donne un résultat plus détaillé sur la comparaison du nombre de nœuds dans l'arbre de recherche de la solution optimale des différents propagateurs. Dans ce tableau, nous indiquons le nombre de fois où le propagateur de gauche a généré moins de nœuds que celui de dessus sur chacune des instances, la totalité des instances étant prise en compte. Seules les instances pour lesquelles les deux propagateurs considérés ont

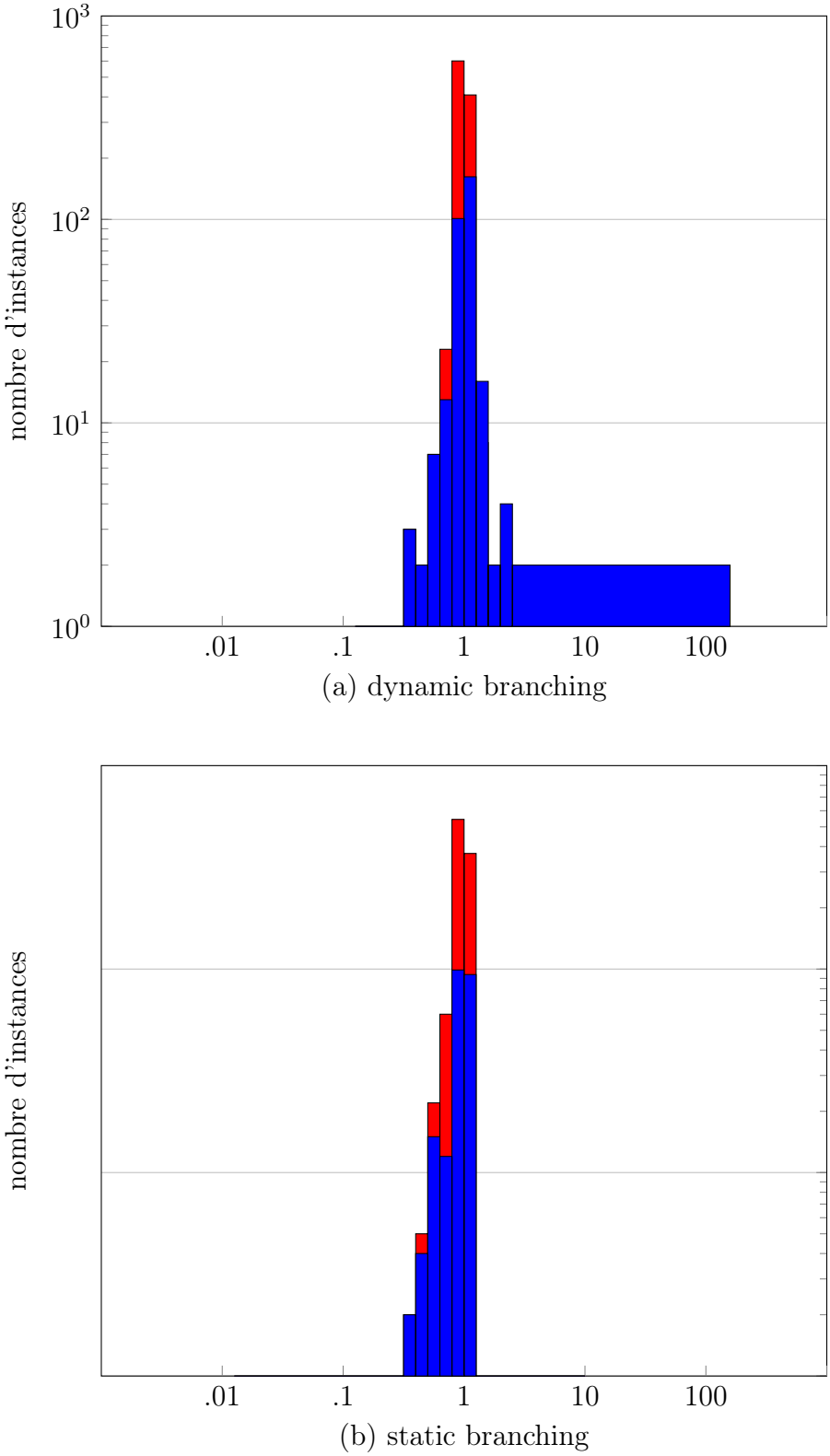


FIGURE 3.3 – Distribution de la vitesse de QUAD sur TTEF avec branchement (a) dynamique, et (b) statique.

trouvé la solution optimale avant le temps limite sont reportées dans ce tableau.

Dans [40], nous signalions qu'il y a un petit nombre d'instances sur lequel le nombre de nœuds entre THETA et QUAD diffère, particulièrement, dans environ 1% des cas. Ceci est contradictoire au résultat obtenu ici car il apparaît que les deux propagateurs ont exactement le même nombre de nœuds, donc atteignent le même point fixe à chaque nœud de l'arbre de recherche. Une suite d'enquêtes a révélé que cette différence de nœuds était dû à l'implémentation de l'algorithme de Θ-tree de [75] qui (dans les versions antérieurs à 3.7.2) manquait quelques ajustements (pour une description plus détaillée des manquements voir [67]).

En utilisant Gecode 3.7.3, le nombre de nœuds de QUAD et THETA est identique dans tous les cas, exactement comme on l'avait espéré pour deux propagateurs effectuant les mêmes ajustements. La différence de nœuds entre ces deux propagateurs et MVH était aussi à espérer. Ceci est dû à la règle (EF1) ajoutée seulement plus tard dans les deux autres algorithmes.

Le nombre de nœuds de TTEF et QUAD est différent sur plusieurs instances, en faveur de l'un comme de l'autre. Ceci prouve que les deux algorithmes effectuent différents degrés d'ajustement, confirmant ainsi la non domination du timetable edge finding sur l'edge-finding.

Table II Comparaison détaillé du nombre de nœuds dans l'arbre de recherche de la solution optimale.

	MVH	THETA	QUAD	TTEF
MVH	—	0	0	43
THETA	13	—	0	46
QUAD	13	0	—	46
TTEF	89	92	96	—

Table III Comparaison de THETA, QUAD et TTEF pour le branchement statique.

	J30			J60			J90			BL		
	solve	time	nodes	solve	time	nodes	solve	time	nodes	solve	time	nodes
THETA	340	0	317	311	0	301	**316**	0	309	33	0	1
QUAD	345	**191**	319	312	105	301	**316**	71	309	**35**	3	1
TTEF	**348**	157	**343**	**315**	**211**	**308**	**316**	**245**	**314**	**35**	**32**	**35**

3.3.2 Branchement statique

3.3.2.1 Le temps CPU et le nombre de nœuds

Dans l'optique de minimiser l'effet du branchement sur la propagation des contraintes (avec un branchement dynamique, même un faible algorithme de filtrage peut avoir de très bon résultats), nous avons testé nos algorithmes avec un branchement statique. Pour ce cas, nous avons laissé de côté le propagateur MVH qui était un peu plus lent que les autres.

Le Tableau III résume le nombre d'instances dont chacun des propagateurs a déterminé la solution optimale avant le temps limite (solve), avec un meilleur temps (time), en parcourant un arbre de recherche réduit (nodes) pour le branchement statique.

Etant donné que les propagateurs THETA et QUAD effectuent les mêmes ajustements, le changement de système de branchement n'a aucune influence sur ces propagateurs. C'est pourquoi nous allons nous focaliser sur la comparaison de TTEF et QUAD.

En général, avec un branchement statique, le propagateur TTEF reste meilleur dans la plupart des cas mais avec une différence qu'ici, l'écart est un peu réduit par rapport au cas du branchement dynamique. Le Tableau III montre que TTEF est en mesure de solutionner 6 instances supplémentaires par rapport à QUAD. Sur J30, QUAD reste le propagateur le plus rapide alors que sur J60, J90 et BL, TTEF est le plus rapide. Sur 36 instances de PSPLib et 34 instances de BL, il y a réduction du nombre de nœuds du propagateur TTEF alors qu'il n'existe pas d'instances sur lesquelles le nombre de nœuds de QUAD est inférieur à celui de TTEF.

La Figure 3.3 (b) montre la distribution du rapport de la vitesse de QUAD sur TTEF

pour le branchement statique. On s'aperçoit que le résultat est similaire à celui du branchement dynamique. La distribution a un pick au dessus de 1 (représentant le cas où QUAD et TTEF ont pratiquement le même temps d'exécution) et un étalage vers la gauche (représentant le cas où TTEF est meilleur par rapport à QUAD). Il apparaît qu'en général, TTEF est plus rapide que QUAD mais cette fois ci avec une marge d'écart un peu réduite. Le temps d'exécution de TTEF est dans la plupart des cas au plus égal à 2 fois celui de QUAD.

3.3.2.2 Nombre de propagations

Nous avons démontré au Chapitre 2 que la complexité de notre algorithme pour l'ajustement maximal est de $\mathcal{O}(n^3)$. Donc, ce n'est qu'après n itérations que notre algorithme pourra effectuer ce que THETA fait en une itération. Dans la pratique, notre algorithme nécessite rarement des itérations supplémentaires pour effectuer l'ajustement maximal. Nous avons pour cela compté le nombre de propagations de chaque propagateur. La Figure 3.4 résume les résultats. Dans la plupart des cas, QUAD et TTEF ont le même nombre de propagations que THETA. Il y a une faible proportion d'instances où le nombre de propagations est différent et cette différence est assez faible. On remarque qu'il y a des cas pour lesquels les propagateurs itératifs (QUAD ou TTEF) ont un nombre réduit de propagations par rapport au propagateur complet THETA. On peut aussi observer que TTEF est plus itératif que QUAD comme l'avait déjà indiqué Vilím dans [77].

3.4 Evaluation de l'extended edge-finding

Il est démontré dans [9, 10] que l'edge-finding ne domine pas l'extended edge-finding. C'est pourquoi il est nécessaire de combiner les deux algorithmes de filtrage pour plus de filtrage. Il est question ici d'évaluer l'apport de l'extended edge-finding lorsqu'il est combiné à l'edge-finding. Pour cela nous comparons notre algorithme quadratique d'extended edge-finding combiné à l'edge-finding à notre algorithme d'edge-finding seul [40]. Les propagateurs suivants ont été implémentés :

- EEF : C'est le propagateur de la contrainte cumulative pour des tâches de durées fixées qui utilise une implémentation de l'algorithme d'edge-finding de complexité

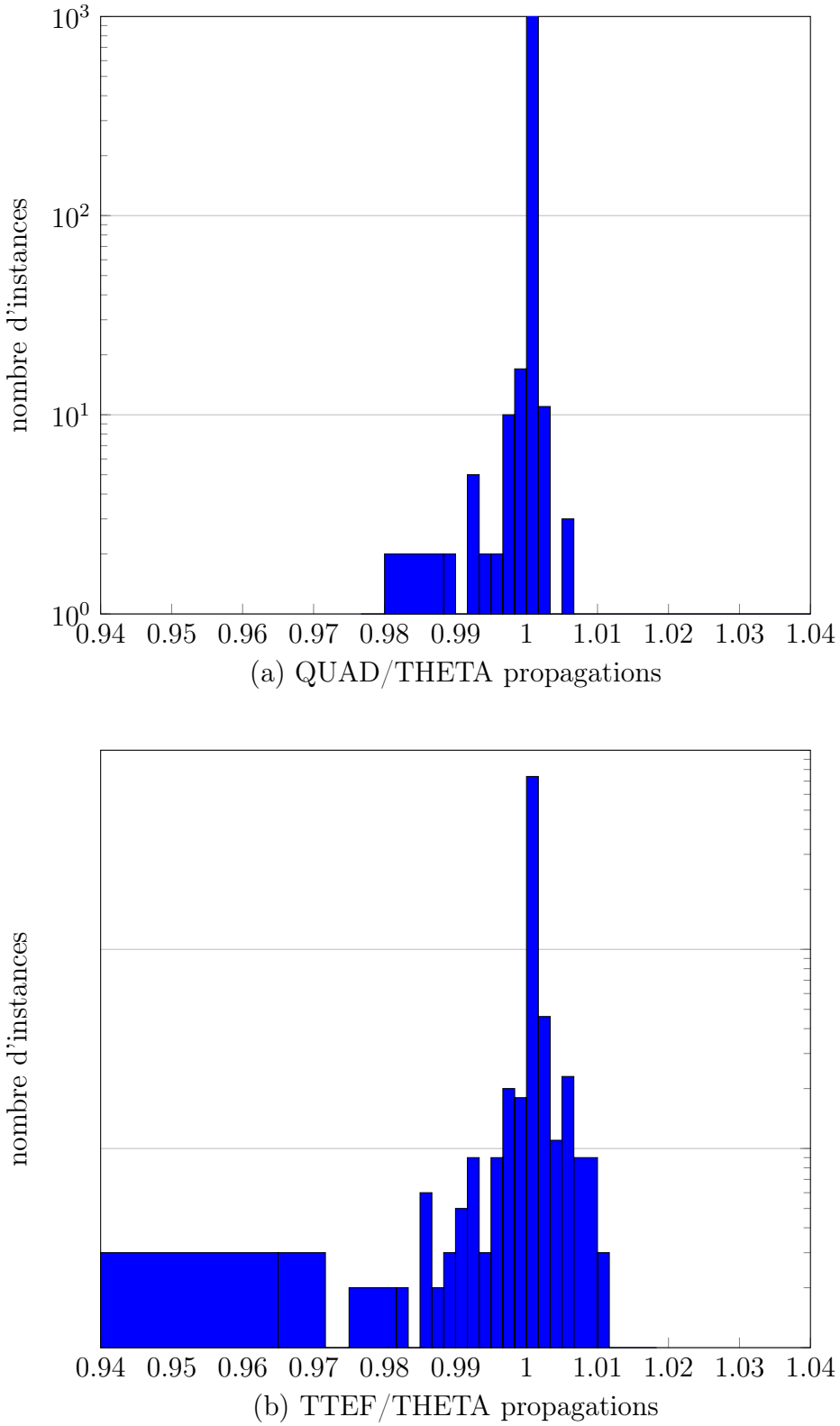


FIGURE 3.4 – Comparaison du nombre de propagations de THETA vs. (a) QUAD, et (b) TTEF. Seules les instances ayant le même nombre de nœuds sont considérées.

$\mathcal{O}(n^2)$ de [40] et l'algorithme d'extended edge-finding de [43] (c'est l'algorithme quadratique proposé dans cette thèse), combiné à l'algorithme de vérification de l'intégrité des intervalles (overload checking) et le timetabling. A chaque point fixe de l'edge-finding, nous exécutons l'Algorithme 4 et ce procédé est répété jusqu'à ce que le point fixe global soit atteint.

– EF : C'est une version modifiée de THETA dans laquelle l'algorithme de filtrage d'edge-finding utilisant le Θ-tree est substitué par l'algorithme d'edge-finding de complexité $\mathcal{O}(n^2)$ de [40].

Les tests ont été effectués sur les ensembles J30, J60, J90 des instances de la librairie bien connue PSPLib [49] et l'ensemble BL de la librairie de Baptiste et Le Pape [8], sur un Intel Pentium(R) Dual Cores processor, CPU 280 GHz, 1 GB de RAM avec un temps limite de 120 secondes. De plus, toutes les instances de J30, J60 et J90 (resp. BL) nécessitant plus de 5000 backtracks (resp. 20000 backtracks) sont comptées comme des échecs.

Nous distinguons pour cette évaluation empirique deux types de branchement : le branchement dynamique et le branchement statique.

– Pour le branchement dynamique, on choisissons à chaque nœud de l'arbre de recherche, la tâche dont la longueur du domaine sur le degré de contraintes est minimale. S'il en existe plusieurs vérifiant ce critère, alors nous choisissons la tâche ayant la plus petite borne supérieure. Pour cette tâche, nous lui attribuons sa plus petite valeur.
– Au branchement statique, nous choisissons la première tâche non ordonnancée prise dans l'ordre chronologique. Pour cette tâche, nous lui attribuons sa plus petite valeur.

3.4.1 Branchement dynamique

3.4.1.1 Le temps CPU

Le Tableau IV donne le nombre d'instances pour lequel chaque propagateur détermine la solution optimale (solve), avec le temps CUP le plus réduit (time), et en parcourant le plus petit nombre de nœuds (nodes). Il apparaît que EF est en général plus rapide, mais il existe quelques instances sur lesquelles EEF est plus rapide que EF. On remarque

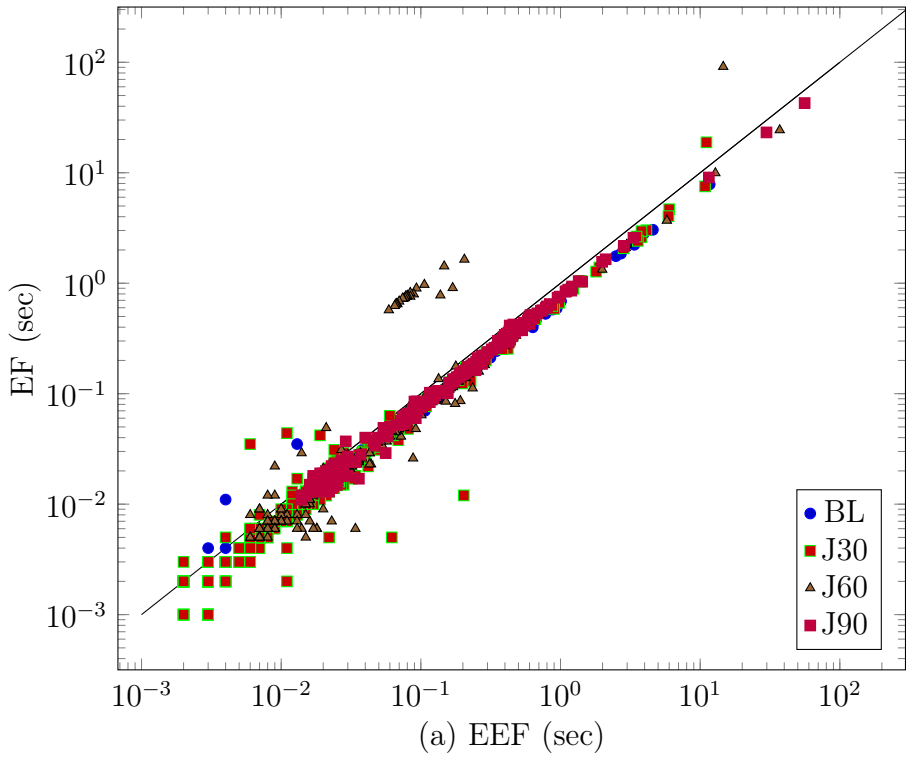


FIGURE 3.5 – Temps d'exécution de EF vs. EEF pour les instances résolues par les deux propagateurs lorsque le branchement est dynamique.

également que EEF parcourt exactement le même nombre de nœuds que EF sur J30, J60 et J90.

Dans les Figures 3.7 et 3.5 les coordonnées des points du nuage de points sont :

- la durée d'exécution du propagateur EEF pour l'abscisse ;
- la durée d'exécution des propagateurs EF pour l'ordonnée.

La droite d'équation $y = x$ (la première bissectrice) est représentée pour servir de référentiel. Ainsi, lorsqu'un point se trouve au dessus de cette droite, alors la caractéristique du propagateur en abscisse est inférieure à celle du propagateur en ordonnée.

La Figure 3.5 compare le temps d'exécution de EF et EEF pour le branchement dynamique. Il apparaît que EF est plus rapide que EEF dans la plupart des instances, mais sur quelques instances, le propagateur EEF est plus rapide que EF. La vitesse du propagateur EF est en moyenne 1,474 fois plus rapide que celle du propagateur EEF sur BL, 1,16 fois plus rapide sur J30 et 1,296 fois plus rapide sur J90. Par contre, le propagateur EEF est en moyenne plus rapide que EF sur J60 avec un facteur de 1,57.

Table IV Comparaison de EF et EEF pour la recherche d'une solution optimale lorsque le branchement est dynamique.

	J30			J60			J90			BL		
	solve	time	nodes	solve	time	nodes	solve	time	nodes	solve	time	nodes
EF	**354**	**335**	0	**334**	**301**	0	**321**	**319**	0	**30**	**25**	0
EEF	**354**	19	0	**334**	33	0	**321**	2	0	**30**	0	**4**

3.4.1.2 Le nombre de nœuds

D'après le Tableau IV, la combinaison de l'edge-finding et de l'extended edge-finding n'apporte aucun gain en terme de nœuds sur les instances des ensembles J30, J60 et J90. Par contre, sur les instances de l'ensemble BL, 4 des 30 instances résolues ont vu leur nombre de nœuds réduit. Ceci est compréhensif car les instances de BL sont réputées fortement cumulatives [8].

3.4.1.3 Le nombre de propagations

D'après la Figure 3.6 (a), la combinaison de l'extended edge-finding et de l'edge-finding a réduit le nombre de propagations dans certains cas du branchement dynamique. Mais en majorité, les deux propagateurs ont le même nombre de propagations.

3.4.2 Branchement statique

3.4.2.1 Le temps CPU

Comme pour le cas dynamique, le Tableau V donne le nombre d'instances pour lesquelles chaque propagateur détermine la solution optimale (solve), avec le temps CUP le plus réduit (time), et en parcourant le plus petit nombre de nœuds (nodes). Il apparaît que EF est en général plus rapide, mais il existe quelques instances sur lesquelles EEF est plus rapide que EF. On remarque également que EEF parcourt exactement le même nombre de nœuds que EF dans tous les cas.

La Figure 3.7 compare le temps d'exécution de EF et EEF pour le branchement sta-

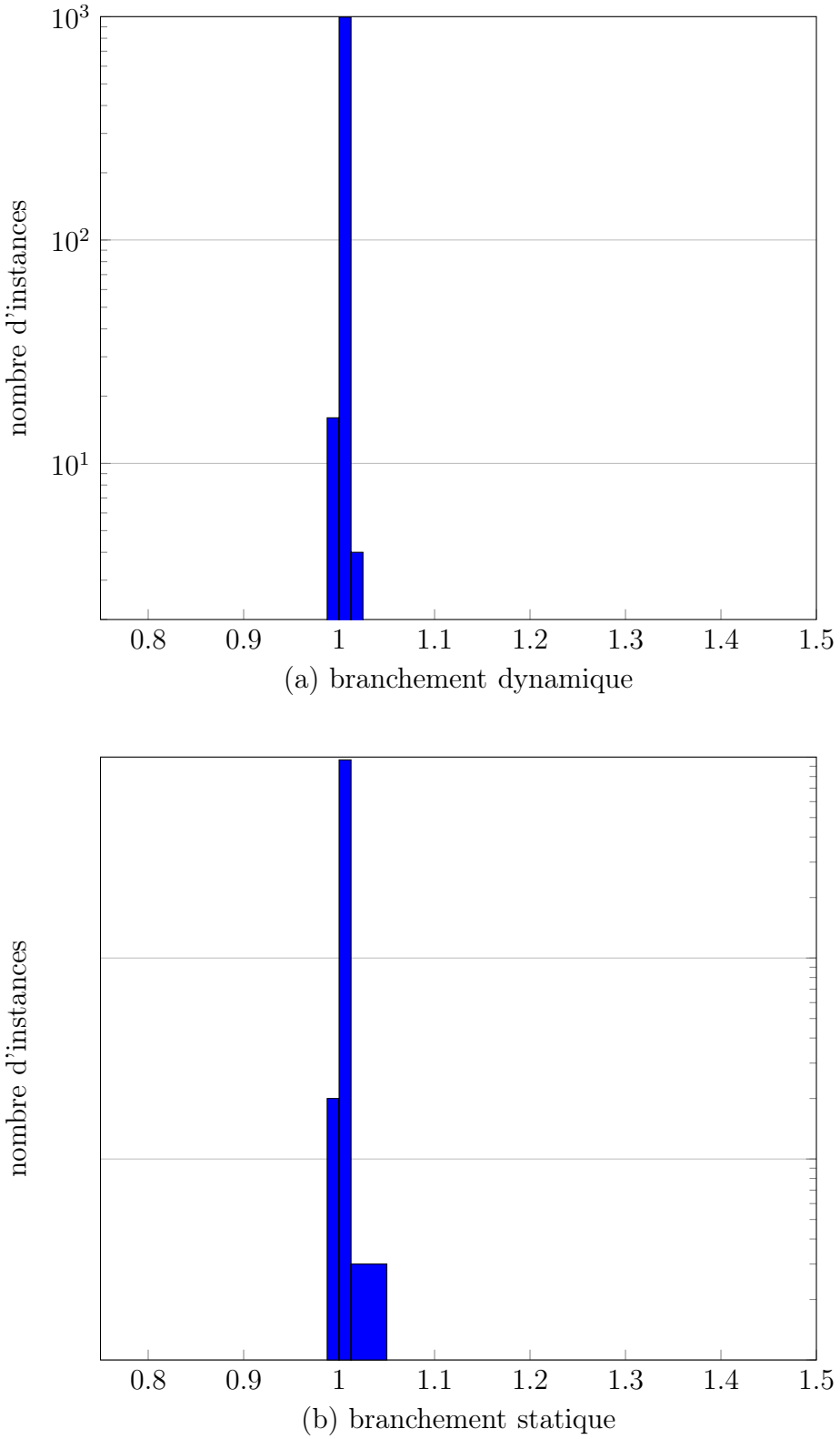


FIGURE 3.6 – Distribution représentant le rapport du nombre de propagations de EF sur EEF pour le branchement (a) dynamique et (b) statique.

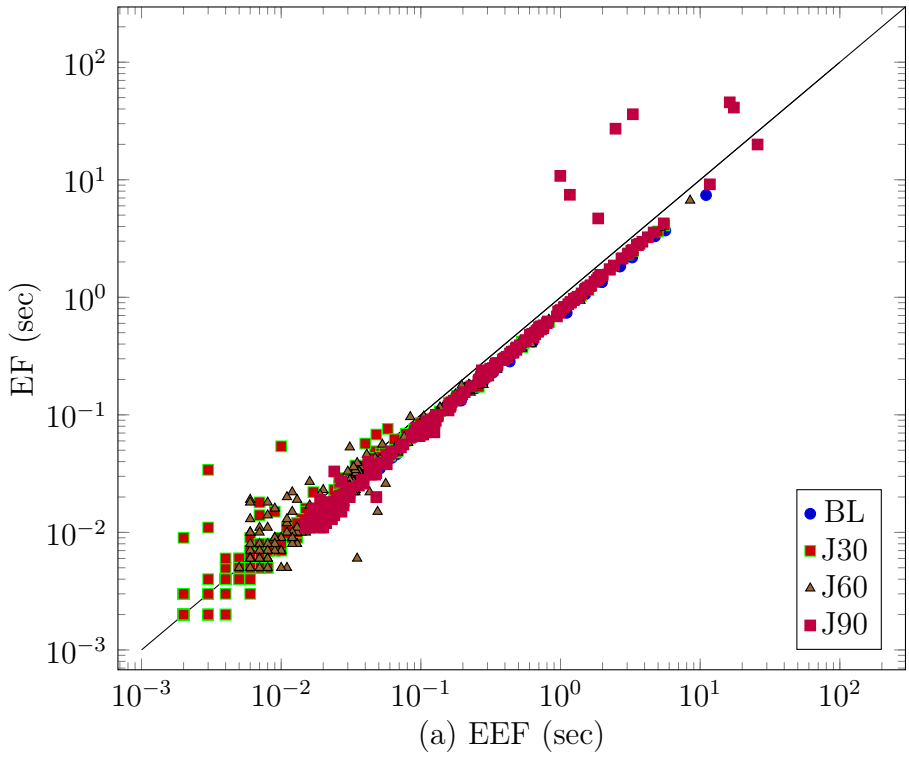


FIGURE 3.7 – Temps d'exécution de EF vs. EEF pour les instances résolues par les deux propagateurs lorsque le branchement est statique.

tique. Il apparaît que EF est plus rapide que EEF dans la plupart des instances, mais sur quelques instances, le propagateur EEF a été plus rapide que EF. La vitesse du propagateur EF est en moyenne 1,479 fois plus rapide que celle du propagateur EEF sur BL, 1,303 fois plus rapide sur J30, 1,294 fois plus rapide sur J60. Par contre, le propagateur EEF est en moyenne plus rapide que EF sur J90 avec un facteur de 1,33.

3.4.2.2 Le nombre de nœuds

Lorsque les effets du branchement dynamique sont annulés, on remarque que, la combinaison de l'edge-finding et de l'extended edge-finding n'apporte aucun gain en terme de nœuds sur toutes les instances de nos deux librairies (voir Tableau V).

3.4.2.3 Le nombre de propagations

Comme c'était déjà le cas pour le branchement statique, la Figure 3.6 (b) montre que la combinaison de l'extended edge-finding et de l'edge-finding a réduit le nombre de propagations dans certains cas. Mais en majorité, les deux propagateurs ont le même

Table V Comparaison de EF et EEF pour la recherche d'une solution optimale lorsque le branchement est statique.

	J30			J60			J90			BL		
	solve	time	nodes	solve	time	nodes	solve	time	nodes	solve	time	nodes
EF	**332**	**256**	0	**319**	**280**	0	**325**	**315**	0	**23**	**23**	0
EEF	**332**	76	0	**319**	39	0	**325**	10	0	**23**	0	0

nombre de propagations.

3.5 Evaluation du not-first/not-last

Nous comparons dans cette section les deux algorithmes de not-first/not-last présentés au Chapitre 2. Il est aussi question d'évaluer l'apport du not-first/not-last lorsqu'il est combiné à l'edge-finding.

Les tests ont été effectués sur les ensembles J30, J60 et J90 des instances de la librairie PSPLib [49] et l'ensemble BL de la librairie de Baptiste et Le Pape [8]. Les tests sont effectués sur un Intel Pentium(R) Dual Cores processor, CPU 280 GHz, 1 GB de RAM avec un temps limite de 120 secondes. Toutes les instances de J30, J60 et J90 (res. BL) nécessitant plus de 5000 backtracks (resp. 20000 backtracks) sont comptées comme des échecs. Nous considérons uniquement le branchement dynamique, i.e., choisissons la tâche pour laquelle le rapport de la longueur du domaine sur le degré de contraintes est minimal. S'il existe plusieurs tâches vérifiant ce critère, alors nous choisissons la tâche ayant la plus petite borne supérieure. Pour cette tâche, nous lui attribuons sa plus petite valeur.

Les propagateurs suivants ont été implémentés :

- EF : C'est le propagateur de la contrainte cumulative pour des tâches de durées fixées qui utilise une implémentation de l'algorithme d'edge-finding de complexité $\mathcal{O}(n^2)$ de [40] combinée avec le timetabling et l'overload checking.
- N2L : C'est une version modifiée de EF dans laquelle l'algorithme itératif de not-first/not-last (Algorithme 6) est ajouté. A chaque nœud de l'arbre de recherche, si le point fixe de l'edge-finding est atteint, alors l'Algorithme 6 [38, 42] est exécuté.

Ce procédé est répété jusqu'à ce que le point fixe global soit atteint.

– N2HL : C'est une version modifiée de N2L dans laquelle l'algorithme itératif de not-first/not-last (Algorithme 6) est substitué à l'algorithme complet de not-first/not-last (Algorithme 5) [39].

3.5.1 Le temps CPU

Le Tableau VI donne le nombre d'instances pour lesquelles chaque propagateur détermine la solution optimale (solve), avec le temps CUP le plus réduit (time), et en parcourant le plus petit nombre de nœuds (nodes). Il apparaît que EF est en général plus rapide, mais il existe quelques instances sur lesquelles N2L est plus rapide que EF. On remarque également que N2L parcourt très peu de nœuds que EF dans la plupart des instances de l'ensemble BL.

Dans les Figures 3.8 (a) et (b), les coordonnées des points du nuage de points sont :

– la durée d'exécution du propagateur N2L pour l'abscisse ;

– la durée d'exécution des propagateurs EF ou N2HL respectivement pour l'ordonnée.

La droite d'équation $y = x$ (la première bissectrice) est représentée pour servir de référentiel. Ainsi, lorsqu'un point se trouve au dessus de cette droite, alors la caractéristique du propagateur en abscisse est inférieure à celle du propagateur en ordonnée.

La Figure 3.8 (a) compare le temps d'exécution de EF et N2L. Il apparaît que EF est plus rapide que N2L dans la plupart des instances, mais sur quelques instances de J60 et BL, le propagateur N2L est plus rapide que EF. La moyenne des temps d'exécution du propagateur EF sur celle du propagateur N2L est de 0,687 sur BL, 0,541 sur J30, 0,724 sur J60 et 0,371 sur J90.

La comparaison du temps d'exécution de N2L et N2HL est donnée sur la Figure 3.8 (b). Nos résultats montrent que le propagateur N2L est plus rapide que N2HL dans tous les cas avec une vitesse moyenne croissante avec la taille de l'instance. La moyenne des temps d'exécution du propagateur N2L sur celle du propagateur N2HL est de 3,21 sur BL ; 5,42 sur J30 ; 9,40 sur J60 et 13,93 sur J90.

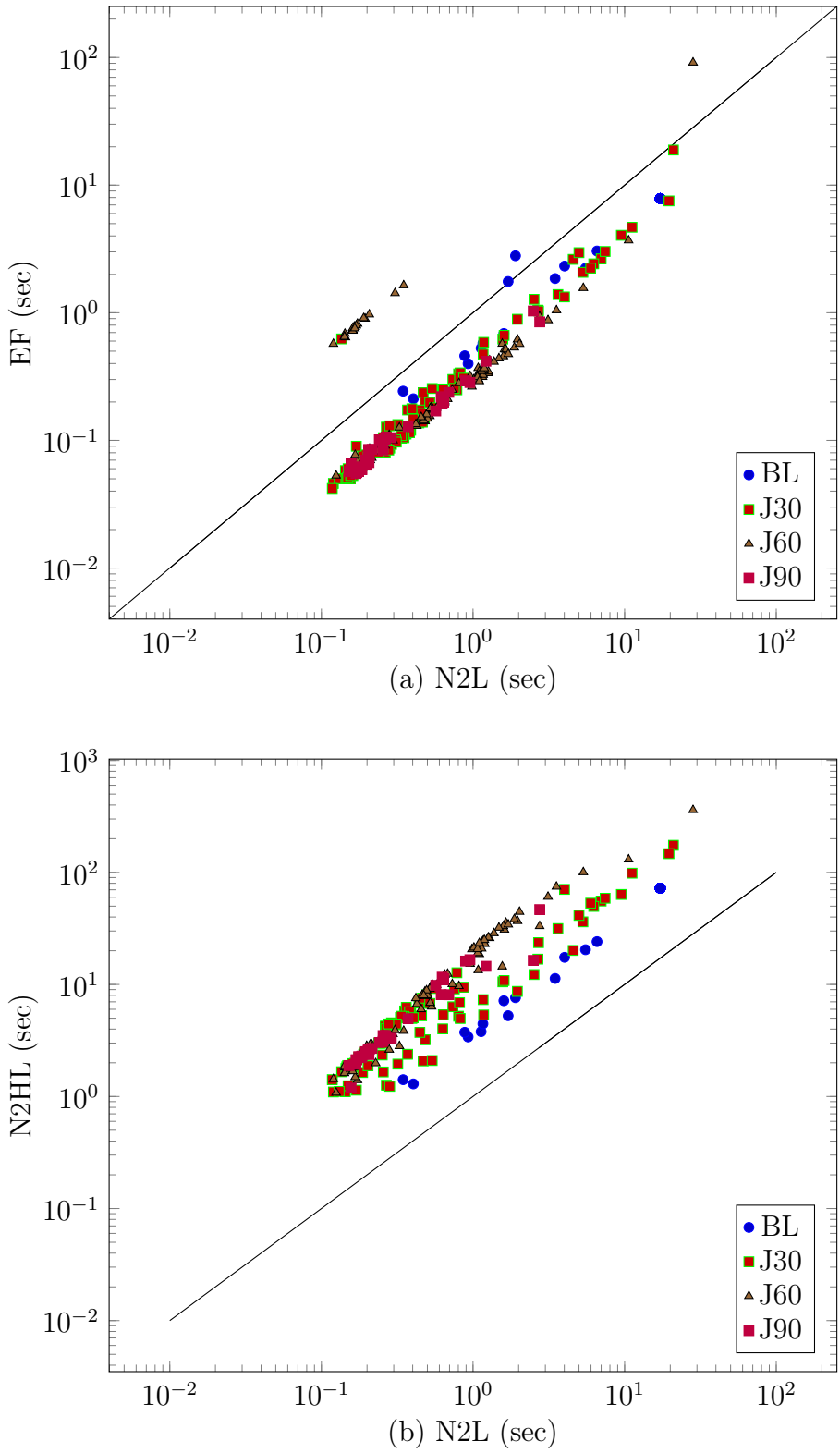


FIGURE 3.8 – Temps d'exécution de (a) N2L vs. EF et (b) N2L vs. N2HL pour les instances résolues par les deux propagateurs.

Table VI Comparaison de EF, N2L et N2HL pour la recherche d'une solution optimale.

	J30			J60			J90			BL		
	solve	time	nodes	solve	time	nodes	solve	time	nodes	solve	time	nodes
EF	**354**	**346**	0	**333**	**304**	0	**322**	**321**	0	**30**	**25**	0
N2L	**354**	8	**12**	**333**	29	**3**	321	0	**6**	**30**	5	**20**
N2HL	352	0	**12**	331	0	**3**	316	0	**6**	**30**	0	**20**

3.5.2 Le nombres de nœuds

D'après le Tableau VI, la combinaison de l'edge-finding et du not-first/not-last a permis la réduction du nombre de nœuds dans l'arbre de recherche par rapport à l'utilisation de l'edge-finding seul sur la majorité des instances de BL. Ceci est comprehensible car les instances de BL sont réputées fortement cumulatives [8], par conséquent plus adéquat pour la propagation. 67% des instances de BL sont influencées par l'ajout du l'algorithme de not-first/not-last alors que seulement 2,1% des instances de la librairie PSPLib le sont. (voir Figure 3.9).

Pour ce qui est de la comparaison du nombre de nœuds de N2L et N2HL, il apparaît que les deux propagateurs ont le même nombre de nœuds. Ceci implique que les deux propagateurs atteignent le même point fixe à chaque nœud de l'arbre de recherche.

3.5.3 Le nombre de propagations

D'après la Figure 3.10 (a), la combinaison du not-first/not-last et de l'edge-finding a réduit le nombre de propagations dans certains cas. Le nombre de propagations de EF est toujours inférieur où égal au nombre de propagations de N2L ou N2HL. Ceci se traduit par l'étalement de la distribution à la droite de 1 et peut être interprété comme suit : certains ajustements du not-first/not-last sont puissants au point de réduire le nombre de propagations de tout le système.

Dans la Figure 3.10 (b), On peut observer que les propagateurs N2HL et N2L ont le même nombre de propagations dans la large majorité des cas. Mais, il existe quelques instances sur lesquelles le nombre de propagations de N2HL est inférieur à celui de N2L.

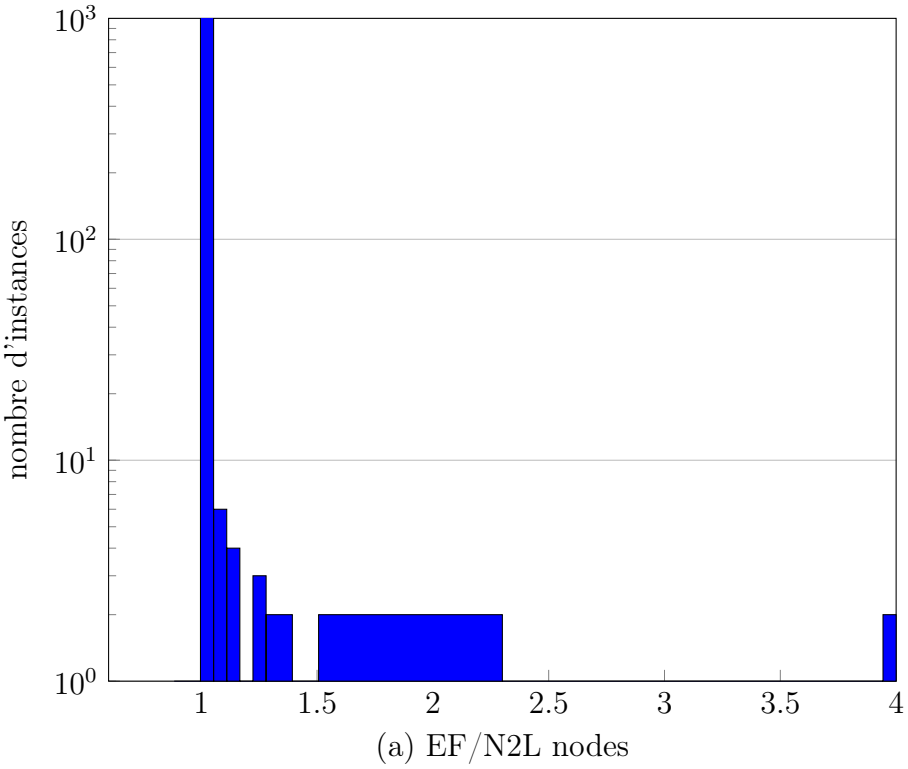


FIGURE 3.9 – Distribution représentant le rapport du nombre de nœuds de EF vs. N2L

Ceci est tout à fait normal, car on se rappelle que N2L contient le not-first/not-last itératif qui effectue l'ajustement maximale de N2HL après $|H|$ itérations où H désigne l'ensemble de différentes dates de fin au plus tôt des tâches. Le nombre d'itérations requis par N2L pour l'ajustement maximal atteint rarement $|H|$ dans la pratique.

3.6 Conclusion

Dans ce chapitre, nous avons effectué une évaluation empirique des algorithmes présentés au Chapitre 2. Il ressort que, malgré la complexité cubique pour l'ajustement maximal, notre algorithme quadratique d'edge-finding est toujours plus rapide que l'algorithme Θ-tree d'edge-finding de Vilím. Nous avons également apporté une preuve que le timetable edge finding ne domine pas l'edge-finding. Les résultats montrent aussi que lorsque notre algorithme quadratique d'extended edge-finding est combiné à l'edge-finding, il y a dans cetains cas réduction du temps d'exécution et pour les instances fortement cumulatives, réduction du nombre de nœuds. Enfin, malgré le fait que nos algorithmes de not-first/not-last ont la même complexité pour l'ajustement maximal, l'algorithme itératif est plus

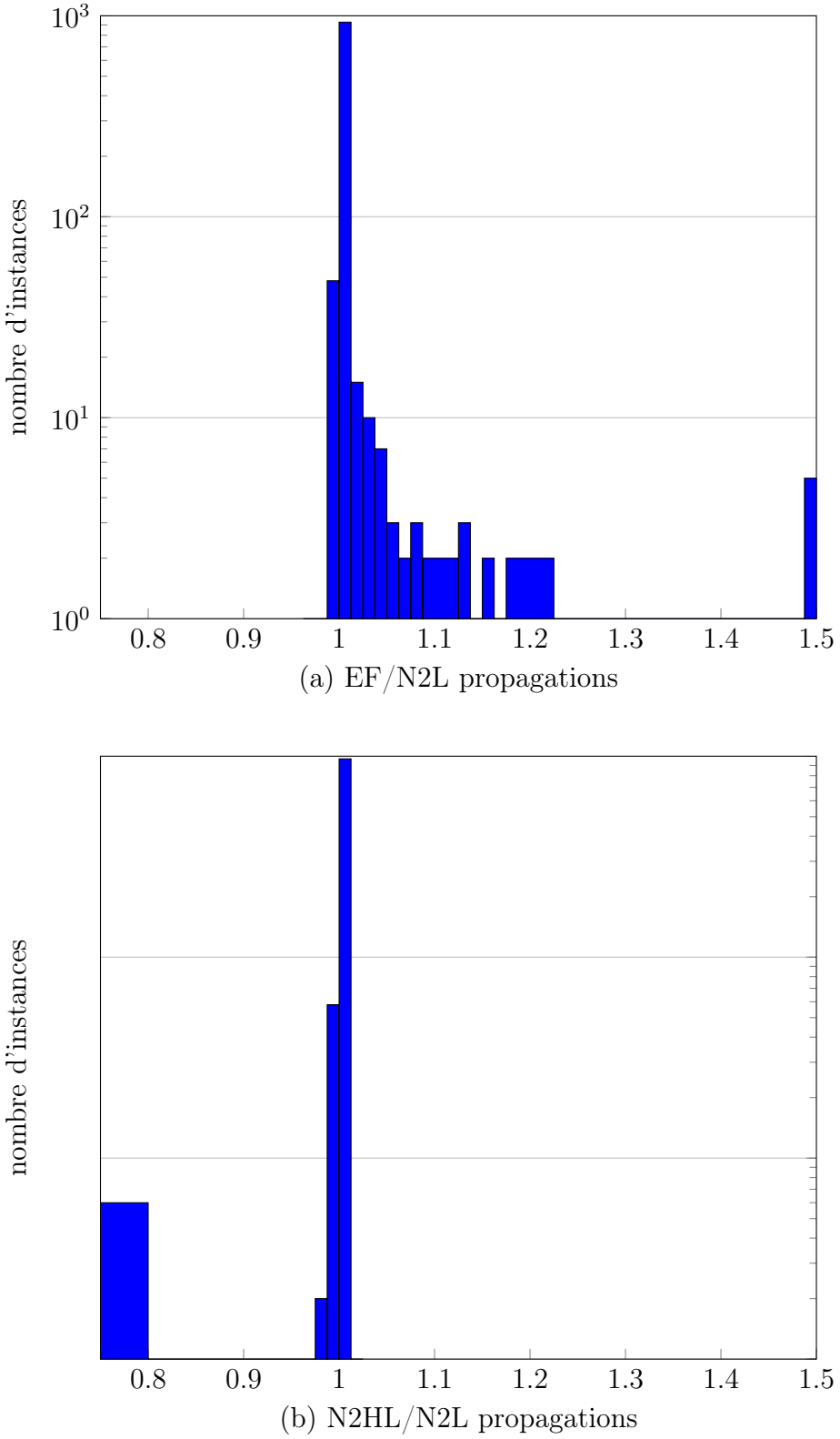


FIGURE 3.10 – Distribution représentant le rapport du nombre de propagations de (a) EF sur N2L, (b) N2HL sur N2L.

rapide dans la pratique. L'apport de cette règle lorsqu'elle est combinée à l'edge-finding est significatif en terme de temps et de nœuds.

Conclusion générale

Dans cette thèse, nous avons présenté plusieurs algorithmes de filtrage de la contrainte de ressource pour les règles d'edge-finding, d'extended edge-finding et de not-first/not-last. Pour chacune des règles, nous avons mené une comparaison théorique et expérimentale des différents algorithmes de propagation avec ceux de la littérature.

Pour ce qui est de l'edge-finding, nous avons proposé un algorithme quadratique basé sur une approche originale utilisant les notions de densité maximale et de marge minimale. Le nouvel algorithme n'effectue pas nécessairement l'ajustement maximal à la première itération, mais s'améliore aux itérations subséquentes. Alors que la complexité de notre algorithme ne domine pas strictement celle de Vilím [75], les résultats expérimentaux montrent qu'il est en moyenne trois fois plus rapide que celui-ci. Nous avons démontré que le timetable edge finding ne domine pas l'edge-finding. Ce qui contredit le résultat annoncé par Vilím [77] suivant lequel la conjonction de l'edge-finding et l'extended edge-finding est dominée par le timetable edge finding.

Dans le cas de l'extended edge-finding, nous avons montré que la seconde phase de l'algorithme d'extended edge-finding de Mercier et Van Hentenryck [54] est incorrect. Nous avons proposé un algorithme quadratique d'extended edge-finding similaire à celui de l'edge-finding. Le nouvel algorithme utilise la notion de marge minimale et est plus efficace au point fixe de l'edge-finding. Nous avons montré que le point fixe de nos algorithmes d'edge-finding et d'extended edge-finding n'est pas dominé par la conjonction des deux règles. Comme pour l'edge-finding, nous avons démontré que le timetable edge finding ne domine pas l'extended edge-finding. Ce qui contredit le résultat annoncé par Vilím [77].

Dans le cas du not-first/not-last, nous proposons deux algorithmes l'un complet et l'autre itératif. L'algorithme complet est de complexité $\mathcal{O}(n^2|H|\log n)$, [39] améliorant

ainsi la complexité du meilleur algorithme complet de la littérature qui est $\mathcal{O}(n^3 \log n)$ [64] car $|H| \leq n$. Nous proposons par la suite un algorithme itératif de not-first/not-last de complexité $\mathcal{O}(n^2 \log n)$ n'effectuant pas nécessairement l'ajustement maximal à la première itération. Alors que sa complexité pour l'ajustement maximal est de $\mathcal{O}(n^2|H| \log n)$, les résultats expérimentaux montrent que ce nouvel algorithme est plus rapide que la version complète.

Des trois règles de filtrage étudiées dans cette thèse, il ressort que dans chaque cas, les algorithmes itératifs lorsqu'ils sont de bonne complexité sont plus rapides que les algorithmes complets dans la pratique. Ce fut déjà le cas pour le not-first/not-last disjonctif [70, 74]. Il serait intéressant d'adapter de telles approches pour les autres variantes des problèmes d'ordonnancement cumulatifs (RCPSP/max etc...) afin de voir si ce résultat se généralise dans les autres cas.

Alors que les techniques de propagation des contraintes sont montrées très efficaces dans le cas disjonctif, elles restent encore prometteuses dans la cas cumulatif. C'est pourquoi, il serait intéressant d'explorer la possibilité d'intégrer dans ces algorithmes de filtrage, du raisonnement énergétique (tout en conservant la complexité) pour booster le propagateur et réduire davantage l'arbre de recherche. Un début de solution est déjà proposé dans [36, 44, 77], mais un approfondissement de cette voie apportera probablement un bon résultat.

Malgré nos efforts, la complexité des algorithmes de propagation que nous avons proposés reste élevée (quadratique dans le meilleur des cas) en comparaison au cas disjonctif. Nous comptons poursuivre nos recherches sur la résolution des problèmes d'ordonnancement avec la programmation par contraintes pour réduire davantage la complexité des algorithmes de filtrage et augmenter leur puissance par l'ajout du raisonnement énergétique.

Bibliographie

[1] Aggoun, A. and Beldiceanu, N. : Extending CHIP in order to solve complex scheduling and placement problems. Mathematical and Computer Modelling, 17(7) :57-73, (1993).

[2] Applegate, D. and Cook W. : A computational study of job-shop scheduling, ORSA Journal on Computing, 3(2) :149-156, (1991).

[3] Apt, K. : Principles of Constraint Programming.University Press, Cambridge, England, (2003).

[4] Artigues, C. : Optimisation et robustesse en ordonnancement sous contraintes de ressources, HdR, Université d'Avignon et des Pays de Vaucluse, (2004).

[5] Artigues, C., Huguet, MJ. and Lopez, P. : Generalized disjunctive constraint propagation for solving the job shop problem with time lags. Engineering Applications of Artificial Intelligence, 24(2) :220-231, (2010).

[6] Bacchus, F. and Van Beek, P. : On the conversion between Non-Binary and Binary Constraint Satisfaction Problems. in Proc. National Conference on Artifical Intelligence (AAAI-98), Madison, Wisconsin, (1998).

[7] Baker, K.R. and Trietsch, D. : Principles of Sequencing and Scheduling. John Wiley & Sons, Hoboken, New Jersey (2009).

[8] Baptiste, P. and Le Pape, C. : Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. In : Smolka, G. (ed.) CP97. LNCS, vol. 1330. Springer, Heidelberg (1997).

[9] Baptiste, P. : Une étude théorique et expérimentale de la propagation de contraintes de ressources, Thèse de doctorat, Université Technologique de Compiègne, (1998).

[10] Baptiste, P., Le Pape, C. and Nuijten, W. : Constraint-based scheduling : applying constraint programming to scheduling problems. Kluwer, Boston (2001).

[11] Baptiste, P. : Résultats de Complexité et Progammation par Contraintes pour l'Ordonnancement. HDR thesis. Compiègne University of Technology. (2002).

[12] Beldiceanu, N. and Carlsson, M. : A new multi-resource cumulatives constraint with negative heights. In Proceedings CP'02, pages 63-79, Ithaca, NY, USA , (2002).

[13] Beldiceanu, N., Carlsson, M. and Rampon, J.-X. : Global constraint catalog. Technical Report T2005-06, Swedish Institute of Computer Science, (2005).

[14] Brucker, P. and Knust, S. : Complex scheduling. Springer-Verlag Berlin, Heidelberg, (2006).

[15] Carlier, J. and Pinson, E. : Adjustment of Heads and Tails for the Job-shop Problem. European Journal of Operational Research,78 :146-161. (1994)

[16] Caseau, Y. and Laburthe, F. : Improved CLP scheduling with task intervals. In Van Hentenryck, P. (ed.) ICLP94, pp. 369–383. MIT Press, Boston (1994).

[17] Caseau, Y. and Laburthe, F. : Cumulative scheduling with task intervals. In Proceedings of the Joint International Conference and Symposium on Logic Programming, pages 363-377. The MIT Press, (1996).

[18] Chip v5. : `http://www.cosytec.com`.

[19] Choco Team. : Choco : an open source java constraint programming library. `http://choco.mines-nantes.fr`, (2011).

[20] Christofides, N. : Graph theory : an algorithmic approach, Academic Press, London, (1975).

[21] Conway, R. W., Maxwell, W. L. and Miller, L. W. : Theory of Scheduling. Addison-Wesley. Massachusetts, (1967).

[22] Demassey, S. : Méthodes hybrides de programmation par contraintes et de pro- grammation linéaire pour le problème d'ordonnancement de projet à contraintes de ressources. Thèse de doctorat, Université d'Avignon, (2003).

[23] Dechter, R. : Constraint Processing. Morgan Kaufmann, San Francisco,(2003).

[24] Dorndorf, U., Pesch. E. and Phan Huy, T. : Constraint propagation techniques for the disjunctive scheduling problem, Artificial Intelligence, 122 :189-240, (2000).

[25] Dorndorf, U., Pesch. E. and Phan Huy, T. : A branch-and-bound algorithm for the resource-constrained project scheduling problem, Math Meth Oper Res 52 : 413-439, (2000).

[26] The eclipse constraint programming system. `http://eclipseclp.org`.

[27] Esquirol, P., Lopez, P. et Huguet M.-J. : Ordonnancement de la production, chapitre Propagation de contraintes en ordonnancement, Hermès Science Publications, Paris, (2001).

[28] Fruhwirth, T. and Abdennadher, S. : Essentials of Constraint Programming. Springer, Berlin, (2003).

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

[30] Gecode. `http://www.gecode.org`. Accessed 30 August, 2012.

[31] Gotha-groupe RCPSP. : Gestion de projet à contraintes de ressources : approches et méthodes de résolution, `http://www.ocea.li.univ-tours.fr/eocea/rcpsp`, (2006).

[32] Hidri, L., Gharbi, A. and Haouari, M. : Energetic reasoning revisited : application to parallel machine scheduling. Journal of Scheduling 11, pp. 239-252, (2008).

[33] Hooker, J. N. : Integrated Methods for Optimization, Springer Science + Business Media, LLC, 233 Spring Street, New York, NY 10013, USA, (2007).

[34] IBM : IBM Ilog Cplex CP Optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/`, (2011).

[35] Kameugne, R. and Fotso, L. P. : Not-First/Not-Last Algorithm For Cumulative Scheduling Problems. In Proceedings of the joint International Conference FRANCORO/ROADEF, pages 149-155, (2007).

[36] Kameugne, R. and Fotso, L. P. : Energetic edge-finder for cumulative resource constraint. Proceeding of CPDP 2009 Doctoral Program, pages 54-63, Lisbone, Portugale, (2009).

[37] Kameugne, R., Kouakam, E. and Fotso, L. P. : La Condition Not-First/Not-Last en Ordonnancement Cumulatif. In proceeding of ROADEF 2010, pages. 89-97 , Toulouse, France (2010).

[38] Kameugne, R. and Fotso, L. P. : A not-first/not-last algorithm for cumulative resource in $\mathcal{O}(n^2 \log n)$. Accepted to CPDP 2010 Doctoral Program, St Andrews, Scotland (2010).

[39] Kameugne, R. and Fotso, L. P. : A Complete Filtering Algorithm for Cumulative Not-First/Not-Last rule in $\mathcal{O}(n^2|H| \log n)$. Proceeding of CSCLP 2010, pages 31-42, Berlin, Germany, (2010).

[40] Kameugne, R., Fotso, L. P., Scott, J., Ngo-Kateu, Y. : A Quadratic Edge-Finding Filtering Algorithm for Cumulative Resource Constraints. In : Lee, J. (ed.) CP 2011, LNCS vol. 6876, pp. 478–492. Springer, Heidelberg (2011).

[41] Kameugne, R., Fotso, L. P., Scott, J., Ngo-Kateu, Y. : A Quadratic Edge-Finding Filtering Algorithm for Cumulative Resource Constraints, (Extended version of the Kameugne et al. (2011)), *Constraints*. Vol. **18**. No. 4, Springer, (2013). In press, DOI :10.1007/s10601-013-9157-z.

[42] Kameugne, R. and Fotso, L. P. : A Cumulative Not-First/Not-Last Filtering Algorithm in $\mathcal{O}(n^2 \log(n))$, Indian J. Pure Appl. Math., **44**(1) : pp 95-115. Springer-Verlag (2013), DOI :10.1007/s13226-013-0005-z.

[43] Kameugne, R., Fotso, L. P., Scott, J. : A Quadratic Extended Edge-Finding Filtering Algorithm for Cumulative Resource Constraints, *International Journal of Planning and Scheduling*. Vol. **1**. No. 4, pp. 264-284, (2013). `http://www.inderscience.com/info/ingeneral/forthcoming.php?jcode=ijps`

[44] Kameugne, R., Fetgo, B. S. and Fotso, L. P. "Energetic Extended Edge Finding Filtering Algorithm for Cumulative Resource Constraints," American Journal of Operations Research, Vol. 3 No. 6, 2013, pp. 589-600. DOI : 10.4236/ajor.2013.36056.

[45] Rinnooy Kan, A. H. G. : Machine Scheduling Problems : Classification, Complexity and Computation. Nijhoff, The Hague, (1976).

[46] Koné, O. : Nouvelles approches pour la résolution du problème d'ordonnancement de projet à moyens limités. PhD thesis, Université Toulouse III - Paul Sabatier, (2009).

[47] Kooli, A., Haouari, M., Hidri, L. and Néron, E. : IP-based energetic reasoning for the resource constrained project scheduling problem. Electronic Notes in Discrete Mathematics 36 (2010) 359-366 ISCO 2010 - International Symposium on Combinatorial Optimization, (2010).

[48] Kumar, V. : Algorithms for constraint satisfaction problems : A survey. AI Magazine, 13(1) :32-34, (1992).

[49] Kolisch, R. and Sprecher, A. : PSPLIB – A project scheduling problem library. European Journal of Operational Research, 96(1) :205–216, (1997).

[50] Laborie, P. : Algorithms for propagation resource constraints in AI planning and scheduling : Existing approaches and new results. Artificial Intelligence, 143 :151 ?188, (2003).

[51] Lopez, P. Approche énergétique pour l'ordonnancement de tâches sous contraintes de temps et de ressources, Thèse de doctorat, Université Paul Sabatier, Toulouse, (1991).

[52] Lopez, P. : Approche par contraintes des problèmes d'ordonnancement et d'affectation : structures temporelles et mécanismes de propagation, HdR, Institut National Polytechnique de Toulouse, (2003).

[53] Lhomme, O. : Consistency Techniques for Numeric CSPs. Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, Chambéry, France, (1993).

[54] Mercier, L. and Van Hentenryck, P. : Edge Finding for Cumulative Scheduling. INFORMS Journal on Computing 20(1), pp. 143-153, (2008).

[55] Meseguer, P. : Constraint satisfaction problems : An overview. AI Communications, 2 : 3-17, (1989).

[56] Mistral. `http://www.4c.ucc.ie/~ehebrard/mistral/doxygen/html/main.html`.

[57] Néron, E. : Du Flow-shop hybride au problème cumulatif , Thèse de doctorat, Université Technologique de Compiègne, (1999).

[58] Néron, E., Baptiste, P. and Gupta, J.N.D. : Solving hybrid flow shop problem using energetic reasoning and global operations, Research report, University of Technology of Compiègne, (2000).

[59] Néron, E., Artigues, C., Baptiste, P., Carlier, J., Demassey, S and Laborie, P. : Perspectives in Modern Project Scheduling, volume 92 de International Series in Operations Research & Management Science, chaptitre Lower bounds computation for RCPSP. Józefowska and Joanna and Weglarz and Jan (Eds.), springer édition, (2006).

[60] Nuijten, W. : Time and resource constrained Scheduling : A constraint Satisfaction Approach. PhD thesis, Eindhoven University of Technology. (1994).

[61] Régin, J. C. : Modélisation et Contraintes Globales en Programmation par Contraintes. Hdr, Université de Nice, (2004).

[62] Rossi, F., Dahr, V. and Petrie, C. : On the equivalence of constraint satisfaction problems. in Proc. European Conference on Artificial Intelligence (ECAI-90), Stockholm, 1990. Also MCC Technical Report ACT-AI-222-89. (1990).

[63] Rossi, F., van Beek, P. and Walsh, T. : editors. Handbook of Constraint Programming. Foundations of Artificial Intelligence. Elsevier, Amsterdam, (2006).

[64] Schutt, A., Wolf, A. and Schrader, G. : Not-first and Not-last Detection for Cumulative Scheduling in $O(n^3 log(n))$. Proceedings 16th. International Conference on Applications of Declarative Programming and Knowledge,INAP 2005, 66-80, Fukuoka, Japan. (2005).

[65] Schutt, A. and Wolf, A. : A New $O(n^2 \log n)$ Not-First/Not-Last Pruning Algorithm for Cumulative Resource Constaints. Principles and Practice of Constraint Programming - CP 2010, Lecture Notes in Computer Science, 2010, Volume 6308/2010, 445-459, DOI : 10.1007/978-3-642-15396-9_36 (2010).

[66] Schutt, A., Feydy, T., Stuckey, P.J., and Wallace, MG . : Explaining the cumulative propagator. Constraints, 16(3) :250-282, (2011).

[67] Scott, J. : Filtering Algorithms for Discrete Cumulative Resources. Masters Thesis, Uppsala University and SICS, (2010).

[68] Sicstus prolog. `http://www.sics.se/isl/sicstuswww/site/index.html`.

[69] Stergiou, K.and Walsh, T. : Encodings of Non-Binary Constraint Satisfaction Problems. in Proc. National Conference on Artifical Intelligence (AAAI-99), Orlando, Florida, (1999).

[70] Torres, P. and Lopez, P. : On Not-First/Not-Last Conditions in Disjunctive Scheduling. European Journal of Operational Research, Vol 127(2) :332–343. (2000).

[71] Tsang, E. P. K. Foundations of Constraint Satisfaction. Academic Press, New York, (1993).

[72] Ugo Montanari. : Network of Constraints : Fundamental Properties and Applications to Picture Processing. Information Sciences, 7 :95-132, (1974).

[73] Van Hentenryck, P., Deville, Y. and Teng, C. M. : A General Arc-Consistency Algorithm and its Specializations. Artificial Intelligence, 57(3) :291-321, (1992).

[74] Vilím, P. : Global Constraints in Scheduling. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, (2007).

[75] Vilím, P. : Edge Finding Filtering Algorithm for Discrete Cumulative Resources in $O(kn \log n)$. In : Ian P. Gent (ed) CP 2009. LNCS 5732. pp. 802-816. Springer, Heidelberg (2009).

[76] Vilím, P. : Max energy filtering algorithm for discrete cumulative resources. In : van Hoeve, W.J., Hooker, J.N. (eds) CPAIOR 2009. LNCS, vol. 5547. pp. 66-80. Springer, Heidelberg (2009).

[77] Vilím, P. : Timetable Edge Finding Filtering Algorithm for Discrete Cumulative Resources. In : Achterberg T., Beck J.C., eds., CPAIOR 2011, LNCS, vol. 6697, pp. 230-245. Springer (2011).

[78] Wolf, A. and Schrader, G. : $O(n \log n)$ overload checking for the cumulative constraint and its application. In 16th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2005, pages 88-101, Fukuoka, Japan, October 2005. Springer. ISBN 3-540-69233-9. (2005).

Liste des publications

Articles de Conférences

1. Kameugne, R and Fotso, L. P. : Not-first/not-last algorithm for cumulative scheduling problems. In Proceedings of the join International Conference FRANCORO/ROADEF, pages 149-155, (2007).
2. Kameugne, R. and Fotso, L. P. : Energetic edge-finder for cumulative resource constraint. Proceeding of CPDP 2009 Doctoral Program, pages 54-63, Lisbone, Portugale, (2009).
3. Kameugne, R., Kouakam, E. and Fotso, L. P. : La condition not-first/not-last en ordonnancement cumulatif. In proceeding of ROADEF 2010, pages. 89-97 , Toulouse, France (2010).
4. Kameugne, R. and Fotso, L. P. : A not-first/not-last algorithm for cumulative resource in $\mathcal{O}(n^2 \log n)$. Accepted to CPDP 2010 Doctoral Program, St Andrews, Scotland (2010).
5. Kameugne, R. and Fotso, L. P. : A complete filtering algorithm for cumulative not-first/not-last rule in $\mathcal{O}(n^2|H| \log n)$. Proceeding of CSCLP 2010, pages 31-42, Berlin, Germany, (2010).
6. Kameugne, R., Fotso, L. P., Scott, J., Ngo-Kateu, Y. : A quadratic edge-finding filtering algorithm for cumulative resource constraints. In : Lee, J. (ed.) CP 2011, LNCS vol. 6876, pp. 478–492. Springer, Heidelberg (2011).

Articles de Journaux

1. Kameugne, R. and Fotso, L. P. : A Cumulative Not-First/Not-Last Filtering Algorithm in $\mathcal{O}(n^2 \log(n))$, Indian J. Pure Appl. Math., **44**(1) : pp 95-115. Springer-Verlag (2013), DOI :10.1007/s13226-013-0005-z.

2. Kameugne, R., Fotso, L. P., Scott, J., Ngo-Kateu, Y. : A Quadratic Edge-Finding Filtering Algorithm for Cumulative Resource Constraints, (Extended version of the Kameugne et al. (2011)), *Constraints*. Vol. **19**. No. 3, pp 243-269. Springer, (2014), DOI :10.1007/s10601-013-9157-z.

3. Kameugne, R., Fotso, L. P., Scott, J. : A Quadratic Extended Edge-Finding Filtering Algorithm for Cumulative Resource Constraints, *International Journal of Planning and Scheduling*. Vol. **1**. No. 4, pp. 264-284, (2013). `http://www.inderscience.com/info/ingeneral/forthcoming.php?jcode=ijps`

4. Kameugne, R., Fetgo, B. S. and Fotso, L. P. : Energetic Extended Edge Finding Filtering Algorithm for Cumulative Resource Constraints, *American Journal of Operations Research*, Vol. **3**. No. 6, 2013, pp. 589-600. DOI : 10.4236/ajor.2013.36056.

Annexe

Cette thèse a été déposée en Février 2012 et les premiers rapports (04) tous positifs obtenus en Juin 2012. Malheureusement, c'est en Janvier 2014 que nous avons déposé une nouvelle fois le travail suite à la création du Centre de Recherche et de Formation Doctoral (Sciences, Technologies et Géosciences). Nous avons obtenu une autorisation de soutenance en Août 2014 et la soutenance à eu lieu le 30 Septembre 2014.

Pendant cette période seul notre papier [1] était accepté mais ce n'est qu'en Février 2013 qu'elle a parue. Les autres publications [2, 3] tirées de la thèse, ont été finalisées durant la longue attente et l'extension annoncé dans les perspectives amorcé dans [4]. Au cours de cette même période, plusieurs travaux ont été réalisés citant notre algorithme d'edge-finding. C'est par exemple la cas des thèses de Alexis De Clercq et de Arnaud LETORT [5, 6] et le mémoire de master de Pierre Ouellet [7]. Dans [5, 6], les auteurs s'intéressent particulièrement du problème de balayage alors que Pierre Ouellet propose un algorithme d'extended edge-finding de complexité $\mathcal{O}(nk\log(n))$ qu'il étant à celui du timetable-extended edge-finding sans modifier la complexité. Cette extension est basée sur la notion de décomposition des tâches et augmente la puissance de la règle de filtrage.

Bibliographie

[1] Kameugne, R. and Fotso, L. P. : A Cumulative Not-First/Not-Last Filtering Algorithm in $\mathcal{O}(n^2 \log(n))$, Indian J. Pure Appl. Math., **44**(1) : pp 95-115. Springer-Verlag (2013), DOI :10.1007/s13226-013-0005-z.

[2] Kameugne, R., Fotso, L. P., Scott, J., Ngo-Kateu, Y. : A Quadratic Edge-Finding Filtering Algorithm for Cumulative Resource Constraints, (Extended version of the Kameugne et al. (2011)), *Constraints*. Vol. **19**. No. 3, pp 243-269. Springer, (2014), DOI :10.1007/s10601-013-9157-z.

[3] Kameugne, R., Fotso, L. P., Scott, J. : A Quadratic Extended Edge-Finding Filtering Algorithm for Cumulative Resource Constraints, *International Journal of Planning and Scheduling*. Vol. **1**. No. 4, pp. 264-284, (2013). `http://www.inderscience.com/info/ingeneral/forthcoming.php?jcode=ijps`

[4] Kameugne, R., Fetgo, B. S. and Fotso, L. P. : Energetic Extended Edge Finding Filtering Algorithm for Cumulative Resource Constraints, *American Journal of Operations Research*, Vol. **3**. No. 6, 2013, pp. 589-600. DOI : 10.4236/ajor.2013.36056.

[5] Alexis De Clercq. : Ordonnancement cumulatif avec dépassements de capacité Contrainte globale et décompositions. Thèse de Doctorat, École des mines de Nantes, Université de Nantes Angers Le Mans (2012).

[6] Arnaud LETORT. : Passage à l'échelle pour les contraintes d'ordonnancement multi-ressources. Thèse de Doctorat, École des mines de Nantes, Université de Nantes Angers Le Mans (2013).

[7] Pierre Ouellet. : Le filtrage des bornes pour les contraintes Cumulative et Multi-Inter-Distance. Mémoire de Master, Université Lava, (2014).